An Operational Theory of Relative Space Efficiency

Adam Bakewell

Submitted for the degree of Doctor of Philosophy

University of York

Department of Computer Science

December 2001
Abstract

This thesis presents a theory for defining and comparing the space usage of different program evaluation strategies.

Language implementations that automate memory management, especially those with dynamic memory management, can have surprising or unpredictable space behaviour. Programs which use more space than expected are said to have *space leaks*.

GraphKit is an equational term-graph rewriting system for specifying the operational semantics of an implementation technique as a graph evaluator. A GraphKit evaluator and garbage collector define the *space semantics* of an implementation accurately enough for analysis of space complexity.

Several features of the non-strict functional language Haskell and implementation techniques for lazy evaluation are modelled in GraphKit.

Space leaks are defined formally as a relative property of two evaluators with respect to a *translation*. This general-purpose definition includes worse space complexity through poor evaluation strategy, through inadequate garbage collection, through space-unsafe program transformation and through space-inefficient encoding of data or algorithms.

A framework is developed for defining translations between different graph languages such that translation preserves space complexity. These translations form the basis of *simulation* proofs that one evaluator is not leakier than another. The limitations of this technique are explored and solutions to some of its restrictions are investigated. Space complexity preserving transformations are also presented as a means of specifying non-standard garbage collection strategies.

Space leaks are classified according to the behaviour that exposes them. An automated procedure is described which proves that an evaluator is leakier than another for those leaks which are exposed by a single witness program. Such leaks are classified further and a search procedure is presented to find proofs of *active* space leaks.

The no-leak and leak proof techniques are applied to show how the space behaviours of the GraphKit models of lazy evaluation are related.

**Keywords** Space usage, space complexity, operational semantics, garbage collection, space leaks, term-graph rewriting, disequality, simulation proof, lockstep evaluation, completion, non-termination analysis, proof search, proof planning, lazy evaluation, functional programming.
# Contents

Contents

List of Figures

List of Tables

Acknowledgements

Declaration

1 Introduction

1.1 The space usage problem ........................................ 11

1.2 Contributions and organisation ................................. 12

## I A Concise History of Space

2 Automated Space Control

2.1 From static to dynamic memory management .................... 17

2.1.1 Static allocation ........................................... 17

2.1.2 Stack allocation ............................................ 18

2.1.3 Tail recursion .............................................. 19

2.1.4 Dynamic allocation and garbage collection ................. 20

2.2 From dynamic to static memory management ................... 22

2.2.1 Compile-time evaluation .................................... 22

2.2.2 Compile-time garbage collection ............................ 23

2.2.3 Compile-time optimisations cause havoc .................... 25

2.3 Space control and laziness .................................... 26

2.3.1 Laziness is leakiness ....................................... 26

2.3.2 Laziness is not leakiness ................................... 27

2.4 Summary ..................................................... 29

3 Reasoning About Space Usage

3.1 Operational Semantics .......................................... 31

3.1.1 Natural semantics ........................................... 32

3.1.2 Natural space semantics .................................... 34

3.1.3 Abstract machines .......................................... 36

3.2 Modelling and reasoning ...................................... 38

3.2.1 Space-safe transformation .................................. 38

3.2.2 Space-safe implementation ................................. 42
II Term Graphs and Space Usage

4 GraphKit

4.1 Graph grammars .............................................. 56
4.2 Term graphs ..................................................... 57
  4.2.1 Term construction ......................................... 57
  4.2.2 Term-graph construction ................................. 58
  4.2.3 Garbage collection ....................................... 59
4.3 Graph evaluators .............................................. 60
  4.3.1 Term patterns ............................................. 60
  4.3.2 Graph-rewrite rules ..................................... 62
  4.3.3 Term and graph contexts ............................... 64
  4.3.4 Evaluation relations ................................... 65
4.4 Input and output ............................................... 67
4.5 Summary ....................................................... 69

5 Space Usage in GraphKit ....................................... 71

  5.1 Measuring space and time usage ............................ 71
  5.1.1 Size, space, time and complexity ...................... 71
  5.1.2 Accurate evaluators .................................... 73
  5.2 Deallocation .................................................. 76
  5.2.1 Garbage-generating rules ............................... 76
  5.2.2 Evaluator trimming ...................................... 77
  5.2.3 Stack categories ........................................ 78
  5.3 Determinism .................................................. 79
  5.3.1 Graph left-pattern unification ......................... 80
5.4 Summary ....................................................... 84

6 Models of Implementations .................................... 87

  6.1 A space semantics for Haskell ............................ 87
  6.1.1 Translation and transformation ......................... 87
  6.1.2 Algebraic data types ................................... 89
  6.1.3 Primitive functions and numbers ....................... 91
  6.1.4 Monadic IO ............................................... 92
  6.1.5 The rest of Haskell ..................................... 95
  6.2 Space semantics of lazy evaluators ....................... 96
  6.2.1 Eliminating the stack .................................. 96
  6.2.2 Separate control expression ........................... 96
  6.2.3 Indirection nodes ....................................... 97
  6.2.4 Black holing ............................................ 99
  6.2.5 Projection elimination ................................. 100
  6.2.6 Environment trimming .................................. 100
  6.3 Wider application and limitations ....................... 101
  6.3.1 Modelling other languages ............................. 101
6.3.2 Modelling environments ............................... 102
6.4 Summary .................................................. 103

7 Space Leaks .................................................. 105
7.1 Relative space leaks ....................................... 105
7.1.1 Translation relations ................................... 106
7.1.2 Space-leak definition ................................... 106
7.2 Comparing space-fault criteria ......................... 110
7.2.1 Space leaks and IO ..................................... 110
7.2.2 Space-fault criteria .................................... 112
7.2.3 Node usage versus size usage ....................... 113
7.3 Summary ..................................................... 115

III Relatively No Leaks ............................................. 117

8 Space Relations .................................................. 119
8.1 From translation relations to space relations ......... 119
8.1.1 Bisimulation relations ................................. 119
8.1.2 Exponential space relations ......................... 120
8.1.3 Space relations ......................................... 121
8.2 Garbage collection by space relation .................. 122
8.3 Desired properties of space relations ................. 125
8.3.1 Garbage collection property ......................... 125
8.3.2 Local reasoning properties ......................... 126
8.3.3 Higher-order graph translation ..................... 128
8.4 Proportional space relations ......................... 128
8.4.1 Translating null terms ................................. 132
8.5 Properties of Proportional Space Relations .......... 133
8.6 Summary ..................................................... 136

9 Checking No Leak ................................................. 137
9.1 Simulation proofs ......................................... 137
9.1.1 Lockstep simulations .................................. 138
9.2 Simulation proof squares ................................ 139
9.3 Graph-pattern translation ............................... 141
9.3.1 Graph translation ...................................... 142
9.3.2 Square-top construction ............................. 143
9.3.3 Garbage collection for pattern configurations .. 146
9.4 Configuration simplification ......................... 147
9.4.1 Renaming nodes ....................................... 147
9.4.2 Square-right-side rule ............................... 149
9.4.3 Renaming hole substitutions ....................... 150
9.4.4 Copying and sharing ................................. 151
9.5 Summary ..................................................... 154
10 Advanced No Leak Proofs 155
10.1 Completion of Proportional Space Relations .................. 155
10.1.1 PSR rule inference ........................................ 157
10.1.2 Controlling PSR completion ................................. 159
10.2 Global reasoning ................................................ 160
10.2.1 Enriched space relations ................................... 160
10.2.2 Simulation-supporting lemmas ............................... 163
10.3 Non-lockstep reasoning ......................................... 164
10.3.1 Non-lockstep termination ................................... 164
10.3.2 Non-lockstep and global reasoning ......................... 165
10.3.3 \(k\)-steps to 1-step simulation .............................. 167
10.3.4 Unlimited-steps to 1-step simulation ....................... 169
10.4 Summary ....................................................... 170

IV Looking for Leaks 173

11 Classification of Space Leaks 175
11.1 Leaky translation ............................................... 175
11.2 Space leaks and PSR completion ................................ 176
11.3 Leaky evaluation ................................................ 177
11.4 Leaky evaluation sequences .................................... 180
11.5 Summary ....................................................... 182

12 Checking Leak Witnesses 183
12.1 Self-feeding loop construction ................................. 183
12.1.1 Super-rules ................................................. 184
12.1.2 Self-feeding loops ......................................... 185
12.2 Graph pattern unification ..................................... 188
12.2.1 More specific unifiers ..................................... 188
12.2.2 Holes with substitutions .................................. 189
12.3 Candidate leak witness construction ........................... 192
12.4 Checking candidate leak witnesses ............................ 194
12.4.1 Candidate witness translation ............................. 194
12.4.2 Termination analysis ...................................... 194
12.4.3 Non-termination analysis .................................. 195
12.5 Passive-leak witness checking .................................. 195
12.6 Summary ....................................................... 197

13 Searching for Leak Witnesses 199
13.1 Searching for a witness ........................................ 199
13.2 Looking for differences ........................................ 201
13.3 Next-rule analysis ............................................. 203
13.4 Permutation-loop removal ...................................... 205
13.5 Expanding sequences .......................................... 207
13.6 Approximate candidate sequences ............................ 208
13.7 Summary ....................................................... 210
Conclusions

14 Assessment ................................................................................. 215
  14.1 Summary of contributions ................................................. 215
    14.1.1 Term graphs for space semantics ............................ 215
    14.1.2 Defining and classifying space leaks ................. 216
    14.1.3 Proving no-leak by simulation ............................. 217
    14.1.4 Leak-witness checking and search ....................... 218
  14.2 Further work ................................................................. 219
    14.2.1 Improving GraphKit ............................................. 219
    14.2.2 Better no-leak proofs ........................................ 220
    14.2.3 Finding more leaks ........................................... 222
  14.3 Conclusion ................................................................. 223

A Using GraphKit ............................................................................. 225

B Relatively-No-Leak Proofs .......................................................... 231
  B.1 Fully automated proof checks ..................................... 231
  B.2 Fully automated proof inference ................................. 232
  B.3 Proofs using global reasoning .................................. 232
  B.4 Proofs using non-lockstep reasoning ....................... 233
  B.5 Simulation proof squares ......................................... 235

C Candidate Leak Witnesses .......................................................... 247

Glossary of Notation and Definitions ........................................... 251

Bibliography ................................................................................. 257
List of Figures

3.1 Natural semantics of lazy evaluation .......................... 32
3.2 Natural space semantics of lazy evaluation ..................... 35
3.3 Sestoft’s mark 1 machine for lazy evaluation ................... 36
3.4 Three simple term graphs .................................. 46
3.5 Two term graphs ........................................... 47

4.1 Example GraphKit input ...................................... 54
4.2 Example GraphKit output .................................... 55
4.3 Pictorial evaluation trace ................................... 70

7.1 Lattice of leak criteria ......................................... 116

8.1 Proportional Space Relation rule restrictions ................. 129
8.2 Example PSR rewrite trace .................................. 131

10.1 No-leak hierarchy of lazy evaluators ......................... 171

11.1 Venn diagram of leak classes ................................ 182

13.1 Leak hierarchy of lazy evaluators ........................... 211
List of Tables

10.1 Details of proofs in no-leak hierarchy. ................................. 171
13.1 All sequences witness search to length 3. ............................ 200
13.2 All sequences candidate search to length 5. ........................... 201
13.3 Distinct sequences witness search to length 4. ....................... 202
13.4 Next-rule table of lazy. .................................................. 203
13.5 Next-rule-table sequences witness search to length 5. .............. 204
13.6 Next-rule-table sequences candidate search to length 6. .......... 204
13.7 Permutation-loop-free sequences witness search to length 6. ... 205
13.8 Permutation-loop-free sequences candidate search to length 7. ... 206
13.9 Expanding sequences sequences witness search to length 7. ....... 207
13.10 Expanding sequences candidate search to length 7. ............... 208
13.11 Approximate candidate sequences witness search to length 8. ... 209
13.12 Approximate candidate sequences candidate search to length 8. .... 209
13.13 Details of proofs in leak hierarchy .................................... 211

C.1 Candidate leak witnesses of lazy ................................. 247
Acknowledgements

I am grateful to my supervisor Colin Runciman for his guidance through the maze of computer science; for the freedom to investigate what pleased me how and when suited me and for the many opportunities I have had to see and take part in computer science at the national and international levels.

Thanks to the members of the Programming Languages and Systems group at York for their advice and opinions. Thanks to Oege de Moor for the opportunity to present an embryonic version of this work to the Programming Research Group at Oxford in February 2000. To David Sands and Jörgen Gustavsson for organising and funding a visit to Chalmers in June 2001 and for their helpful discussions on and relating to the work presented here, thank you.

This work was funded by a research studentship from the Engineering and Physical Sciences Research Council.
Declaration

The work presented here is all my own. Some sections are developed from papers written in collaboration with my supervisor Colin Runciman. Parts of chapters 4 and 5 are based on [BR00b]; Section 6.1 is based on [BR01]; chapters 8 and 9 are based on [BR00a] and chapters 12 and 13 are based on [Bak01].
Chapter 1

Introduction

1.1 The space usage problem

“You can’t say ‘all my knowledge of Star Trek and Dr Who, I’ll wipe that and I’ll fill that space with something more interesting’. You can do that on a computer but you can’t do that with your brain.” [Coc01]

Computers are admired for their ability to forget the irrelevant. But they are not always forgetful enough. Our subject is the dynamic space usage of compiled programs. Forget correctness; forget time complexity; the choice of programming language is not important. The space behaviour of a program is determined by the space semantics it is given by a compiler; the combination of evaluation rules and garbage collection strategy that implement the program.

A compiler should turn a program into an executable which preserves the intended space complexity of the algorithms described by the program. But in more advanced (especially declarative and in particular lazy functional) languages, the program by itself does not describe its own space complexity. The compiler takes the tedious and fraught tasks of memory management away from the programmer, automating all the details regarding where data is stored, how it is stored and for how long it is stored. The run-time properties of a program are only fully defined once it is compiled.

A compiler may also optimise a program. Various combinations of transformations may be applied to a program to make it faster or smaller. These transformations can distort the relationship between a program and its run-time space complexity further. They could make the space usage of a particular algorithm much worse than expected. The program would then be accused of having a space leak if it does not run within the space available. But the source of the space leak is very unclear in these circumstances: it could be a fundamental property of the program; it could be a badly chosen algorithm; it could be an unfortunate combination of compiler optimisations; it could be the way the program is executed or it could be a result of the way that memory is managed in the compiled program.

The problem is how to decide which version of a program has better space usage; how to decide which optimisation package gives the better space usage; how to decide which implementation technique gives a program the best space usage; how to decide whether there is a space leak.

This thesis provides theoretical modelling and proof techniques for tackling these problems. We investigate how to model the space semantics of an implementation abstractly such that its space behaviour can be reasoned about mathematically. We
look at the question of what space leaks are, see how they are manifested in a theoretical model and develop general-purpose proof methods for registering their presence or absence.

1.2 Contributions and organisation

Part I: A concise history of space

The first part looks at how space usage is controlled in compiled languages and how it is modelled and reasoned about.

Chapter 2 reviews some of the important advances in automated memory management to show why it has become so difficult to reason about the space usage of programs. Dynamic allocation, garbage collection and lazy evaluation all assist programmers by simplifying the production of a correct program while complicating the analysis of space usage.

Chapter 3 investigates the techniques available for modelling and reasoning about space usage in the presence of dynamic memory management. The basic tool is operational semantics and we look at different methods available for specifying the space semantics of lazy evaluation. Within operational semantics, specific frameworks for modelling space behaviour have been devised. We look at abstract machine frameworks and term and graph rewriting frameworks that provide an accurate theoretical model of space usage. Reasoning work about space currently includes techniques for showing that local program transformations are space-safe (assuming a fixed semantics) and techniques for showing that a global transformation or implementation technique is space-safe (relative to some semantics). These proofs usually assume specific abstract machines or particular abstract machine frameworks. This leaves open the question of how well-suited a general-purpose operational model such as term-graph rewriting really is for defining and reasoning about space usage.

Part II: Term graphs and space usage

The second part is concerned with modelling space usage and defining space leaks.

Chapter 4 gives an operational description of GraphKit, an equational term-graph rewriting system which is designed for modelling the space usage of various implementation techniques. The GraphKit tool takes an input graph grammar and evaluator, which are checked, and an initial graph which is checked and evaluated. GraphKit includes a standard reachability-based garbage collector and a facility for IO operations. Theoretically, evaluation in GraphKit is described using a notation for graph contexts and the theory of disequality to disambiguate graph patterns. This work reformulates and extends the basic version of GraphKit presented in [BR00b].

Chapter 5 discusses aspects of GraphKit relating to space usage. A standard notion of graph size and graph space usage are provided and an accuracy condition is developed to ensure that evaluators are realistic models of both time and space usage. Simple rules for the trimming of graphs and evaluators are described. An analysis is used to provide explicit deallocation where it can show that structures are being used in a stack-like manner.

Chapter 6 uses GraphKit to model various aspects of lazy evaluation. In [BR01] we gave a simple GraphKit model of some of the core features of the non-strict program-
ming language Haskell. That work is extended by modelling data constructors and projections, primitive types, IO and error handling. We also present models of several implementation techniques for lazy evaluation. These are used as the examples for comparing space usage in following chapters.

Chapter 7 gives a formal definition of the idea of a space leak. A leak is defined as a relative property of a GraphKit evaluator — relative to another evaluator and a graph translation relation. This definition is justified by arguing that a leak in our model corresponds to appreciably worse space usage in its implementation for some programs. In order not to leak, an evaluator must consistently match or better the space usage of the evaluator it is being compared with. Other ways of defining a space fault are considered and we see where our definition fits into the range of possible space fault criteria.

Part III: Relatively no leaks

The third part is concerned with how to prove that an evaluator does not leak space.

Chapter 8 lays the foundations for our no-leak proof method. Leakiness is defined relative to a translation between evaluator states. The approach is to arrange for the translation to carry properties which ensure that translating a graph does not increase its space requirements by more than some constant factor. We call such translations space relations. Translations include further properties to simplify the proof of no-leak (such proofs can ignore garbage, for example). A framework for defining space relations with all the properties we need as configuration rewrite systems is given. We also show that space relations have very similar properties to garbage collectors. This forms the basis of a GraphKit extension in which non-standard garbage collectors can be specified in a concise notation based on the space-relation framework.

Chapter 9 describes the no-leak proof method. This is an automated procedure which takes two GraphKit evaluators and a translation relation defined in the framework of Chapter 8. The method is an adaption of the standard simulation method for showing the correctness of small-step semantics. No-leak proofs use an algorithm for graph translation; this is also the mechanism used for implementing non-standard garbage collection by space relation in GraphKit. The proof method is not complete so a number of axioms are introduced to enable the completion of some examples. This proof system is developed from the technique we described in [BR00a].

Chapter 10 extends the no-leak proof method to handle evaluator comparisons that do not quite fit the restrictions of the automated method. We use completion to allow under-specification of the translation so that only initial states are translated by the user of the proof system. The automated method requires an exact correspondence between the steps taken by the evaluators it compares; this is relaxed in various ways. The automated method localises the reasoning to consider only the effect of an evaluator rewrite on the graph. Some comparisons need forms of global reasoning that are not catered for in this framework, so we look at enriching the framework and supporting it with additional global information. Part III concludes with a hierarchy of no-leak comparisons for the lazy evaluation techniques modelled in Chapter 6.
Part IV: Looking for leaks

The fourth part is concerned with how to prove that an evaluator does leak space.

Chapter 11 investigates leakiness in the abstract by classifying leaks according to their cause. We distinguish between inefficiency caused by poor encoding and inefficiency caused by a bad evaluation strategy. We look briefly at whether the no-leak proof process says anything about the presence of a leak in the other direction. We identify those leaks which can be recognised by finding a single witness graph. Finally, we separate leaks which are caused purely by bad evaluation rules from those which are caused by applying bad evaluation rules to certain graphs.

Chapter 12 describes an automated procedure for checking that an evaluator is leakier than another by calculating the space behaviour of a single witness graph. The development includes an algorithm for the unification of higher-order graph patterns and a simple non-termination analysis.

Chapter 13 conducts automated searches for witness graphs to prove that an evaluator is leakier than another. The basic method is a systematic search of all graphs whose evaluation conforms to a certain pattern: the class of graphs identified in Chapter 11 whose evaluation trace repeatedly follows the same sequence of steps. Inevitably, the search space is huge even for our modest models so a number of proof planning techniques are introduced to cut the space without losing any witnesses. The search experiments find leaks among our lazy evaluation models. This work is developed from the ideas presented in [Bak01]. Part IV concludes with a hierarchy of leakiness comparisons for the lazy evaluation techniques modelled in Chapter 6.

Conclusions

Chapter 14 reviews the contributions of this thesis. It identifies areas for improvement, wider application and further research. It finishes by commenting on the power, flexibility and usefulness of the ideas that this thesis presents.

Supporting material

Appendix A summarises the implementation work that supports this dissertation. GraphKit, the basic no-leak proof method and the leak-witness search technique are all available for experimentation and they were used to generate most of the examples that appear in the preceding chapters.

Appendix B presents the details of the no-leak proofs omitted from Chapters 9 and Chapter 10.

Appendix C summarises the results of the looking-for-leaks experiments omitted from Chapter 13.

A glossary lists the symbols and unusual notations that we define, and the graph evaluators, translation relations and garbage collectors used as examples throughout.
Part I

A Concise History of Space
Chapter 2

Automated Space Control

Compiled languages offer automated memory management. This means that decisions about where data is stored, how it is stored and for how long it is stored are not taken in a program; the compiler makes them. Advances in compiler technology can make it difficult for programmers to understand space behaviour.

Section 2.1 looks at the development of automated memory management, including the ideas of garbage and garbage collection, which separate data deallocation from the text of a program and simultaneously make predicting space usage difficult. Section 2.2 looks at some of the work that attempts to tame automated memory management by reducing the need for garbage collection. Section 2.3 looks at lazy evaluation which relies more heavily on garbage collection than other evaluation strategies and for which little work on the prediction or control of space usage exists.

This provides background for our work on understanding space usage and detecting space leaks. The important concepts introduced here are the reliance of certain reduction strategies on garbage collection and the difficulty of reasoning about space in the presence of garbage collection. We assume that the reader is familiar with functional programming languages such as Haskell [Pe99] and the λ-calculus [Bar84].

2.1 From static to dynamic memory management

The evolution of programming languages — from means of specifying sequences of operations with the effect of producing a result to means of specifying a result with the requisite operations deduced by a compiler — changes the problem of predicting program space behaviour from trivial afterthought to intractable afterpain.

2.1.1 Static allocation

In the beginning programming languages used static allocation: the location and size of all pieces of data and control structures needed when a program runs were determined when the program was compiled. This approach to programming language design has many well known flaws — for a comprehensive roasting of the von Neumann style see [Bac78]. Regarding space and data storage, calculating how much space the running program needs is simple. The problem is that some re-use of locations is needed. The reasoning problem is to make sure that nothing is overwritten before its last use.
Example 2.1 (A sub-routine for square root)

\[
.sq rt \text{ var } n : \text{int} \\
\text{ var } r : \text{address} \\
\text{ var } i : \text{int} ::= 0 \\
\text{ var } s : \text{int} ::= 0 \\
\text{ while } s < n \\
\text{ do } i ::= i + 1 \\
\text{ s ::= i } \times i \\
\text{ goto } r
\]

The sub-routine \( \text{sqrt} \), written above in a fictitious assembly language, defines a simple method for computing \( \lceil \sqrt{n} \rceil \). The intention is that an integer is assigned to \( n \) and a return address to \( r \); control passes to \( \text{sqrt} \) which computes the square of every \( i \) starting at 0 until \( i^2 \geq n \); then \( i \) holds the result and control continues from \( r \).

A compiler for this language could easily compute the program space usage statically: three integer variables and an address. Much more difficult is the argument that the last use for \( i \) occurs before the next run of \( \text{sqrt} \).

2.1.2 Stack allocation

Algorithmic languages, in which programs are collections of function or procedure definitions, introduce a control stack to the running program. When a function is entered a new stack frame is pushed to hold its arguments, any temporary results and a return address. The result of a function and any global variables that need to survive after the function returns are written into frames further down the stack.

The memory management is still said to be static because the operations which create and destroy stack frames are part of the compiled program. Stack allocation solves the reasoning problem associated with static allocation: memory re-use is automatic and safe. Storage for variables local to a function is disposed after the function returns and the local variables for the next function called occupy the same memory space. But stack allocation also introduces a new reasoning problem: there is a risk that the running program will run out of memory.

Example 2.2 (A functional square root)

\[
\text{sqrt } n = f 0 \\
\text{ where } f i = \text{if } i \times i \geq n \text{ then } i \text{ else } f (i + 1)
\]

The function \( \text{sqrt} \) to compute \( \lceil \sqrt{n} \rceil \) shown above could be written in many algorithmic or functional languages. The memory management of variables is implicit: the space for the argument \( n \) is part of the stack frame for \( \text{sqrt} \). The stack frame for each call of the auxiliary function \( f \) — corresponding to the while loop in Example 2.1 — includes the space for its argument \( i \), any temporary results like \( s \) in Example 2.1 and a return address.

Reasoning is still straightforward. A bound for the maximum amount of stack space needed can be found by multiplying the maximum stack frame size by the maximum stack depth. This can be formalised by using the techniques of axiomatic semantics to deduce or confirm the space complexity of a program [NN92]. For example, we might prove the assertion \( \{ i \geq 0 \} f i \{ n \downarrow f i \geq \sqrt{n} \} \) which says that assuming that \( i \) is
not negative, \( fi \) can be computed in \( O(n) \) space and its result is not less than \( \sqrt{n} \). To do this calculation requires a model of execution which says that each call of \( f \) uses a constant amount of stack space (we assume bounded integers here). If greater precision was required we might try to prove the assertion \( \{ i \geq 0 \} fi \{ \sqrt{n} \downarrow fi \geq \sqrt{n} \} \): the result is computed in \( O(\sqrt{n}) \) space.

### 2.1.3 Tail recursion

Not removing a stack frame until a function \( f \) returns is wasteful if the last thing done by \( f \) is to return the result of calling another function \( g \). The stack frame for \( f \) sits in the memory, its temporary variables never to be used again, its only purpose to supply the next return address when \( g \) returns to it. The space occupied by such a stack frame cannot be used for anything else and it serves no useful purpose; in a recursive function such as \( f \) in Example 2.2, this effect can happen at each call. It is as if the stack memory space has sprung a leak and more and more of its bytes are pouring out with each function call, never to be seen until the chain of calls finally terminates. In a long computation the entire memory will drain away and the result of the program is an error message: stack overflow.

This problem discourages the use of functional recursive programs in favour of imperative programs built from loop control structures. If programs are written in a functional style a huge proportion of function calls can exhibit this wasteful behaviour [Cli98]. They are known as tail calls and compilers for functional languages such as Scheme or ML include a special optimisation to prevent them from wasting space.

**Definition 2.1 (Tail expressions and calls [Cli98])**

In a simple functional programming language consisting of the \( \lambda \)-calculus extended with if-expressions, data constructors, case expressions and primitive types and operations such as characters, numbers and multiplication (e.g. the core of Scheme), tail expressions and calls are defined as follows:

- The body of a \( \lambda \)-expression is a tail expression.
- The then and else branches of an if-expression are tail expressions if that if-expression is a tail expression.
- An alternative right-hand-side of a case expression is a tail expression if that case expression is a tail expression.
- A tail expression that is a function call is a tail call.

**Example 2.3 (A tail recursive function)**

Analysing the definition of \( f \) in Example 2.2 with the rules in Definition 2.1 we find that the body of \( \lambda i.\text{if } i \times i \geq n \text{ then } i \text{ else } f(i + 1) \) is a tail expression therefore \( i \) and \( f(i + 1) \) are tail expressions. The variable \( i \) is not a tail call but \( f(i + 1) \) is. A function with a recursive tail call is known as tail recursive.

Executing a tail call does not consume stack space. Instead, the new stack frame is written over the current stack frame. After digesting these new rules for the space usage
of a program we can reason that evaluating $\sqrt{n}$ from Example 2.2 uses constant space. Again, this could be formalised with the techniques of axiomatic semantics [NN92] and we could prove the assertion $\{i \geq 0\} f \downarrow f i \geq \sqrt{i}$. So reasoning about a precise bound for space usage becomes more complex. But there is still a clear relationship between a program and its space usage and the tail-call optimisation can only improve on worst-case assumptions about space complexity.

### 2.1.4 Dynamic allocation and garbage collection

Sometimes the result of a function is a data structure whose size is not known, or not easily calculated, when the function is called. For situations like this the run-time system of many programming languages offer a heap. A data structure can be stored in the heap by allocating memory for it at run-time; it is referred to by a pointer variable. The heap will fill eventually; in languages like C the program must call a special function to return memory space to the run-time system when a data structure is not needed any more. From the point of view of reasoning about space usage and program correctness this approach returns to the problems of static allocation for those programs that do not obey the stack discipline.

More advanced programming languages, beginning with LISP [McC60], have run-time systems which manage the heap automatically. Data structures are always allocated in the heap and the run-time system keeps track of all the structures that the program needs. Deallocation is also automatic: the run-time system includes a garbage collector to remove data structures that are needed no longer.

**Example 2.4 (Dynamically managed square root)**

$$\sqrt{n} = f (0..n)$$

$$f x = \begin{cases} y : y s & \text{if } y \times y \geq n \text{ then } y \text{ else } f y s \\ \end{cases}$$

$$m..n = \begin{cases} \emptyset & \text{if } m \geq n \text{ then } \emptyset \text{ else } m : (m + 1..n) \\ \end{cases}$$

The functional program above specifies $\sqrt{n}$ as the application of a function $f$ to the list of numbers from 0 to $n$. The function $f$ repeatedly takes the front element of the list $x s$ and, if $y^2 \geq n$ it returns $y$ as the result as in previous versions of $\sqrt{n}$. Otherwise it moves on to the next element of the list. The function $m..n$ generates the list of numbers from $m$ to $n$ recursively. Evaluating $\sqrt{4}$ with strict semantics is shown below.

$$\sqrt{4}$$

$$= f (0..4)$$

$$= f [0, 1, 2, 3, 4]$$

$$= \begin{cases} 0 \times 0 \geq 4 \text{ then } 0 \text{ else } f [1, 2, 3, 4] \\ 1 \times 1 \geq 4 \text{ then } 1 \text{ else } f [2, 3, 4] \\ 2 \times 2 \geq 4 \text{ then } 2 \text{ else } f [3, 4] \\ 2 \\ \end{cases}$$

Each step reduces an expression. First the list argument is generated, occupying $O(n)$ space, then $f$ is applied to each element of the list until the head element is 2
when it returns a result. The recursive reductions of $f$ are tail calls so the control stack needed to run this reduction sequence occupies constant space. The list plays the part of a loop counter. After its elements are used they disappear. In a functional language the removal of such elements is achieved by garbage collection.

Garbage collection usually means removal of those data structures which are unreachable from the root of the expression being evaluated. In example 2.4 this means that the first element of the list could be removed after the fourth step shown and the second element after the fifth step.

Garbage collection is good for reasoning about program correctness: data structures are kept for as long as they are needed and there is no risk of accidentally overwriting a structure before its last use. For reasoning about space there are two main issues to note about garbage and garbage collection.

1. Garbage structures are those pieces of data that are not going to be used in any future computation step. Not just those that are unreachable. In Example 2.4 elements 3 and 4 of the list are garbage from the point of their creation but they are not garbage collected until the last step terminates. This discrepancy between garbage and its collection is inevitable (see Proposition 2.1) and it means that all practical implementations sometimes retain structures unnecessarily. In Example 2.4 it raises the space complexity from $O(\sqrt{n})$ to $O(n)$.

2. Garbage collection is separate from program evaluation. Predicting space usage is very difficult because allocation and deallocation of data structures are not tied to the beginning and end of the execution of some function in the way that stack frame allocation and deallocation are so it is difficult to analyse the space usage of a program axiomatically. Even the approximate reachability-based collectors are not always very predictable. In advocating the use of functional languages, Backus complains that “to understand a conventional program [meaning one written in an algorithmic language and probably with static allocation] one must mentally execute it” [Bac78]. Well, to understand the space usage of a program with dynamic memory management we have to mentally execute it, and an execution model which includes automated allocation and garbage collection can be quite difficult to visualise.

**Proposition 2.1 (Garbage is undecidable)**

Determining if a heap binding is garbage in an arbitrary $\lambda$-calculus program is undecidable [MFH95].

**Proof**

Morrisett et al prove this by reduction of the halting problem. An arbitrary $\lambda$-term is to be evaluated to weak head normal form then applied to some argument term stored in the heap. If the term halts the argument is needed, if it does not then the argument heap binding is garbage. □

As for how to implement garbage collection which removes unreachable structures, there are many approaches, see [JL96]. For functional languages two important techniques are copying collection and generational collection. For copying collection the heap is partitioned into a from-space and a to-space. Allocation occurs in the from-space. The collector traces through the reachable parts of the from-space, copying
the data structures encountered into the to-space. Then the spaces exchange roles. If most of the from-space is garbage then copying reachable structures is much faster than removing unreachable structures. By increasing the heap size the average time taken to deallocate a structure by copying collection becomes negligible [App87].

Generational collectors [LH83] divide the heap into several generations. Allocation happens in the youngest generation. When a generation is full, copying garbage collection promotes its reachable structures to the previous generation. Thus longer lived data are not considered by the collector so often while short-lived data are collected more aggressively. This approach seems to fit the way functional programs operate where there are typically many short-lived structures.

2.2 From dynamic to static memory management

Dynamic memory management can be slow and difficult to reason about. Much research has gone into reducing the reliance of programming languages on this irrational part of their run-time systems without taking freedom of expression from programmers. Very broadly there are two approaches. Section 2.2.1 looks at how a compiler can re-formulate programs so that their need for dynamic allocation and garbage collection is reduced. Section 2.2.2 looks at techniques for turning dynamic memory management into static memory management. Our main interest in these tricks is how they affect our ability to reason about space usage and Section 2.2.3 looks at some work relating to this point.

2.2.1 Compile-time evaluation

It is often possible to simplify or re-formulate a program to reduce the amount of dynamic memory management it does. The simplest examples of this are the transformations compilers use to remove redundant code. These range from global optimisations which prune libraries of those functions not used by a program to the local version (2.1) which can be applied repeatedly throughout the text of a program. This removes the local definition of $x$ whenever $x$ cannot be needed because it is not mentioned in the expression $e$ [App92].

\[
\text{let } x = e' \text{ in } e \implies e, \text{ if } x \notin \text{fv } e
\]

(2.1)

In a strict functional language implementation (e.g. [App92]) this can save a lot of space because evaluating $e'$ could involve much allocation and garbage collection work. More cunning transformations are also used to turn programs into more efficient versions of themselves. For example, Appel [App92] uses a dead argument elimination transformation which can remove the first argument from the definition of $f$:

\[
\begin{align*}
  f \ x \ y &= \text{if } y > 0 \text{ then } f \ x \ (y - 1) \text{ else } 0 \\
  &\implies f \ y = \text{if } y > 0 \text{ then } f \ (y - 1) \text{ else } 0
\end{align*}
\]

(2.2)

Like (2.1) this can eliminate computation work, including stack and heap manipulation, from the running program. This technique of optimising a source program by repeatedly applying simple performance-enhancing transformations is a popular approach to compilation. It also provides a different way of thinking about space usage. Instead of trying to calculate the exact space usage of a program we could use
an approximate upper bound and then let the optimising compiler apply space usage improving (or non-worsening) transformations; these are known as safe-for-space transformations [App92]. A more aggressive transformation known as deforestation [Wad90] eliminates transient intermediate data structures.

**Example 2.5 (Deforested square root)**

\[
T \left[ \text{sqrt } n \right] = f \ n \ 0 \\
\text{where} \\
f \ n \ m = \begin{cases} 
\text{error} & \text{if } m \geq n \\
\text{else if } m \times m \geq n \text{ then } m \text{ else } f \ n \ (m + 1) 
\end{cases}
\]

An application of the deforestation transformation \( T \) to the program from Example 2.4 is shown above. The result is essentially the version of \( \text{sqrt} \) from Example 2.2. The listless deforested program does not need to generate the intermediate list and its function \( f \) is tail recursive, so the program runs in constant space. \( \square \)

These optimising transformations amount to doing bits of computation work once when the program is compiled rather than every time it is run. Often this saves space where some of a data structure is redundant or where the simplified program does not need a data structure. Such transformations are simple instances of partial evaluation [JGS93], a field of program analysis devoted to finding techniques for compile-time evaluation. The examples considered here seem not to worsen space complexity, but they still do not answer the questions of how to calculate or compare space complexity as they do not eliminate garbage collection. In general it is not clear when evaluation at compile-time improves space complexity (Section 3.2.1 looks at work relating to this point).

### 2.2.2 Compile-time garbage collection

Garbage collection makes reasoning about space difficult because it separates the point of deallocation from the program. With enough careful calculation it is often possible to work out where in a program the ideal deallocation point for a data structure should be. Automating such calculations in a compiler is known as compile-time garbage collection. This calculation can also be seen as a way of explicating the space usage of a program because it restores the link between program text and memory management operations so reasoning about space usage becomes a calculation based on the text of the compile-time collected program.

There are many variations on this idea. One of the earliest versions was just used as a way to reduce the amount of work done by a reference-counting garbage collector [Bar77]. In this system, as well as making calls to the run-time system to allocate data, the compiled program has to make calls when it increases or decreases the number of references to data. The optimisation described in [Bar77] removes any such calls that cancel each other out.

An explicit deallocation optimisation works with a heap allocator that maintains a free list of unused locations. It annotates a program text with instructions which return data structure heap cells to the free list after their last use. This eliminates much of the garbage collection work. However, not all deallocation can be eliminated, so the space usage of the optimised program may not always be easier to reason about.
Example 2.6 (Compile-time collected square root)

One of the earliest compile-time deallocation analyses [ISY88] calculates an expression to describe which parts of a list data structure escape in the result of a function. The parts that do not escape are deallocated explicitly when the function returns. For the \texttt{sqrt} program of Example 2.4 this results in the following program.

\[ \text{\texttt{sqrt}} \ n = \text{\textit{RECLAIM1}} \* (f \ (\text{\textit{HOLD}} \ (0..n))) \]

When evaluated strictly the functions \textit{RECLAIM1}*, \textit{f} and \textit{HOLD} are pushed then the list \((0..n)\) is generated as normal. The \textit{HOLD} function stores a pointer to the list on a special stack. Then the result of applying \textit{f} to the list is calculated as in Example 2.4. Then the \textit{RECLAIM} function pops the held pointer to the list and returns to the free list those cells described by the expression — the whole list for \(1^*\).

This optimisation does not improve the space complexity of \texttt{sqrt} but it does improve predictability: enclosing \textit{f} in \textit{RECLAIM1}* says that space for the list generated by \((0..n)\) is needed for the duration of the computation of \textit{f}. To minimise space usage the compile-time analysis needs to nest the \textit{RECLAIM} functions as tightly as possible.

\[ \square \]

So we can think of compile-time garbage analysis as a way of making dynamically allocated structures fit the stack principle by tying allocation and deallocation of heap space to points in the execution of a program; the heap structures acquire a program-determined order of deallocation; the heap becomes a kind of stack of data structures. This also fits in with generational garbage collection which maintains a stack of heap generations.

Therefore another way to garbage collect at compile-time is to pre-empt a generational collector: if the program knows whether a data structure will be long lived or short lived then it can tell the run-time heap allocator which heap generation to allocate in and therefore save the generational collector some work in copying long-lived structures back into older generations.

A simple way to work out which generation a data structure belongs to is to run the program once, recording all the allocation sites and how long lived their offspring are. This profile-directed analysis has been applied to C programs [BZ93].

The alternative is to analyse a program to work out how the data structures allocated by a program should be organised into a stack of heap generations. An early version of this [RM88] uses a static abstract interpretation analysis to calculate which data structures are not used outside of a function. As well as allocating stack frames, function calls allocate heap regions into which non-escaping data structures are allocated. The whole region is deallocated on function return.

A type-based approach to this stack of heap-regions method was formulated by Tofte et al [TT94] (or [Tof98] for an overview) who have developed it into a practical compiler for ML programs [TBE+97]. A heap region \(\rho\) is associated with the evaluation of an expression \(e\) by writing \texttt{letregion} \(\rho\ \text{in} \ e \ \text{end}\) and \(e\) can allocate its result in \(\rho\) by writing \(e\ \text{at} \ \rho\). The region type system is a global analysis which nests \texttt{letregion} annotations as tightly as it can prove safe.
Example 2.7 (Region annotated square root)

letregion $\rho_t, \rho_p, \rho_r$

in letrec $\sqrt{[\rho_n, \rho_i]} \at \rho_r \ (n : \text{int}, \rho_n) =$

letrec $f[\rho_m] \at \rho_f \ (m : \text{int}, \rho_m) =$

if letregion $\rho_o, \rho_s$

in $(m \times m) \at \rho_s) \geq n \at \rho_t$

end

then $m$

else letregion $\rho_c$

in $f[\rho_m] \at \rho_c$

letregion $\rho_o$

in $(m + (1 \at \rho_o)) \at \rho_m$

end

end

in letregion $\rho_z$

in $f[\rho_i] \at \rho_z \ (0 \at \rho_k)$

end

end

end

This region-annotated program computes $\sqrt{4}$. The definitions of $\sqrt{\cdot}$ and $f$ are extended with region parameters in square brackets to say where their argument and result are stored. All memory management is explicit and handled in a stack-like manner (no garbage collection). The argument 4 is stored in the region $\rho_t$ which is referred to as $\rho_n$ within $\sqrt{\cdot}$.

Recursive calls of $f$ work by: storing a closure for $f$ in a new region $\rho_c$; computing the argument by writing a number $m + 1$ into $\rho_m$, region $\rho_o$ is a temporary variable to hold the 1 for this calculation; then retrieving the closure for $f$ and evaluating its definition with the new argument. When $f$ returns its result is the number $m$, the value most recently stored in the region $\rho_m$.

Regions expand as new values are written into them, then the whole region is deallocated when the program reaches the end of the letregion block which declares it. It follows that the space needed for region $\rho_q$ (referred to as $\rho_m$ within $f$) is $O(\sqrt{n})$: the arguments for successive recursive calls pile up in $\rho_q$; other regions require bounded space.

So regions ease space analysis by providing a link between a program and its points of allocation and deallocation and they guarantee safe deallocation. Unfortunately, in this simple example the basic analysis turns an $O(1)$ space program into an $O(\sqrt{n})$ space program.

\[ \square \]

2.2.3 Compile-time optimisations cause havoc

Changing dynamic memory management into static memory management is usually done with the intention of reducing the running time of programs. The problem is this seems not to happen.
"The most effortless form of memory management is to do none at run-
time. A considerable amount of research has gone into compile-time tech-
niques to discover when objects can be discarded or reused. Most of this
work has been theoretic and, as yet, we believe that there has been little
evidence of substantial performance gains." [JL96]

Researchers who have done practical studies of the differences in performance be-
tween garbage collection and explicit deallocation analyses also come to this conclusion
[App87, JT93]. Essentially, heap allocation can be as fast as stack allocation, and if
the heap size is large enough and most of its contents are garbage then the time needed
to deallocate by using a copying collector can be less than the time needed to do stack
deallocation! These studies suggest that if the heap is at least 8 or 15 times the size of
its typical reachable content then garbage collection is faster than stack deallocation.

Even worse, turning heap allocation into stack allocation can change the space
complexity of a program, often for the worse [Cha88]. We saw this in Example 2.7
where a simple region analysis fails to be properly tail recursive. The Region Kit
[TT94] actually includes an optimisation to avoid this situation. A special annotation
is added to the \( \rho_m \) parameter of \( f \) to indicate that region \( \rho_m \) should be reset before
the next argument is written into it, restoring proper tail recursion and \( O(1) \) space
usage.

Aiken et al [AFL95] go further, presenting an optimisation to the region system
which allows regions to be allocated late and deallocated early, relaxing the stack
discipline enough to prevent many programs suffering an increase in space complex-
ity when they are compiled with regions. In [TBE+97] some methods are presented
for eliminating space leaks in region-compiled programs by advising programmers of
situations where the introduction of global variables should be delayed (by rewriting
the program). These techniques complicate reasoning about the space usage of
region-based programs in the same way that a tail-recursion optimisation complicates
reasoning about stack-based programs.

The key point is that region-compiling a program can make its space complexity
different (maybe better or worse) from its space complexity with garbage collection.
So regions offer improved predictability at the expense of slower and potentially more
space-hungry evaluation.

2.3 Space control and laziness

Most of the discussion so far concerns call-by-value languages. Lazy functional lan-
guages are perhaps the most heavily reliant on dynamic memory management. They
have been shown amenable to the generational garbage collection technique but still
spend around 10-20% of execution time in garbage collection [SP93], compared with 5-
10% for strict languages [App92]. They are also less amenable to compile-time garbage
collection; but does reduced predictability mean better or worse space usage?

2.3.1 Laziness is leakiness

Lazy languages have a bad reputation for causing programs to have space faults. This
makes sense if we imagine the lazy evaluation of a program: function arguments are
repeatedly put aside unevaluated until further procrastination is not an option; once
a data structure has been computed it is then stored in case a further use is necessary. All this delaying and storing costs a lot of space. An online dictionary of computing defines a space leak as follows.

“A data structure which grows bigger, or lives longer, than might be expected. Such unexpected memory use can cause a program to require more garbage collections or to run out of heap. Space leaks in functional programs usually result from excessive laziness. For example, the Haskell function \( \text{sum }[] = 0; \text{sum } (x:xs) = x + \text{sum } xs \) when applied to a list will build a chain of closures for the additions and only when it reaches the end of the list will it perform the additions and free the storage.” [How93]

Unwanted laziness is a problem acknowledged by designers of languages like Haskell which allow programmers to specify strict evaluation if they are aware of such a problem. A more ambitious scheme is to use strictness analysis in a compiler to discover parts of the program that can be executed strictly without changing the result [Wad87b]. Now, if strictness is enabled then so are all of the compile-time garbage collection techniques discussed in the previous section [Arg90]. So turning a lazy program into a strict one can be seen as a way to make its space usage that bit more predictable. In practice, strictness analysis will result in a hybrid of lazy and strict evaluation which makes reasoning about space usage even trickier. Another problem is that a strictness analyser might remove wanted laziness that was deliberately deployed to keep space usage low.

Growing larger or living longer than expected means that some structure thought to be transient turns out to be persistent. The key to understanding space usage with lazy evaluation is to distinguish the two [Oka98]. Unfortunately the optimisations performed by many compilers such as full laziness [Pey87] can change the sharing in a program and thus turn a transient structure into a persistent one and hence increase space complexity. Of course there is a time-space tradeoff here, as in any language. A classic example is the use of sharing in an implementation of a powerset program which makes a constant-factor saving in time in exchange for an exponential worsening in space usage [Mei85].

Where a structure is shared but large parts of it are not going to be used, it is the gap between garbage and garbage collection into which space leaks. An important line of research in garbage collection for both strict and lazy evaluation is the development of collectors that can recognise more garbage. For example, garbage collectors can prevent such space leaks by shortcutting projections to data structures [Wad87a].

2.3.2 Laziness is not leakiness

Lazy languages are admired for their demand-driven execution model which can often result in very low space usage. This makes sense if we imagine the lazy evaluation of a program: instead of producing a whole data structure and then processing each element, the processing begins and elements of the data structure are produced when they are needed. At any point the elements already produced but not needed again are garbage and the elements not yet processed have not even been created. Researchers often note this effect in their lists of reasons for laziness:
"Using lazy evaluation allows some programs to run in constant space, which with a strict evaluation strategy would require space proportional to the input." [Joh87]

Lazy evaluation can make programs self-optimising: the transformations for redundant-code elimination discussed in Section 2.2.1 do not prevent significant amounts of redundant computation with lazy evaluation (though they may reduce space usage by enabling earlier garbage collection of structures referred to by the redundant code). Deforestation does not reduce space complexity with lazy evaluation owing to the way that transient data structures are processed.

**Example 2.8 (Lazy square root)**

\[
\sqrt{4} = f \ (0..4) \\
= \text{if } 0 \times 0 \geq 4 \text{ then } 0 \text{ else } f \ (1..4) \\
= \text{if } 1 \times 1 \geq 4 \text{ then } 1 \text{ else } f \ (2..4) \\
= \text{if } 2 \times 2 \geq 4 \text{ then } 2 \text{ else } f \ (3..4) \\
= 2
\]

Returning to the version of \( \sqrt{ } \) defined in Example 2.4, the lazy evaluation of \( \sqrt{4} \) is depicted above. Elements of the list \( (0..4) \) are generated as successive calls of \( f \) demand them; at the next call previous elements are garbage. So the intermediate list is a transient structure: it could be eliminated by deforestation but lazy evaluation is doing the deforestation at run-time. \( \Box \)

So laziness sometimes offers better space complexity than strictness, but what about predictability and control? What about compile-time garbage collection for lazy evaluation? There is no known equivalent of region analysis for laziness, which suggests that lazy evaluation is fundamentally less predictable — the kind of analysis that discovers the point at which a data structure becomes garbage is of little use if that data structure has not even been generated.

Lazy evaluation combines call-by-name semantics with sharing. Where sharing can be done away with, predictability is often improved greatly. Banerjee and Schmidt note that call-by-name evaluation can be implemented with a stack environment [BS94]. Deforestation eliminates the heap management work for a transient data structure. Avoiding the re-computation of a structure by storing it for future uses takes some work. This work can be avoided when it can be shown that a structure is only used once. Compile-time garbage marking [Ham95], or update avoidance [Gus98] optimisations detect such structures. Similarly, an explicit deallocation analysis can be used for data structures that are not shared [Ham95].

**Example 2.9 (Lazy explicitly deallocated square root)**

\[
f \ x s = \text{case } x s \text{ of} \\
( y : d \ y s ) \rightarrow \text{if } y \times y \geq n \text{ then } y \text{ else } f \ y s
\]

Applying an explicit deallocation analysis [Ham95] to the version of \( \sqrt{ } \) defined in Example 2.4 annotates the definition of \( f \) as shown above. The \( d \) attached to the cons pattern indicates that as soon as the case-expression has identified that \( x s \) begins \( (y:ys) \) it can deallocate that cons cell. This analysis works because each cons cell of the list \( (0..n) \) is used once. \( \Box \)
Programmers using lazy languages are disadvantaged because lazy evaluation is more difficult to predict and control than strict evaluation. They are privileged in that lazy evaluation offers simple ways to reduce space complexity either unwittingly through its self-optimising behaviour or through pre-mediated use of transient structures.

2.4 Summary

To have a convenient programming style and a language which guarantees data are not changed or overwritten before their last use we need automated memory management. The simplest way to arrange for safe re-use of data structures is to allocate dynamically in a heap and run a garbage collector to remove structures which have passed their last point of use. It is not always possible to tell exactly when a structure can be collected so all practical implementations use more space than necessary for some programs.

When more space is used than expected a program is said to have a space leak. There is the possibility of a space leak if an implementation does not stick to rules of the language. Failing to be properly tail recursive can cause a space leak. Using a compile-time garbage collection optimisation which extends the lifetime of some structure can leak space. An insufficiently powerful garbage collector can leak space. Choosing (or, a compiler deciding to use) lazy evaluation instead of strict, or vice-versa, could cause a space leak. Choosing to share a structure could spring a leak. Choosing one version of a function over another could open up a leak.

The problem is how to express what the expected space usage should be and then how to reason about programs and implementations. For stack-based languages this means understanding which are the tail calls. For region-compiled programs it means understanding which regions require constant space and which can grow. For lazy evaluation with garbage collection it means understanding what is shared after the compiler has optimised, which structures will be transient at run-time, when the lazy evaluation order will delay the creation of some structures and keep others and how much garbage the collector will be able to spot. For computation in general we need a model which is accurate enough to describe space usage and a way to determine what constitutes a space leak.
Chapter 3

Reasoning About Space Usage

We want to use programming languages with dynamic memory management and garbage collection, but we also want to reason about space usage — to decide whether programs or implementations have space leaks. This chapter reviews existing techniques that are used for modelling and analysing space usage in languages with garbage collection.

The first thing we need is a way to model a programming language which defines how much space programs use as they run, an operational semantics. These come in a variety of styles which permit different kinds of proofs. Section 3.1 looks at big-step semantics and small-step abstract machines, concentrating on how lazy evaluation can be modelled and how an accurate model of space usage can be provided.

Reasoning about space usage can take many forms. The problem of showing that an implementation preserves the space usage described by an operational model must be addressed for: ensuring an evaluation strategy is sufficiently space efficient; ensuring a garbage collector is powerful enough and checking that optimising transformations do not adversely affect space usage. Showing that transforming a program does not damage its space complexity includes transformation to a different algorithm, transformation to a different language or transformation to a generalised or specialised version of the same program.

Section 3.2 looks at a number of formal frameworks which purport to provide a suitable basis for such reasoning. It examines some of the reasoning techniques that have been realised and the ways that ideas like space improvement and space leaks have been formalised.

3.1 Operational Semantics

An operational semantics models how a program is executed. It is an abstract description of an implementation technique, as opposed to a denotational semantics which maps programs to their meanings without concern for resources. Operational semantics usually need to include a structure which models the memory in an implementation and therefore they can be used as a basis for reasoning about space.

Broadly there are two kinds of operational semantics. Section 3.1.1 looks at big-step natural semantics, a high-level approach which is close to a denotational specification. Section 3.1.2 shows how it can be adapted to describe space usage. Section 3.1.3 looks at abstract machines, a simple form of structural semantics, which can be thought of as an abstract implementation.
\[ \Gamma : \lambda x.e \downarrow \Gamma : \lambda x.e \]  

(Lambda)

\[ \Gamma, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n : e \downarrow \Delta : z \]

(Let)

\[ \Gamma : e \downarrow \Delta : z \]

(Variable)

\[ \Gamma : e \downarrow \Delta : \lambda y.e' \quad \Delta : e'[x/y] \downarrow \Theta : z \]

(Application)

\[ \{ i \mapsto \lambda y.y \} : \lambda x.x \downarrow \{ i \mapsto \lambda y.y \} : \lambda x.x \]  

(Lambda)

\[ \vdots \]

\[ \{ \} : \lambda y.y \downarrow \{ \} : \lambda y.y \]  

(Lambda)

\[ \{ i \mapsto \lambda y.y \} : \{ i \mapsto \lambda y.y \} : \lambda y.y \]

(Variable)

\[ \{ i \mapsto \lambda y.y \} : \{ i \mapsto \lambda y.y \} : \lambda z.z \]

(Application)

\[ \{ \} : \text{let } i = \lambda y.y \text{ in } (\lambda x.x) i \downarrow \{ i \mapsto \lambda y.y \} : \lambda z.z \]  

(Let)

Figure 3.1: Natural semantics of lazy evaluation [Lau93] with example derivation.

### 3.1.1 Natural semantics

A natural semantics [Kah87, NN92] specifies the relationship between the initial and final states of an execution. That is, we begin with a configuration containing a program and some initialised structures and end with a configuration containing a collection of structures that represent the result of the program. The semantics is a collection of natural-deduction-style inference rules and axioms which describe how this relationship between initial and final states is derived. Thus the execution of the program is modelled by using the semantics to construct a proof tree. If no tree can be constructed then the program is divergent — its value is \( \bot \) in denotational terms.

A natural semantics for a call-by-value language Mini-ML is given in [Kah87], and one for an imperative language While in [NN92]. We are interested in reasoning about lazy functional languages, so we consider the seminal paper which defines a natural semantics for lazy evaluation [Lau93].

**Example 3.1 (Natural semantics of lazy evaluation)**

To specify lazy evaluation, Launchbury defines a natural semantics whose configurations are a set which models the runtime heap then a colon separator then a control expression. A heap \( \Gamma, \Delta \) or \( \Theta \) maps variables to expressions. Expressions are written in the following subset of the \( \lambda \)-calculus extended with let for sharing and (mutual) recursion.

\[ x, y \in \text{Var} \]

\[ e \in \text{Exp} := \lambda x.e \mid e \mid x \mid \text{let } x_1 = e_1; \cdots; x_n = e_n \text{ in } e \]  

(3.1)

All pure functional programs can be transformed to this tiny language. The only unusual restriction is that function arguments must be variables. A normalisation
transformation enforces the restriction, so $(\lambda x.x)(\lambda y.y)$ becomes let $i = \lambda y.y$ in $(\lambda x.x) i$ for example. This restriction (and the choice not to restrict the rest of the grammar in any way) simplifies the presentation of the semantics. A semantics without it such as [SI96] has slightly more complex rules and and needs more steps to evaluate some programs.

Launchbury's natural semantics is a collection of four rules, one for each language construct, shown in Figure 3.1 with an example derivation. Each rule has a conclusion and zero, one or two premises. A conclusion or premise is a sequent comprising two configurations joined by the evaluates-to symbol $\downarrow$. The value of a configuration is found by using the rules to construct a derivation tree.

- (Lambda) says that a $\lambda$-expression with some heap $\Gamma$ evaluates to itself.

- (Let) says let-expressions are evaluated by adding the $n$ let-bound expressions to the heap (in one step) and then evaluating the body expression.

- (Variable) finds the value of $x$ by looking it up in the heap, removing its binding, evaluating $e$ to its value $z$, re-introducing $x$ to the heap, now bound to $z$ so that further uses of $x$ do not have to re-compute $e$, and presenting a bound-variable-renaming of $z$ (written $\hat{z}$) as the result.

- (Application) evaluates $e$ to a function value then substitutes the argument $x$ for the formal parameter $y$. Sharing is preserved in the reduction step because the restricted syntax guarantees $x$ to be a variable and the semantics preserve the restriction.

There are a few quirks in this specification — issues such as whether function arguments should be restricted to variables and when it is best to do the variable renaming (which ensures that variables added to the heap by (Let) do not clash with existing bindings).

An important issue for modelling space behaviour is hidden in (Variable). Removing $x \mapsto e$ from the heap while computing $e$ corresponds to an implementation technique known as black holing [Jon92]. This has two benefits: if $x$ is directly self-dependent (e.g. the heap binding is $x \mapsto x$) the infinite loop is detected because $x$ becomes undefined in the configuration after it is requested for the first time. Secondly, it can improve space behaviour by preventing duplication and hence potentially reduce the lifetimes of structures in the heap.

So this semantics has the potential to be used as a model of the space behaviour of an implementation. The only objection to its model of black holing is that $x$ cannot really be removed from the heap in an implementation because — assuming that references to heap variables are pointers to make heap lookup a constant time operation — the updated value $z$ must be written into the same place that $e$ was originally. In this case, calculating the heap size is more complicated than counting its bindings so it is not clear how the semantics models space. Alternatively, if heap lookup is implemented through a data structure associating variables with addresses then the space model becomes more complicated and heap lookup, removal and insertion might not be constant-time operations. Then the semantics would be a misleading model of time usage.
Proofs about natural semantics work by induction on the depth of the derivation tree [NN92]: a property is proved for each axiom in the semantics and for the conclusion of each inference rule, assuming it holds for all the premises. Launchbury [Lau93] shows that the lazy semantics is correct with respect to a denotational semantics [Abr90], and suggests how it could be extended for modelling space and time. Mountjoy [Mou98] uses the semantics to derive others which model the STG-machine for lazy evaluation by extending application and abstraction to have many arguments; this work also supports the claim that the semantics is realistic. A similar natural semantics [SI96] is used to demonstrate the correctness and type safety of lazy evaluation.

3.1.2 Natural space semantics

To model the space behaviour of lazy evaluation we need to model garbage collection somewhere. Launchbury shows how this can be done by adding a garbage collection rule which can be applied to any configuration during the generation of a proof tree. For example, in the proof tree in Figure 3.1, the binding for $i$ is not needed anywhere down the right side, so the garbage collector could remove it.

Then we need to add formulas to the rules to calculate the space usage of the computation represented by the construction of a proof tree. These formulas should assume *maximal garbage collection* as collectible nodes cannot contribute to any space leak. Basically: decide which parts of a configuration require bounded resources and ignore them; decide which parts can vary in size, affecting the space complexity of the computation, and count them. Then add formulas to the semantic rules to calculate the maximum configuration size needed during a proof.

Then do we have a *space semantics*? Not quite, the natural semantics framework has other surprises for us. The (Variable) rule carries the name of $x$ from the left configuration of its conclusion to the right but without mentioning it anywhere in the premise. So $x$ must be stored somewhere while the premise proof tree is computed, costing some space. Similarly, the (Application) rule carries the variable $x$ from the conclusion left configuration to the second premise. This too must be stored somewhere while the first premise is computed (we assume that premises evaluate left-to-right). These considerations correspond to a run-time stack. With garbage collection, a measure of size and an analysis of the hidden space costs we can define a space semantics.

Example 3.2 (Space semantics for lazy evaluation)

Figure 3.2 shows our space semantics. The configurations are the same heap and expression pairs as the original. Following the style of a natural space semantics for call-by-value [Min00], we extend the sequents to judgements of the form:

$$R \vdash \Gamma : e \Downarrow_s \Delta : z$$

This means that the configuration $\Gamma : e$ evaluates to the configuration $\Delta : z$ in space $s$ where $R$ is a set of root variables. We define the size of a configuration as the number of bindings in the configuration heap which are reachable from the configuration expression or $R$.

$$size (R, \Gamma : e) = \# \Gamma_{reach (\Gamma, R \cup \text{fv } e)}$$

where $reach (\Gamma, X) = \text{fixFrom} (\lambda V. V \cup \text{fv } (\Gamma|_V)) X$  \hspace{1cm} (3.2)
\[
R \vdash \Gamma : \lambda x.e \Downarrow_{\text{size}} \{r, \Gamma : \lambda x.e\} \quad \Gamma : \lambda x.e \ (\text{Lambda'})
\]

\[
R \vdash \Gamma, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n : e \Downarrow_s \Delta : z
\]

\[
R \vdash \Gamma : \text{let } x_1 = e_1; \ldots; x_n = e_n \in e \Downarrow_{\max \{s, \text{size } \{r, \Gamma : \text{let } x_1 = e_1; \ldots; x_n = e_n \in e\}\}} \Delta : z \quad (\text{Let'})
\]

\[
R \vdash \Gamma : e \Downarrow_s \Delta : z
\]

\[
R \vdash \Gamma : e : x \Downarrow_{s+1} \Delta, x \mapsto z : z \quad (\text{Variable'})
\]

\[
\frac{\Gamma \vdash x \Downarrow_s \Delta : \lambda y.e' \quad R \vdash \Delta : e'[x/y] \Downarrow_{s'} \Theta : z}{\Gamma \vdash e \ x \Downarrow_{\max \{s+1, s'\}} \Theta : z} \quad (\text{Application'})
\]

Figure 3.2: A natural semantics for lazy evaluation with space costs.

\[
\text{fixFrom } f \ x = \begin{cases} 
  \ x, & \text{if } x = f(x) \\
  \text{fixFrom } f \ f(x), & \text{otherwise}
\end{cases} \quad (3.3)
\]

This size measure encodes the garbage collector in its reach function. We define fixFrom as the least fixpoint of a function f starting from x. So reach finds the smallest subset of \( \Gamma \) whose free variables (fv) are disjoint from its domain and whose domain includes X. Variables in a heap domain are bound. The fixpoint exists because (\( \lambda V, V \cup f v (\Gamma |v) \)) is monotonic. The calculation of s in the semantic rules adds the maximum configuration size to any other space needed to represent the run-time stack. The size of R, the control expression and the run-time machinery needed to evaluate and garbage collect are ignored. It is easy to use these rules to verify the following judgement.

\[
\{\} \vdash \{\} : \text{let } i = \lambda y.y \in (\lambda x.x) \ i \Downarrow_2 \{i \mapsto \lambda y.y\} : \lambda z. z
\]

So the example in Figure 3.1 runs in 2 units of space: one is for the binding for \( i \) and the other stores the argument \( i \) during (Application). We cannot read too much into the figure — a program using 3 units might use slightly more or less space in practice — but we can conclude that the example produces the correct value within a small number of steps and within a small space bound.

\[
\text{Deriving the space semantics from the natural semantics is in principle a simple logical process, but care must be taken to make sure that the hidden costs are counted properly and that the units represent bounded blocks of space. Another way to account for the hidden costs would be to add a model of a run-time stack to the configurations (a semantics in [Jon01] does this). This might simplify the space-semantics calculations at the expense of a lower-level presentation.}
\]

The space semantics depends on the initial program size — we can have less heap bindings if their definitions are larger — but this is acceptable because we are only claiming that the semantics is accurate enough to describe space complexity.

There are a few papers which reason about space using a natural space semantics. Belloch and Greiner [BG96] give a time-and-space semantics for NESL. Minamide [Min00] uncovers a mistake in this specification when he compares it with a continuation-passing semantics. The mistake regards counting constant rather than variable space for elements of the run-time stack which are hidden by the natural
\[ (\Gamma, e, p, S) \implies (\Gamma, e, p : S) \quad \text{(app)} \]
\[ (\Gamma, \lambda y.e, p : S) \implies (\Gamma, e[p/y], S) \quad \text{(app)} \]
\[ (\Gamma[p \mapsto e], p, S) \implies (\Gamma, e, \#p : S) \quad \text{(var)} \]
\[ (\Gamma, \lambda y.e, \#p : S) \implies (\Gamma[p \mapsto \lambda y.e], \lambda y.e, S) \quad \text{(var)} \]
\[ (\Gamma, \text{let} \{ x_i = e_i \}_{i=1}^n \text{ in } e, S) \implies (\Gamma[p_i \mapsto e_i[p_j/x_j]_{j=1}^n, e[p_i/x_i]_{i=1}^n, S) \quad \text{(let)} \]

\[ \{ \}, \quad \text{let } i = \lambda y.y \text{ in } (\lambda x.x) i, \quad \epsilon \]
\[ \implies (\{ p \mapsto \lambda y.y \}, (\lambda x.x) p, \quad \epsilon ) \quad \text{(let)} \]
\[ \implies (\{ p \mapsto \lambda y.y \}, (\lambda x.x), \quad p : \epsilon ) \quad \text{(app)} \]
\[ \implies (\{ p \mapsto \lambda y.y \}, p, \quad \epsilon ) \quad \text{(app)} \]
\[ \implies (\{ \}, \quad \lambda y.y, \quad \#p : \epsilon ) \quad \text{(var)} \]
\[ \implies (\{ p \mapsto \lambda y.y \}, \quad \lambda y.y, \quad \epsilon ) \quad \text{(var)} \]

Figure 3.3: Sestoft’s mark 1 machine for lazy evaluation [Ses97] and an example trace.

Presentation. This reinforces the remarks about hidden costs in our discussion and suggests that natural-semantics may be too abstract to give a simple space semantics. Another way to adapt Launchbury’s rules to modelling space usage is Sansom’s costcentre semantics [San94].

### 3.1.3 Abstract machines

A small-step structural operational semantics [Plo81] specifies how the individual steps of a computation take place. An abstract machine is a structural semantics whose rules are flat: each rule rewrites a configuration to the next configuration in the evaluation trace. Evaluating a program is modelled by beginning with an initial configuration and then applying rules for as long as possible. This evaluation trace terminates with a configuration that represents the result of a program.

**Example 3.3 (An abstract machine for lazy evaluation)**

Sestoft [Ses97] derives an abstract machine for lazy evaluation from Launchbury’s natural semantics. This machine is a rewrite relation \( \implies \) on configurations comprising heap and expression components (built exactly as in the natural semantics), plus a stack component. Sestoft chooses to separate the category of variables which are bound in expressions from the category of pointer variables that form the domain of the heap. The stack is a list of pushed application-argument variables and update markers for variables written \#x. Stack elements are separated by colons and the empty stack is written \( \epsilon \). The initial configuration for evaluating expression \( e \) is \( \{ \}, \epsilon, \epsilon \).

There are five rules shown in Figure 3.3 with the evaluation trace for the program from Example 3.1.

- \( \text{(app)} \) begins to evaluate an application, corresponding to moving from the left conclusion configuration to the left premise configuration in the natural rule (Application). The pointer-variable argument of the application is pushed on top of the stack and the applied expression moves into the control position.

- \( \text{(app)} \) completes the work of the natural (Application) rule. When the control expression is a \( \lambda \)-value and the top of the stack is a pushed argument, the
argument is popped and substituted for the bound variable in the body which becomes the control expression. This substitution is an in-place update.

- \((\text{var}_1)\) begins to find the value of a variable \(p\). Like the \((\text{Variable})\) rule, it finds (in constant time, presumably) the current definition of \(p\) in the heap, removes the binding for \(p\) to model black holing and copies \(e\) into the control position. It also pushes an update marker for \(p\) onto the stack.

- \((\text{var}_2)\) finishes the evaluation of a variable. When the control expression is a \(\lambda\)-value and the top of the stack is an update marker for \(p\), the stack is popped, the binding for \(p\) is re-instated and the value is copied back into the heap. This must be a copy to make the destructive reduction in \((\text{app}_2)\) safe — there is no hidden sharing of expressions or sub-expressions.

- \((\text{let})\) adds locally defined variables unevaluated to the heap. That is, \(n\) new heap locations are created and the expressions \(e_i\) are copied into those locations. This semantics renames variables in the allocation step, rather than using the \(\hat{e}\) operation of the natural rule \((\text{Variable})\). Renaming in \((\text{let})\) seems preferable as we can imagine the heap allocator supplying the new names \(p_i\) from a free list. If a variable is used more than once then renaming a variable at allocation is more efficient — though less lazy — than renaming its value each time it is requested.

If a trace ends in a terminal configuration of the form \((\Gamma, \lambda x. e, e)\) it has a value. This machine differs from the natural semantics in that a valueless program still has an evaluation trace. The trace might not terminate. Or, if there is an error in the program or a variable is self-dependent, it ends in a non-terminal configuration. \(\square\)

The abstract machine is quite low level but it still ignores many practical implementation issues. It uses term substitution without defining how it works — no need to model environments or de-bruijn indices. It copies whole terms in one step. It treats the heap as a series of boxes of unlimited capacity — it is perfectly safe to update variables with expressions that are larger than their original definition. These points might suggest that the semantics is not a very realistic model but Sestoft shows how a much lower-level abstract machine can be derived from this one which is very close to real implementations of lazy evaluation. Mountjoy [Mou98] derives an abstract machine which models the STG-machine for lazy evaluation and which essentially has the same time behaviour as Sestoft’s machine.

Proofs about abstract machines work by induction on the length of the evaluation trace [NN92]: show that initial states have a property and that the rewrite rules of the semantics preserve that property.

Making an abstract machine space semantics is easier than making a natural space semantics because the size calculations can be separated from the semantics. As before, we define a garbage collector for a configuration and the size of a configuration. There are no hidden costs to worry about because a configuration contains all the information needed to derive the next configuration in the trace — it includes the stack.

**Example 3.4 (Abstract machine space semantics)**

We define the size of a configuration as (3.4), the definition of reach is the same as in (3.2). Again, the heap size is the number of reachable bindings, but this time reachable
just means reachable from a free variable of $e$ or a pushed argument on the stack. The stack size is its length.

$$\text{size } (\Gamma, e, S) = \#S + \#\text{reach } (\Gamma, f, S \cup \emptyset, e)$$

We can define the space usage of a configuration (3.5) as the largest configuration generated during its evaluation.

$$\text{space } (\Gamma, e, S) = \max \{\text{size } C \mid (\Gamma, e, S) \rightarrow^* C\}$$

It is very easy to verify that the space usage of the example program in Figure 3.3 is 2, which agrees with the result in Example 3.2 (the truth is that the natural semantics’ space measure is derived from the abstract machine space measure). \(\square\)

Apart from their uses for showing the correctness of implementations, abstract machines have been used for reasoning about space as we see in Section 3.2. Some recent compile-time garbage collection techniques are defined by operational semantics and abstract machines [Han98], rather than denotational store semantics. Sestoft’s machine is also the basis for an analysis to detect unnecessary variable updates [Gus98]. Both big-step and small-step semantics can be used to describe space usage but it is clear that small-step space semantics are much simpler.

### 3.2 Modelling and reasoning

Armed with an accurate space semantics we can do some proofs. We might make a reference implementation of the operational semantics and run programs to determine what their space usage should be. We might formulate rules for calculating bounds for the space usage of a program when it runs.

We can also do comparative reasoning. We could devise techniques to compare the space usage of two algorithms. Or we could work out which of two implementation techniques gives better space usage.

All these lines of work are quite difficult in the presence of dynamic memory management. This section reviews some of the work that tackles them based on the approach of defining what we call a space semantics.

#### 3.2.1 Space-safe transformation

Program transformation is used heavily in compilers for functional programming languages. It can be important that these transformations do not cause an unexpectedly large increase in the space usage of a program — they should not cause space leaks.

**Continuation passing and conservative transformations**

Appel’s book [App92] describes the implementation of an optimising compiler for Standard ML. This compiler does a lot of transformation and a chapter is devoted to the subject of which transformations are space safe. The basic compilation method transforms Core ML programs into continuation passing style (CPS) programs. The CPS semantics provide some useful features to aid reasoning about space.
“CPS is a concise and easily manipulable representation that quite precisely characterises liveness properties for garbage collection.” [App92]

The CPS transformation in its simplest incarnation [Min00] turns call-by-value λ-calculus programs into programs built from the following grammar.

\[
M ::= M(x, \ldots, x) \mid \text{let } x = \lambda x \ldots x.M \text{ in } M
\]  

(3.6)

**Example 3.5 (continuation passing style program)**  
Converting the λ-calculus term \((\lambda x.x)\ y\) to CPS using the rules in [Min00] gives the following program.

\[
\begin{align*}
\text{let } k_1 & = \lambda f. \text{ let } k_2 = \lambda a.f\langle a, \text{stop}\rangle \\
& \quad \text{in } k_2\ y \\
& \quad \text{in } f = \lambda x k. k\langle x\rangle \\
& \quad \text{in } k_1\langle f\rangle
\end{align*}
\]  

(3.7)

Every intermediate result and every function needed during evaluation — everything that needs to be stored somewhere — has a name in the CPS language. Better still, some memory management is explicit in the program because the evaluation of a let-expression corresponds to the allocation of a function closure in the heap. Functions never return as such, instead one of their arguments is a continuation which gets applied to their result. In the example, the applied function \((\lambda x.x)\) is named \(f\) and extended with an extra argument \(k\), the continuation which it applies to \(x\). Thus the run-time stack is represented more explicitly in the program by allocating a continuation function to push stack and applying one to pop stack. The final continuation, which denotes the end of the stack, is a special function \textbf{stop}. □

A CPS semantics (Appel’s denotational store semantics [App92] or an operational semantics such as Minamide’s [Min00]) still needs a model of the heap so that sharing can be represented properly and it still needs a garbage collector but it makes a good space semantics for several reasons [App92].

- The tail recursion optimisation is explicit in CPS programs: \(\eta\)-reduction of programs makes them properly tail recursive in the CPS semantics.

- Explicit environment trimming: all the heap bindings kept by a reachability-based garbage collector are reachable from the free variables of the current continuation.

- There is a well-defined notion of a space-safe conservative transformation.

Appel discusses the space-safety of several local program transformations. No exact space-safety proof method is given but some basic necessary conditions are. These safe conservative transformations do not extend the set of reachable variables from a term, so they do not affect the reachability or lifetime of other terms. In addition to this it is necessary to argue somehow that the transformation is safe \textit{in any context} — that replacing a sub-term with its transform cannot increase the space usage of the whole program unreasonably.
Example 3.6 (Space-safe transformations)
Two simple transformations that are space-safe are \((\beta\text{-reduction})\) and
\((\text{dead-binding-elimination})\).

\[
\begin{align*}
\text{let } f &= \lambda x_1 \ldots x_n. M \text{ in } \cdots f(y_1, \ldots, y_n) \cdots \\
\implies \text{let } f &= \lambda x_1 \ldots x_n. M \text{ in } \cdots M[y_i/x_i]_{i=1}^n \cdots \quad (\beta\text{-reduction}) \\
\text{let } f &= V \text{ in } M \implies M, \text{ if } x \text{ is not free in } M \quad (\text{dead-binding-elimination})
\end{align*}
\]

Both transformations are safe in the left-to-right direction only. The rather complicated CPS rendering of \((\lambda x. x) y\) that we obtained in Example 3.5 can be simplified using these transformations. We can think of \((\beta\text{-reduction})\) as compile-time evaluation and \((\text{dead-binding-elimination})\) as compile-time garbage collection. The result of applying both transformations once is shown below.

\[
(3.7) \implies \text{let } k_1 = \lambda f. \text{let } k_2 = \lambda a. f (\text{\texttt{stop}}) \text{ in } f(y, \text{\texttt{stop}}) \\
\text{in } \text{let } f = \lambda x k. k(x) \\
\text{in } k_1(f) \\
\implies \text{let } k_1 = \lambda f. f (y, \text{\texttt{stop}}) \text{ in } f = \lambda x k. k(x) \\
\text{in } k_1(f)
\]

After two more applications of each transformation the term reduces down to \text{\texttt{stop}}(y). The transformations eliminate all the reduction steps needed to apply the identity function to \(y\).

Intuitively, these transformations are space-safe: before the transformed expression is evaluated \((\beta\text{-reduction})\) can make heap bindings reachable only from \(\{y_i\}_{i=1}^n\) unreachable; \((\text{dead-binding-elimination})\) can make heap bindings reachable only from \(f v V\) unreachable; neither transformation introduces free variables that could increase the lifetime of any heap binding; at the point of evaluation the transformed terms lead to the same state as the untransformed terms.

In the right-to-left direction, \((\beta\text{-reduction})\) is dangerous when the function body does not mention some \(x_i\) so it could increase the lifetime of \(y_i\). Similarly, \((\text{dead-binding-elimination})\) may introduce free variables in \(V\) which could extend the lifetime of some heap bindings. Repeated use of either transformation also increases term size which is dangerous if there is no limit to the number of transformations.

Space improvement theory

Gustavsson and Sands [GS00, GS01, Gus01] define an algebra for space-safe transformation of lazy functional programs. This is based on a similar algebra for time-safe transformation [MS99]. The space semantics of lazy evaluation they use is almost Sestoft's Mark 1 machine which we reviewed in Section 3.1.3. The difference is a small extension to the garbage collector to remove \textit{dead update markers} from the stack component. The garbage collector is defined by making update markers binding occurrences for variables and finding the smallest closed configuration.

The space usage of a program is defined as a pair \((h, s)\): the minimum heap size \(h\) and the minimum stack depth \(s\) needed to evaluate it assuming maximal garbage
collection. Programs that fail, or fail to terminate, are not part of the theory. To compensate for the way that the semantics models black holing (discussed in Example 3.1), heap size is defined as the number of bindings plus the number of (live) update markers on the stack — another example of how care needs to be taken when calculating space costs.

Then a space-improving local program transformation is expressed by writing $M \triangleright N$ if $N$ uses no more than some constant factor as much space than $M$ in any context. This might not sound much like improvement — allowing some worsening — but the space semantics is only accurate enough to describe complexity. So $M \triangleright N$ means that the space complexity of $N$ is no worse than $M$.

To show improvement, a context lemma is used to localise the reasoning: the behaviour of a term before and after transformation is compared at the point of evaluation. Like the space-safe conservative transformations [App92] this also requires that the transformation does not introduce any free variables so it does no damage before the point of evaluation. The proof method uses a stronger definition of an improving transformation defined $M \triangleright N$ if $N$ uses no more space than $M$. To show a strong improvement, terms are annotated with symbols to represent various aspects of space usage and these are manipulated by laws which guarantee preservation of space usage.

Example 3.7 (Space-safe transformation)

\[
\begin{align*}
\text{let } i &= \lambda x.x \text{ in } i y \\
\triangleright &\approx \text{let } i &= \lambda x.x \text{ in } (\lambda x.x) y \quad \text{(value-\textbeta)} \\
\triangleright &\approx (\lambda x.x) y \quad \text{(gc)} \\
\triangleright &\approx y \quad \text{(reduction-eq)}
\end{align*}
\]

The example program transformation above uses three space-safe steps. Some steps like (value-\textbeta) can increase the term size slightly; like the classic loop unrolling transformations of imperative compilers this is safe provided it is only done a bounded number of times; more importantly, the transformation does not affect the lifetime of terms elsewhere in the context.

Improvement theory does not directly help with reasoning about when a transformation is space worsening. If a transformation cannot be shown improving by the various laws, to show worsening it is necessary to find example programs whose space complexity is increased by the transformation.

Example 3.8 (Space-unsafe transformation)

Part of the motivation for the work in this dissertation was a normalising transformation scheme which we designed in [BR99]. A good rationale for choosing which direction to apply normalising transformations in is to make the normal form of a program the version with the best space complexity. We did not tackle this problem in [BR99] but consider one of its transformations (3.8) which eliminates some repeated computation work.

\[
\begin{align*}
\text{case } e_{\text{of}} \{ \cdots ; k \ v_1 \cdots v_n \to C(e) ; \cdots \} \\
\implies \text{case } e_{\text{of}} \{ \cdots ; k \ v_1 \cdots v_n \to C(k \ v_1 \cdots v_n) ; \cdots \} \quad (3.8)
\end{align*}
\]
The right-to-left direction is not a space improvement. To see this consider the following program.

\[
\begin{align*}
\text{repeat } x &= \text{let } xs = \text{repeat } x \text{ in } x : xs \\
\text{take } 0 \ l &= [] \\
\text{take } n \ [] &= [] \\
\text{take } n \ (x : xs) &= \text{take } (n - 1) \ xs \\
\text{list} &= \text{take } n \ (\text{repeat } []) \\
\text{main} &= \text{case } (\text{last list}) \text{ of } \{ [] \to \text{last list} \}
\end{align*}
\]

The data structure list is persistent in this program: it is computed when the case selector in main is evaluated and remains reachable until the expression last list in the case alternative is evaluated: the program has \( O(n) \) space usage. Applying the transformation (3.8) changes the definition of main to case (last list) of \{ [] \to [] \}. Now list is transient: the case selector computes in constant space because there are no other references to list; the space complexity of the program is \( O(\log(n)) \), the space needed to store \( n \).

So (3.8) should be applied left-to-right? Unfortunately this direction is not an improvement either. The following program runs in \( O(\log(n)) \) space as both lists of \( n \) zeros are transient.

\[
\begin{align*}
\text{main} &= \text{case } (\text{let } l = \text{take } n \ (\text{repeat } 0) \text{ in } (\text{head } l, \text{last } l)) \text{ of} \\
&\quad \{ (f, s) \to s + \text{snd } (\text{let } l = \text{take } n \ (\text{repeat } 0) \text{ in } (\text{head } l, \text{last } l)) \}
\end{align*}
\]

Applying the transformation gives the following program in which \( l \) is persistent so the space complexity is \( O(n) \).

\[
\begin{align*}
\text{main} &= \text{case } (\text{let } l = \text{take } n \ (\text{repeat } 0) \text{ in } (\text{head } l, \text{last } l)) \text{ of} \\
&\quad \{ (f, s) \to s + \text{snd } (f, s) \}
\end{align*}
\]

\[\square\]

Many useful transformations can damage space usage in some circumstances. Examples include full laziness [Pey87] and strictness transformations [Wad87b]. In general optimising the space behaviour of a program is an inherently global problem.

The local transformations listed as safe by Appel [App92] and those listed as improvements by Gustavsson and Sands [GS00] are remarkably similar. Work regarding the limits of space-improvement theory [Gus01, GS01] tackles more complex transformations involving recursive functions and shows up some of the differences between the space usage of strict and lazy evaluation. The examples include different representations of cyclic data structures (which are impracticable in call-by-value); a deforestation transformation which does not affect space complexity (in call-by-value it can improve complexity); and the use of a strictness annotation to improve space complexity (deforestation is a bit like a laziness transformation for call-by-value). So the principles of the space-safe transformation algebra would carry over to call-by-value but the results do not.

### 3.2.2 Space-safe implementation

After transformation and compilation the space usage of a program still depends on how the language is implemented: the combination of evaluation and garbage collection strategies that rewrite the program to its result.
For example, there are a number of ways in which CPS can be implemented. Morrisett and Harper [MH97] describe a language $\lambda_{gc} \rightarrow \nu$ — a Core ML with types — and an abstract machine for its evaluation. The abstract machine has configurations of the form $(H, S, E, e)$, that is (heap, stack, environment, control expression). It is a small-step semantics so it also provides a space semantics for ML. Their garbage collector is specified as another abstract machine; it is a copying collector with configurations comprising from-space, to-space and frontier set.

The abstract machine and collector can be seen as a reference implementation which could be used for comparing the space usage of different programs. It can also be seen as a framework for specifying different implementations of $\lambda_{gc} \rightarrow \nu$ which may have different space usage.

**Example 3.9 (Proper tail call elimination in $\lambda_{gc} \rightarrow \nu$)**

Appel [App92] notes that $\eta$-reduction in CPS programs corresponds to the space-saving tail-call optimisation. Morrisett and Harper show how this can be incorporated into the $\lambda_{gc} \rightarrow \nu$ abstract machine by defining a translation rule to remove identity continuations from its stack:

$$(H, S_1 \circ (\{E', \lambda x : t.x\}) \circ S_2, E, e) \implies (H, S_1 \circ S_2, E, e)$$

This translation is shown not to affect the outcome of the computation. A better version of the abstract machine is presented in the same framework which never produces any configuration containing an identity continuation.  

Clinger [Cli98] takes this approach further. He uses essentially the same language as $\lambda_{gc} \rightarrow \nu$ and a similar four-component abstract-machine framework for specifying the implementation of Core Scheme programs. The space semantics of the framework are defined by giving functions to measure all of the components and the space usage of any program that does not lead to an error state is defined as the maximum configuration size during evaluation. Six different implementations are compared. Implementation $X$ is shown to be consistently no worse than Implementation $Y$ by the argument discussed below and Implementation $Y$ is shown to be asymptotically worse than Implementation $X$ by stating a program which exposes the inefficiency of $Y$.

The proof technique for showing that $X$ is never worse than $Y$ for any program needs to solve several technical problems. Clinger outlines his solution which is essentially a simulation proof technique: the evaluation trace of a program on $X$ is translated to its evaluation trace on $Y$. To argue that the space usage is not improved by this translation, garbage and its collection need to be considered. Maximal garbage collection can make the evaluation trace translation difficult to establish. So the evaluation trace translation assumes no garbage collection, and a separate argument says that garbage collection on $X$ removes at least as much as garbage collection on $Y$. Another problem is that the traces can have different lengths — the tail-call optimisation (Example 3.9) eliminates the steps that evaluate the identity continuations it removes. Clinger handles this problem by using a global argument: the evaluation traces are translated whole.

Similar work in this area includes Minamide’s comparison of call-by-value evaluation of $\lambda$-calculus terms with their CPS translation [Min00], showing that the CPS translation neither damages nor improves space complexity.
As well as specifying different abstract machines, the \( \lambda_{ge}^{c,\gamma} \) framework can describe different garbage collectors. Morrisett, Felleisen and Harper [MFH95] specify a type-based collection strategy which is shown to remove some reachable garbage (therefore it is better than the standard copying collector, though they do not discuss how much better in the way that Clinger’s proofs do).

To summarise, \( \lambda_{ge}^{c,\gamma} \) is a detailed solution to the problem of how to model space usage which can be used to describe and compare the space behaviour of different variations on its implementation and garbage collection by global proof arguments. The proof methods could be extended to other similarly expressed semantics. Clinger’s approach of presenting a hierarchy which shows how the space usage of one implementation relates to others is appealing. But the proofs often need to be tailored carefully for the particular implementations under comparison. So it would be interesting to see if a general means of comparing \( X \) with \( Y \) can be established — a framework for the proofs as well as the semantics.

### 3.2.3 Term rewriting with sharing

Frameworks like \( \lambda_{ge}^{c,\gamma} \) are quite low-level, making assumptions about the languages they describe and how their implementations work. For a space semantics which is closer to the program level a more abstract framework is desirable. This can be equipped with a standard notion of space usage which is proven realistic once and for all so an operational semantics automatically becomes a space semantics. The framework may have other properties which support any reasoning about the space behaviour of different programs or semantics.

Term rewriting is a popular abstract model for studying many aspects of computation. For functional programming languages this typically means reasoning about call-by-name evaluation of \( \lambda \)-calculus terms, e.g. [Abr90]. However, term rewriting does not seem a good choice for studying space because it ignores sharing, so defining state size as term size could be a very inaccurate over-estimate. Ignoring copying by assuming that all occurrences of identical sub-terms occupy the same physical space is equally dubious because implementing term rewriting to work in this way makes each rewrite a global operation. So the semantics would not be a realistic time-usage model.

Rose [Ros96] shows how term rewriting can be adapted to model space. The first change is to introduce address labels to indicate that a sub-term is shared. For example, in \( \lambda x. (\text{square} . x)^a \times (\text{square} . x)^a \), both occurrences of \( \text{square} . x \) reside at address \( a \) so evaluating one of them evaluates both simultaneously. To support this change, terms must satisfy a well-formedness condition which says that any two sub-terms with the same address are the same. Term-rewrite rules must preserve this well-formedness condition. For example, \( \beta \)-reduction can be written as follows.

\[
(\lambda x.Z(x)) \ X \rightarrow Z(X^a)
\]

Term variables are written in upper-case, \( Z(x) \) is a term context which may contain several occurrences of the variable \( x \). The application is reduced by substituting argument for parameter as normal but an address label is added so that instead of each occurrence of \( x \) in \( Z \) getting a separate copy of \( X \), they all share the same one. But if the function \( (\lambda x.Z(x)) \) is shared we do not want to reduce all its other uses too. So a new rule duplicates shared sub-terms before they are reduced:
\[ Z^v \ X \longrightarrow (Z \parallel Z^v) \ X \] (3.11)

The operation \((Z \parallel Z^v)\) results in a copy of \(Z\). All other occurrences of \(Z^v\) are unchanged, so this rewrite is still a local operation which requires bounded work to implement.

As well as sharing and copying, Rose uses a notation to allow cyclic references. The term \(\bullet^a\) indicates such a reference to address \(a\).

**Example 3.10 (Cyclic shared term rewriting)**

Defining the function \(\text{sqrt}\) as the term \(\lambda n. (\lambda x. \text{if} (\text{square} \ x) \geq n \ \text{then} \ x \ \text{else} \ \bullet^f (x + 1))^f 0\)

we can visualise the evaluation of \(\text{sqrt} \ 0\) as follows.

\[
\begin{align*}
(\lambda n. (\lambda x. \text{if} (\text{square} \ x) \geq n \ \text{then} \ x \ \text{else} \ \bullet^f (x + 1))^f 0) \ 0 \\
\longrightarrow (\lambda x. \text{if} (\text{square} \ x) \geq 0^a \ \text{then} \ x \ \text{else} \ \bullet^f (x + 1))^f 0 \\
\longrightarrow (\lambda x. \text{if} (\text{square} \ x) \geq 0^a \ \text{then} \ x \ \text{else} \ \bullet^f (x + 1))^f (0^b + 1) \\
\longrightarrow^* 0
\end{align*}
\]

The first step replaces \(n\) with the argument \(0\) stored at a new address \(a\). The second step takes a copy of the sub-term at address \(f\), but does not duplicate \(0^a\) — both occurrences refer to the same place. The third step reduces this copy of \(f\), replacing \(x\) with the argument \(0\) stored at new address \(b\) (so \(0^a\) and \(0^b\) occupy different addresses). Other rules reduce this term to the result. \(\square\)

As well as rewrite rules, we need a strategy to decide where to apply them. This aspect of computation affects space because it usually means maintaining a stack or a continuation. Rose shows how an abstract machine similar to the lazy small-step semantics [Ses97] can be built in this notation. The difference is that there is no need for a separate heap component because the addressed sub-terms in the main term and the stack collectively represent the heap.

**Reasoning about space**

So with some extra notation term rewriting can specify a space semantics. In his thesis [Ros96], Rose defines a number of abstract machines which can be thought of as reference models for the space usage of functional programs. However, he does not discuss the space behaviour of these machines or how they could be used for reasoning about space.

In a paper [BLR96], Rose and others define a *space leak* formally. A calculus (operational semantics) is said to be *space-leak free* if no unreachable term in their standard model stays indefinitely in the calculus. This is an interesting definition because it intuitively captures the meaning of *leak*: some space disappearing forever. But it does not really guarantee any worsening in practice because an abstract calculus is not accurate enough; it also does not guarantee worse space complexity in theory because a single unit of space leaking counts as a leak; it may also ignore worse space complexity if the leak is temporary.

An example calculus which models environment-based reduction is presented in [BLR96]. It includes *environment trimming* operations which are shown not to cause
leaks for any of several weak-reduction evaluation strategies. By showing that some of the trimming operations are unnecessary for the call-by-need strategy they argue that the STG-machine (an environment-based implementation of call-by-need) is not leaky.

3.2.4 Term-graph rewriting

“Graph reduction very accurately models the essential behaviour of most implementations of functional languages.” [ESP96]

The implementation of functional languages is often described as graph reduction and graphs are a natural model of sharing. Therefore it seems appropriate to develop graph rewriting systems which are suitable for defining space semantics. Representing configurations as graphs we can easily measure their size. An operational semantics can be defined as a collection of graph-rewrite rules.

The basic graph model is a term graph [Plu99, BEG+87]. A term graph is a collection of nodes which each contain a function symbol and arcs which interconnect nodes. A graph may also have a root node to indicate where the value it represents begins, or to indicate where rewrite rules should be applied in the case of rooted graph rewriting.

Example 3.11 (Term graphs)
The compound term $F(G)$ can be represented as a graph containing two nodes. One node contains $F$ and the other $G$. A directed arc connects the $F$ node to the $G$ node and the root of the graph is the $F$ node.

Sharing is represented by arranging for two arcs to point to the the same node, so $H(G,G)$ can be defined as a graph of two nodes and two arcs. A cyclic term like $(F(e'))'$ is represented directly by a cyclic graph. □

So graphs are naturally the right model for describing many aspects of space usage. The shared term notation of Rose [Ros96] is just another way of describing a term graph. A heap in the operational semantics for laziness or in $\lambda^{cy}_{pc}$ is yet another way of describing a graph. The stacks in those semantics are acyclic graphs. In short, graphs are the general purpose notation for describing space usage.

Equational term graph rewriting

To present a space semantics as a graph-rewrite system, we need to describe how sharing is established and how sub-terms are copied. For higher-order languages we
also want graphs to describe higher-order terms and semantics to describe substitution as neatly as term rewriting does. These concepts are difficult to define with plain term graphs.

**Example 3.12 (sub-graph copying problem)**
The left graph in Figure 3.5 shows one way to represent the following program as a term graph.

\[
\text{let } \{ f = \lambda x.g \, x, g = \lambda y.f \} \text{ in } f \, g
\]

To apply \( \beta \)-reduction to this graph we want a graph-rewrite rule that can take a copy of the sub-graph representing the body of \( f \) and substitute arcs to \( g \) for occurrences of the bound variable \( x \). This produces the right graph in Figure 3.5. Such a rule can be devised easily enough for \( \beta \)-reduction of the graph representing \( f \). So we could define the space semantics of a functional programming language by a transformation which compiles a program into a collection of rooted graph-rewrite rules [ESP96].

To describe an operational semantics directly as a collection of graph-rewrite rules requires a generic \( \beta \)-reduction rule. This is problematic with graphs of function symbols because there is nothing to define the limits of the sub-graph that represents the term that we want to copy and substitute into. The equational term-graph model [AK96, AK97, Blo01], also known as sharing graphs [Has97], provides one solution to this problem. Rose’s notation for terms with sharing is another solution.

**Example 3.13 (Equational term-graph)**
An equational term-graph is a set of recursion equations: a set which associates variables with terms. The graphs of Figure 3.5 can be represented equationally as follows.

\[
\langle a \rangle \{ f = \lambda x.g \, x, g = \lambda y.f, a = f \, g \} \rightarrow \langle a \rangle \{ f = \lambda x.g \, x, g = \lambda y.f, a = g \, g \}
\]

These equational graphs-of-terms solve the scope problem. The scope of bound variable \( x \) in node \( f \) is restricted to the sub-term \( g \, x \). We can tell that the term \( \lambda y.f \)
is not in the scope of $x$ because it occurs in a separate graph node. These graphs could still be represented pictorially as Figure 3.5, to represent the scope information we could add boxes to indicate which nodes are in the scope of each lambda [Blo01]. Another way to think of this is to distinguish external nodes, which have addresses in the equational presentation, from internal nodes, which are sub-terms in the equational presentation.

Graphs with a notion of sub-term or scope enable the definition of operational semantics as equational graph rewrite rules in which copying and sharing are explicit, and arguably easier to visualise than in Rose’s notation for terms with sharing.

**Example 3.14 (Equational graph rewriting)**

A rooted equational term graph-rewrite rule for $\beta$-reduction can be written as follows.

$$\langle a\{a = \lambda x. Z(x) \ Y \} \cup G \rangle \longrightarrow \langle a\{a = Z(b), b = Y \} \cup G \rangle \quad (3.12)$$

The argument term which matches $Y$ is copied to a new node $b$ and the node $a$ is overwritten with the term matching $Z$ where arcs to $b$ replace any occurrences of $x$. The variable $G$ stands for the rest of the graph which is unchanged by the rewrite.

If $a$ contains a cyclic reference this rewrite is dangerous because the value of the term in node $a$ changes. This rule does not describe the reduction shown in Figure 3.5; that rewrite can be defined by (3.13) which leaves the definition of $f$ intact. Alternatively, a rule similar to Rose’s (3.11), or the $(\text{var}_1), (\text{var}_2)$ variable lookup then update regime of Sestoft’s semantics (Figure 3.3), could be used to duplicate a function body before it is reduced.

$$\langle a\{a = f b, f = \lambda x. Z(x) \} \cup G \rangle \longrightarrow \langle a\{a = Z(b), f = \lambda x. Z(x) \} \cup G \rangle \quad (3.13)$$

Reasoning work in graph rewriting theory, especially in the more recent equational formulation, tends to concentrate on fundamental questions such as what kinds of confluence properties graph rewriting has [Blo01]. Modelling work is much more extensive [Has97, AA95, Ros96]. If we extend the definition of an equational term graph to include any structure which can be thought of as a set of recursion equations then most of the work mentioned in this chapter fits the bill.

Space leaks are considered by van Eekelen et al [ESP96] in their discussion of graph rewriting semantics. They say that “space leaks are a mythical problem due to lack of graph semantics”. They go on to note some different causes of space leaks.

1. Design errors: including the use of time-for-space trading techniques such as tabulation.

2. Programmer errors: accidental inefficiency caused by misunderstandings about the operational behaviour of programs.

3. Implementation errors: caused by deviation from the graph semantics.

This classification scheme is helpful but the different kinds of leak are not defined formally and there is no discussion of reasoning techniques for detecting them.
3.3 Summary

An operational semantics can be used to define the space semantics of a programming language implementation by extending it with a garbage collector and a measure of state size and proving that the model is realistic. Big-step or small-step formulations are possible; the small-step formulation is often simpler as it includes most of the relevant information directly and space calculations can be separated from the semantic rules.

Then ideas like space-safety and space leaks can be given a formal definition and techniques devised to prove that changing a program or implementation is safe. Space-improving transformations do not damage the space complexity of a program in any context. The alternative is to consider global transformation or comparison of different evaluation strategies; the reasoning is similar whichever question is being asked.

Frameworks like $\lambda_{dc}$ or equational term-graph rewriting can be used to specify a whole range of implementations and to formalise notions like space leaks. Graph rewriting declares itself the model of choice for such activities but it is not so clear how well suited term-graph rewriting is as a basis for reasoning about space and how such proofs would work. The following chapters investigate these questions.
Part II

Term Graphs and Space Usage
Chapter 4

GraphKit: A Term-Graph Rewriting System for Operational Semantics

GraphKit is a term-graph rewriting tool designed for specifying structural operational semantics for space analysis. This chapter explains abstractly how GraphKit works and some of the theory in this formulation of graphs and rewriting.

The GraphKit input specifies a graph grammar, an initial graph and a graph evaluator — a programming language, an initial program and the operational semantics of the language. These three components are explained in Sections 4.1 to 4.3. GraphKit also includes a simple stream-based IO model which is explained in Section 4.4. For more information on GraphKit see Appendix A.

All the grammars, graphs, evaluators and evaluation traces in this and subsequent chapters are generated by GraphKit. We begin with Example 4.1 which shows how GraphKit can be used to specify a structural operational semantics for lazy evaluation. This example evaluator is our canonical call-by-need semantics which forms the basis for the variants introduced and compared in later chapters.

Example 4.1 (A GraphKit session)
Figure 4.1 shows a GraphKit input file which specifies a simple programming language: the \( \lambda \)-calculus extended with let-expressions, an example graph program and a set of graph rewrite rules which specify lazy (call-by-need) evaluation.

The graph rewrite rules — which we refer to collectively as the evaluator [lazy] — are a term-graph interpretation of Sestoft’s structural semantics for lazy evaluation [Ses97] (see Figure 3.3). The differences are: our let-expressions can only allocate one binding; the stack, heap and control expression are all part of a graph which has a root for the control expression and a root for the stack top; variable lookup works by moving the control address to the variable instead of copying the variable’s definition into the control position. A proof that this semantics is operationally equivalent to Sestoft’s semantics is given in [BR01] or Section 10.3.

The initial graph shows how the boolean values \( True \), \( False \) and \( not \) can be encoded in the \( \lambda \)-calculus. The graph has one node whose expression defines combinators to represent \( True \) and \( False \) and then applies the combinator representing \( not \) to \( True \).

The Latex-formatted GraphKit output for this input is shown in Figure 4.2. The rest of this chapter explains how GraphKit works with reference to this example.  

53
-- Graph grammar definition
-- Graphs have an expression root and a stack root
ROOT ::= x s

-- Expressions are lambda terms extended with let and a Bot constant
X ::= LAM x . X "\lambda \#1. \#1" "\#1. \#1"
  | APP X x "@1 \: \#2" "@1 \@2"
  | VAR x "@1" "@1"
  | LET x . X X "\{rm let\}:\#1=\#1,\{rm in\}:\#2" "let \#1=\#1 in \#2"
  | BOT "\bot" "_ _"

-- Stack terms hold pushed arguments and variable addresses
S ::= PSH x s "@1 : \#2" "@1: \#2"
  | UDM x s "\# \#1 \: \#2" "\#\#1 \#2"

-- Initial graph definition
-- Graph program to compute (not True)
a = LET true . (LAM x . (LAM y . (VAR x)))
  (LET false . (LAM x . (LAM y . (VAR y)))
  (APP (LAM x . (APP (APP (VAR x) false) true)) true))
root = a, null

-- Graph evaluator definition
-- Call-by-need operational semantics
\{a = APP F x\}a,s -> \{a = F, t = PSH x s\}a,t "Push"
\{a = LAM y.E, s = PSH x t\}a,s -> \{a = E[x/y]\}a,t "Reduce"
\{a = VAR x\}a,s -> \{a = BOT, t = UDM a s\}x,t "Lookup"
\{a = LAM x,E, y = BOT, s = UDM y t\}a,s -> \{y = LAM x.E\}y,t "Update"
\{a = LET y . E X\}a,s -> \{a = X[b/y], b = E[b/y]\}a,s "Let"

Figure 4.1: Example GraphKit input. The lines beginning with two dashes are comments. The rest are the BNF-style syntactic category definitions (explained in Section 4.1), equations defining the initial graph (explained in Section 4.2) and rewrite rules defining a graph evaluator (explained in Section 4.3).
\[ \text{ROOT} ::= \langle x,s \rangle \]
\[ X ::= \text{LAM } x . X \quad \lambda x . X \]
\[ \quad \text{APP } X \ x \quad X \ x \]
\[ \quad \text{VAR } x \quad x \]
\[ \quad \text{LET } x . X \ X' \quad \text{let } x = X \text{ in } X' \]
\[ \quad \text{BOT} \quad \bot \]
\[ S ::= \text{PSH } x \ s \quad x : s \]
\[ \quad \text{UDM } x \ s \quad \# x \ s \]

\[ \{ a \mapsto F \ x \} a, s \quad \rightarrow \quad \{ a \mapsto F, t \mapsto x : s \} a, t \quad \text{(Push)} \]
\[ \{ a \mapsto \lambda y . E, s \mapsto x : t \} a, s \quad \rightarrow \quad \{ a \mapsto E[x/y] \} a, t \quad \text{(Reduce)} \]
\[ \{ a \mapsto x \} a, s \quad \rightarrow \quad \{ a \mapsto \bot, t \mapsto \# a \ s \} x, t \quad \text{(Lookup)} \]
\[ \{ a \mapsto \lambda x . E, y \mapsto \bot, s \mapsto \# y \ t \} a, s \quad \rightarrow \quad \{ y \mapsto \lambda x . E, a \mapsto \lambda x . E \} y, t \quad \text{(Update)} \]
\[ \{ a \mapsto \text{let } y = E \text{ in } X \} a, s \quad \rightarrow \quad \{ a \mapsto X[b/y], b \mapsto E[b/y] \} a, s \quad \text{(Let)} \]

\[ \{ a \mapsto \text{let } \text{true } = \lambda x . \lambda y . x \text{ in } \text{let } \text{false } = \lambda x . \lambda y . y \text{ in } (\lambda x . \text{false } b) \ b \ b \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Let)} \]
\[ g \in \{ a \mapsto (\lambda x . c \ b) \ b, c \mapsto \lambda x . \lambda y . y, b \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Let)} \]
\[ g \in \{ a \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Push)} \]
\[ g \in \{ a \mapsto \lambda x . \lambda y . x \} b, c \mapsto \lambda x . \lambda y . y, b \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Push)} \]
\[ g \in \{ a \mapsto \lambda x . \lambda y . x \} b, c \mapsto \lambda x . \lambda y . y, b \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Push)} \]
\[ g \in \{ a \mapsto \lambda x . \lambda y . x \} b, c \mapsto \lambda x . \lambda y . y, b \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Push)} \]
\[ g \in \{ a \mapsto \lambda x . \lambda y . x \} b, c \mapsto \lambda x . \lambda y . y, b \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Push)} \]
\[ g \in \{ a \mapsto \lambda x . \lambda y . x \} b, c \mapsto \lambda x . \lambda y . y, b \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Push)} \]
\[ g \in \{ a \mapsto \lambda x . \lambda y . x \} b, c \mapsto \lambda x . \lambda y . y, b \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Push)} \]
\[ g \in \{ a \mapsto \lambda x . \lambda y . x \} b, c \mapsto \lambda x . \lambda y . y, b \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Push)} \]
\[ g \in \{ a \mapsto \lambda x . \lambda y . x \} b, c \mapsto \lambda x . \lambda y . y, b \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Push)} \]
\[ g \in \{ a \mapsto \lambda x . \lambda y . x \} b, c \mapsto \lambda x . \lambda y . y, b \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Push)} \]
\[ g \in \{ a \mapsto \lambda x . \lambda y . x \} b, c \mapsto \lambda x . \lambda y . y, b \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Push)} \]
\[ g \in \{ a \mapsto \lambda x . \lambda y . x \} b, c \mapsto \lambda x . \lambda y . y, b \mapsto \lambda x . \lambda y . x \} a, \epsilon \quad \text{(Push)} \]

Figure 4.2: Example GraphKit output. The grammar is displayed, both the raw term definitions and the LaTeX-macro formatted versions are shown so that the two can be related. Primes are added to category names to make this relationship clear. The evaluator is checked according to the grammar and displayed. The initial graph is also displayed, followed by its evaluation trace and a summary of its resource usage if it terminates. The parts of a graph that change are underlined in the trace.
4.1 Graph grammars

The grammar specified in a GraphKit input (like Figure 4.1) defines a number of syntactic categories in a BNF-like notation where function symbol definitions are separated by vertical bars. A function-symbol declaration is followed by the category names of the variables it binds and then a dot separator and then the category names of its arguments. Function symbols have a fixed arity. Their arguments are either variables whose category name is written in lower-case or they are terms whose category name is written in upper-case in the function-symbol declaration. This distinction lets us construct graphs of terms — rather than graphs of function symbols. The scope of a bound variable is restricted to the arguments of the function symbol where it is bound (this is explained formally in Definition 4.1).

The function-symbol declaration is followed by two quoted strings, its display macros. The first is for Latex documentation and the second is for ASCII display. Whenever GraphKit displays a function symbol, the appropriate macro is applied. Its $i$th bound variable replaces the occurrence of $\#i$ in the macro and the $i$th argument replaces the occurrence of $@i$ in the macro. Of course, macros are just a presentation device. From now on we only show the Latex macro-formatted versions, but it is important to remember that beneath the presentation the nice let-expressions and so on are directly encoded as ugly terms like LET — displayed expressions are not a meta-notation that needs translating to become real terms.

The grammar specification does not include any precedence or fixity information. Our examples are mostly based on Haskell, so expression terms are bracketed as Haskell expressions. The only restriction on display macros is that they must include every bound variable and every argument exactly once. This prevents the displayed terms and graphs from being too misleading — if an argument was not included then reachable graph nodes might appear to be unreachable; including arguments more than once would distort term size and the in-degree and out-degree of graph nodes. Function symbols, such as VAR in the example, may be invisible when displayed to make displayed terms look more natural but we should not forget about their existence.

The special ROOT category must always be defined. Its definition is a sequence of $n$ category names, indicating that graphs must have a sequence of $n$ root variables. For our theoretical treatment, a graph grammar is a set which satisfies Definition 4.1. This is a non-standard use of the term graph grammar, for traditional graph grammars see [Roz97]

**Definition 4.1 (Graph grammar)**

A graph grammar $G$ maps syntactic category names to their definitions.

$G(C)$ gives the definition of category $C$.

A syntactic category definition is a set of function symbol definitions.

Let $FSymbols$ be a denumerable set of function symbols. A function symbol definition $(Fc_1 \cdots c_n.A_1 \cdots A_n)$ says that $F \in FSymbols$ binds $b$ variables and takes $a$ arguments. The category of the $i$th bound variable is $c_i$. If the $i$th argument should be a term of category $C'$ then $A_i = TArg C'$; if it is a variable then $A_i = VArg C'$. A grammar must include a root category whose sole member is an $n$-tuple of category names: $G(\text{ROOT}) = \{ \langle c_1, \ldots, c_n \rangle \}$. $\square$

There are a few naming restrictions on graph grammars: categories and function symbols are upper-case strings and each name may be defined at most once, ROOT
has a special purpose and the string NULL cannot be used.

A grammar must be closed: all the categories mentioned in its function symbol definitions must be defined. GraphKit also checks that all the categories defined are reachable from the \textit{ROOT} declaration. Sometimes this check can help locate typing mistakes in a complicated input file where a function declaration with the wrong argument renders some category unreachable. A warning is issued if any category is \textit{empty}. For example, \textit{EMPTY} ::= 0 \textit{EMPTY} looks like a mistake (0 is a function symbol) as there is no way to construct a finite term (without resorting to the null term of Definition 4.3).

## 4.2 Term graphs

Terms are constructed from a grammar in the obvious way. Every term belongs to a category, so the grammar serves as a very simple type system for the purpose of term checking. Sub-terms should be variables or terms of the appropriate category. Graphs are sets of addressed terms paired with an \textit{n}-tuple of root variables. The equations in a GraphKit input are the addresses and terms in the initial graph. The formal definitions below explain the subtleties of term-graph construction.

### 4.2.1 Term construction

**Definition 4.2 (Term)**

\[
\begin{align*}
(F \ c_1 \cdots c_b \ . \ A_1 \cdots A_a) & \in \mathcal{G}(C) & \Delta = \{x_i \mapsto c_i \}_{i=1}^b \cup \Gamma & \text{function } \Delta \\
(A_i = VArg \ C') & \Rightarrow T_i \in \mathcal{V} & \Delta(T_i) = C' & (A_i = TArg \ C') \Rightarrow \mathcal{G}, \Delta \vdash T_i : C'
\end{align*}
\]

Term \( T \) belongs to category \( C \), written \( T : C \), if there is a variable environment \( \Gamma : \mathcal{V} \rightarrow dom \mathcal{G} \), \( (dom \ f \ is \ the \ domain \ of \ f, \ rng \ f \ is \ the \ range \ of \ f) \) which maps all the free variables of \( T \) to their categories, such that \( \mathcal{G}, \Gamma \vdash T : C \). Any variables bound at a function symbol must have distinct names. They must also be distinctly named from any other bound variables which \( T \) is in the scope of (those in \( \Gamma \)). Bound-variable names and free-variable names do not overlap (so \( \Delta \) is a \textit{function}). These restrictions make substitution much simpler. Two bound variables can only have the same name if they occur in different sub-terms of a term.

Null terms and variables are also permitted. They are useful for ending recursive structures and closing graphs. They also ease the theory in some proofs later on. Both nulls play the role of a null arc when we make graphs of terms — free variables correspond to arcs and sub-terms correspond to \textit{internal nodes} in other higher-order graph frameworks such as [Blo01].

Null terms are defined by the inference rule below. To include null variables \( \epsilon \), we change the first part of the second premise line in Definition 4.2 to \( (A_i = VArg \ C') \Rightarrow T_i \in \mathcal{V} \land \Delta(T_i) = C' \lor T_i = \epsilon \). Note that nulls cannot be bound anywhere.

**Definition 4.3 (Null term \( \mathcal{E} \))**

\[
\begin{align*}
\mathcal{C} & \in dom \mathcal{G} \\
\mathcal{G}, \Gamma \vdash \mathcal{E} : \mathcal{C}
\end{align*}
\]

Null terms may belong to any category in a grammar.
Example 4.2 (Valid and invalid terms)
The following are valid terms in the grammar of Example 4.1: \( \lambda x.\mathcal{E} \), \( \lambda x.\lambda y. y \), \( \mathcal{E} \epsilon \).
These are not valid terms: \( \lambda \epsilon. x \), \( \lambda x.\lambda x. x \), \( \epsilon \mathcal{E} \). \( \square \)

Definition 4.4 (Term Properties)
\( \text{fv } T \), \( \text{bv } T \) and \( \text{var } T \) are the free, bound and all variables of term \( T \).
\( \text{fv } x = \{ x \} \)
\( \text{fv } ( F x_1 \ldots x_n. T_1 \ldots T_n ) = \bigcup_{i=1}^{n} \text{fv } T_i - \{ x_i \}_{i=1}^{n} \)
\( \text{fv } \epsilon = \text{fv } \mathcal{E} = \emptyset \)
\( \text{bv } x = \text{bv } \epsilon = \text{bv } \mathcal{E} = \emptyset \)
\( \text{bv } ( F x_1 \ldots x_n. T_1 \ldots T_n ) = \{ x_i \}_{i=1}^{n} \cup \bigcup_{i=1}^{n} \text{bv } T_i \)
\( \text{var } T = \text{fv } T \cup \text{bv } T \)

Applying a substitution \( \theta = [y_i/x_i]_{i=1}^{n} \) of variables for variables to a term \( T \) is written \( \theta(T) \). The variables in \( \theta \) must be disjoint from \( \text{bv } T \).
\([y_i/x_i]_{i=1}^{n}(x) = y_i \), if \( x = x_i; \) \( x \), otherwise
\( \theta(F x_1 \ldots x_n. T_1 \ldots T_n) = F x_1 \ldots x_n. \theta(T_1) \ldots \theta(T_n) \)
\( \theta(\epsilon) = \epsilon; \theta(\mathcal{E}) = \mathcal{E} \)

\( T \equiv S \) means that terms \( T \) and \( S \) are identical.
\( T = S \) means they are equal up to renaming of bound variables. \( \square \)

Example 4.3 (Term properties)
\( \text{fv } (\lambda x. y) = \text{bv } (\lambda y. x) = \text{var } (\lambda y. x) = \{ y \} \)
\( \lambda x. z \equiv [z/y] (\lambda x. y) \)
\( (\lambda x. \text{let } y = x \text{ in } y) = (\lambda a. \text{let } b = a \text{ in } b) \neq (\lambda x. \text{let } y = x \text{ in } y) \) \( \square \)

4.2.2 Term-graph construction
A rooted term graph \( G \) is a set of nodes followed by a sequence of root variables. Each node \( (x \mapsto T) \in G \) has an address \( x \) and a term \( T \). The free variables of \( T \) are arcs in the graph — pointers to nodes. The roots are there to guide an evaluator or a garbage collector and must be consistent with the \( \text{ROOT} \) category of the grammar.

Definition 4.5 (Rooted graph)
\( \langle c_1, \ldots, c_r \rangle \in \mathcal{G}(\text{ROOT}) \)
\( \{ x_i \mapsto T_i \}_{i=1}^{n} \)
\( \text{dom } \Gamma \downarrow \text{bv } \{ x_i \mapsto T_i \}_{i=1}^{n} \)
\( \exists \mathcal{\Delta} = \{ x_i \mapsto c'_i \}_{i=1}^{n} \cdot v_i = \epsilon \lor (\Delta \cup \Gamma)(v_i) = c_i \land (\mathcal{G}, \Delta \cup \Gamma \vdash T_i : c'_i) \)
\( \mathcal{G}, \Gamma \vdash \text{Graph } \{ x_i \mapsto T_i \}_{i=1}^{n} v_1, \ldots, v_r \)

A graph is a set of nodes and a sequence of root variables. The category of a root variable \( v \) matches the category of the term at address \( v \). The free variables of the terms in the body of the graph are either the addresses of other nodes, in which case they must have the appropriate category, or they are free in the graph and they must have the category defined in the environment \( \Gamma \). Graph nodes all have distinct addresses; bound variables in a term may have the same name as bound variables in other terms; free variables are distinct from all bound variables (\( \not\epsilon \) denotes disjoint sets). \( \square \)

Example 4.4 (Valid and invalid rooted graphs)
Example 4.1 offers plenty of examples of valid graphs. The following are invalid graphs in the grammar of Example 4.1 for the reasons given.
In \( \{a \mapsto \lambda x.a\}a, a \) the second root points to a category \( X \) node.  
In \( \{a \mapsto \lambda x.a, a \mapsto \lambda x.x\}a, \epsilon \) the graph is not a function.  
In \( \{a \mapsto \lambda x.b, s \mapsto y : b\}a, s \) the category of \( b \) cannot be \( X \) and \( S \).

A term graph models the state of a machine as an infinitely extensible set of nodes each of unrestricted size. The definition of a graph evaluator and the restrictions described in Chapter 5 ensure that this powerful model is accurate enough for our purposes.

**Definition 4.6 (Graph properties)**

\[
egin{align*}
fv \left( \left\{ x_i \mapsto T_i \right\}_{i=1}^n \parallel v_1, \ldots, v_r \right) &= \left( \left\{ v_i \right\}_{i=1}^r \cup \bigcup_{i=1}^n fv T_i \right) - \left\{ x_i \right\}_{i=1}^n \vspace{2mm} \\
ibv \left( \left\{ x_i \mapsto T_i \right\}_{i=1}^n \parallel v_1, \ldots, v_r \right) &= \bigcup_{i=1}^n ibv T_i \cup \left\{ x_i \right\}_{i=1}^n \vspace{2mm} \\
wf G &\text{ if } fv G = \{} \end{align*}
\]

The definitions of free and bound variables extend to graphs as above. An extra property for graphs is well-formedness (no free variables). Write \( G \equiv H \) if graphs \( G \) and \( H \) are identical and \( G = H \) if they are identical up to renaming of bound variables. Note that the graph nodes are an unordered set and the graph domain variables are bound (this is standard in the graph literature, e.g. [AA95]), so nodes in equal graphs may be named differently.

Well-formedness is an important property because a closed graph must include all the information needed during evaluation so it helps to substantiate the claim that graphs are an accurate model of space usage.

**Example 4.5 (Graph properties)**

\[
egin{align*}
fv \left( \{a \mapsto \lambda x.y, y \mapsto z\}a, s \} \right) &= \{z, s\} \vspace{2mm} \\
ibv \left( \{a \mapsto \lambda x.y, y \mapsto z\}a, s \} \right) &= \{a, x, y\} \vspace{2mm} \\
\{a \mapsto \lambda x.a\}a, \epsilon &= \{b \mapsto \lambda y.b\}b, \epsilon \vspace{2mm} \\
\{a \mapsto \lambda x.a\}a, s &\neq \{b \mapsto \lambda y.b\}b, t \vspace{2mm} \\
wf \{a \mapsto \lambda x.a\}a, \epsilon \\
\neg wf \{a \mapsto \lambda x.y, y \mapsto z\}a, s \end{align*}
\]

**4.2.3 Garbage collection**

Graphs are rooted so that graph evaluation can work by rewriting the nodes at the root rather than searching the whole graph to find a redex. This means that any nodes not reachable from a root can be safely removed. Definition 4.7 introduces a language-independent garbage collector \( gc \) for this purpose.

This fixed garbage collector makes GraphKit more suitable for modelling the space behaviour of implementations that assume reachability-based garbage collection than implementations which work in a completely different way, or which need a more complicated garbage collector, such as region-based evaluation [TT94].

**Definition 4.7 (Reachable nodes, Garbage collection)**

\[
egin{align*}
reach(G, X) &= fixFrom (\lambda V. V \cup \mathit{fv} G|_V) X \\
gc(Gv_1, \ldots, v_r) &= (G|_{reach(G, \{v_1, \ldots, v_r\})})v_1, \ldots, v_r
\end{align*}
\]

The set \( \text{reach}(G, X) \) contains all the node addresses reachable in graph \( G \) starting from those in the set \( X \). All root-unreachable graph nodes are removed by \( gc \).
Proposition 4.1 (Garbage collection preserves well-formedness)
If $G_{v_1}, \ldots, v_r$ is well-formed and $G' = gc(G_{v_1}, \ldots, v_r)$ then $G'$ is well-formed.
Further, if $G'' \subset G'$ then $G''$ is not well-formed.

Proof
By the definition of $fixFrom$, we reach a set of variables $V$ such that $fv\ G|_V \subseteq V$. Therefore $fv\ G|_V \subseteq fv\ G$ so if $G_{v_1}, \ldots, v_r$ is well-formed then $dom\ G|_V \supseteq \{v_1, \ldots, v_r\} - \{\}$ and $gc(G_{v_1}, \ldots, v_r)$ is well-formed. If $a \in dom\ G' - G''$ then $a \in fv\ G''$ by the definition of $reach$. \hfill \Box

4.3 Graph evaluators

The third part of a GraphKit input defines a graph $evaluator$ as a set of rewrite rules. Evaluators are checked as described in this section (some of the details pertinent to space usage are deferred to Chapter 5). GraphKit displays the checked rules, as in Figure 4.2 for example, then uses them to evaluate the supplied graph. Maximal garbage collection ensures that the evaluation trace reflects the space usage of the graph.

Evaluator rules have a left pattern and a right pattern, where patterns are graphs of term patterns.

4.3.1 Term patterns

Term patterns are terms which may contain holes. A hole $h \in \text{Holes}$ is a special variable which matches an arbitrary term. A variable in a pattern only matches a variable in a term. When GraphKit displays a rewrite rule, the holes are shown in upper-case and the variables in lower-case (see Figure 4.2). The example input in Figure 4.1 also follows this convention (this is not required: whether a symbol is a hole, variable or function symbol is inferred from the grammar). To define patterns rigorously we introduce variable and hole environments.

Definition 4.8 (Variable and hole environments)
An environment $E$ stores information about the variables and holes in a pattern during checking. It helps us to check that a rule cannot generate invalid graphs. Environments contain three kinds of mapping (these could be separated into three environments but it is simpler not to):

1. A bound variable maps to its category. Separate occurrences of the same bound variable must have the same category.

2. A free variable $x$ maps to its category paired with $\beta$, the set of bound variables whose scope includes $x$. Pattern matching can capture a free variable so a rewrite rule must not reduce the set of capture variables if it is to preserve well-formedness. We define the capture variables of $x$ in $E$ as:
   $cv\ x\ E = \bigcup\{dom\ \beta|(x \mapsto (C, \beta)) \in E\}$.

3. A hole maps to a triple containing its category, its capture variables and its substitution. A rewrite-rule right pattern may need to substitute into the term matching a hole to preserve well-formedness (substitutions never occur in rule left patterns). The capture variables of hole $h$ in $E$ are:
   $cv\ h\ E = \bigcup\{dom\ \beta|(h \mapsto (C, \beta, \theta)) \in E\}$.
\[ \bigoplus_n^m E_i = E_n + \cdots + E_m \text{ if } E_i(x) = C \land E_j(x) = C' \Rightarrow C = C' \]
\[ E_i(x) = (C, \beta) \land E_j(x) = (C', \beta') \Rightarrow C = C' \]
\[ E_i(x) = (C, \beta, \theta) \land E_j(x) = (C', \beta', \theta') \Rightarrow C = C' \]

Sequence concatenation is denoted by +. The operator \( \bigoplus \) combines environments. To succeed, separate occurrences of the same variable or hole must have the same category.

**Definition 4.9 (Term pattern)**
Term patterns are defined by judgements of the form \( \mathcal{G}, \beta \vdash \text{Pat } P : C, E \) which says \( P \) is a term pattern in the context of grammar \( \mathcal{G} \) and in the scope of the capture variables in \( \beta \). \( P \) matches terms of category \( C \). The environment \( E \) has an entry for each occurrence of each variable and hole in \( P \).

\[
\frac{h \in \text{Holes} \quad C \in \text{dom } \mathcal{G} \quad D \supseteq \{y_i \neq y_j \mid 1 \leq i < j \leq n\}}{\mathcal{G}, \beta \vdash \text{Pat } h[x_i/y_i]_{i=1}^n || D : C, \langle h \mapsto (C, \beta, [x_i/y_i]_{i=1}^n) \rangle} \tag{4.1}
\]

A hole \( h \) (4.1) is a pattern which matches any term of the right category. Its category is inferred from the surrounding term or graph structure. It may have a suspended substitution of variables for variables \([x_i/y_i]_{i=1}^n\). The variables in the domain of this substitution must be distinct. Terms may have sets of disequations attached to them and two vertical bars to introduce such constraints. The capture variables \( \beta \) and substitution are recorded in the environment to aid further checks.

\[
\frac{(F \ c_1 \cdots c_n, A_1 \cdots A_n) \in \mathcal{G}(C) \quad E_0 = \langle x_i \mapsto c_i \rangle_{i=1}^n \quad B = E_0 + \beta \quad \text{function } B \quad (A_i = VArg \ C') \Rightarrow P_i \in \text{Var} \cup \{\epsilon\} \land E_i = \langle P_i \mapsto (C', B) \rangle | P_i \neq \epsilon \quad (A_i = TArg \ C') \Rightarrow \mathcal{G}, B \vdash P_i || D : C', E_i}{\mathcal{G}, \beta \vdash \text{Pat } (F \ x_1 \cdots x_n, P_1 \cdots P_a) || D : C, \bigoplus_0^n E_i} \tag{4.2}
\]

A constructed pattern (4.2) is built like a term (Definition 4.2). Its bound variables are recorded in environment \( E_0 \). They must be distinctly named from each other and any variables in \( \beta \). Arguments are variables (or null variables \( \epsilon \)) or term patterns as appropriate. Variable arguments are free-variable patterns (not holes), they are added to the environment \( E_i \). Term arguments are constructed term patterns or holes (never variables). The sub-environments combine successfully.

\[
\frac{C \in \text{dom } \mathcal{G}}{\mathcal{G}, \Gamma \vdash \text{Pat } \mathcal{E} : C, \langle \rangle} \tag{4.3}
\]

Finally, the null term is a pattern of every category (4.3).

**Example 4.6 (Term pattern)**
The term pattern \( \lambda y.B \) is checked by the judgement below where \( \mathcal{G}_\lambda \) stands for the grammar of Example 4.1.

\[
\mathcal{G}_\lambda, \langle \rangle \vdash \text{Pat } \lambda y.B : X, \langle y \mapsto X, B \mapsto (X, \{y\}, []) \rangle
\]

The variable \( y \) matches a variable and hole \( B \) matches any category \( X \) term, which may include occurrences of the variable matching \( y \). The pattern \( B[x/y] \) checked as shown below is simply a hole.
\[ G, \{ \} \vdash \text{Pat } B [x/y] : X, \langle B \mapsto (X, \{ \}, [x/y]) \rangle \]

This pattern matches a term in which free occurrences of the variable matching \( y \) are replaced by the variable matching \( x \). Neither example needs any disequality constraints. A pattern that does is \( B[x/y, w/z][y \neq z] \); the pattern \( B[x/y, w/z] \) is not valid because not every instance of its substitution is a function. Note that constraints are only needed at the outermost level of a pattern; the GraphKit input does not specify them, but they are added internally by the checker to make sure that evaluators (and the more complicated things we do with rules in later chapters) do not go wrong. \( \square \)

### 4.3.2 Graph-rewrite rules

Evaluators are sets of graph-rewrite rules. Each rule consists of a left pattern and a right pattern. The construction of rule patterns is explained next.

**Definition 4.10 (Graph pattern)**

\[
\begin{align*}
G, \emptyset \vdash \text{Pat } P_i \mid D : C_i, E_i \quad E_j(x_i) = (C, \beta) \Rightarrow C = C_i & \quad D \supset \{ x_i \neq x_j | 1 \leq i < j \leq n \} \\
G \vdash \text{UnrootedPat } \{ x_i \mapsto P_i \}_{i=1}^{n} \mid D : \bigoplus_{i} E_i & \quad \langle C'_1, \ldots, C'_r \rangle \in G(\text{ROOT}) \quad E_0 = \langle v_i \mapsto (C'_i, \emptyset) | v_i \neq e \rangle_{i=1}^{r} & \quad G \vdash \text{UnrootedPat } G \mid D : E_1 \\
G \vdash \text{GraphPat } v_1, \ldots, v_r \mid D : \bigoplus_{i} E_i & \quad \langle C'_1, \ldots, C'_r \rangle \in G(\text{ROOT}) \quad E_0 = \langle v_i \mapsto (C'_i, \emptyset) | v_i \neq e \rangle_{i=1}^{r} & \quad G \vdash \text{UnrootedPat } G \mid D : E_1
\end{align*}
\]

An unrooted graph pattern is a set of addressed term patterns. The categories of free variables and roots must match the categories of the term patterns they address, this determines the categories of all reachable nodes. Patterns have constraint sets to indicate disequalities amongst their variables introduced by two vertical bars. As well as including the constraints needed by the terms patterns \( P_i \), the constraints in \( D \) guarantee that the nodes in the pattern match distinct graph nodes. Rooted graph patterns have a sequence of \( r \) root variables (which may be null). \( \square \)

**Definition 4.11 (Left pattern)**

\[
G \vdash \text{GraphPat } L : E \quad L = \text{gc}(L) \quad E(h) = (C, \beta, \emptyset) \Rightarrow \emptyset = [] \quad \text{function } E
\]

A left pattern \( L \) is fully connected, none of its holes may have a substitution and it is linear in all variables and holes. \( \square \)

Left-linearity is not necessary for a graph-rewrite system (not even for the accuracy condition discussed in Chapter 5), but it makes evaluation faster and simpler to define. Linearity means that left patterns contain exactly one internal arc to each node.

**Definition 4.12 (Right pattern)**

\[
\begin{align*}
G \vdash \text{GraphPat } R : E' & \quad \text{dom } R \supset \text{dom } L \\
E'(x) = C & \Rightarrow E(x) = C \\
E'(x) = (C, \beta') & \Rightarrow x \in \text{dom } R \lor \exists \beta \cdot E(x) = (C, \beta) \land \beta' \supset \beta \\
E'(h) = (C, \beta', [y_i/x_i]_{i=1}^{n}) & \Rightarrow \exists \beta \cdot E(h) = (C, \beta, []) \\
& \land \{ x_i \}_{i=1}^{n} = \text{dom } (\beta - \beta') \\
& \land \beta(x_i) = C' \Rightarrow \\
& \exists \beta'' \cdot (E + E')(y_i) = (C', \beta'') \land \beta'' \subseteq \beta'
\end{align*}
\]

\[
G, L, E \vdash \text{RhsPat } R
\]
4.3. Graph Evaluators

A right pattern \( R \) is defined with reference to its left pattern \( L \). No nodes are deallocated by \( R \) (Section 5.2 discusses how this is enforced in more detail). Terms in \( R \) may not bind any variable not bound in \( L \). Free variables in its terms are arcs to nodes in \( R \) or they occurred free in \( L \) and their capture variables include at least the capture variables of their occurrence in \( L \).

Holes retain their category. If its capture variables in \( L \) include \( x_i \) but its capture variables in \( R \) do not then \( x_i \) must be substituted for in \( R \) to preserve well-formedness. the replacement is a variable \( y_i \) which cannot itself be freed in the context of the capture variables \( \theta' \).

\[ \]

**Example 4.7 (Left and right patterns)**

The five rewrite rules in Example 4.1 illustrate what left and right patterns may contain. The constraint sets are added internally by GraphKit, so the right pattern of \( \text{(Push)} \) is actually \( \{ a \mapsto F, t \mapsto x : s \} a, t \} \{ a \neq t \} \). The following rule is permissible.

\[ \{ a \mapsto \lambda x.E y \} a, s \rightarrow \{ a \mapsto E[a/x \ a] a \} a, s \]

This rule cannot cause a graph to become ill-formed thanks to the substitution for \( x \) into the hole \( E \). The left and right patterns are checked by the following judgements.

\[ G \vdash LhsPat \{ a \mapsto \lambda x.E y \} a, s : \langle x \mapsto X, a \mapsto (X, \emptyset), s \mapsto (S, \emptyset), y \mapsto (X, \emptyset), E \mapsto (X, \{x\}, []) \} \]

\[ G \vdash GraphPat \{ a \mapsto E[a/x \ a] a \} a, s : \langle a \mapsto (X, \emptyset), s \mapsto (S, \emptyset), E \mapsto (X, \emptyset, [a/x]) \} \]

It follows that \( \{ a \mapsto E[a/x \ a] a \} a, s \) is a right pattern of \( \{ a \mapsto \lambda x.E y \} a, s \).

\[ \]

**Definition 4.13 (Rule)**

\[ \exists E \cdot G \vdash LhsPat L : E \quad G, L, E \vdash RhsPat R \]

\[ G \vdash Rule L \rightarrow R \]

A graph-rewrite rule is a left pattern and a right pattern.

\[ \]

**Definition 4.14 (Evaluator)**

\[ \{ G \vdash Rule r_i \}^n_{i=1} \]

\[ G \vdash Evaluator \{ r_i \}^n_{i=1} \]

A graph evaluator \( \mathcal{A} = \{ r_i \}^n_{i=1} \) is a set of rules.

\[ \]

**Example 4.8 (Evaluators)**

The rules in Example 4.1 define an evaluator \( \text{lazy} \):

\[ \text{lazy} = \{(\text{Push}), (\text{Reduce}), (\text{Lookup}), (\text{Update}), (\text{Let})\} \].

\[ \]

**Definition 4.15 (Graph pattern properties)**

The graph properties — \( fv, bv \), identity and equality — all extend to graph patterns without modification. The variables in pattern constraints are ignored by these functions. But \( var \) is extended to include all variables in a pattern, including those in constraint sets or the domain and range of any hole substitutions. Garbage collection of a graph pattern removes constraints involving variables which are not part of the graph, and the symmetry of \( \neq \) is used to eliminate other redundant constraints. Disequalities between variables of different categories also do not need to be stored explicitly. Substitution into a hole is suspended: \( \theta(h\theta') = h(\theta \cup \theta') \). We define holes \( P \)
to give a sequence of all holes occurring in $P$:
\[ \text{holes } x = \text{holes } \epsilon = \text{holes } E = \emptyset \]
\[ \text{holes } h\theta = \{h\} \]
\[ \text{holes } (F x_1 \cdots x_n T_1 \cdots T_n) = \text{holes } T_1 + \cdots + \text{holes } T_n \]
\[ \text{holes } (\{x_i \mapsto T_i\}_{i=1}^n \nu_1, \ldots, \nu_r \mid D) = \text{holes } T_1 + \cdots + \text{holes } T_n \]

\[ \square \]

**Example 4.9 (Graph pattern properties)**

Substitution into holes works as follows. \[ [d/x] E[b/y] = E[b/y, d/x] \]
To illustrate the other properties, let $G = \{a \mapsto E[b/y], b \mapsto E[b/y]\} a, s$.

\[ \text{fv } G = \{s\} \]
\[ \text{bv } G = \{a, b\} \]
\[ \text{var } G = \{a, b, y, s\} \]
\[ G = \{a \mapsto E[x/y], x \mapsto E[x/y]\} a, s \]
\[ G \neq \{a \mapsto X[x/y], x \mapsto X[x/y]\} a, s \]
\[ \text{holes } G = \{E, E\} \]

\[ \square \]

### 4.3.3 Term and graph contexts

Contexts are a useful notation for describing the operation of evaluators. Putting pattern $P$ in a context $C$ produces a more specific pattern or maybe a graph — a hole-free pattern. Roughly, evaluation rewrites an instance of a left pattern to the same instance of a right pattern. That is, if we can find a context $C$ which maps a rule left pattern to a given graph $G$ then we can rewrite $G$ to $C$ applied to the rule right pattern.

**Definition 4.16 (Term context)**

A term context $\mathbb{T} = (\mathcal{M} \mid D)$ consists of a mapping $\mathcal{M}$ and a disequality constraint set $D$. Application of a term context is defined by the following partial function.

\[ \mathcal{M} \mid D)(T) \mid C = \mathcal{M}(T) \mid D', \text{ if } \wedge D' \text{ where } D' = D \cup \{\mathcal{M}(x) \neq \mathcal{M}(y) \mid (x \neq y) \in C\} \]
\[ \mathcal{M}(x) = y, \text{ if } (x \mapsto y) \in \mathcal{M}; x, \text{ otherwise} \]
\[ \mathcal{M}(h[x_{i1}/y_{i1}]_{i=1}^n) = [\mathcal{M}(x_{i1})/\mathcal{M}(y_{i1})]_{i=1}^n(\mathcal{M}(h)) \]
\[ \mathcal{M}(F x_1 \cdots x_n . P_1 \cdots P_n) = F \mathcal{M}(x_1) \cdots \mathcal{M}(x_n) . \mathcal{M}(P_1) \cdots \mathcal{M}(P_n) \]

We can specialise a pattern by applying a context to it, recursively applying the mapping $\mathcal{M}$ to the pattern and its constraints. The context can only be applied successfully if its constraint set $D$ and the specialised constraints of the pattern are consistent.

\[ \square \]

**Definition 4.17 (Graph context)**

A graph context $G$ is a function which maps a graph pattern to a more specific graph pattern. Its definition is a pair including a term context $\mathbb{T}$ and an extra set of graph nodes $G$ to add to the specialised graph. Application of a graph context $G = (G, \mathbb{T})$ is defined by the following partial function.

\[ (G, (\mathcal{M} \mid D))(\{a_i \mapsto P_i\}_{i=1}^n \nu_1, \ldots, \nu_r \mid C) \]
\[ = (G \cup \{\mathcal{M}(a_i) \mapsto \mathcal{M}(P_i)\}_{i=1}^n \mathcal{M}(\nu_1), \ldots, \mathcal{M}(\nu_r) \mid D' \]
\[ \text{if } \wedge D' \text{ where } D' = D \cup \{\mathcal{M}(x) \neq \mathcal{M}(y) \mid (x \neq y) \in C\} \]

As with term contexts, a graph context is applied by recursively applying its mapping function to all parts of the pattern and checking that the combined constraint set is consistent.

\[ \square \]
Example 4.10 (Contexts)
This example shows how a graph context \( G \) maps \( L \) — the left pattern of \( (\text{Let}) \in \text{lazy} \) — to the second graph in the evaluation trace shown in Figure 4.2.

\[
T = \{ a \mapsto a, y \mapsto \text{false}, E \mapsto \lambda x.\lambda y. y, X \mapsto (\lambda x.\lambda y. \text{false}) b, s \mapsto \epsilon \}\}
\]

\[
G = \{ b \mapsto \lambda x.\lambda y. x \}, T \}
\]

\[
L = \{ a \mapsto \text{let} \ x = \text{E} \ \text{in} \ X \} a, s \}
\]

\[
G(L) = \{ a \mapsto \text{let} \ false = \lambda x.\lambda y. y \ \text{in} \ (\lambda x.x \ \text{false}) b, b \mapsto \lambda x.\lambda y. x \} a, s, \epsilon \}
\]

The disequality constraints imposed on graph patterns mean that the context which maps a pattern to a graph is a graph isomorphism (rather than a homomorphism), i.e. pattern matching is injective. We can define patterns which allow homomorphic matches by omitting some disequality constraints but this is not safe in general. For example, we do not want rules with left pattern \( \{ x \mapsto y, y \mapsto T \} x, z \) to match graphs like \( \{ a \mapsto a \} a, \epsilon \). The problem is that the right pattern might try to update \( a \) with two different terms and so evaluation without the isomorphic matching (enforced by adding the constraint \( \{ x \neq y \} \) to this pattern) could produce invalid graphs.

4.3.4 Evaluation relations

An evaluator defines an evaluation relation, as explained by Definition 4.18.

**Definition 4.18 (Evaluation)**

\[
(L \rightarrow R) \in \mathbb{A} \quad \mathbb{A} : (\text{dom } R - \text{dom } L) \rightarrow (\text{Var} - \text{var } G - \text{var } L) \ \text{injective } \mathbb{A}
\]

\[
G(L) \rightarrow_{\mathbb{A}} G(\mathbb{A}(R))
\]

The evaluation relation defined by \( \mathbb{A} \) rewrites graphs \( G \) which are identical to a left pattern \( L \) in some context \( G, G \equiv G(L) \), to the corresponding right pattern \( R \) in the same context. An extra context \( \mathbb{A} \) — an injective substitution — is applied to the right pattern to allocate any new variables introduced by \( R \). The new variables are distinct from all variables occurring anywhere in \( G \).

\[
\rightarrow_{gc} = \{(G, gc \ H) \mid G \rightarrow H\}
\]

Any evaluation relation \( \rightarrow \) can be extended to include maximal garbage collection, giving the relation \( \rightarrow_{gc} \) defined as above.

Once GraphKit has checked its three input components, the supplied graph is evaluated according to the supplied evaluator. The full evaluation trace can be displayed as shown in Figure 4.2. Maximal garbage collection ensures that the trace is an accurate depiction of space usage.

**Example 4.11 (Evaluation step)**

Now we can complete the second evaluation step in Example 4.1. Example 4.10 found the context \( G \) which maps the left pattern of \( (\text{Let}) \) to the graph. We need another context \( \mathbb{A} = \{ b \mapsto c \} \) to allocate a new node \( c \) in the graph (\( \mathbb{A} \) is a graph context which adds no new graph nodes or constraints, it is shorthand for \( \{(\}, \{(b \mapsto c)\}||\{(\})\)}\)). Applying these contexts to \( R \) — the right pattern of \( (\text{Let}) \) — gives us the third graph in the evaluation trace.

\[
R = \{ a \mapsto X[b/y], b \mapsto E[b/y] \} a, s, \{ a \neq c \}
\]
\( G(\mathbb{A}(R)) = G(\{a \mapsto X[c/y], c \mapsto E[c/y]\} a, s | \{a \neq c\}) = \{a \mapsto (\lambda x.x c b) b, c : \lambda x.\lambda y.y, b : \lambda x.\lambda y.x\} a, \epsilon \)

The garbage collector runs over the graph after the evaluation step. In this case there is nothing for it to remove. \( \square \)

The correctness of GraphKit evaluators is explained by Proposition 4.2: graph evaluation cannot generate invalid graphs and the user-specified grammar is preserved. A corollary is that if the initial graph is well-formed then evaluation also preserves well-formedness.

**Proposition 4.2 (Evaluation preserves well-formedness)**

\( \mathcal{G}, \Gamma \vdash Graph \ G \land G' \vdash \text{Evaluator} \ \mathbb{A} \ \land G \rightarrow A G' \Rightarrow \mathcal{G}, \Gamma \vdash Graph \ G' \)

Therefore if \( G \) is well formed then evaluation preserves well-formedness (replacing \( \Gamma \) with \( \emptyset \) in the formula).

**Proof**

(Outline) \( G' \) is a function. Its roots have the correct categories. Its nodes belong to appropriate categories and interconnecting arcs are self-consistent. The free variables of \( G' \) are a subset of those in \( G \) and they have the correct categories. \( \square \)

It is worth noting that we can apply evaluation to graph patterns as well as graphs. Discussion of issues like complexity are deferred to Chapter 5. The obvious missing element from our definition of evaluation is that we have no concept of initial or final states: when evaluation terminates, GraphKit cannot tell whether the resulting graph represents some sort of value or an error state. We conclude this section with a new example which extends the system defined in Example 4.1.

**Example 4.12 (Boolean constructors and if expressions)**

Suppose we want to write expressions including boolean constructors and if-expressions. The expression category of the grammar, \( X \), is extended as shown below to include \( BOOL \) constructors and a ternary if with an arbitrary expression for its selector argument and its two alternatives. The new category \( BOOL \) defines the constructor constants. Category \( S \) is extended with a new term to hold the alternatives of an if-expression while its selector is evaluated.

\[
  X ::= \ldots \mid BOOL \mid \text{if} \ X \ X' \ X'' \\
  BOOL ::= True \mid False \\
  S ::= \ldots \mid \{X; X'\} \ : \ s
\]

A new evaluator \( \mathbb{H} \) is defined by extending \( \mathbb{A} \) with four new rules. Boolean constructors are values, so we need \( UpdateCtr \) — a new version of \( Update \) — to copy them at the end of a variable lookup. An if-expression is evaluated by storing its alternative expressions in a new stack node and commencing evaluation of its selector by \( PushIf \). When the boolean value of the selector is found, the alternatives are taken from the stack node and the first or second is evaluated as appropriate by \( IfTrue \) or \( IfFalse \). The category system prevents some possible type errors but it is nowhere near as restrictive as a strong type system so it is possible that the subject of an if-expression could evaluate to a \( \lambda \)-expression, causing evaluation to terminate.
$\mathbb{I} = \text{lazy} \cup \{(\text{UpdateCtr}), (\text{PushIf}), (\text{IfTrue}), (\text{IfFalse})\}$

\[
\{a \mapsto B, y \mapsto \bot, s \mapsto \#y : t\} a, s \quad \longrightarrow \quad \{y \mapsto B, a \mapsto B\} y, t \quad \text{(UpdateCtr)}
\]

\[
\{a \mapsto \text{if } E \times Y\} a, s \quad \longrightarrow \quad \{a \mapsto E, t \mapsto \{X; Y\} : s\} a, t \quad \text{(PushIf)}
\]

\[
\{a \mapsto \text{True}, s \mapsto \{X; Y\} : t\} a, s \quad \longrightarrow \quad \{a \mapsto X\} a, t \quad \text{(IfTrue)}
\]

\[
\{a \mapsto \text{False}, s \mapsto \{X; Y\} : t\} a, s \quad \longrightarrow \quad \{a \mapsto Y\} a, t \quad \text{(IfFalse)}
\]

Now the graph from Example 4.1, which encodes a boolean expression as a $\lambda$-calculus term can be written as shown below. The \textit{not} combinator becomes a simple function $\lambda x.\text{if} \ x \ \text{false true}$.

\[
\{a \mapsto \text{let true } = \text{True in let false } = \text{False in } (\lambda x.\text{if} \ x \ \text{false true}) \text{ true}\} a, e
\]

This graph evaluates in 10 steps to give the result $\{a \mapsto \text{False}\} a, e$. It needs two fewer steps than the $\lambda$-calculus version because both the alternatives of the if-expression are pushed in one step and popped in one step.

\[
\square
\]

### 4.4 Input and output

Modelling IO is important for reasoning about complexity and helps to make evaluators more convincing models of real implementations — treating the input as part of the program is an inadequate solution because we could never have a sub-linear space complexity. GraphKit supports IO by allowing rules to have an \textit{effect}: a \textit{put} or \textit{get} action which is executed after the context mapping the rule left pattern to a graph is found and before the graph is rewritten. The effect \textit{put} $C \ h$ writes the category $C$ term matching hole $h$ to the output stream; the effect \textit{get} $C \ h$ attempts to read a category $C$ term from the input stream and binds it to hole $h$.

To ensure that effects only do a bounded amount of work IO is restricted to \textit{base category} terms. This way, the number of function symbols in the input stream is the size of the input and the number of \textit{get} effects in a trace represents the amount of input read; this is more accurate than counting the input size as, say, the length of a list which might contain variable-sized elements.

**Definition 4.19 (Base category)**

$C \in \text{BaseCats } \mathcal{G}$ if $(F \ x_1 \ldots x_n. A_1 \ldots A_n) \in \mathcal{G}(C) \Rightarrow a = b = 0$

$C$ is a base category if all its function symbols bind no variables and have no arguments.

\[
\square
\]

This is very restrictive but it is intuitive and fits well with real programming languages which typically allow only character or byte IO at the lowest level. On a practical note, GraphKit uses the ASCII display macro of a function symbol to display or parse function symbols as they are written or read. For the theoretical treatment rule definitions are extended to include effects.

**Definition 4.20 (Rule with a side effect)**

\[
\begin{align*}
\exists E \cdot \mathcal{G} \vdash \text{LhsPat } L : E \\
C \in \text{BaseCats } \mathcal{G} \quad \exists \beta \cdot E(h) = (C, \beta, []) \\
\mathcal{G}, L, E \vdash \text{RhsPat } R \\
\mathcal{G}, L \vdash \text{Rule } L^{\text{put } C^\mathcal{H} h} \quad \mathcal{G}, L \vdash \text{Rule } L^{\text{get } C^\mathcal{H} h} \quad R
\end{align*}
\]

\[
\begin{align*}
\exists E \cdot \mathcal{G} \vdash \text{LhsPat } L : E \\
C \in \text{BaseCats } \mathcal{G} \quad h \notin \text{dom } E \\
\mathcal{G}, L, E \vdash \langle h \mapsto \langle C, \emptyset, [] \rangle \rangle \vdash \text{RhsPat } R \\
\mathcal{G}, L, E \vdash \langle h \mapsto \langle C, \emptyset, [] \rangle \rangle \vdash \text{RhsPat } R
\end{align*}
\]

\[
\mathcal{G}, L \vdash \text{Rule } L^{\text{put } C^\mathcal{H} h} \quad \mathcal{G}, L \vdash \text{Rule } L^{\text{get } C^\mathcal{H} h} \quad R
\]
A put effect outputs the function symbol bound to the hole \( h \) in the left pattern; a term of the base category \( C \). A get effect reads a function symbol of the base category \( C \) into a new hole \( h \).

Now we define the evaluation of graph \( G \) with an input stream \( I \) and an output stream \( O \). Both streams are sequences of function symbols (a useful extension would be to allow separate streams for each category). We assume that both effects always succeed. If a symbol of the requested category is not available at the start of the input stream then get returns \( E \) by default.

**Definition 4.21 (Evaluation with side effects)**

\[
\begin{align*}
(L \rightarrow R) &\in \mathbb{A} \\
(G(L), I, O) &\rightarrow A (G(A(R)), I, O) \\
(L, h^R_{\text{put}}) &\in \mathbb{A} \quad G = (G, T) \\
(G(L), I, O) &\rightarrow A (G(A(R)), I, O + \langle T(h) \rangle) \\
(L, h^R_{\text{get}}) &\in \mathbb{A} \quad G = (G, T) \quad T \in \mathcal{G}(C) \\
(G(L), \{I\} + I, O) &\rightarrow A (G(\{h \mapsto T\}(A(R))), I, O) \\
(L, h^R_{\text{get}}) &\in \mathbb{A} \quad G = (G, T) \quad I = \{T\} + I' \Rightarrow T \notin \mathcal{G}(C) \\
(G(L), I, O) &\rightarrow A (G(\{h \mapsto E\}(A(R))), I, O)
\end{align*}
\]

These rules modify Definition 4.18 to evaluate a graph tuple with input and output streams which may have side effects. The allocation context \( A \) in all the rules is an injective function \( A : (\text{dom } R - \text{dom } L) \rightarrow (\text{Var} - \text{var } G - \text{var } L) \). For put effects, the term matching hole \( h \) is added to the end of the output stream. For get effects, the first term \( T \) in the input stream is removed. If \( T \) belongs to category \( C \) then \( T \) is substituted for \( h \) in \( R \). If \( T \) belongs to another category or the input is empty then the null term \( E \) is substituted for \( h \) instead.

**Example 4.13 (Call-by-need evaluation with boolean IO)**

This example extends the system defined in Example 4.12 to do IO on boolean constructors. The expression category \( X \) is extended with get and put expressions. The function symbol \( \text{putBool} \) takes an expression as an argument so the stack category \( S \) is extended with a term to store a pending application of \( \text{putBool} \) while its argument is computed.

\[
\begin{align*}
X &::= \cdots \mid \text{getBool} \mid \text{putBool} \ C \\
S &::= \cdots \mid \text{put} : s
\end{align*}
\]

The new evaluator \( \text{eval} \) introduces the following three new rules. \( \text{(GetBool)} \) reads from the input stream, replacing the \( \text{getBool} \) expression with the value read. This means that if the node \( a \) is used twice it has the same value on both occasions; the second use does not cause another value to be read. A \( \text{putBool} \) is evaluated by \( \text{(PushPutBool)} \) which finds the value of its argument \( X \). If this turns out to be a boolean value \( B \) (the BOOL function symbol is invisible in the displayed patterns) it is written out by \( \text{(ReducePutBool)} \) and returned.

\[
\begin{align*}
\text{eval} &\mathrel{=} \text{eval} \cup \{(\text{GetBool}), (\text{PushPutBool}), (\text{ReducePutBool})\} \\
\{a \mapsto \text{getBool}\}a, s &\rightarrow (\text{get BOOL } B)\{a \mapsto B\}a, s \quad (\text{GetBool}) \\
\{a \mapsto \text{putBool} X\}a, s &\rightarrow \{a \mapsto X, t \mapsto \text{put } : s\}a, t \quad (\text{PushPutBool}) \\
\{a \mapsto B, s \mapsto \text{put } : t\}a, s &\rightarrow (\text{put BOOL } B)\{a \mapsto B\}a, t \quad (\text{ReducePutBool})
\end{align*}
\]
An example graph evaluation is summarised below. The function or repeatedly reads boolean values from the input until it gets a True, at which point it outputs True and halts. If the input only contains False values then there is no output. If the input is also finite evaluation will halt when getBool returns \( \varepsilon \); there is no output.

\[
\begin{align*}
( & \{ \text{or} \mapsto \lambda x. \text{if getBool (putBool True) (or } x) \}, \text{main} \mapsto \text{or } \varepsilon \{ \text{main}, \varepsilon \}, \\
& (\text{False}, \text{True}, \text{False}), () \} \\
\end{align*}
\]

Evaluating this graph with the input stream shown: first a False is read, so or calls a new instance of itself; the next value read is True so or outputs a True and returns to main; evaluation terminates with the node main containing True.

\( \checkmark \)

### 4.5 Summary

We have defined the version of term-graph rewriting implemented by the GraphKit tool. Graph grammars permit the definition of higher-order sorted terms; essentially this is very similar to other definitions of equational term-graph rewriting [AK96, Blo01, Has97]. Unlike Blom’s framework [Blo01], the sub-terms in our framework do not have their own addresses and we do not need to define which nodes are in the scope of another node, because our grammars separate terms from variables. Unlike Hasegawa [Has97], we are restricted in that graph nodes, roots, bound variables and term arguments may only belong to one syntactic category. Graphs have the advantage that they can be drawn. Figure 4.3 offers a suitable pictorial notation.

The definition of evaluators and evaluation is quite low-level. Patterns are generalisations of graphs, adding disequality constraints to ensure that the mapping from patterns to graphs cannot go wrong. This mapping is defined as a context, a common idea in operational definitions of reduction [San00]. This produces a more detailed definition than the graph-rewriting literature usually offers.

Finally, we added a stream-based IO model to the graph-rewriting system to enable fuller modelling of real implementations. There are a number of small extensions which might be useful that we have omitted for simplicity: precedence and fixity definitions in the grammars, the addition of a type system, definitions of valid initial and final states and the option to specify an alternative garbage collector. Some of these issues are returned to later on (space relations for initial states in Section 9.1 and as garbage collectors in Section 8.2). Another useful extension would be to allow meta graph-rewrite rules. For example, consider the Haskell let-expression. Currently, a separate rule is needed for each possible number of bindings. With a suitable meta-rule notation and a family of let-function symbols, or perhaps a variadic let, we could specify a rewrite rule to handle let with any number of bindings.
Figure 4.3: Pictorial representation of the evaluation trace in Figure 4.2, showing (top to bottom) the initial graph, the graph after 6 steps and the final graph. Nodes are drawn as boxes containing terms, rather than splitting terms into trees of function symbols. Node position indicates address and arcs are drawn. This representation clearly shows how the graph changes during evaluation and works well for identifying which nodes are allocated, updated or deallocated.
Chapter 5

Space Usage in GraphKit

GraphKit is designed to support specification of, and reasoning about, the space usage of graph evaluators. This chapter explores that claim by looking at some relevant properties of GraphKit evaluators.

Section 5.1 defines a resource-usage model for GraphKit evaluators and shows that with an accuracy condition we can use this model to reason about the space and time usage of the implementation modelled by an evaluator. Section 5.2 investigates the extent to which GraphKit evaluators need to rely on a garbage collector. Section 5.3 shows how GraphKit checks that evaluators are deterministic. This requires a unification algorithm which finds various other uses in later chapters.

5.1 Measuring space and time usage

A GraphKit evaluator (plus the garbage collector) is such an accurate model of space and time usage that we can use it as a basis for comparing different implementation strategies. This section examines this claim, defining our space-usage model and showing that the space usage of a literal interpretation of a GraphKit evaluator is within a constant factor of the model. Thus the space model can be used for arguing about the asymptotic space behaviour of the implementations represented by an evaluator.

5.1.1 Size, space, time and complexity

**Definition 5.1 (Size)**

\[
\begin{align*}
\text{size } x &= \text{size } e = \text{size } \mathcal{E} = 1 \\
\text{size } h &\theta = 0 \\
\text{size } (F \, x_1 \cdots x_n, T_1 \cdots T_n) &= 1 + b + \sum_{i=1}^a \text{size } T_i \\
\text{size } (\{ x_i = T_i \}_{i=1}^n) v_1, \ldots, v_r &= \sum_{i=1}^n \text{size } (T_i)
\end{align*}
\]

These equations define the size of terms, graphs and patterns. Holes have size 0 so that \( \text{size } \{ h \mapsto T \}(h) = \text{size } h + \text{size } T \). \( \Box \)

To measure the storage used by a graph, Definition 5.1 gives the size of terms and graphs in the obvious way. This definition assumes that the grammar is finite (so the function symbol size is limited) and that we have an infinite set of variables which each occupy a unit of space. Well-formedness is also relevant — we assume that the grammar, graph and evaluator contain all the information needed during evaluation.
So size is reasonably accurate and it is language-independent which is important for making a fair comparison of different languages.

**Definition 5.2 (Evaluation space and time usage)**

- space \( G \square \) = nodeUsage \( G \square \) = \( \max \{ \#G' | G \xrightarrow{\varphi}^* G' \} \)
- sizeUsage \( G \square \) = \( \max \{ \text{size} G' | G \xrightarrow{\varphi}^* G' \} \)
- time \( G \square \) = \( \begin{cases} \infty, & \text{if } \forall G' \cdot G \xrightarrow{\varphi}^* G' \Rightarrow \exists G'' \cdot G' \xrightarrow{\varphi} G'' \\ \max \{ n | G \xrightarrow{\varphi}^* G' \not\xrightarrow{\varphi} G' \}, & \text{otherwise} \end{cases} \)

The space needed to evaluate a graph is its node usage: the maximum number of nodes of any graph in its evaluation trace (# denotes set cardinality). As an alternative, a more accurate measure of space is given by measuring size usage. The time usage is the number of steps in its evaluation trace. These measures can be infinite, so their range is \( \mathbb{N} \cup \{ \infty \} \).

The amount of space and time needed to evaluate a graph are defined by the simple functions in Definition 5.2. We do not count the space needed for the graph evaluation machinery and the rewrite rules; once the language and evaluator have been specified these parts of the system do not need any additional space.

A garbage collector will need some auxiliary space, but any reasonable collector will not increase space requirements by more than some pre-determined constant factor. We also ignore garbage collection can also be ignored as far as time complexity is concerned because the complexity of most garbage collectors is bounded by graph size and — assuming there is no thrashing — most of the graph should have been allocated since the previous collection so garbage collection should only increase time by some constant factor. In practice GraphKit runs the collector after every evaluation step (or nearly, see Section 5.2) which makes its time complexity depend on graph size.

**Example 5.1 (Space and time usage)**

As Figure 4.2 shows, GraphKit displays the space and time usage of a graph if its evaluation terminates. In that example the node usage was 6, the maximum graph size was 26 and the evaluation took 12 steps.

The problem with all this is that space is not always accurate enough so we need to restrict our general graph rewriting system to make both space and time sufficiently accurate. Another issue is whether nodeUsage or sizeUsage is more accurate. For a literal interpretation of graph rewriting, it is tempting to just measure size usage, then it would obviously be true that the implementation is within a constant factor of the model. The more abstract nodeUsage measure is more appealing for reasoning. Implementations of functional languages based on environments are somewhere between the two measures: the room needed by a node in the environment model is determined by the number of its free variables in the graph model. So if we can guarantee that the nodeUsage measure is accurate enough then we can claim to be modelling not only literal graph rewriting interpretations but also those which have a more compact representation of graph nodes.

To support the time measure — to see how accurate graph rewriting is as a model of execution time — we look at the complexity of a rewrite step. Proposition 5.1 highlights the tradeoff between language complexity and time complexity: we can make a step faster by having a simpler grammar and less rewrite rules, but unless there is
some redundancy we will need more steps. The constant factors in Proposition 5.1 are determined when the grammar and evaluator are specified. So time should be accurate enough for comparing the asymptotic behaviour of different programs or evaluators.

**Proposition 5.1 (Complexity of a Rewrite Step)**

If $\mathcal{A} = \{L_i \rightarrow R_i\}_{i=1}^n$ then $G \rightarrow_A H$ is $O(t \times a \times f \times \sum_{i=1}^{n}(\text{size}(L_i R_i)))$

where $t$ is the maximum term size in $G$, $a$ is the size of an address and $f$ is the size of a function symbol.

**Proof** (outline). This result is based on a simple implementation of graph rewriting. We try each rule in turn. Attempting to match a left pattern $L_i$ requires time proportional to its size. Because it is fully connected and linear, no search in the graph is needed and no comparison of the terms which match holes in the pattern are needed, so matching does not depend on graph size. The amount of work needed to compare function symbols depends on their size, $f$, which depends on the size of the grammar. The amount of work needed to manipulate graph arcs also depends on their size, $a$, which in the most flexible implementation will depend on the logarithm of the space available for the graph. In this sense rewriting does depend on graph size to a small extent. However, the limitations of computer implementations mean that $a$ is always a predetermined constant. Once a match is found for $L_i$, the rewrite time depends on the size of $R_i$ and the size of the largest term matching a hole in $R_i$, hence the $t$ term in the equation. Allocation is a constant-time operation. □

If time and space are accurate to within a constant factor then they are adequate for reasoning about asymptotic behaviour. So we want to define asymptotic complexity classes such that a graph has order $f$ space usage if its space is bounded by $f(x)$. The question is what is $x$? It cannot be initial graph size — we could never have a sub-linear space complexity. Instead, Definition 5.3 uses the IO model to specify asymptotic complexity; $x$ is the length of the input — the number of function symbols in the input stream. Since IO is only permitted on base categories, any structured input must be broken down so this is a fair measure of the size of the input file.

**Definition 5.3 (Asymptotic space usage)**

space extends to graphs with input and output streams as follows:

space $(G, I, O) \mathcal{A} = \max \{ \#G', (G, I, O) \rightarrow^*_{A'} (G', I', O') \}$

The space usage of $G$ is $O(f)$ if $\exists k \cdot \forall I \cdot \text{space } (G, I, \langle \rangle) \mathcal{A} \leq k \times f(\#I + 1)$ □

### 5.1.2 Accurate evaluators

Unfortunately, space and time are not accurate measures in general. The problem is that evaluation can cause the size of a graph node to grow, so there is no limit on the size of a term matching a hole in a rewrite rule. Copying the term matching a hole into a new node, or substituting into it, could therefore require more and more work as evaluation proceeds. It follows that there is no upper bound on the maximum node size during the evaluation of a graph. A further concern is that failing to limit the work at each step could invalidate sizeUsage because allowing arbitrarily complex steps might artificially lower the size needed to store a graph, though it seems that this cannot happen with our definition of graph rewriting. To make our measures valid we define accurate evaluators.
Definition 5.4 (Accurate evaluator)

accurate $\mathbf{A}$ if $\exists k \in \mathbb{N} \cdot \forall G$, sizeUsage $G \mathbf{A} \leq k \times \text{size} G \times \text{space} G \mathbf{A}$.

uniformAccurate $\mathbf{A}$ if $\exists k \in \mathbb{N} \cdot \forall G$, sizeUsage $G \mathbf{A} \leq k \times \text{space} G \mathbf{A}$.

Evaluator $\mathbf{A}$ is accurate if its node usage (space) bounds its size usage: there is some constant $k$ which, when multiplied by the initial graph size, bounds the size of all graph nodes in the evaluation trace. If we do not need to consider the initial graph size in this equation then we say $\mathbf{A}$ is uniformly accurate. □

By making sure that all evaluation traces have a maximum node size in this way, we ensure that space and time are accurate to within a constant factor depending on the program, language and evaluator.

Proposition 5.2 (Accurate evaluators are accurate time models)

For any evaluator $\mathbf{A} = \{I_i \rightarrow R_i\}_{i=1}^n$, let $k = a \times f \times \Sigma_{i=1}^n(\text{size}(L_i R_i))$ where $a$ and $f$ are defined as in Proposition 5.1.

If $\mathbf{A}$ is accurate then $\forall G \exists \cdot G \rightarrow^*_{\mathbf{A}} G' \rightarrow_{\mathbf{A}} G'' \Rightarrow G' \rightarrow_{\mathbf{A}} G''$ is $O(t \times k)$.

If $\mathbf{A}$ is uniformly accurate $\forall G \cdot G \rightarrow^*_{\mathbf{A}} G' \rightarrow_{\mathbf{A}} G'' \Rightarrow G' \rightarrow_{\mathbf{A}} G''$ is $O(k)$.

Proof

For every graph $G$ there is a constant $t \in \mathbb{N}$ which is greater than the size of any graph node in the evaluation trace of $G$ by definition 5.4. Therefore the complexity of each evaluation step will be $O(t \times k)$, by Proposition 5.1. For uniformly accurate evaluators $t$ is a graph-independent constant so we omit it from the calculation. □

So by using accurate evaluators and the space measure, we can be sure that the space complexity of a graph using the space measure is its space complexity on a real implementation. The actual space usage of the implementation lies somewhere between sizeUsage and nodeUsage. We return to the relationship between sizeUsage and nodeUsage in Section 7.2. Unless stated otherwise, all our example evaluators are accurate.

Example 5.2 (Inaccurate evaluator)

Suppose we redefine the lazy system from Example 4.1 with the stack component rewritten to build up a stack in a single graph node instead of a chain of nodes (an advantage is that we can be sure that the stack will not be a cycle if we make this change). So category $S$ is redefined $S ::= x : S | \#x S$ and we define the evaluator stack as follows.

$$\text{stack} = \{(S\text{Push}), (S\text{Reduce}), (S\text{Lookup}), (S\text{Update}), (\text{Let})\}$$

$$\begin{align*}
\{a \mapsto F \text{ } x \text{ }, s \mapsto S\}a, s &\rightarrow \{a \mapsto F, s \mapsto x : S\}a, s \quad (S\text{Push}) \\
\{a \mapsto \lambda y. E \text{ }, s \mapsto x : S\}a, s &\rightarrow \{a \mapsto E[x/y], s \mapsto S\}a, s \quad (S\text{Reduce}) \\
\{a \mapsto x \text{ }, s \mapsto S\}a, s &\rightarrow \{a \mapsto \perp, s \mapsto \#a S\}x, s \quad (S\text{Lookup}) \\
\{a \mapsto \lambda x. E \text{ }, y \mapsto \perp, s \mapsto \#y S\}a, s &\rightarrow \{y \mapsto \lambda x. E, a \mapsto \lambda x. E, s \mapsto S\}y, s (S\text{Update})
\end{align*}$$

Now, deliberately, there is no bound on the size of the stack term. In effect we have rewritten the graph-rewriting model of a stack as a term-rewriting model of a stack. Unfortunately as the stack gets bigger it will take more and more work to update the $S$-holes in the right patterns of the rules; stack is not accurate. □
Accuracy is probably undecidable in general so GraphKit uses the simple sufficient condition defined by Proposition 5.3. This check is easily verified by the user. A kind of linearity in right patterns is demanded: all the holes in any right-pattern node must occur in some node of the left-pattern; the size of that right-pattern node cannot be greater than the size of the corresponding left-pattern node to ensure that no node grows.

Proposition 5.3 (Accuracy check)
If for all rules $(L \rightarrow R) = (\{(x_i \mapsto P_i)_{i=1}^{n}\}y \rightarrow (\{y_i \mapsto Q_i\}_{i=1}^{m})w) \in \mathbb{A}$, $1 \leq i \leq m \Rightarrow (\exists P_j \cdot \text{holes } Q_i \subseteq \text{holes } P_j \wedge \text{size } Q_i \leq \text{size } P_j)$ or holes $Q_i = \emptyset$ then $\mathbb{A}$ is accurate.

Proof
The key observation is that $\text{size } \mathbb{T}(P) = \text{size } P + \sum \{\text{size } \mathbb{T}(h) | h \in \text{holes } P\}$.

Therefore, if $G \rightarrow_{A} H$ and $\mathbb{A}$ satisfies the accuracy check then the maximum node size in $H$ is no larger than the maximum node size in $G$, or the maximum hole-free pattern in any right pattern of $\mathbb{A}$.

It follows by induction on the derivation sequence that the maximum node size in any graph generated during the evaluation of $G$ is less than the maximum node size in $G$ or $\max \{\text{size } Q | (L \rightarrow R) \in \mathbb{A}, (a \mapsto Q) \in R, \text{holes } Q = \emptyset\}$, whichever is greater.

\[ \square \]

Sometimes an obviously accurate evaluator fails the check which requires a rule to be re-formulated, typically using more graph nodes, to get round the limitations of the check.

Example 5.3 (Uncheckably accurate)
We can define call-by-value evaluation as follows. Change the expression grammar of Example 4.1 to $X ::= \lambda x.X \mid X \cdot X' \mid x \mid \bot$, extend the stack category according to $S ::= \cdots \mid X; s$. The evaluation of an application first stores the argument and evaluates the function $(\text{EvalFun})$, then $(\text{EvalArg})$ evaluates the argument, storing the function on the stack meanwhile. When the argument has evaluated the function body is specialised and evaluated by $(\text{EvalBody})$.

\[ \text{cbv} = \{(\text{EvalFun}), (\text{EvalArg}), (\text{EvalBody}), (\text{Lookup}), (\text{Update})\} \]

\[ \begin{align*}
\text{a } &\mapsto \text{F } X \text{a, s } \rightarrow \text{ a } \mapsto \text{F, b } \mapsto \text{X, t } \mapsto \text{b } \mapsto \text{s } \text{a, t } \quad \text{(EvalFun)} \\
\text{a } &\mapsto \text{\lambda x.E, s } \mapsto \text{y } : \text{t } \text{a, s } \rightarrow \text{ a } \mapsto \text{y, s } \mapsto \text{\lambda x.E; t } \text{y, s } \quad \text{(EvalArg)} \\
\text{a } &\mapsto \text{\lambda x.E, s } \mapsto \text{\lambda y.B; t } \text{a, s } \rightarrow \text{ a } \mapsto \text{B[b/y], b } \mapsto \text{\lambda x.E; a, t } \quad \text{(EvalBody)}
\end{align*} \]

Clearly $\text{cbv}$ is accurate. The stack node $s$ modified by $(\text{EvalArg})$ can never have a size greater than the largest $\lambda$-value node in the graph plus two. However, it is larger than node $a$ in the left pattern, so it fails the accuracy check. To model this evaluation strategy we need to change the new stack term to hold the address of a function while its argument evaluates: $S ::= \cdots \mid x; s$ and reformulate the rules as shown below.

\[ \begin{align*}
\text{a } &\mapsto \text{\lambda x.E, s } \mapsto \text{y } : \text{t } \text{a, s } \rightarrow \text{ a } \mapsto \text{\lambda x.E, s } \mapsto \text{a; t } \text{y, s } \quad \text{(EvalArg)} \\
\text{a } &\mapsto \text{\lambda x.E, f } \mapsto \text{\lambda y.B, s } \mapsto \text{f; t } \text{a, s } \rightarrow \text{ a } \mapsto \text{\lambda x.E, f } \mapsto \text{B[a/y]; f, t } \quad \text{(EvalBody)}
\end{align*} \]

\[ \square \]

IO further complicates accuracy. Holes introduced by get effects do not appear in the left pattern. However, we only allow IO on base-category terms, so we simply count their size as one in the right patterns and remove them from the list of holes in each right pattern for the purpose of the accuracy check.
5.2 Deallociation

In principle, an evaluator rule replaces the sub-graph matching its left pattern with a
sub-graph matching it right pattern. This means that any nodes in the left pattern
but not the right are explicitly deallocated. The evaluator right-pattern definition
(Definition 4.12) insists that all nodes in the left pattern are also in the right pattern,
to guarantee preservation of well-formedness. GraphKit enforces this condition by
applying the extend rule (5.1) so that unchanged nodes do not need to be mentioned
in right patterns. For example, the definition of node $a$ is omitted from the right
pattern of the definition of (Update) in Figure 4.1. Hereafter we omit such nodes to
simplify evaluator definitions.

\[
\text{extend}(Lu_1, \ldots, u_n \rightarrow Rv_1, \ldots, v_n) = (Lu_1, \ldots, u_n \rightarrow (R \cup E_{\text{dom } E}v_1, \ldots, v_n) \quad (5.1)
\]

Relying on a garbage collector for all deallocation is not desirable. Practically, it makes a GraphKit evaluation rather slow because it uses maximal garbage collection. Theoretically, using a garbage collector does not damage the validity of the time or space measures, but the extent to which garbage collection is relied on is some indication of the unpredictability of the space usage of an evaluator.

5.2.1 Garbage-generating rules

Some evaluation rules do not produce any new garbage because everything reachable
from the left pattern always remains reachable in the right pattern. GraphKit labels
a rule as garbage generating if there could be some new garbage to collect after it is applied.

**Definition 5.5 (Garbage-generating rule)**

\[
ggen(L \rightarrow R) = \exists G \cdot G = gc G \land G \rightarrow G' \land G' \neq gc G'
\]

A rule is garbage generating if applying it to a garbage-collected graph can result in a
graph which contains garbage.

Evaluation in GraphKit uses the relation $\overline{\text{gc}} \rightarrow$ which we now redefine as follows.

\[
\overline{\text{gc}} \rightarrow A = \rightarrow A \cup \{ (G, gc G') | G \rightarrow A' G' \}
\]

where $\overline{\text{A}} = \{ r \in A | \overline{\text{ggen}}(r) \} \land \overline{\text{A'}} = \{ r \in A' | \text{ggen}(r) \} \quad (5.2)

Testing whether a rule is garbage generating is more difficult than it first appears.
GraphKit uses the simple check described in Proposition 5.4 which tests whether all
free variables and holes in a left pattern remain reachable in the right pattern.

**Proposition 5.4 (Garbage-generating rule check)**

If $\text{ggen}(Lu_1, \ldots, u_n \rightarrow Rv_1, \ldots, v_n)$ then:
1. $\exists x \cdot x \in \text{fv} Lu_1, \ldots, u_n \rightarrow \text{fv} (gc Rv_1, \ldots, v_n)$ or
2. $\exists x \cdot x \in \text{dom} L \rightarrow (\text{dom} L - \text{dom} R) \rightarrow \text{dom} (gc Rv_1, \ldots, v_n)$ or
3. $\exists h \cdot h \in \text{holes} L \land h \notin \text{holes} (gc Rv_1, \ldots, v_n)$.

**Proof**

1. If $x$ is free (and therefore reachable) in $L$ but not reachable in $R$ then it could be
the address of a node which is garbage after the rewrite. Note that $\text{fv} h \theta = \emptyset$ so nodes
in \( R \) only reachable via a substitution are not guaranteed to be reachable in all cases (an example is the node allocated by (\textit{Let}) in lazy).

2. An address in \( L \) which is not explicitly deallocated and might be unreachable in \( R \) could be garbage.

3. If a hole \( h \) in \( L \) is unreachable in \( R \) then any free variables in the term matching \( h \) could be garbage after the rewrite. If \( h \) belongs to a base category, or a category whose terms may have no free variables, then the rule is not really garbage-generating, so this check is only an approximation. \( \square \)

\textbf{Example 5.4 (Garbage-generating rules in lazy)}

In lazy (Example 4.1), (\textit{Push}) and (\textit{Lookup}) are not garbage-generating; all holes and free variables in their left patterns are reachable in their right patterns. (\textit{Let}) can be garbage-generating: the binding it allocates might not be reachable from its body: for example, in \( G\{ a \mapsto \text{let } y = z \text{ in } \lambda x.x \} \), \( s \) not only is the new node allocated by (\textit{Let}) going to be garbage after the next evaluation step, but anything in \( G \) reachable from \( z \) could become garbage too. Fortunately, a simple static analysis can eliminate let-expressions that will behave in this way. (\textit{Reduce}) always makes the top stack node garbage. If the function body does not mention its parameter then anything reachable from its argument might become garbage. (\textit{Update}) also makes the top stack node garbage. If there are no further references to the updated address \( a \) then node \( a \) will be garbage too. Nothing elsewhere in the graph can become garbage, so this rule can only generate garbage locally, but finding out whether node \( a \) is garbage still requires a global search of the graph unless a reference-counting scheme is used. Another approach to this problem is to use a static update analysis to avoid unnecessary update steps [Gus98]. \( \square \)

\section{5.2.2 Evaluator trimming}

We have not yet ruled out the possibility that an evaluator rule could allocate garbage. GraphKit applies the guaranteed-garbage elimination rule (5.3) to remove any such nodes. Note that (5.3) cannot use the usual definition of \textit{reach} (Definition 4.7) to find the possibly-reachable variables because it needs to keep any nodes which are reachable only via a substitution. The variable naming restrictions on patterns ensure that variables bound in the reachable part of \( R \) cannot include any variables in the domain of the non-reachable part of \( R \).

\[
(L \longrightarrow Rv_1, \ldots, v_n) = (L \longrightarrow (R|_{\text{from } (\lambda \nu \text{ var } R \{ v \}) \{ v_1, \ldots, v_n \}}) v_1, \ldots, v_n) \quad (5.3)
\]

Another practical improvement included in GraphKit is evaluator trimming — removing rules that cannot be needed to evaluate the initial graph supplied in the input. Trimming improves the evaluation speed in cases where the evaluator has a lot of rules which are not needed. Trimming also encourages the user to re-use and extend existing evaluators.

\textbf{Definition 5.6 (Evaluator trimming)}

Trimming is another approximation. We define symbols in the obvious way, to find the set of function symbols that occur in a graph or a pattern. A rewrite rule \( L \longrightarrow R \) is needed only if symbols \( L \) is a subset of symbols \( G \) where \( G \) is the initial graph, or if
symbols \( L \) is a subset of the symbols introduced by other rules that might have been used. The application \( \text{trim}(A, G) \) finds all such rules.

\[
\text{trim}(A, G) = \{(L \rightarrow R) \in A \mid \text{symbols } L \subseteq S\text{reach}\}
\]

where

\[
S\text{reach} = \text{fix} (\lambda S. \text{symbols } G \cup S \cup \bigcup \{\text{symbols } R \mid (L \rightarrow R) \in A, \text{symbols } L \subseteq S\})
\]

Where a rule has a get effect, \( L \xrightarrow{\text{get } C_{j_k}} R \), it is also necessary to include all the function symbols in category \( C \) of the grammar in symbols \( L \). \( \square \)

**Proposition 5.5 (Evaluator trimming is safe)**

\( G \rightarrow^n_A H \Rightarrow G \rightarrow^n_I H \) where \( I = \text{trim}(A, G) \)

**Proof**

By induction on the derivation sequence. \( \square \)

**Example 5.5 (Trimming)**

If we use \( \text{hlo} \) to evaluate the graph in Example 4.1 then all the extra rules involving boolean expressions and IO are removed, reducing it to the original definition of \( \text{lazy} \).

In the example \( \text{hlo} \) evaluation trace summarised in Example 4.13, trimming removes the \( \text{(Let)} \) rule from \( \text{hlo} \) because the initial graph contains no let-expressions and none of the rules introduce them. Of the remaining rules, all are used during evaluation except for \( \text{(UpdateCtr)} \) which is defined in Example 4.12.Trimming cannot detect that this rule is not needed because all the symbols in its left pattern can and do occur during evaluation. \( \square \)

Evaluator trimming is also one approach to solving the problem of how to represent meta-rules. It is common to write rules like this in operational specifications (e.g. Sestoft’s \( \text{(let)} \) in Figure 3.3):

\[
\{a \mapsto \text{let } y_1 = E_1, \ldots, y_n = E_n \text{ in } X\} a, s \rightarrow \{a \mapsto X[b_i/y_i]_{i=1}^n, b_i \mapsto E_i[b_i/y_i]_{i=1}^n\} a, s
\]

GraphKit cannot directly represent such a rule. Rather, a family of rules must be defined, which give a finite approximation to the infinite number of rules defined by the meta-rule. With trimming in place we can define the family of rules and any that cannot be needed are removed. Further, providing that the index \( i \) is not modified in the meta-rule right pattern, we can be sure that the trimmed evaluator will always be finite because the new indexed function symbols introduced by the \( i \)th rule must also be indexed by \( i \).

### 5.2.3 Stack categories

Sometimes we know that applying a rule will make garbage of certain nodes in its left pattern. If GraphKit can prove this then it may do safe explicit deallocation, making rules less likely to be garbage-generating and thus reducing the garbage collection overhead.

Our example evaluators model stacks as chains of nodes. With this in mind, GraphKit includes an analysis to detect when a structure is being used in a stack-like manner to enable explicit deallocation.
Definition 5.7 (Stack category)

S is a stack category of grammar \( \mathcal{G} \): \( S \in \text{StackCat} \mathcal{G} \), if: only one root variable points to S-nodes; every function symbol \( (F \, x_1 \cdots x_0, A_1 \cdots A_a) \in S \) refers to at most one S-node: \( \forall i \cdot A_i = \text{Var} S \land j \neq i \Rightarrow A_j \neq \text{Var} S \) and \( \forall i \cdot A_j \neq \text{Tag} S \); and no other function symbols have category S arguments.

If S is a stack category then S-nodes form a chain. They could be a cycle or become a cycle after an evaluation step. If we can prove this will not happen then evaluator rules can explicitly deallocate S-nodes which are locally garbage (Proposition 5.6). Like trimming, this analysis needs to know about the initial graph.

Proposition 5.6 (Stack node deallocation)

Let \( \text{vrg} \ G_{v_1}, \ldots, v_n \) be the sequence of variables occurring in G and \( \{v_1, \ldots, v_n\} \), excluding the domain of G.

If: S is a stack category; in the initial graph G, \( \text{vrg} \ G \) is linear in S-variables; and in the evaluator \( \{L_i \rightarrow R_i\}_{i=1}^{n} \), each \( \text{vrg} \ (\text{gc} R_i) \) is linear in S-variables and S-holes. Then any S-node not in \( \text{gc}(R_i) \) may be removed in the evaluation of \( G_{v_1}, \ldots, v_n \).

Proof

If \( G \rightarrow H \) then \( \text{vrg} \ (\text{gc} H) \) is linear in S-variables. The sub-graph updated by the rule has a locally reachable part (the gc \( R_i \) above), which must still be linear in S-node references. It may also have a locally unreachble part; any S-nodes here are no longer reachable because the only references to them in G were in the sub-graph updated by the rule. The remaining graph is linear in S-node references. By induction on the derivation sequence the graph will remain linear in S-node references.

Example 5.6 (Stack node deallocation in \textit{lazy} )

In Figure 4.2, GraphKit has recognised that S is being used as a stack category and explicitly removed nodes s from the right patterns of (Reduce) and (Update). If instead we define the initial graph as \( \{a \mapsto \lambda x.y, s \mapsto \epsilon : s\}a, s \) then the evaluator \textit{lazy} is trimmed as shown below. The S-nodes no longer form a stack so (Reduce) and (Update) cannot remove s anymore. The evaluation trace of this unusual graph shows the same argument being substituted for both \( \lambda \)-bound variables \( x \) and \( y \) then a variable lookup step before evaluation terminates.

\[
\begin{align*}
\{a \mapsto \lambda y.E, s \mapsto x : t\}a, s & \rightarrow \{a \mapsto E[x/y], s \mapsto x : t\}a, t \quad \text{(Reduce)} \\
\{a \mapsto x\}a, s & \rightarrow \{a \mapsto \perp, t \mapsto \#a\ s\}x, t \quad \text{(Lookup)} \\
\{a \mapsto \lambda x.E, y \mapsto \perp, s \mapsto \#y\ t\}a, s & \rightarrow \{y \mapsto \lambda x.E, s \mapsto \#y\ t\}y, t \quad \text{(Update)}
\end{align*}
\]

\[
\begin{align*}
\{a \mapsto \lambda x.y, s \mapsto \epsilon : s\}a, s \\
\frac{L}{\rightarrow} \{a \mapsto \lambda y.y, s \mapsto \epsilon : s\}a, s \quad \text{(Reduce)} \\
\frac{L}{\rightarrow} \{a \mapsto \epsilon, s \mapsto \epsilon : s\}a, s \quad \text{(Reduce)} \\
\frac{L}{\rightarrow} \{a \mapsto \perp, b \mapsto \#a\ s, s \mapsto \epsilon : s\}\epsilon, b \quad \text{(Lookup)}
\end{align*}
\]

space = 3, time = 3

\[
\frac{L}{\rightarrow} \alpha
\]

\[
\frac{L}{\rightarrow} \alpha
\]

5.3 Determinism

GraphKit is designed for reasoning about the space behaviour of deterministic evaluators. Determinism is not strictly necessary for our definitions of time or space but
a non-deterministic evaluator specification is incomplete in that it does not describe exactly how evaluation should proceed.

Non-determinism is a useful way to describe the semantics of parallel evaluation \cite{BFKT00}, or evaluation with backtracking. This thesis does not address the space usage of parallel systems. Allowing backtracking would affect space so we cannot permit its inclusion without an explicit control strategy. Therefore GraphKit evaluators must satisfy definition 5.8.

**Definition 5.8 (Deterministic evaluator)**

\[ \mathcal{A} \] is deterministic if only one rule can match any graph:
\[
deterministic \mathcal{A} \text{ if } (L \rightarrow R) \in \mathcal{A} \land (L' \rightarrow R') \in \mathcal{A} \land \\
\exists G, H \cdot G(L) \equiv H(L') \Rightarrow (L \rightarrow R) \equiv (L' \rightarrow R'). \quad \square
\]

An alternative to determinism would be to treat graph evaluators as prioritised rewrite systems where the first rule to match is the one that gets used. That might reduce the number of rules needed sometimes but the implementations we are trying to model do not work like this. It is more efficient in practice to use the graph structure at the root to determine which rule to apply than to run through all rules looking for a match. Therefore determinism re-enforces the argument that evaluators are accurate models of implementations. It also means that space and time are functions rather than upper bounds on the resource usage of all possible evaluation traces of a graph. To check determinism we need a graph-pattern unification algorithm (which also finds many uses in subsequent chapters).

### 5.3.1 Graph left-pattern unification

To test for determinism we only need to unify graph left patterns. This simplifies matters because unlike right patterns, left patterns do not contain hole substitutions. Unifying arbitrary graph patterns (which we return to in Section 12.2) involves solving equations like \( h[x/y] = h'[a/b] \). For now we specify a version of unification which is sufficient for deciding determinism.

**Definition 5.9 (Graph pattern unifier)**

Graph context \( \mathbb{P} \) unifies graph patterns \( P \) and \( Q \) iff \( \mathbb{P}(P) \equiv \mathbb{P}(Q) \). It is a most general unifier if for any other unifying context \( \mathbb{P}' \), there is a context \( \mathbb{P}'' \) such that \( \mathbb{P}'(P) \equiv \mathbb{P}''(\mathbb{P}(P)) \equiv \mathbb{P}'(Q) \equiv \mathbb{P}''(\mathbb{P}(Q)) \). \( \square \)

Graph contexts (see Section 4.3) formalise the notion of generality on graphs. If there is a context specialising \( P \) to \( P' \) then \( P \) is more general than \( P' \), or they are equally general where the context just renames some variables or holes. A pattern with more constraints is also less general.

**Definition 5.10 (Term pattern unification)**

The application \( \text{unify} \ P \ Q \) yields a set of term contexts. If the set is empty then \( P \) and \( Q \) do not unify. Otherwise it contains a term context \( \mathbb{T} \) which is a most general unifier of \( P \) and \( Q \). For the simple hole-substitution-free terms in left patterns, this is a mapping from variables to variables and holes to terms, but in general unifiers will need to have constraint sets too; there might also be more than one unifier.
5.3. DETERMINISM

\[
\begin{align*}
\text{unify } & \epsilon \quad \epsilon \quad = \{\{\}\} \\
\text{unify } & x \quad \epsilon \quad = \{\{x \mapsto \epsilon\}\} \\
\text{unify } & \epsilon \quad y \quad = \{\{y \mapsto \epsilon\}\} \\
\text{unify } & x \quad y \quad = \{\{x \mapsto y\}\} \\
\text{unify } & h \quad Q \quad = \{\{h \mapsto Q\}\} \odot \{\{\}\}\} \\
\text{unify } & P \quad h \quad = \{\{h \mapsto P\}\} \odot \{\{\}\}\} \\
\text{unify } & \mathcal{E} \quad \mathcal{E} \quad = \{\{\}\} \\
\text{unify } & F \quad F \quad = \{\{\}\} \\
\text{unify } & (F \ P \ \bar{P}) \quad (F \ Q \ \bar{Q}) \quad = \text{unify } P \ Q \ \odot \text{unify } (F \ \bar{P}) \ (F \ \bar{Q}) \\
\text{unify } & (F \ x \ \bar{x}, \ P) \quad (F \ y \ \bar{y}, \ Q) \quad = \{x \mapsto y\} \odot \text{unify } (F \ \bar{x}, \ P) \ (F \ \bar{y}, \ Q) \\
\text{unify } & P \quad Q \quad = \{\} \\
\end{align*}
\]

The only significant difference between this term-unification algorithm and standard definitions is the distinction between holes and variables in term graphs. Our graph grammars ensure that variables only ever unify with variables (or null variables) whereas holes unify with arbitrary constructed terms or other holes or null terms but never variables. Constructed terms unify in the normal way: they should have the same function symbol and their corresponding sub-terms unify. Unifiers are composed with the operator \(\odot\) defined next. The occur check for holes is performed by the composition operator (Definition 5.11).

\[\square\]

As patterns may have constraint sets, it is necessary to specify a disunification algorithm — unification with disequations. The unifiers must preserve and satisfy the constraints. This part of the problem can be solved by checking and merging constraints after normal unification [BB94] but as our unifiers include constraint sets it is convenient to include the constraint manipulation in the unifier composition algorithm (Definition 5.11) which adapts the rules of Comon [Com91] to graph-pattern unifiers.

**Definition 5.11 (Unifier composition)**

\[U \odot U' = \text{fixFrom } (\text{merge } \odot \text{sub } \odot \text{norm}) \left\{ \mathcal{M} \mathcal{N} \mathcal{C} \mathcal{D} \mid \mathcal{M} \mathcal{C} \in U, \mathcal{N} \mathcal{D} \in U' \right\}\]

where

\[\text{merge } U \{\mathcal{M}\{h \mapsto S, h \mapsto T\}\mathcal{C} \mathcal{D} \mid \mathcal{M}\mathcal{C} \in U, \mathcal{N}\mathcal{D} \in \text{unify } ST\} = U\{\mathcal{M}\mathcal{N}\mathcal{C} \mathcal{D} \mid \mathcal{N}\mathcal{D} \in \text{unify } ST\}\]

\[\text{merge } U = U, \text{ if } \mathcal{M}\mathcal{C} \in U \Rightarrow \text{function } \mathcal{M}\]

\[\text{sub } U = \text{fixFrom } (\text{sub } \odot \text{occurs } U)\]

\[\text{sub } U = \{\{h \mapsto \mathcal{M}(T) \ | \ (h \mapsto T) \in \mathcal{M}\}\{x \mapsto y\} \mid \mathcal{M}\mathcal{C} \ C \mathcal{D} \in U\}\]

\[\text{occurs } U = \{\mathcal{M}\mathcal{C} \in U \mid (h \mapsto \theta) \in \mathcal{M} \Rightarrow h = h'\}\]

\[\text{norm } U = U' \text{ where } U \sim^* U' \not\leftrightarrow\]

\[U\{\mathcal{M}\{x \mapsto x\}\mathcal{C}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \Rightarrow U\{\mathcal{M}\mathcal{C}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \quad (\text{del}_1)\]

\[U\{\mathcal{M}\{x \mapsto y, y \mapsto x\}\mathcal{C}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \Rightarrow U\{\mathcal{M}\{x \mapsto y\}\mathcal{C}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \quad (\text{del}_2)\]

\[U\{\mathcal{M}\{x \mapsto y, y \mapsto z\}\mathcal{C}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \Rightarrow U\{\mathcal{M}\{x \mapsto z, y \mapsto z\}\mathcal{C}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \quad (\text{merge}_1)\]

\[U\{\mathcal{M}\{x \mapsto y, z \mapsto z\}\mathcal{C}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \Rightarrow U\{\mathcal{M}\{x \mapsto y, z \mapsto y\}\mathcal{C}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \quad \text{if } z \neq \epsilon \quad (\text{merge}_2)\]

\[U\{\mathcal{M}\{h \mapsto h\}\mathcal{C}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \Rightarrow U\{\mathcal{M}\mathcal{C}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \quad (\text{del}_3)\]

\[U\{\mathcal{M}\mathcal{C}\{x \neq x\}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \Rightarrow U\{\mathcal{M}\mathcal{C}\{x \neq y\} \mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \quad (\text{fallacy})\]

\[U\{\mathcal{M}\mathcal{C}\{x \neq y\}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \Rightarrow U\{\mathcal{M}\mathcal{C}\{x \neq y, y \neq x\} \mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \quad (\text{sym})\]

\[U\{\mathcal{M}\{x \mapsto y\}\mathcal{C}\{x \neq z\}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \Rightarrow U\{\mathcal{M}\{x \mapsto y\}\mathcal{C}\{x \neq z, y \neq z\} \mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \quad (\text{spread}_1)\]

\[U\{\mathcal{M}\{x \mapsto y\}\mathcal{C}\{y \neq z\}\mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \Rightarrow U\{\mathcal{M}\{x \mapsto y\}\mathcal{C}\{x \neq z, y \neq z\} \mathcal{D} \mid \mathcal{C}\mathcal{D} \in U\} \quad (\text{spread}_2)\]
To compose two sets of unifiers $U$ and $U'$ we combine every unifier in $U$ with every unifier in $U'$ then repeatedly normalise, substitute and merge until a stable solution is reached.

The normalisation phase $\text{norm}$ is a rewrite system $\leadsto$. Its first four rules deal with the variable-to-variable mappings in the unifiers: $(\text{del}_1)$ removes any identities; $(\text{del}_2)$ eliminates any symmetry (to prevent cycles); $(\text{merge}_1)$ eliminates chains; $(\text{merge}_2)$ eliminates forks. These rules orient the variable mappings such that all variables equal under the unifier map directly to the same variable. $(\text{del}_3)$ removes any identity hole mappings. The remaining rules involve the disequality constraints in a unifier. If a composed unifier is inconsistent then it is eliminated by $(\text{fallacy})$. Between them $(\text{syn})$, $(\text{spread}_1)$ and $(\text{spread}_2)$ ensure that if $x$ and $y$ are equal under the unifier and $x \neq z$ then $y \neq z$, so any inconsistency will be detected.

The substitution phase $\text{subst}$ includes the occur check which eliminating any unifier in which a hole maps to a term containing itself. The exception is where a hole maps to itself plus a substitution because that does not lead to an infinite unification, $\{h \mapsto h\theta\}(h) = h\theta$ (this situation does not arise until we consider arbitrary graph-pattern unification in Section 12.2). The occur check alternates with $\text{sub}$ until a fix point is reached.

The merge phase unifies any two distinct substituted for the same hole. \qed

**Definition 5.12 (Graph pattern unification)**

\[
\text{unify} \left( Pu_1, \ldots, u_n \mid \{C\} \ (Qv_1, \ldots, v_n \mid D) \right) = \left\{ (R, \mathcal{T}) \bigg| \mathcal{T} = \{ (u_i \mapsto v_j)_{i,j=1}^n \mid \{CD\}, \mathcal{T}' \in \mathcal{T} \cap \text{unify} \mathcal{T}(PQ), \ R = \mathcal{T}'(PQ) - (\mathcal{T}'(P) \cap \mathcal{T}'(Q)), \{e\} \notin \text{dom} \mathcal{T}'(PQ) \right\}
\]

To unify two graph patterns we first form a term context $\mathcal{T}$ that maps the roots of the first pattern to the roots of the second and which includes the pattern constraint sets. $\mathcal{T}$ is applied to the graph bodies which are combined and unified by the rules below. This results in a set of term contexts which are composed with $\mathcal{T}$ to get a set of unifying term contexts. The graph unifier is a graph context which includes any of the unified nodes which are not common to both patterns, $R$, and applies a unifying term context $\mathcal{T}'$. Any constraints not common to both patterns are introduced by $\mathcal{T}'$. Null variables must not occur in the domain of the unified graphs.

\[
\text{unify} \ P \{a \mapsto S, a \mapsto T\} = \{ \mathcal{T}' \mid \mathcal{T} \in \text{unify} \ ST, \mathcal{T}' \in \mathcal{T} \cap \text{unify} \mathcal{T}(P \{a \mapsto S\}) \}
\]

\[
\text{unify} \ P = \{ \{\} \} \text{[function } P]\}
\]

A set of pattern nodes is unified by repeatedly unifying any two nodes with the same address. The result is a set of term contexts. \qed

**Example 5.7 (Unification examples)**

Consider the following simple grammar for defining graphs of binary and nullary terms.

\[
\text{ROOT} ::= (t) \\
T ::= (t,t) | ()
\]

The patterns $\{a \mapsto (b, c), b \mapsto (), c \mapsto ()\}a$ and $\{d \mapsto (e, e), e \mapsto ()\}d$ overlap. This is confirmed by the unification algorithm as follows.

To find $\text{unify} \ (\{a \mapsto (b, c), b \mapsto ((), c \mapsto (())a\} \ (\{d \mapsto (e, e), e \mapsto ()\}d)$, we begin by constructing $\mathcal{T}_1 = \{a \mapsto d\}$ to equate the root variables. Then we apply $\mathcal{T}_1$ to the graph bodies.
To find \( \text{unify} \{ d \mapsto (b, c), b \mapsto (), c \mapsto (), d \mapsto (e, e), e \mapsto () \} \) we unify the definitions of the two nodes with address \( d \). \( T_2 \in \text{unify} \ (b, c) (e, e) = \{b \mapsto e, c \mapsto e\} \). Applying this unifier \( T_2 \) to the graph body we find that it becomes a function and so there are no nodes left to be unified: \( \text{unify} \{ d \mapsto (e, e), e \mapsto () \} = \{\} \).

The overall unifying term context \( T_3 \) is formed by composing \( T_1 \) with \( T_2 \) to give: \( T_3 = \{a \mapsto d, b \mapsto e, c \mapsto e\} \).

The unifying graph context applies the term context \( T_3 \) to its arguments and adds any nodes common to both patterns. In this case there are no common nodes so the unifying graph context is \( \{\}, T_3 \), i.e. \( T_3 \). There is one unifier so the patterns overlap.

However, if these patterns are used as left patterns in a GraphKit evaluator then they are extended with constraint sets as shown below.

\[
\text{unify} \quad \{a \mapsto (b, c), b \mapsto (), c \mapsto ()\} \# \{a \neq b, a \neq c, b \neq c\}
\]

\[
\{d \mapsto (e, e), e \mapsto ()\} \# \{d \neq e\}
\]

The unification proceeds as before but \( T_1 = \{a \mapsto d\} \# \{a \neq b, a \neq c, b \neq c, d \neq e\} \). Now at the point where we try to create \( T_3 \) by composing \( T_1 \) with \( \{b \mapsto e, c \mapsto e\} \) we get an empty set of unifiers because a consistent unifier cannot be formed. One possible normalisation trace leading to this conclusion is shown below.

\[
\{a \mapsto d, b \mapsto e, c \mapsto e\} \# \{a \neq b, a \neq c, b \neq c, d \neq e\}
\]

\[
\{a \mapsto d, b \mapsto e, c \mapsto e\} \# \{a \neq b, a \neq c, b \neq c, d \neq e, c \neq b\} \quad \text{(sym)}
\]

\[
\{a \mapsto d, b \mapsto e, c \mapsto e\} \# \{a \neq b, a \neq c, b \neq c, d \neq e, c \neq b, e \neq b\} \quad \text{(spread)}
\]

\[
\{a \mapsto d, b \mapsto e, c \mapsto e\} \# \{a \neq b, a \neq c, b \neq c, d \neq e, c \neq b, e \neq b\} \quad \text{(spread)}
\]

\[
\{a \mapsto d, b \mapsto e, c \mapsto e\} \# \{a \neq b, a \neq c, b \neq c, d \neq e, c \neq b, e \neq b\} \quad \text{(fallacy)}
\]

Although this unification algorithm cannot handle patterns that involve holes with substitutions it is sufficient for all other patterns, including non-linear and unconnected patterns, as confirmed by Proposition 5.7 and Proposition 5.8. We claim that the algorithm is terminating. Regarding confluence, we claim the composition algorithm is confluent modulo the way in which variable substitutions are oriented, e.g. if \( x \) maps to \( y \) and vice-versa it is not determined which mapping remains in the (\( \text{del}_2 \)) rule. These claims are not proven but they are supported by the implementation of GraphKit.

**Proposition 5.7 (Unifiers are correct)**

If \( G \in \text{unify} \ G \ H \) then \( G(G) = G(H) \).

**Proof**

By induction on the structure of terms and graphs, assuming that composition has the following property. \( \mathcal{T}(S) = \mathcal{T}(T) \land \mathcal{T} \in \{\mathcal{T}\} \circ U = \mathcal{T}'(S) = \mathcal{T}'(T) \). To prove this we consider each of the \( \leadsto \) rules and show that if \( \mathcal{T}(x) = \mathcal{T}(y) \) then composing any mapping \( \{x' \mapsto y'\} \) with \( \mathcal{T} \) (where either \( x' \) or \( y' \) or both may equal \( x \) or \( y \)) results in a unifier \( \mathcal{T}' \) under which \( \mathcal{T}'(x) = \mathcal{T}'(y) \). Similarly where an arbitrary hole-to-term mapping or variable disequation is added.

**Proposition 5.8 (Unifiers are complete and most general)**

If \( G \) is a unifier of \( G \) and \( H \) and \( G' \in \text{unify} \ G \ H \) then there is a graph context \( G'' \) such that \( G(G) = G''(G'(G)) \).
Proof
By induction on the structure of terms and graphs, assuming that composition has
the following property. If $\mathbb{T}$ is an instance of a member of $U$ and of a member of $U'$
then $\mathbb{T}$ is an instance of a member of $U \cap U'$, by case analysis on the $\sim$ rules.  \[ \square \]

It follows that this unification algorithm is also adequate for deciding determinism.

Proposition 5.9 (Determinism check)
An evaluator $\{L_i \rightarrow R_i\}_{i=1}^n$ is deterministic if for all $i$ and $j$ such that $1 \leq i < j \leq n$,
unify $L_i, L_j = \{\}$.  

Proof
If the evaluator is non-deterministic there must be some graph which is an instance
of two distinct left patterns $L_i$ and $L_j$ (Definition 5.8). Assuming that the variables
and holes in $L_i$ and $L_j$ are disjoint, there must be a graph context $G$ such that
$G(L_i) = G(L_j)$. This cannot happen if the unification of every pair of left patterns is
empty, by the completeness of unification.  \[ \square \]

Unless stated otherwise, all the evaluators described in this thesis are deterministic.
Non-determinism is slightly harder to create than first appears, as our pattern unifica-
tion in Example 5.7 illustrates. The key to avoiding non-determinism in a GraphKit
input is to make sure that the rules can be re-ordered without affecting any evaluation
traces. Unfortunately, determinism does rule out a lot of useful little short cuts we
might want to use when defining evaluators.

Example 5.8 (A non-deterministic evaluator)
In $\llbracket lazy \rrbracket$, we could reduce the length of many evaluation traces by merging common
combinations of rules. Quite often when a function is applied, its definition is in a
separate graph node so it would be useful to have the rule $(VPush)$ defined below
for the case where a variable is applied to a variable. This rule saves the function
argument and looks up the function $f$ all in one step.

$$\llbracket V\text{l}azy \rrbracket = \llbracket lazy \rrbracket \cup \{(VPush)\}$$

$$\{a \mapsto f\, x\} a, s \mapsto \{a \mapsto \bot, u \mapsto \#a, t, t \mapsto x : s\} f, u \quad (VPush)$$

Unfortunately $\llbracket V\text{l}azy \rrbracket$ is non-deterministic — even though exactly the same result
is generated whether $(VPush)$ or the usual $(Push)$ then $(Lookup)$ is used. To build a
deterministic version of $\llbracket lazy \rrbracket$ we would have to replace the old $(Push)$ with new rules for
applying $\lambda$-expressions, applying let-expressions and applying function applications.  \[ \square \]

It seems that making evaluators confluent — rather than deterministic — pro-
duces more elegant evaluator specifications. But confluence is difficult to prove (much
of Blom’s thesis [Blo01] is about confluence in graph rewriting) and neither it nor
determinism are really essential for our arguments about space usage.

5.4 Summary
We have equipped GraphKit with a model of space and time usage. However, these
measures are only valid for accurate evaluators because unrestricted term-graph rewor-
ting can do arbitrary amounts of work at each step.
Space usage can be controlled by introducing some trimming operations and explicit deallocation operations. These optimisations seem useful in practice; their limitations help to show why we need to rely on a garbage collector to describe space usage properly.

A simple unification algorithm for graph patterns without hole substitutions provides a way to test determinism, reinforcing the argument that GraphKit evaluators are realistic models of implementations.
Chapter 6

Models of Implementations

GraphKit is designed for specifying small-step operational semantics. It is also intended to be suitable for modelling implementations of functional programming languages, especially lazy languages, so that we can reason about their space usage.

Section 6.1 investigates the use of GraphKit for modelling the operational behaviour of Haskell, extending our paper on this subject [BR01]. We take a basic model of lazy evaluation and try to extend it to include other key features of Haskell.

Section 6.2 introduces a series of example evaluators which model the space behaviour of different implementation techniques for lazy evaluation. These evaluators are the subjects of the proofs in subsequent chapters.

Section 6.3 looks briefly at the wider applicability of GraphKit as a framework for defining the semantics of other languages and other implementation techniques.

The evaluators defined in this chapter are listed in the glossary at the end of the thesis for ease of reference.

6.1 A space semantics for Haskell

One of the aims of this work is to model the space behaviour of lazy functional programs. Here we tackle that problem by building a GraphKit model of the core features of the non-strict language Haskell [Pe99]. The evaluator \texttt{lazy} of Example 4.1 is sufficient for studying many aspects of lazy evaluation. All Haskell programs can in principle be transformed to the \(\lambda\)-calculus with sharing and cycles. However, this approach overlooks some important aspects of the language and so \texttt{lazy} can hardly be called its semantics. The following sub-sections present a number of extensions of \texttt{lazy} to cater for more of Haskell.

6.1.1 Translation and transformation

The first problem is that the semantics of a Haskell program is determined by translating it into a simpler core language. The Haskell Report suggests a translation scheme but it is not one-to-one and the core language is not formally defined. The second problem is that after translation a compiler may apply optimising transformations to a program. The only operational requirement Haskell imposes is non-strict semantics. This means that a compiler may change the degree of sharing, so the operational semantics could be call-by-name, call-by-need, some kind of optimal evaluation or a hybrid strategy.
Many parts of the Haskell syntax are deliberately superfluous and their removal by translation is quite straightforward: if-expressions, do-expressions and guards for example. The question is whether translating out a construct can make any program less space-efficient than it would be had that construct been implemented differently. The onus is on implementors to make sure the translation scheme used cannot cause worse space usage than the worst possible allowed by the rules in the language definition. The language designer is responsible for ensuring that the standard translation scheme is not known to be inefficient.

Example 6.1 (Inefficient translation scheme)

\[
\text{case } e_0 \text{ of } \{ x + k \rightarrow e; e' \rightarrow e' \} \\
= \begin{cases} 
& \text{if } e_0 \geq k \text{ then let } x' = e_0 - k \text{ in } e[x'/x] \text{ else } e' \text{ (} x' \text{ is a new variable)} 
\end{cases} \quad (s)
\]

The Haskell Report [Pe99] includes a collection of formal rules which define the semantics of case expressions. Until April 2001, rule (s) was stated as shown above. It eliminates \((n + k)\) patterns from case expressions. A possible problem is that duplicating the case selector expression \(e_0\) as suggested could lead to an unlimited increase in space usage: not only will \(e_0\) be re-computed, but it might have free variables which could artificially extend the lifetime of some data structure.

\[
\text{case } v \text{ of } \{ x + k \rightarrow e; e' \rightarrow e' \} \\
= \begin{cases} 
& \text{if } v \geq k \text{ then } (\lambda x. e)(v - k) \text{ else } e' \text{ (} x' \text{ is a new variable)} 
\end{cases} \quad (s)
\]

The solution shown above is used in the amended Report. It applies the transformation only when \(e_0\) is known to be a value — a number which has almost no re-computation cost and no free variables. An alternative would be to lift the let-expression in the original version and use \(x'\) in both the selector and the then-branch of the transformed expression. \(\square\)

The transformation problem is more of a minefield. The work of Gustavsson and Sands into space-safe lazy program transformations [GS00, GS01] demonstrates that it is possible to provide a set of safe and useful transformations. However, the really useful runtime-saving transformations seem to consistently fall outside such a theory. A transformation like full laziness [Pey87] which increases sharing is bound to increase space-usage requirements in many cases. The same goes for strictness transformations [Wad87b] which eliminate the overhead of suspending evaluation of an argument where call-by-value can be proved to generate the same result.

Perhaps space-unsafe transformations should be banned. But this position is difficult to defend. Languages like Haskell are supposed to take control of the details of evaluation out of programmers hands. Programmers may be quite prepared to accept that compiled code sometimes perform badly and sometimes does much better than expected. To do no transformation betrays the principle of functional programming and forces people to start worrying about operational details. Perhaps the most realistic solution is to make sure that programmers can control sharing and evaluation-order behaviour when absolutely necessary. The strict application construct is a good example of how Haskell currently supports this view.

Rather than getting bogged down in trying to resolve the ambiguities of Haskell, we choose to go in at the core level. Assume that a program is presented fully translated and transformed, ready to run. The question is whether \texttt{lazy} is sufficient as a model for the evaluation of Core Haskell programs.
6.1.2 Algebraic data types

Constructors are usually included in Core Haskell. The benefits include preservation of type information and a nicer encoding of constructor values than using \( \lambda \)-terms. In Chapter 4 we defined \( \texttt{[]} \) which adds booleans and if-expressions to \( \texttt{[] lazy} \). Other algebraic data types are added in a similar way. For each data type \( \tau \) we define a new category \( T \) for its constructors (it is standard to assume that constructors are saturated) and add a function symbol \( CTR_\tau T \) to expression category \( X \). A case-expression is needed to destruct each type. We add a function symbol \( \text{case}_X X \) of \( \{ A_{r,1}; \cdots ; A_{r,n} \} \) which has a selector expression \( X \) and an alternative for each of the \( n \) constructors of type \( \tau \). The \( i \)th alternative is defined \( A_{r,i} := C_i x_1 \cdots x_j \to X \), the number of bound variables \( j \) being the number of arguments of the \( i \)th constructor in \( T \). A new version of the (\textit{Update}) rule is needed for type \( T \) constructors, a rule to start evaluating the selector of a type \( T \) case and rules for choosing each possible alternative once the selector value is found.

Example 6.2 (Adding a list data type)

Adding the data type \( \textit{List} \alpha = \alpha : (\textit{List} \alpha)[[]] \) to \( \texttt{[] lazy} \) extends the grammar as shown below. \( \textit{LIST} \) defines the \textit{List} constructors, \( \textit{ALN} \) and \( \textit{ALC} \) define the case alternatives for \( [] \) and \( (:) \). A stack term \( \{;\} : s \) is added to \( S \) to hold the alternatives of a \textit{List} case while its selector evaluates.

\[
\begin{align*}
X & ::= \cdots \mid \text{\textit{LIST}} \mid \text{\textit{case } } X \text{ of } \{ \text{\textit{ALN}; } \text{\textit{ALC}} \} \\
\text{\textit{LIST}} & ::= [] \mid x : x' \\
\text{\textit{ALN}} & ::= [] \to X \\
\text{\textit{ALC}} & ::= (x : x') \to X \\
S & ::= \cdots \mid \{ \{ \text{\textit{ALN}; } \text{\textit{ALC}} \} : s \}
\end{align*}
\]

The evaluator \( \texttt{list} \) has four new rules. List constructors are values so they need an update rule (\textit{LUpdate}). A case expression is evaluated by storing its alternatives and evaluating its selector expression (\textit{LPush}). When the selector has evaluated to a list constructor then (\textit{LReduce[]} or (\textit{LReduce}:)) chooses the appropriate alternative. Where the selector is a (\( : \)), its arguments are substituted for the pattern-bound variables \( v \) and \( w \) before \( Y \) is evaluated.

\[
\texttt{list} = \texttt{[]} \cup \{ (\texttt{LUpdate}), (\texttt{LPush}), (\texttt{LReduce}[]), (\texttt{LReduce}:) \}
\]

\[
\begin{align*}
\{ a \mapsto L, y \mapsto \bot, s \mapsto \#y t \} a, s & \rightarrow \{ y \mapsto L \} y, t \quad \text{(\textit{LUpdate})} \\
\{ a \mapsto \text{case } E \text{ of } \{ N; C \} \} a, s & \rightarrow \{ a \mapsto E, t \mapsto \{ N; C \} : s \} a, t \quad \text{(\textit{LPush})} \\
\{ a \mapsto [ ], s \mapsto \{ [] \to X; C \} : t \} a, s & \rightarrow \{ a \mapsto X \} a, t \\
\{ a \mapsto p : q, s \mapsto \{ N; (v : w) \to Y \} : t \} a, s & \rightarrow \{ a \mapsto Y[p/v, q/w] \} a, t \quad \text{(\textit{L Reduce}[])}
\end{align*}
\]

\( \square \)

This encoding is a little unsatisfactory as the grammar depends on the program, but it is acceptable because both are known at compile time (and therefore the evaluator is finite for any program by evaluator trimming — see Section 5.1).

Projection functions for constructors are not directly a standard part of Haskell but \textit{pattern bindings} are. References to pattern-bound variables can be translated out easily enough, e.g. \( \text{let } (h : t) = \textit{list in } h \equiv (\text{case } \textit{list of } \{ [] \to \bot; (h : t) \to h \}) \), but it is worth keeping them in the Core language and giving them a direct semantics because, as we will see later, there is a space fault associated with them.
Example 6.3 (Adding a cons pattern binding)

\[
\begin{align*}
X ::= \cdots & \mid \text{let } (x : x') = X \text{ in } X' \mid \pi_h x \mid \pi_t x \\
S ::= \cdots & \mid \#\pi_h x s \mid \#\pi_t x s
\end{align*}
\]

Here the expression grammar is extended with a cons pattern binding and head and tail projection functions. The stack category is extended to include pushed projection functions.

\[
\text{proj} = \text{list} \cup \{(\text{LetCons}), (\text{PihLook}), (\text{PitLook}), (\text{PihUpdate}), (\text{PitUpdate})\}
\]

\[
\begin{align*}
\{a \mapsto \text{let } (x : y) = E \text{ in } X\}a, s & \longrightarrow \\
\{a \mapsto X[b/x, c/y], b \mapsto \pi_h d, c \mapsto \pi_t d, d \mapsto E[b/x, c/y]\}a, s & \quad (\text{LetCons}) \\
\{a \mapsto \pi_h c\}a, s & \longrightarrow \{a \mapsto \bot, t \mapsto \#\pi_h a s\}c, t & \quad (\text{PihLook}) \\
\{a \mapsto \pi_t c\}a, s & \longrightarrow \{a \mapsto \bot, t \mapsto \#\pi_t a s\}c, t & \quad (\text{PitLook}) \\
\{a \mapsto c : d, s \mapsto \#\pi_h b t, b \mapsto X\}a, s & \longrightarrow \{b \mapsto c\}b, t & \quad (\text{PihUpdate}) \\
\{a \mapsto c : d, s \mapsto \#\pi_t b t, b \mapsto X\}a, s & \longrightarrow \{b \mapsto d\}b, t & \quad (\text{PitUpdate})
\end{align*}
\]

The evaluator \(\text{proj}\) has five new rules. The \(\text{(LetCons)}\) rule allocates a pattern binding. The bound variables are replaced with projections which point to the expression \(E\). When the head projection is needed, \(\text{(PihLook)}\) stores it in a new stack node, black-holes the projection node by overwriting it with \(\bot\) and proceeds to evaluate the expression at \(c\); essentially the same as variable lookup. When the expression has evaluated to a cons value, \(\text{(PihUpdate)}\) copies its head argument into the projection node \(b\). When the tail projection is requested, \(\text{(PitLook)}\) and \(\text{(PitUpdate)}\) are used in an analogous way. When both projections have been calculated, the cons node becomes garbage. If only one is mentioned then the other becomes garbage after the pattern binding is allocated.

\[
\begin{align*}
\{ \text{listof} & \mapsto \lambda x.\text{let } \text{lofx} = \text{listof } x \text{ in } x : \text{lofx}, \\
\text{finite} & \mapsto \lambda l.\text{case } l \text{ of } \{[] \rightarrow \text{True}; (h : t) \rightarrow \text{finite } t\}, \\
\text{nulls} & \mapsto \text{listof } \epsilon, \\
\text{main} & \mapsto \text{let } (x : y) = \text{nulls } \text{in if } (\text{finite } y) \text{ x } \epsilon \text{main}, \epsilon
\end{align*}
\]  

(6.1)

The example graph (6.1) uses constructors, case expressions and pattern bindings. The function \(\text{listof}\) lazily generates an infinite list of its argument; \(\text{finite}\) returns the boolean constructor \(\text{True}\) if its argument list is finite; the \(\text{main}\) expression returns the head of \(\text{nulls}\) if the tail of \(\text{nulls}\) is a finite list; \(\text{nulls}\) is an infinite list whose elements are undefined.

\[
(6.1) \xrightarrow{\text{proj}} 14 \{ \text{nulls} \mapsto \text{let } \text{lofx} = \text{listof } \epsilon \text{ in } \epsilon : \text{lofx}, \\
\text{listof} \mapsto \cdots; \text{finite} \mapsto \cdots, \\
c \mapsto \bot, b \mapsto \bot, a \mapsto \pi_h c, \text{main} \mapsto \bot
\]

(6.2)

\[
\begin{align*}
k & \mapsto \#c j, j \mapsto \#\pi_t b i, i \mapsto \#\text{main } g, \\
g & \mapsto \{[] \rightarrow \text{True}; (h : t) \rightarrow \text{finite } t\} : d, d \mapsto \{a ; \epsilon : \epsilon\} \text{nulls}, k
\end{align*}
\]

After 14 evaluation steps (6.1) becomes (6.2). Expression nodes are shown above the gap and stack nodes below, unchanged nodes are abbreviated \(\cdots\). The first element of the list \(\text{nulls}\) is about to be generated. It will be copied into node \(c\) which was
allocated by the let-expression in the original \textit{main} expression. Its tail projection will then be copied into node \( b \) as the process of finding out if \( b \) is finite begins. The head projection of \( c \) in node \( a \) is still reachable via the then-branch of the if-expression which is now waiting in stack node \( d \) for a verdict on whether \textit{nulls} is finite or not.

\[
(6.2) \quad \delta \xrightarrow{35_{\text{proj}}} \{ \begin{array}{l}
c \mapsto \epsilon : o, o \mapsto \epsilon : s, s \mapsto \epsilon : ca,
ca \mapsto \text{let } \text{lofx} = \text{listof } \epsilon \text{ in } \epsilon : \text{lofx},
\text{listof} \mapsto \ldots, \text{finite} \mapsto \ldots
\end{array}
\quad \begin{array}{l}
a \mapsto \pi_h c, \text{main} \mapsto \bot,
\end{array}
\]

\[
\begin{array}{l}
g a \mapsto \#\text{main} f a,
f a \mapsto \{[[] \rightarrow \text{True}; (h : t) \rightarrow \text{finite t} : d, d \mapsto \{a; \epsilon\} : \epsilon \} ca, ga
\end{array}
\]

Picking up the trace 35 steps later (6.3), the fourth element of \textit{nulls} is about to be generated at \( ca \). All four elements at \( c, o, s \) and \( ca \) are reachable via the head projection at \( a \), still reachable via stack node \( d \). The original \textit{nulls} node and its tail projection \( b \) have been collected. The space fault associated with programs like this is that a list can unwind and stay in memory because it is still reachable via a head projection. In our example, nodes \( o \) and \( s \) (and \( ca \) and the rest of the list as it unwinds) are semantically garbage because they are never going to be used. Wadler [Wad87a] suggests a solution to this problem by letting the garbage collector short-cut any projections of constructors (GraphKit does allow a user-defined garbage collector, we return to this point in Section 8.2). Another solution to the projection problem is presented when we define the evaluator \texttt{prsh} in Section 6.2. \( \Box \)

GraphKit can model features like data constructors neatly, allowing us to provide a formal basis for discussing the space behaviour of projections.

### 6.1.3 Primitive functions and numbers

Computers offer much faster operations on numbers represented in a native format than can be achieved by modelling them with constructors. It is common to include numbers in the core language to simplify code generation. It is also important to make the operational assumptions about primitive functions explicit. A classic example is that the choice of which argument of \texttt{+} gets evaluated first may affect the space complexity of a program [GS00].

It is easy enough to add \texttt{Int} and \texttt{Char} types and the appropriate primitive functions to an operational semantics[Lau93, GS00]. It is much more difficult to add them to a GraphKit evaluator without resorting to meta-notation. Variadic function symbols would almost be the right model except that we need limits on the index for these large-yet-finite primitive types. Instead we treat them as large algebraic types.

**Example 6.4** (\( \mathbb{Z}_0 \) extension)

\[
\begin{array}{l}
X ::= \cdots \mid \text{NAT} \mid X + X'
\end{array}
\begin{array}{l}
\text{NAT} ::= 0
\end{array}
\begin{array}{l}
S ::= \cdots \mid (\text{+}X) : s \mid (0+) : s
\end{array}
\]

The grammar is extended with a new category \texttt{NAT} to hold the natural number(s), expressions are extended to include natural numbers and addition expressions. There
are two new stack terms: one to hold the right argument of + and one to hold the
evaluated left argument of + (always zero in this example).

\[
\text{pint} = \text{proj} \cup \{(\text{UpdateInt}), (\text{PrimLeft}), (\text{PrimRight}), (\text{PrimAdd})\}
\]

\[
\begin{align*}
\{a \mapsto N, y \mapsto \bot, s \mapsto \#y t\}a, s & \longrightarrow \{y \mapsto N\}y, t & \text{(UpdateInt)} \\
\{a \mapsto X + Y\}a, s & \longrightarrow \{a \mapsto X, t \mapsto (+Y) : s\}a, t & \text{(PrimLeft)} \\
\{a \mapsto 0, s \mapsto (+Y) : t\}a, s & \longrightarrow \{a \mapsto Y, s \mapsto (0+) : t\}a, s & \text{(PrimRight)} \\
\{a \mapsto 0, s \mapsto (0+) : t\}a, s & \longrightarrow \{a \mapsto 0\}a, t & \text{(PrimAdd)}
\end{align*}
\]

Four new rules are introduced by \text{pint}. Numbers are values so \text{(UpdateInt)} copies
them when they are looked up. Three rules handle +: first the left argument is
evaluated by \text{(PrimLeft)} which stores the right argument in a new stack node. When
the value of the left argument is found, it is swapped with the right argument which is
then evaluated by \text{(PrimRight)}. When the right argument’s value is also known, the
addition is executed by \text{(PrimAdd)}.

To extend this encoding to \(\mathbb{Z}_n\) in GraphKit we need to define a \text{(PrimAdd)} for each
of the \(n^2\) possible additions. No other new rules are needed but \text{(PrimRight)} has to
be formulated differently to get round the accuracy check.

The next example uses \text{pint} to look at the space leak problem associated with the
commutativity transformation.

\textbf{Example 6.5 (Commutativity is not space safe)}

\[
\begin{align*}
\{ \text{listof} \mapsto \lambda x. \text{let } lofx = \text{listof } x \text{ in } x : lofx, \\
\text{last} & \mapsto \lambda l. \text{case } l \text{ of } \{[] \mapsto \bot; (h : t) \mapsto \text{case } t \text{ of } \{[] \mapsto h; (t : tt) \mapsto \text{last } t\}\}, \\
\text{zeros} & \mapsto \text{listof } \text{zero}, \\
\text{zero} & \mapsto 0, \\
\text{main} & \mapsto (\text{let } (h : t) = \text{zeros in } h) + \text{last } \text{zeros}\} \text{main}, \epsilon
\end{align*}
\]

Graph (6.4) defines an infinite list \text{zeros}; the \text{main} expression tries to add the first
element of \text{zeros} to the last. This never terminates. It keeps cycling through the
same sequence of 17 evaluation rules as the list of \text{zeros} is repeatedly unwound by
one element and found not to be empty. The space usage is constant. The largest
graph generated has 13 nodes because as \text{last} moves onto the next element in \text{zeros} the
previous element becomes garbage — the only reference to the growing list is through
\text{last}.

Replacing the \text{main} expression with \text{last zeros} + (\text{let } (h : t) = \text{zeros in } h), we find
that the program no longer runs in constant space. Asking for the last element first
causes the unwinding list \text{zeros} to remain reachable in the graph via the unevaluated
head projection.

\section*{6.1.4 Monadic IO}

Now we come to an issue at the edge of the Haskell specification, and something usually
ignored by theoretical modelling work. The notable exception is Gordon [Gor92] who
gives a big-step semantics for IO and Monadic IO in his \(\mu\nu ML\) meta-language. IO is
necessarily under-specified as the exact nature of the facilities provided and the kind
of errors that can occur depend on the operating system. However, it is important
to give a formal semantics to the parts that are considered standard. Regarding space usage, we want a semantics which ensures that the IO interface cannot buffer unlimited amounts of input or output. Reading ahead or delaying writing for efficiency are acceptable, but reading a whole file of arbitrary size as soon as its first character is asked for should be classified as a space fault.

In [BR01] we gave a very simple semantics to the primitives **getChar** and **putChar**. This section extends the investigation to show how we can incorporate a simple model of the Haskell IO error handling mechanism into a GraphKit evaluator.

The Haskell Report [Pe99] defines the simplest version of the IO system as four primitives: **getChar**, **putChar**, **ioError** and **catch**. The function **getChar** :: **IO Char** reads a character from the input; **putChar** :: **Char -> IO ()** evaluates its argument to a character then writes it to the output, returning the unit constructor as its result. Either operation may result in an error. The type **IOError** is system defined. We assume that the only error is end of file (**EOF**). When an IO operation generates an error a handler function is activated. By default this is the primitive function **ioError** :: **IOError -> IO a**. Alternatively, the user can define a handler **H** which is activated by any IO errors generated during the evaluation of expression **X** by using the function **catch** :: **IO a -> (IOError -> IO a) -> IO a**.

At a slightly higher level, the IO type is a monad: IO operations are built from objects of type **IO a** which do some IO and return an object of type **a**. In particular, a compiled program usually has the type **IO ()**, indicating that it uses IO operations to get input and produce output and if it terminates the only value we get is the unit constructor (**()**). Monadic programs are sequenced with (**>>=** :: **IO a -> (a -> IO b) -> IO b**) which is the basis of the do construct. A sequence of monadic operations typically ends with an application of the function **return** :: **a -> IO a**. See [Tho96] for more details of how to write a monadic program.

Encoding and translating IO

We define the categories **CHAR**, **IOERROR**, **IO** and **UNIT** to model the necessary primitive data types, with case defined for each type as explained in Section 6.1.2.

\[
\begin{align*}
CHAR & ::= \cdots \mid A \mid \cdots \\
IOERROR & ::= EOF \\
IO & ::= IO x \\
UNIT & ::= () \\
X & ::= \cdots \mid \text{getChar } x s \mid \text{putChar } x s X \mid \text{Handler } x s x' \\
& \mid \text{Catch } x s X X' \mid \text{getOK } x s x' \mid \text{getErr} \\
S & ::= \cdots \mid \text{putChar } x s : s'
\end{align*}
\]

To specify exception handling we must make sure that the right handler is called and that the stack root is in the right place when the handler is called. To this end, we extend getChar and putChar with two new arguments: the first is the address of the handler to call if it raises an error and the second is the address the stack top should revert to if the handler is called. We also introduce the function **Handler** to make the necessary re-arrangements when a handler is called. Outside of the scope of a **catch** expression, getChar becomes (**getChar ioError e**), putChar **X** becomes (**putChar ioError e X**) and (**ioError e**) becomes (**Handler ioError ee**), indicating that the default system-defined handler **ioError** should be used.
An expression \( catch \ X \ H \) where \( X \) is the expression guarded by handler \( H \) translates to \( Catch \ x \ s X H \) where \( x \) and \( s \) are fresh variables and any occurrences of \( getChar \), \( putChar \ X' \) or \( ioError \ e \) inside the guarded expression \( X \) are replaced by \( getChar \ x \ s \), \( putChar \ x \ s X' \) or \( Handler \ x \ s \ e \) as appropriate. Exceptions raised in the handler expression \( H \) are not handled by \( H \).

The monadic combination functions are translated as follows: \( return \ x \mapsto IO \ x \) and \( a >>= b \mapsto \text{case } a \text{ of } \{ (IO \ x) \mapsto b \ x \} \).

We also add two new functions to read the response from the IO system: \( getOK \) is true if the last IO operation did not generate an error. If it is false then the error code (which is an element of \( IOERROR \)) can be read with \( getErr \).

Example 6.6 (Program with IO)

\[
gcontents = catch \ (\lambda y. \text{case } (getChar h s) \text{ of} \{ (IO \ c) \mapsto \text{let } cs = inp y \text{ in } c : cs \}) \in let \ inp = \text{flip } y \text{ in } c : cs \mapsto (\lambda e. \text{case } e \text{ of } \{ \text{EOF} \mapsto \text{return } [] \}) \quad (6.5)
\]

\[
main = gcontents >>= \text{finite}
\]

The example program (6.5) translates into the graph (6.6). The function \( \text{finite} \) was defined in Example 6.3. The program reads the input lazily into a list using the function \( gcontents \) and tests if the input is finite. The translation extends \( catch \) and \( getChar \) with two new arguments \( h \) and \( s \) as described earlier. The function \( gcontents \) works by repeatedly reading a character from the input and adding it to a list. If an \( EOF \) error is raised by the IO system then its handler returns the empty list.

\[
\{ gcontents \mapsto \text{Catch } h \ s. \ (\lambda \text{let } inp = \text{flip } y. \text{case } (getChar h s) \text{ of } \{ (IO \ c) \mapsto \text{let } cs = inp y \text{ in } c : cs \}) \text{ in let } inpn = \text{flip } e \text{ in } IO \ inpn) \mapsto (\lambda e. \text{case } e \text{ of } \{ \text{EOF} \mapsto \text{let } \text{nil } = [\text{in } IO \ \text{nil}] \}) \}, \quad (6.6)
\]

\[
\text{finite } \mapsto \ldots
\]

\[
main \mapsto \text{case } gcontents \text{ of } \{ (IO \ l) \mapsto \text{finite } l \}\} \text{main}, e
\]

\[
\square
\]

Semantics of IO

The semantics assume that the user input and output are sequences of \( \text{CHAR} \) symbols. After a symbol is read or written, the IO interface adds a boolean code to the front of the input sequence indicating whether the operation was successful. If this is false, then the IO interface also adds an \( IOERROR \) symbol after it. So the input to the evaluator is a sequence of function symbols described by the expression \( (\text{CHAR True} | \text{False} \ IOERROR)^* \).

The evaluator \( \text{IIO} \) is defined as \( \text{PROJ} \) extended with the \( \text{CHAR}, \ IOERROR, \ IO \) and \( \text{UNIT} \) algebraic data types and the seven new rules described here.

\[
\{ a \mapsto \text{getChar } h \ u \} a, s \mapsto \{ a \mapsto \text{getChar } c \} a, s \quad (\text{GetChar})
\]

\[
\{ a \mapsto \text{getOK } h \ u \ b \mapsto C \} a, s \mapsto \{ a \mapsto \text{getOK } h \ u \ b \mapsto C \} a, s \quad (\text{GetChar})
\]

\[
\{ a \mapsto \text{if } \text{OK } (IO \ b) \ (\text{Handler } h \ u \ c) \mapsto \text{getErr} \} a, s \mapsto \{ a \mapsto \text{if } \text{OK } (IO \ b) \ (\text{Handler } h \ u \ c) \mapsto \text{getErr} \} a, s \quad (\text{GetBool})
\]

\[
\{ a \mapsto \text{getErr} \} a, s \mapsto \{ a \mapsto \text{getErr} \} a, s \quad (\text{GetErr})
\]

\[
\{ a \mapsto \text{getOK } h \ u \ b \mapsto C \} a, s \mapsto \{ a \mapsto \text{getOK } h \ u \ b \mapsto C \} a, s \quad (\text{GetChar})
\]

\[
\{ a \mapsto \text{if } \text{OK } (IO \ b) \ (\text{Handler } h \ u \ c) \mapsto \text{getErr} \} a, s \mapsto \{ a \mapsto \text{if } \text{OK } (IO \ b) \ (\text{Handler } h \ u \ c) \mapsto \text{getErr} \} a, s \quad (\text{GetBool})
\]

\[
\{ a \mapsto \text{getErr} \} a, s \mapsto \{ a \mapsto \text{getErr} \} a, s \quad (\text{GetErr})
\]
The rule \((GetChar)\) reads a character \(C\) from the input, stores it in the new node \(b\) then passes the handler arguments \(h\) and \(u\) along with \(b\) to \(getOK\). Then \((GetBool)\) reads the response code from the IO system. If no error is generated the character \(b\) is wrapped in an \(IO\) constructor and returned. Otherwise the error-handler function \(h\) is raised by evaluating \(Handler\ h\ u\ c\) where \(c\), the argument to \(h\), is found by reading the error code from the IO system with \((GetError)\) when it is demanded.

\[
\begin{align*}
\{ a \mapsto \text{putChar}\ h\ u\ E \} a, s &\longrightarrow \{ a \mapsto E, t \mapsto \text{putChar}\ h\ u : s \} a, t & \text{(PushPut)} \\
\{ a \mapsto C, s \mapsto \text{putChar}\ h\ u : t \} a, s &\longrightarrow \{ \text{putChar}\ c \} a, t \quad \text{(PutChar)}
\end{align*}
\]

The rules for \(\text{putChar}\) work in a similar way. First its argument is evaluated by \((PushPut)\) which stores the \(\text{putChar}\) symbol with its handler arguments in a new stack node. When the argument has evaluated to a character \(C\), it is written out by \((\text{PutChar})\). If the subsequent \(getOK\) succeeds then the unit constructor \(()\) is returned wrapped in an \(IO\) constructor; otherwise the error handler is invoked.

\[
\begin{align*}
\{ a \mapsto \text{Handler}\ h\ u\ x \} a, s &\longrightarrow \{ a \mapsto h\ x \} a, u & \text{(Handler)} \\
\{ a \mapsto \text{Catch}\ h\ u\ X\ H \} a, s &\longrightarrow \{ a \mapsto X[b/h, s/u], b \mapsto H[b/h, s/u] \} a, s & \text{(Catch)}
\end{align*}
\]

\((\text{Handler})\) builds an expression which applies the handler function \(h\), to its argument \(x\) (typically the \(\text{getErr}\) and resets the stack root to \(u\) before evaluating this expression. A \(\text{Catch}\) is reduced in a similar way to a let-expression: \((\text{Catch})\) allocates a new node \(b\) for the handler expression \(H\), replacing occurrences of \(h\) in its scope with \(b\) and occurrences of \(u\) with the current stack root. The substitution into \(H\) is not necessary because in the Haskell specification, any exceptions \(H\) raises are not in the scope of the \(\text{Catch}\). But we cannot represent this idea in GraphKit without using extra function symbols. For example, we could translate \(\text{Catch}\ h\ u\ X\ H\) to let \(h = H\) in \(\text{Catch}'\ u\ X\).

With these expressions referring to stack nodes, GraphKit can no longer prove that category \(S\) is being used as a stack in the sense defined in Section 5.2.

**Example 6.7 (Evaluation with IO)**

\[
((6.6), \langle A, \text{True}, \text{False}, EOF \rangle, \langle \rangle) \xrightarrow{\text{ge} \in \text{hs}} 62 \{ \text{main} \mapsto \text{True} \}\text{main}, \epsilon, \langle \rangle, \langle \rangle
\]

The result of running \((6.6)\), defined in Example 6.6, with a user input stream containing a single character 'A' is summarised above. We extend the input stream to include the IO system responses: first an \(A\) is read then a \(\text{True}\) to indicate a successful read. The next read fails so a \(\text{False}\) is read to indicate that an error occurred and the handler is activated, this reads the error code \(\text{EOF}\) and returns a \([]\) constructor. The result of the program is \(\text{True}\) because the input is finite.

### 6.1.5 The rest of Haskell

There are a few other issues that merit discussion and definition in a space semantics. The state monad and strict evaluation are not represented by the models developed in this section, and both these features affect the space behaviour of programs.

Dictionary arguments are another possible cause of unexpected space behaviour. The issue is whether a function needs to be extended with an extra argument to carry
a dictionary of class methods, and how that argument affects space behaviour. There is potentially another problem if dictionaries are not trimmed of function instances which are no longer needed: these could artificially extend the lifetime of some data structure, possibly causing space leaks.

Although GraphKit evaluators can be quite complicated, they seem to be adequate for describing the features needed for a language like Haskell. Although the lack of a meta-rule or indexed notation is inconvenient, the higher-order term graph syntax is expressive. The guarantee of accuracy is perhaps the most valuable property that using GraphKit gives to a specification intended for reasoning about space.

### 6.2 Space semantics of lazy evaluators

This section defines a number of GraphKit evaluators which model different implementation techniques for lazy evaluation as variants of \texttt{[lazy]}.

#### 6.2.1 Eliminating the stack

We can save some space in a graph semantics by reducing the separation between the stack and heap components. A simple pointer-reversal machine uses the space occupied by a variable to store its update marker.

To implement this scheme we change the root category so there are two expression roots. Remove the stack category \( S \) and add its function symbols to category \( X \): pushed applications record the argument and the address of the previous stack node. Update markers only record the previous stack node address.

\[
\begin{align*}
\text{ROOT} & := \langle x, x \rangle \\
X & := \cdots | x : x' | \#x
\end{align*}
\]

**Definition 6.1 (Evaluator with pointer-reversal stack)**

\( \texttt{host} = \{ (\text{Let}), (\text{Push}), (\text{Reduce}), (\text{NosLookup}), (\text{NosUpdate}) \} \)

\[
\begin{align*}
\{ a \mapsto x \} a, b & \rightarrow \{ a \mapsto \#b \}, a \quad (\text{NosLookup}) \\
\{ a \mapsto \lambda x. E, b \mapsto \#c \} a, b & \rightarrow \{ b \mapsto \lambda x. E \}, b, c \quad (\text{NosUpdate})
\end{align*}
\]

The evaluator \( \texttt{host} \) looks up variables with \( (\text{NosLookup}) \) which overwrites them with an update marker; \( (\text{NosUpdate}) \) then overwrites the marker with the value of the variable, returning the stack root to its previous address. So this optimisation saves space by overwriting expressions with stack terms. It turns out that the benefits of using \( \texttt{host} \) are very slight: the graph in Example 4.1 has a space usage of 5 nodes instead of 6 but its size usage is exactly the same. It seems reasonable to conjecture that any saving in space or time compared with \( \texttt{[lazy]} \) is well within some constant factor.

#### 6.2.2 Separate control expression

The control root (current-expression pointer) is moved about in the graph by \( \texttt{[lazy]} \). We can just as easily fix the control node to obtain a more direct GraphKit version of Sestoft’s Mark 1 machine ([Ses97] or Figure 3.3).
Definition 6.2 (Evaluator with separated control expression)

\[ \text{ses} = \{(\text{Let}), (\text{Push}), (\text{Reduce}), (\text{ScLookup}), (\text{ScUpdate})\} \]

\[ \begin{align*}
    \{a \mapsto x, x \mapsto E\}a, s & \quad \rightarrow \quad \{a \mapsto E, x \mapsto \perp, t \mapsto \#x\}a, t \quad (\text{ScLookup}) \\
    \{a \mapsto \lambda x.E, y \mapsto \perp, s \mapsto \#y\}a, s & \quad \rightarrow \quad \{y \mapsto \lambda x.E\}a, t \quad (\text{ScUpdate})
\end{align*} \]

The disadvantage is that looking up a variable with (ScLookup) involves copying the expression \( E \) as well as black-holing the variable \( x \). It seems likely that \texttt{lazy} and \texttt{ses} have exactly the same space and time usage. However, intermediate states could look superficially quite different, so the interesting question is how to establish such a proof. Section 10.3 addresses this problem; another solution is given in [BR01].

### 6.2.3 Indirection nodes

Copying the expression terms that match holes is inefficient, even in an environment-based implementation. An alternative is to insert an \textit{indirection} term that points to the term we want to copy [Tur79]. For this variant we extend the expression grammar with an indirection function \( I : X ::= \cdots \mid I\,x \).

Definition 6.3 (Evaluator with indirections)

\[ \text{ind} = \texttt{lazy} \setminus \{(\text{Update})\} \cup \{(\text{VUpdate}), (\text{IUpdate}), (\text{IReduce})\} \]

\[ \begin{align*}
    \{a \mapsto \lambda x.E, y \mapsto \perp, s \mapsto \#y\}a, s & \quad \rightarrow \quad \{y \mapsto I\,a\}y, t \quad (\text{VUpdate}) \\
    \{a \mapsto I\,b, y \mapsto \perp, s \mapsto \#y\}a, s & \quad \rightarrow \quad \{y \mapsto I\,b\}y, t \quad (\text{IUpdate}) \\
    \{a \mapsto I\,c, c \mapsto \lambda y.E, s \mapsto x : t\}a, s & \quad \rightarrow \quad \{a \mapsto E[x/y]\}a, t \quad (\text{IReduce})
\end{align*} \]

In the indirection-based evaluator \texttt{ind}, (VUpdate) creates an indirection to a \( \lambda \)-value instead of copying it. When a value lookup reaches an indirection, we could follow the indirection and then use (VUpdate) again, but instead we circumvent the need for two steps by defining (IUpdate) to copy the indirection. Indirected values are reduced by (IReduce) — it would be incorrect to follow the indirection and then reduce the value in-place with (Reduce) because there might be other indirections to that value.

With indirection nodes, the worry is that they might build up into a chain, leading to an unreasonable increase in space usage. Compilers can extend their garbage collectors with a rule like (6.7) to eliminate indirection chains and avoid this situation [Wad87a]. We conjecture that \texttt{ind} cannot cause such a problem and that it would not be significantly improved by using (6.7).

\[ (G\{x \mapsto I\,y\})a, s \quad \rightarrow \quad [y/x]Ga, s \text{ if } x \neq a \quad (6.7) \]

Indirections can be used more aggressively to implement update-marker compaction [Gus01]. When there is already an update marker at the top of the stack, instead of pushing another one on top of it (potentially building up a chain of update markers and black-holed variables), we can overwrite the variable with an indirection and avoid an (Update) step. To understand the problem, consider the following evaluation trace. Using \texttt{ind}, a chain of three update markers builds up:

\[ ... \]

\[ ... \]
\{a \mapsto b, b \mapsto c, c \mapsto d, d \mapsto \lambda x. E\} a, \epsilon

\vdash_{\text{ind}}^{\text{mag\!ic\!}} \{a \mapsto \perp, b \mapsto \perp, c \mapsto \perp, d \mapsto \lambda x. E, s \mapsto \# c t, t \mapsto \# b u, u \mapsto \# a \epsilon\} d, s

\vdash_{\text{ind}}^{\text{mag\!ic\!}} \{a \mapsto I d, b \mapsto I d, c \mapsto I d, d \mapsto \lambda x. E\} a, \epsilon

Ideally we would like an evaluator \textbf{[mag\!ic\!]} which could eliminate all but one of the \textit{(Update)} steps like this:

\{a \mapsto b, b \mapsto c, c \mapsto d, d \mapsto \lambda x. E\} a, \epsilon

\vdash_{\text{mag\!ic\!}}^{\text{ind}} \{a \mapsto \perp, b \mapsto I d, c \mapsto I d, d \mapsto \lambda x. E, s \mapsto \# a \epsilon\} d, s

\vdash_{\text{mag\!ic\!}}^{\text{ind}} \{a \mapsto I d, b \mapsto I d, c \mapsto I d, d \mapsto \lambda x. E\} a, \epsilon

This is not quite possible because we cannot predict where the chain of variables will end as we traverse it (\textit{d} in this case). But if we allow garbage collection of indirections then the next evaluator \textbf{[#ind]} should be acceptable.

**Definition 6.4 (Evaluator with update-marker indirections)**

\[
\text{[#ind]} = \text{[lazy]} - \{ (\text{Lookup}) \} \cup \{ (\text{Indirect}), (\text{Lookup} \epsilon), (\text{Lookup} :) \}
\]

\[
\{ a \mapsto I \, x \} a, s \longrightarrow \{ \} x, s \quad \text{(Indirect)}
\]

\[
\{ a \mapsto x \} a, \epsilon \longrightarrow \{ a \mapsto \perp, x \mapsto \# a \epsilon\} x, s \quad \text{(Lookup} \epsilon)\n\]

\[
\{ a \mapsto x, s \mapsto y : u \} a, s \longrightarrow \{ a \mapsto \perp, t \mapsto \# a s\} x, t \quad \text{(Lookup} :)\n\]

\[
\{ a \mapsto x, s \mapsto \# y u \} a, s \longrightarrow \{ a \mapsto I \, x\} x, s \quad \text{(Lookup} \#)\n\]

Now when we need to look up a variable, if the stack root is \epsilon or the term at the top of the stack is a pushed argument then (\text{Lookup} \epsilon) or (\text{Lookup} :) pushes an update marker as usual. If the top of the stack is an update marker then (\text{Lookup} \#) introduces an indirection and does not push another update marker. Node \textit{s} is unchanged in all three cases. An indirection to \textit{x} is evaluated by just moving the control root to \textit{x} by (\text{Indirect}). Unlike \textbf{[ind]}, no indirection-reduction rule is needed. \hfill \square

Now the example graph can be evaluated as follows.

\[
\{ a \mapsto b, b \mapsto c, c \mapsto d, d \mapsto \lambda x. E\} a, \epsilon
\]

\[
\vdash_{\text{ind}}^{\text{#ind}} \{ a \mapsto \perp, b \mapsto I c, c \mapsto I d, d \mapsto \lambda x. E, s \mapsto \# a \epsilon\} d, s
\]

\[
\vdash_{\text{ind}}^{\text{#ind}} \{ a \mapsto \lambda x. E, b \mapsto I c, c \mapsto I d, d \mapsto \lambda x. E\} a, \epsilon
\]

Note that \textit{b} and \textit{c} can be removed by the standard garbage collector after they are rewritten as indirections, so it seems that while \textbf{[#ind]} can build chains of indirections without (6.7), its space behaviour may not always be worse than \textbf{[ind]}.

It might appear that a better strategy which avoids chains would be to use a more sophisticated update-marker compaction scheme, so that our example graph rewrites to \{ a \mapsto \lambda x. E, b \mapsto \lambda x. E, c \mapsto I b, d \mapsto I b\} a, \epsilon, but this also builds chains: any existing indirections to \textit{d} become double indirections to \textit{b}. Yet another strategy evaluates the example to \{ a \mapsto \lambda x. E, b \mapsto \lambda x. E, c \mapsto I b, d \mapsto \lambda x. E\} a, \epsilon which should be safer as it avoids creating or extending indirection chains. With garbage collection of indirections any of these strategies should be safe.
6.2.4 Black holing

When \texttt{[lazy]} looks up a variable \( x \), it is overwritten with \( \perp \) — \texttt{[lazy]} black-holes \( x \). This has two benefits. Firstly, if \( x \) is directly dependent on itself, this cycle will be detected at run-time when the control root returns to node \( x \) and finds the \( \perp \) which it cannot evaluate. Secondly, it prevents a space fault; it is the lazy evaluation equivalent of the tail-call optimisation. To explore this space fault we define a new evaluator \texttt{[nobh]} which does not black-hole variables.

\textbf{Definition 6.5 (Evaluator without (Lookup) black holing)}

\texttt{[nobh]} = \{(Let), (Push), (Reduce), (LookupNh), (UpdateNh)\}

\[
\begin{align*}
\{a \mapsto x, x \mapsto E \} a, s & \quad \rightarrow \quad \{ t \mapsto \#a, a \mapsto E \} x, t \quad \text{(LookupNh)} \\
\{a \mapsto \lambda x. E, y \mapsto X, s \mapsto \#y \} a, s & \quad \rightarrow \quad \{ y \mapsto \lambda x. E \} y, t \quad \text{(UpdateNh)}
\end{align*}
\]

Another way to avoid black-holing is defined by \texttt{[badbh]}. Instead of maliciously copying an expression during lookup, the variable pointer can just be left in place. We see later that this too can cause a space fault, but only in programs that are directly self-dependent.

\textbf{Definition 6.6 (Evaluator without variable black holing)}

\texttt{[badbh]} = \{(Let), (Push), (Reduce), (LookupBad), (UpdateBad)\}

\[
\begin{align*}
\{a \mapsto x\} a, s & \quad \rightarrow \quad \{ t \mapsto \#a, a \mapsto x \} x, t \quad \text{(LookupBad)} \\
\{a \mapsto \lambda x. E, y \mapsto z, s \mapsto \#y \} a, s & \quad \rightarrow \quad \{ y \mapsto \lambda x. E \} y, t \quad \text{(UpdateBad)}
\end{align*}
\]

The \texttt{[nobh]} space fault is caused by copying an expression and keeping it reachable via the stack. We can get a similar fault by failing to black-hole an application when beginning to evaluate the applied expression. Instead of storing the application argument in a stack node, \( (\text{PushNh}) \) stores a copy of the whole application. This remains in the graph until the value of the applied expression is found and \( (\text{ReduceNh}) \) detaches it.

\textbf{Definition 6.7 (Evaluator without (Push) black holing)}

\texttt{[noah]} = \{(Let), (Lookup), (Update), (PushNh), (ReduceNh)\}

\[
\begin{align*}
\{a \mapsto F \_x \} a, s & \quad \rightarrow \quad \{ a \mapsto F, b \mapsto F \_x, t \mapsto b : s \} a, t \quad (\text{PushNh}) \\
\{a \mapsto \lambda x. E, b \mapsto F \_y, s \mapsto b : t \} a, s & \quad \rightarrow \quad \{ a \mapsto E[y/x] \} a, t \quad (\text{ReduceNh})
\end{align*}
\]

As an experiment in composing evaluation strategies we also define the black-hole-free evaluator \texttt{[noabh]}.

\textbf{Definition 6.8 (Evaluator without (Push) or (Lookup) black holing)}

\texttt{[noabh]} = \{(Let), (LookupNh), (UpdateNh), (PushNh), (ReduceNh)\}

Compiled-code graph reduction machines often need special optimisations to produce the same effect as black-holing. Jones [Jon92] discusses how the G-machine, the TIM and the Spindless Tagless G-machine are modified to prevent the space fault caused by not black holing.
6.2.5 Projection elimination

Recall the space fault mentioned in Example 6.3. Using \(\text{proj}\), an unevaluated head projection of an unwinding list can prevent garbage in the tail of that list from being collected. One solution [Wad87a] is to let the garbage collector short-cut projections of constructors. Another solution suggested by Sparud [Spa93] modifies the evaluator, and this solution can be modelled in GraphKit. First we change the grammar as follows.

\[
X ::= \cdots \mid \text{let } (x : x') = X \text{ in } X' \mid \pi_h x@(:, x') \mid \pi_t x@ (x' :)
\]

\[
S ::= \cdots \mid \#\pi x(x' : x'') s
\]

Projection function symbols are given two arguments: \(x\) is the address of the data structure (which will evaluate to a cons expression in our examples) and \(x'\) is the address of the other projection to that structure. The stack term for a pushed head or tail projection holds the projection address that we are interested in in \(x\), both projection addresses in \(x'\) and \(x''\) and the next stack node address \(s\).

**Definition 6.9 (Projection shortcutting evaluator)**

\[\text{prsh} = \text{list} \cup \{(\text{LetCons'}), (\text{PihPush}), (\text{PitPush}), (\text{PiReduce})\}\]

\[
\{a \mapsto \text{let } (x : y) = E \text{ in } X\}a, s \rightarrow
\{a \mapsto X[b/x, c/y], b \mapsto \pi_h d@(c :), c \mapsto \pi_t d@(b :), d \mapsto E[b/x, c/y]\}a, s \quad (\text{LetCons'})
\]

\[
\{a \mapsto \pi_h p@(b :), b \mapsto X\}a, s \rightarrow \{a \mapsto \perp, b \mapsto \perp, t \mapsto \#\pi a(a : b) s\}p, t \quad (\text{PihPush})
\]

\[
\{a \mapsto \pi_t p@(b :), b \mapsto X\}a, s \rightarrow \{a \mapsto \perp, b \mapsto \perp, t \mapsto \#\pi a(b : a) s\}p, t \quad (\text{PitPush})
\]

\[
\{a \mapsto (d : e), s \mapsto \#\pi i(b : c) t, b \mapsto \perp, c \mapsto \perp\}a, s \rightarrow
\{b \mapsto d, c \mapsto e, a \mapsto (d : e)i, t\} \quad (\text{PiReduce})
\]

The new evaluator \(\text{prsh}\) needs only four rules to evaluate projection expressions. \((\text{LetCons'})\) allocates a pattern binding. As in \(\text{proj}\) there are new nodes for the head projection \(x\), the tail projection \(y\) and the expression \(E\). But now both projections also hold pointers to each other. As \((\text{PihPush})\) begins to evaluate a head projection, both projections are black-holed and their addresses are stored in a new stack node (the head projection address is recorded twice). \((\text{PitPush})\) mirrors this behaviour for a tail projection. The double black-holing ensures that erroneous definitions like \((x : y) = y\) are caught quickly. When a cons value is reached, \((\text{PiReduce})\) updates both projections simultaneously then continues evaluation at the head or tail projection, depending on which version of \((\text{PiPush})\) was used.

The problem discussed in Example 6.3 is fixed by using \(\text{prsh}\). The graph (6.1) now runs in 14 nodes. So \(\text{prsh}\) certainly improves the behaviour of some programs. Sparud’s solution includes a further improvement: instead of just updating the projections with variables, he updates them with indirections which can then be removed by a garbage collector. Without that feature, our solution will perform worse than his in some cases. But it still seems reasonable to conjecture that \(\text{prsh}\) never offers worse space complexity than \(\text{proj}\).

6.2.6 Environment trimming

GraphKit cannot easily model environment-based evaluators such as Sestoft’s Mark 2 machine [Ses97] but we can still model them at a higher level. An environment maps
variables to expressions. When a variable is no longer needed it should be removed from the environment. This happens when a case alternative is selected and one of its pattern-bound variables is not used in the body of that alternative. For example, in the expression case \( x \) of \( (a, b) \rightarrow f(a) \) when the calculation of \( f(a) \) begins the binding for \( b \) should be removed from the environment. Similarly, when a function body is entered: if any parameter is not used then it should be removed from the environment.

Definition 6.10 modifies \texttt{lazy} to model an environment machine that does not trim its environment on function-body entry. The stack is extended with a new term to hold onto an argument while a body is evaluated: \( S ::= \cdots | \Env x s; \langle \text{EnvReduce} \rangle \) adds one of these terms to the stack. When the value of the function body is found (the point where a function in an imperative language would execute a return control statement), the new rule \( \langle \text{EnvReturn} \rangle \) removes the \( \Env \) term from the stack.

\[
\text{Definition 6.10 (No environment trimming)} \quad \texttt{notr} = \texttt{lazy} - \{(\text{Reduce})\} \cup \{(\text{EnvReduce}), (\text{EnvReturn})\}
\]

\[
\begin{array}{ll}
\{a \mapsto \lambda x.E, s \mapsto y : t\}a, s \longrightarrow \{a \mapsto E[y/x], s \mapsto Env y t\}a, s & \langle \text{EnvReduce} \rangle \\
\{a \mapsto \lambda x.E, s \mapsto Env y t\}a, s \longrightarrow \{a, t\} & \langle \text{EnvReturn} \rangle \\
\end{array}
\]

In a real environment machine we could implement a cheap version of trimming by black holing entries for variables we no longer need [FW87]. This way the associated expression in the heap can still be garbage collected if it is dead. We can model this variant too by extending the stack category: \( S ::= \cdots | \Ret s \) and modifying the evaluation rules as follows.

\[
\text{Definition 6.11 (Environment trimming by black holing)} \quad \texttt{bht} = \texttt{lazy} - \{(\text{Reduce})\} \cup \{(\text{RetReduce}), (\text{RetReturn})\}
\]

\[
\begin{array}{ll}
\{a \mapsto \lambda x.E, s \mapsto y : t\}a, s \longrightarrow \{a \mapsto E[y/x], s \mapsto Ret t\}a, s & \langle \text{RetReduce} \rangle \\
\{a \mapsto \lambda x.E, s \mapsto Ret t\}a, s \longrightarrow \{a, t\} & \langle \text{RetReturn} \rangle \\
\end{array}
\]

\subsection*{6.3 Wider application and limitations}

GraphKit is designed for expressing the small-step semantics of lazy evaluation. An important question is how well it can model the space behaviour of other programming languages and of implementations that work in quite different ways.

\subsection*{6.3.1 Modelling other languages}

GraphKit does not cater for some of the concepts which are needed for more abstract operational semantics. This is deliberate; in Section 3.1 we saw how the natural semantics framework is too abstract to describe space usage easily.

A semantics which includes non-determinism is also above our level of abstraction. An example is the semantics of Scheme where the order in which the arguments to a function are evaluated is not specified in the language definition [Cli98]. We cannot model this situation fairly without appealing to an external oracle through the IO model.
Another way to achieve non-determinism is to define an unrooted graph semantics. Again, we cannot represent this scenario. Unrooted rewrite rules are advantageous for describing certain systems such as parallel evaluators where several redexes in a graph can be reduced at once, or for separating reduction from the evaluation strategy as in [BLR96]. But it can be more difficult to reason about space usage where the order of reduction is not fixed. Research into the space usage of parallel languages has only recently reached the definition stage with the publication of some operational semantics, e.g. [BFKT00].

A third source of non-determinism is back-tracking: again, this can only be modelled in a very low-level and explicit manner in GraphKit; back-tracking definitely affects space usage and so the convenience with which we could reason about logic or functional-logic languages is compromised.

At first sight there is no problem with modelling other languages and almost any practical implementation style with a graph-rewriting system: We have experimented with a strict language in [BR00b]. The kind of machines used by Clinger [Cli98] could easily be modelled at the same level of abstraction as our lazy machines — though modelling environments directly in GraphKit is difficult. Similarly, other strategies such as call-by-name evaluation [Abr90] could be described. Combinator-based evaluators such as SKI reduction [Tur79] could be defined. We could also take the approach of transforming a program into a GraphKit evaluator; the space model is evaluator-independent so we could compare different programs fairly.

The dynamic semantics of the region-based evaluation scheme for call-by-value reduction [TT97] is above our level of abstraction as it is a natural semantics which manipulates environments. To reason about this evaluation system we would need to adapt a much lower-level description of a region-based system, such as the Region Kit Abstract Machine [BTV96].

Structural semantics of imperative languages like While [NN92] and its abstract machine semantics translate to GraphKit easily. Abstract machines for functional languages with extra structure for modelling value stacks [Ses97], IO buffers, explicit free lists (for modelling finite machines with limit checks) [SI96] and explicit step-by-step definitions of garbage collectors are possible although complex. We could also include terms to model types in systems that use dynamic type checking or type-based garbage collection [MH97].

The one popular feature of lower-level abstract machines that does seem difficult to model properly in GraphKit is environments.

### 6.3.2 Modelling environments

Instead of making substitutions into expressions, it is often more efficient to delay them until a variable is encountered. At the point where the variable is needed, the delayed substitution is applied to it. The delayed substitution is called an environment. This usually leads to a description of an abstract machine with four components: heap, stack, control expression and a current environment, e.g. [Ses97, Cli98].

To preserve well-formedness in a GraphKit model, we must store an expression in the same node as its environment (if we are to delay substitution from the point where a function is specialised to the point where it requests it argument). With this concession we can imagine redefining loud as an environment-based evaluator like Sestoft’s Mark 2 machine [Ses97]. The required meta-rules are shown below. All
expressions are paired with an environment term \( E \) which maps their free variables to graph-node addresses. The rules \((EPush)\) and \((ELookup)\) use the environment to retrieve the addresses of function arguments and variables respectively.

\[
\begin{align*}
\{ a \mapsto \langle F, x, E \rangle \}, a, s & \quad \longrightarrow \quad \{ a \mapsto \langle F, E, \rangle, t \mapsto E(x) : s \} a, t \quad (EPush) \\
\{ a \mapsto \langle \lambda y.B, E \rangle, s \mapsto x : t \} a, s & \quad \longrightarrow \quad \{ a \mapsto \langle B, E[x/y] \rangle \} a, t \quad (EReduce) \\
\{ a \mapsto \langle \langle x, E \rangle \} a, s & \quad \longrightarrow \quad \{ a \mapsto \langle \bot, t \mapsto \#a \rangle E(x), t \quad (ELookup) \\
\{ a \mapsto \langle \lambda x.B, E \rangle, y \mapsto \bot, s \mapsto \#y \} a, s & \quad \longrightarrow \quad \{ y \mapsto \langle \lambda x.B, E \rangle \} y, t \quad (EUpdate) \\
\{ a \mapsto \langle \text{let } y = X \text{ in } B, E \rangle \} a, s & \quad \longrightarrow \quad \{ a \mapsto \langle B, E[b/y] \rangle, b \mapsto \langle X, E[b/y] \rangle \} a, s \quad (ELet)
\end{align*}
\]

To implement this scheme in GraphKit we could extend expression terms to include a family of \( n \) \( Env \) terms: \( X ::= \cdots \mid \{ Enw : x_1 \cdots x_i, x'_1 \cdots x'_i \}_{i=0}^n \). The \( i \)th \( Env \) term can be displayed as \( \langle X, [x'_i/x_j]_{j=1}^i \rangle \).

Next we need a family of rules for each of the meta-rules. For example, \((EPush)\) can be implemented by a set of rules for each possible environment variable size \( i \) and for each possible argument variable position \( k \) in the environment.

\[
\left\{ \begin{array}{l}
\{ a \mapsto \langle F, y, [x'_j/y_j]_{j=1}^i \rangle \} a, s \longrightarrow \\
\{ a \mapsto \langle F, [x'_j/y_j]_{j=1}^i \rangle, t \mapsto x_k : s \} a, t
\end{array} \right\}_{k=1}^i_{i=0}^{n}
\]

This is tedious but the real problem is that environments can grow, so unless we can somehow prove that there is a maximum environment size, the following \((EReduce)\) rules and the \((ELet)\) rules would break the accuracy condition: evaluator trimming could not turn the infinite family of \((EReduce)\) rules into a finite evaluator because the environment term grows each time the rule is applied.

\[
\left\{ \begin{array}{l}
\{ a \mapsto \langle \lambda y.B, [x'_j/y_j]_{j=1}^i \rangle, s \mapsto x : t \} a, s \longrightarrow \\
\{ a \mapsto \langle B, [x/y] [x'_j/y_j]_{j=1}^i \rangle a, t
\end{array} \right\}_{i=0}^{n-1}
\]

The number of free variables an expression contains is less than its size, so if there is a maximum expression size there should also be a maximum environment size. The naive environment scheme depicted here omits the trimming optimisations needed to guarantee space-safe implementation [Ses97]. To provide a set of environment-based rules to evaluate a particular graph could be possible but there would be lots of them and they would be quite complicated. As with numbers and variadic let or apply expressions, GraphKit is an awkward model of these indexed operations.

There are three main problems: expressions cannot fully be separated from their environments without breaking well-formedness; environments are variadic; environments need to grow and shrink and yet the evaluator must still be provably accurate. The difficulties we have in modelling environments are perhaps an indication of how much care should be taken with implementing them to avoid space faults.

### 6.4 Summary

GraphKit has been applied to give a space semantics to various parts of the core Haskell language and to model several implementation techniques for lazy evaluation. The models are simple graph evaluators, often involving no more than five rules, which capture the space behaviour of a reduction strategy.
We run into difficulties when modelling those ideas that do not fit the graph model naturally — indexed operations and implementation techniques that break from the usual term-binding structure like environments. However, we could still suggest models like the evaluator \texttt{nott} which captures the space behaviour of an environment machine without trimming at a higher level.

The evaluators introduced in Section 6.2 are used as examples for comparing and classifying space behaviour in subsequent chapters.
Chapter 7

Space Leaks

Now that we have some GraphKit models of implementations, we want to prove whether they leak space or not. This chapter argues that a space leak should be seen as a relative property and explains our definition.

Section 7.1 deals with the question of how two evaluators which work in quite different ways can be compared and how we define one as leakier than the other. We explain how our definition of a leak will manifest itself in practice on the implementations modelled by a GraphKit evaluator.

Section 7.2 discusses how to define the severity of a leak by reviewing the hierarchy of possible space-fault classifications, to show where our definition fits in and how it relates to asymptotic worsening. The differences between the size-usage and node-usage models defined in Section 5.1 are also investigated: we consider how our node-usage leaks affect size usage and how accuracy helps to unite these usage measures.

7.1 Relative space leaks

Informally, a program is said to have a space leak if its space usage is worse than expected. Therefore an evaluator, which automates the memory management of a whole language of programs, has a space leak if any one program has worse space usage than expected. Conversely, it does not have a space leak — it is space safe — if the space usage of all programs meets our expectations.

Before giving a formal definition, we must decide how to specify our expectations of what the space usage of a program should be. One approach would be to develop an absolute definition of space safety. We could argue that any evaluation strategy in which all garbage is removed must be safe because the only elements remaining in the graph will be needed by the computation. We could go further than this, classifying retained garbage nodes according to whether they have been used and whether they will be used, as in LDVU profiling [RR96]. But absolute leakiness is flawed because the evaluation strategy may be fundamentally inefficient even if it is garbage free during evaluation. For example, a calculator using unary numbers has a higher space complexity than a calculator using binary numbers and an inefficient binary calculator could still use much less space than a unary calculator which guaranteed garbage-free computation. Worse still, the undecidability of garbage means that all evaluators have space leaks in practice. So to talk of absolute leakiness we would need to define some measure of the severity of a leak.

Instead, a fair way to specify our operational expectations is to give a reference
evaluator — another space semantics — to compare against a proposed implementation. Now we ask if there is a relative leak. This circumvents the problem of deciding how leaky an evaluator is because we can compare different evaluators, testing whether \[ A \] is consistently better or worse than \( B \).

Having decided that leakiness is a relation between evaluators we immediately face another problem. The evaluators under comparison could work in completely different ways. They might even evaluate programs written in completely different languages. We have seen how GraphKit offers a fair and language-independent measure of space usage so it should be possible to make comparisons despite such differences.

### 7.1.1 Translation relations

Translation relations define which \[ A \] states we regard as corresponding to some \( B \) state. This translation \( \equiv \) is a user-defined relation between the graph language of \( A \) and the graph language of \( B \). It might be used in some of the following ways.

- \( \equiv \) is a definition of all valid initial states: it maps all the program states in one graph language to their equivalents in another graph language. This lets us eliminate invalid or meaningless graphs, such as those representing ill-typed programs, from consideration in an evaluator comparison. As a simple example, we could define a translation relation to be the identity on all expression terms in the grammar of Example 4.1 — the \( \lambda \)-calculus with let — this would serve as a useful definition of the possible initial and final states for \( \text{lazy} \) and its variants. Then we can ask if \( \text{lazy} \) has a space leak relative to one of the evaluators in Chapter 6, assuming this translation.

- \( \equiv \) maps an extended language to its core. The process of de-sugaring Haskell discussed in Section 6.1 is this kind of translation. Such a translation could be used to compare the behaviour of \( B \) with \( \text{lazy} \) by defining a translation from if-expressions to the \( \lambda \)-calculus; for example, the translation might include:

\[
\text{if } \text{True} \text{ then } x \text{ else } y \mapsto ((\lambda t.\lambda f.t) x) y.
\]

- \( \equiv \) defines equivalent states of a computation. This kind of translation could show how the control structures used by \( A \) relate to the control structures used by \( B \). For example, if we are comparing the indirection-based evaluator \( \text{ind} \) with \( \text{lazy} \) we might use a translation which relates the following graphs:

\[
\{ a \mapsto I \, b, \, b \mapsto \lambda x.x \} a, \, \epsilon \mapsto \{ c \mapsto \lambda y.y \} c, \, \epsilon.
\]

### 7.1.2 Space-leak definition

Leakiness is a ternary relation between two evaluators and a translation. We call \( A \) leakier than \( B \) with respect to translation \( \equiv \) if there are graphs \( G \) and \( H \) such that \( G \equiv H \) and the space usage of \( G \) on \( A \) is worse than the space usage of \( H \) on \( B \).

The question now is how much worsening constitutes a leak. GraphKit’s space measure is only accurate to within a constant factor which is determined by the combination of evaluator and grammar. We are comparing different evaluators that may use different grammars. Therefore it would be unsound to label \( A \) leaky if it kept its space usage within some constant factor of \( B \). Further, the GraphKit model is supposed to represent a range of implementation styles. It would be unfair to calculate
some allowable constant factor based on the grammar and evaluator definitions because that would assume that any implementation of the model had very similar behaviour to GraphKit — we might accidentally rule out environment-based implementations by doing this, for example.

A programmer will probably complain of a space leak when changing the implementation causes a program to run out of memory. Theoretically this is usually characterised as an asymptotic worsening in space usage. But several standards could be seen as a valid model of the programmer’s leak definition, including: 1. some program is worsened by more than some fixed constant factor; 2. for each constant factor \( k \) there is a program worsened by more than \( k \); 3. some program is worsened by more than any constant factor. Definition 7.1 chooses standard 2, the strongest we can use without assuming a literal interpretation of the graph model. Thus a leak in our model translates to an appreciable worsening in a real implementation. This definition should not be too harsh because the space measure ignores the size of graph nodes.

**Definition 7.1 (\( \equiv \) leakier and repairs \( \Rightarrow \))**

\( \equiv \) is leakier than \( \Rightarrow \) with respect to translation \( \equiv \), written \( \equiv \Rightarrow \) for short, or just \( \equiv \Rightarrow \) if the translation is clear from the context, if:

\[
\forall k \in \mathbb{N} \cdot \exists G, H \cdot G \equiv H \land \text{space } G \equiv k > \text{space } H \Rightarrow .
\]

That is, there is no constant \( k \) for which the space usage of any graph on \( \equiv \) is less than \( k \) times the space usage of its translation on \( \Rightarrow \). Written in the other direction we say \( \equiv \) repairs \( \Rightarrow \); \( \equiv \Rightarrow \Rightarrow \).

\[
\Rightarrow \equiv \]

So \( \equiv \) is not leakier than \( \Rightarrow \) with respect to \( \Rightarrow \), written \( \neq \Rightarrow \) for short, if:

\[
\exists k \in \mathbb{N} \cdot \forall G, H \cdot G \Rightarrow H \Rightarrow \text{space } G \equiv \leq k \times \text{space } H \Rightarrow .
\]

Written in the other direction we say \( \equiv \) does not repair \( \Rightarrow \); \( \equiv \Rightarrow \Rightarrow \).

The leak definition says that for any \( k \) we can find a pair of programs \( G \Rightarrow H \) such that the space usage of \( G \) is more than \( k \) times the space usage of \( H \). We can think of these leak witness graphs as a series of program pairs \( G_i \Rightarrow H_1, \ldots \) such that the difference between the space usage of \( G_{i+1} \) and \( G_i \) is greater than the difference between the space usage of \( H_{i+1} \) and \( H_i \).

Similarly, repair says that for each constant factor the space usage of some program is improved. Repair is not the same as space safety [App92]: repair improves the space usage of some programs; a repair for some programs might cause leaks for other programs.

**Example 7.1 (Examples of space leaks)**

The first example concerns the storage of numbers. Let \( b(n) \) be a graph representing the binary encoding of \( n \) as a chain of binary digits with a single root, for example \( b(2) = \{ a_2 \mapsto 1 : a_1, a_1 \mapsto 0 : [] \} a_2 \). Let \( u(n) \) be a graph representing the unary encoding of \( n \) as a chain of successor nodes terminated with a zero node, for example \( u(2) = \{ a_2 \mapsto \text{succ } a_1, a_1 \mapsto \text{succ } a_0, a_0 \mapsto 0 \} a_2 \).

Imagine (we do not define these evaluators or their languages formally) an evaluator \( \text{binary} \) which reads a number \( n \) from its input, storing it as the graph \( b(n) \). Another evaluator \( \text{unary} \) reads a number \( n \), storing it as \( u(n) \). We have \( \text{unary} \equiv \text{binary} \) with respect to the translation between their initial states: although we can always find a \( k \) by which to multiply the size of any binary number to make it bound the
unary version, we can always find a larger input whose unary encoding is more than \( k \) times the size of its binary encoding.

For a second example, recall the discussion of \( \texttt{pint} \) and the commutativity transformation in Example 6.5. That example showed that \( \texttt{pint} \sim \texttt{pint} \) with respect to the following translation.

\[
(\text{let } (h : t) = \text{zeros in } h) + \text{last zeros} \iff \text{last zeros} + (\text{let } (h : t) = \text{zeros in } h)
\]

This translation turns a program that runs in bounded space into one that needs unbounded space. There is no constant which multiplies its space usage on \( \texttt{pint} \) before translation to be greater than its space usage on \( \texttt{pint} \) after translation. \( \square \)

**Definition 7.2** (\( \sim \) mutually leakier, mutually not leakier \( \preceq \))

\( A \sim B \) if \( A \preceq B \) and \( A \not\leq B \)

\( A \) and \( B \) are mutually leakier than each other (mutually repair).

\( A \preceq B \) if \( A \not\preceq B \) and \( A \not\leq B \)

\( A \) and \( B \) are mutually not leakier than each other (mutually do not repair). \( \square \)

**Example 7.2** (Mutually leaky evaluators)

Lazy evaluation is often blamed for causing space leaks (see Section 2.3). If we define a function \( f(x) = f(x \land \text{True}) \) in a lazy language, evaluating \( f(\text{True}) \) will soon run out of space because the conjunction of \( x \) with \( \text{True} \) at each iteration is delayed. Call-by-value evaluation does not have the same problem because it computes \( x \land \text{True} \) at each iteration, so lazy evaluation is leakier than call-by-value.

But lazy evaluation repairs call-by-value too. A program such as \( \text{finite } l \) where \( l \) is defined as an infinite unwinding list like \( \text{zeros} \) in Example 6.4 can run in constant space under lazy evaluation. But under call-by-value the evaluator tries to find what \( l \) is before deciding whether it is finite and runs out of space.

Encoding these programs as \( \lambda \)-calculus terms would show that \( \texttt{lazy} \sim \texttt{cbv} \) with respect to the identity translation. \( \square \)

We return to the study of leak witnesses in Part IV. Here we look at some properties of \( \preceq \). Not leakiness is reflexive in the sense that \( A \) has the same space usage as itself with respect to the identity translation. Not leakiness can often be shown indirectly by the following transitivity property.

**Proposition 7.1** (\( \preceq \) is transitive)

If \( A \preceq B \) wrt \( \Rightarrow \) and \( B \preceq C \) wrt \( \Rightarrow \) then \( A \preceq C \) wrt \( \Rightarrow \). \( \Rightarrow \)

**Proof**

Just multiply the space-bound constant for \( \Rightarrow \) by the space-bound constant for \( \Rightarrow \) to find a space-bound constant between \( A \) and \( C \). \( \square \)

So to compare two very different evaluators we can introduce a series of intermediate evaluators and prove no overall leak by proving no leak between adjacent intermediate versions. The lack of a symmetry property creates more proof work: we need to prove both \( \preceq \) and \( \preceq \) to show \( \preceq \).

The \( \preceq \) relation is not transitive in general. However, leakiness can be shown indirectly in many cases by the following property.
Proposition 7.2 (Showing $\approx$ indirectly)

If $\text{rng} \xrightarrow{1} \subseteq \text{dom} \xrightarrow{2}$ and $\llbracket A \rrbracket \approx \llbracket B \rrbracket$ wrt $\xrightarrow{1}$ and $\llbracket B \rrbracket \equiv \llbracket C \rrbracket$ wrt $\xrightarrow{2}$

then $\llbracket A \rrbracket \approx \llbracket C \rrbracket$ wrt $\xrightarrow{2} \circ \xrightarrow{1}$.

Proof

From the definitions of $\approx$ and $\equiv$: $\forall k \in G, G \cdot G \xrightarrow{1} G \land \text{space} G \llbracket A \rrbracket > k \times \text{space} G \llbracket B \rrbracket$

and $\exists \ k \in G, H \cdot G \xrightarrow{2} H \Rightarrow \text{space} G \llbracket B \rrbracket \leq k \times \text{space} H \llbracket C \rrbracket$.

Therefore: $\exists \ k \in G, G \cdot G \xrightarrow{1} G' \xrightarrow{2} \text{space} G \llbracket A \rrbracket > (k \div k') \times \text{space} H \llbracket C \rrbracket$

where $\div$ is integer division defined $n \div d = q$ where $q \times d \leq n$ and $(q + 1) \times d > n$ for $d > 0$.


Our examples in following chapters often satisfy this proposition. For example, looking ahead to Figure 13.1, we can deduce noah $\approx$ lazy because the translations for noah $\approx$ noah and noah $\approx$ lazy are both the same identity subset (so their domain and range coincide) and noah $\equiv$ lazy is established in Figure 10.1.

The following chapters discuss practical ways to prove or refute $\approx$. Here we note that $\approx$ is undecidable in general.

Proposition 7.3 (Undecidability of $\approx$)

$\approx$ is undecidable.

Proof

We can reduce the halting problem to $\approx$ as follows. Consider the evaluator counter. Its rules are the same as lazy but it has five roots. The idea is that the third root holds a list of all nodes allocated during evaluation.

$$\text{counter} = \{(C\text{Push}), (C\text{Reduce}), (C\text{Lookup}), (C\text{Update}), (C\text{Let})\}$$

$$\begin{align*}
\{a \mapsto F\ x\} a, s, n, i, j & \quad \rightarrow \\
\{a \mapsto F, t \mapsto x : s, m \mapsto t : n\} a, t, m, i, j & \quad \text{(C\text{Push})} \\
\{a \mapsto \lambda y. E, s \mapsto x : t\} a, s, n, i, j & \quad \rightarrow \\
\{a \mapsto E[x/y]\} a, t, n, i, j & \quad \text{(C\text{Reduce})} \\
\{a \mapsto x\} a, s, n, i, j & \quad \rightarrow \\
\{a \mapsto \#, t \mapsto \# a, s, m \mapsto t : n\} x, t, m, i, j & \quad \text{(C\text{Lookup})} \\
\{a \mapsto \lambda x. E, y \mapsto \#, s \mapsto \# y\ t\} a, s, n, i, j & \quad \rightarrow \\
\{y \mapsto \lambda x. E\} y, t, n, i, j & \quad \text{(C\text{Update})} \\
\{a \mapsto \text{let} y = E \text{ in } X\} a, s, n, i, j & \quad \rightarrow \\
\{a \mapsto X[b/y], b \mapsto E[b/y]\, m \mapsto b : n\} a, s, m, i, j & \quad \text{(C\text{Let})}
\end{align*}$$

Let $E$ be an arbitrary $\lambda$-expression. It is undecidable whether $E$ terminates under lazy evaluation and therefore on counter (This statement is obviously true but the proof is rather indirect: Barendregt [Bar84] shows that the termination of call-by-name reduction is undecidable, Abramsky [Abr90] shows the equivalence of call-by-need and call-by-name, Launchbury [Lau93] shows that a natural semantics implements call-by-need, Sestoft [Ses97] shows that the abstract machine we model as sess implements the natural semantics and we show equivalence of lazy with the Sestoft semantics). Now consider the following translation.

$$\xrightarrow{E} = \{(G_i, G_i) \mid i \in \mathbb{N}\}$$

where $G_i = \{a \mapsto E, p_j \mapsto a_j : p_{j-1} \mid 0 \leq j \leq i, p_{i-1} = \epsilon\} a_i, \epsilon, p_i, \epsilon, p_i$

The graph $G_i$ contains a chain of $i$ stack nodes each pointing to a node containing a copy of $E$. The evaluator never produces any garbage. If $E$ terminates then
space \( G_i \) \( \textbf{counter} \) = \#\( G'_i \) where \( G_i \rightarrow^* \textbf{counter} G'_i \not\rightarrow \textbf{counter} \). Further, the number of nodes in any graph produced by evaluating \( G_i \) is twice the length of the chain of nodes beginning at the third root.

Now consider the evaluator \( \textbf{leaker} \). Starting from a terminal \( \textbf{counter} \) state \( G'_i \) it proceeds to multiply the size of the graph by \( 2^i \). Initially the fourth root is \( \varepsilon \), it is set to \( n \) by (\( CMult \)) and moves all the way down the chain, doubling the chain length by extending it with a new link for each link it traverses using (\( CAdd \)). This behaviour is repeated \( i \) times by moving the fifth root one link down the chain from its original position at each (\( CMult \)).

\[
\begin{align*}
\textbf{leaker} &= \textbf{counter} \cup \{(CAdd), (CMult)\} \\
\{a \mapsto \lambda x. E, i \mapsto b : k\}a, \varepsilon, n, i, j &\rightarrow \{m \mapsto c : n, c \mapsto \lambda x. E\}a, \varepsilon, m, k, j \quad (CAdd) \\
\{a \mapsto \lambda x. E, j \mapsto b : k\}a, \varepsilon, n, \varepsilon, j &\rightarrow \{\}a, \varepsilon, n, n, k \quad (CMult)
\end{align*}
\]

Therefore, if \( E \) terminates on \( \textbf{counter} \) it also terminates on \( \textbf{leaker} \) but its space usage is \#\( G'_i \times 2^i \). If \( E \) does not terminate then the space usage of \( G_i \) on \( \textbf{counter} \) and \( \textbf{leaker} \) are identical. It follows that \( \textbf{leaker} \equiv \textbf{counter} \) wrt \( \not\rightarrow \) iff \( E \) terminates. \( \Box \)

### 7.2 Comparing space-fault criteria

So far we have avoided the question of how leaky an evaluator is. The relative definition of \( \equiv \) does not totally eliminate this issue. There are two dimensions to the problem. Firstly, we may only need one leak witness to call an evaluator leaky. This does not sound very dangerous in a language containing an infinite number of programs! It would be very useful to be able to say what proportion of programs are adversely affected by a leaky evaluator. Even more useful would be a type system that could tell if a particular program is adversely affected. That line of investigation has already been tackled with some success by Pareto [Par01] who defines a type system which only accepts programs guaranteed to run in bounded space. Our contribution to this area of deciding whether a program is adversely affected by a leaky evaluator is the work on recognising leak witnesses in Part IV, but we leave the question of how many programs are affected to further work.

The second dimension in the leakiness question is how much worse are the affected programs on the leaky evaluator. Our definition sets a particular standard which we call leaky and we have justified it to some extent. The rest of this section compares \( \equiv \) with other possible standards by which an evaluator might be deemed to have a space fault, so we can see where it fits into the spectrum. Definition 7.3 formalises this idea of an ordering of space-fault criteria. Finding the smallest criterion under which \( A \) is worse than \( B \) with respect to \( \equiv \) tells us how bad the leak is.

**Definition 7.3 (Space-fault criterion)**

A space-fault criterion relation is a ternary relation on two graph evaluators and a translation relation. We order space-fault criteria by subset. \( \Box \)

#### 7.2.1 Space leaks and IO

Examples 7.1 and 7.2 show how space leaks can arise without IO, but we also stated that an asymptotic worsening in space usage should count as a space leak. Evaluator
Comparisons in the following chapters ignore IO for simplicity. But here we consider how \( \preceq \) relates to the notion of asymptotic worsening and the question of whether it is valid to ignore IO in general.

**Definition 7.4 (Asymptotic space leak)**

\( \mathbb{A} \) is asymptotically leakier than \( \mathbb{B} \) with respect to translation \( \implies \) and we write \( \text{asymleak}(\mathbb{A}, \mathbb{B}, \implies) \) if: \( \exists f, G, H \) such that \( G \implies H \) and \( \text{space} \ (H, I, \langle \rangle) \mathbb{B} \) is \( O(f) \) and \( \text{space} \ (G, I, \langle \rangle) \mathbb{A} \) is not \( O(f) \).

Expanding this definition we get:

\[ \exists f, G, H \cdot G \implies H \land \forall k \exists I \cdot \text{space} \ (G, I, \langle \rangle) \mathbb{A} > k \times f(\#I + 1) \]

\[ \land \exists \forall I \cdot \text{space} \ (H, I, \langle \rangle) \mathbb{B} \leq k \times f(\#I + 1) \]

\[ \Rightarrow \exists G, H \cdot G \implies H \land \exists \forall k \exists I \cdot \text{space} \ (G, I, \langle \rangle) \mathbb{A} > (k \div k') \times \text{space} \ (H, I, \langle \rangle) \mathbb{B} \]

\[ \Rightarrow \exists G, H \cdot G \implies H \land \forall k \exists I \cdot \text{space} \ (G, I, \langle \rangle) \mathbb{A} > k \times \text{space} \ (H, I, \langle \rangle) \mathbb{B} \] \( \square \)

**Example 7.3 (Asymptotic leaks)**

In Example 7.1, \( \text{binary} \) is asymptotically leakier than \( \text{binary} \) because for any \( k \) we can construct an input whose unary representation is more than \( k \) times the size of its binary representation.

We define the translation \( \implies^{ub} \) from unary numbers to binary numbers encoded as in Example 7.1: \( u(n) \implies^{ub} b(n) \). The empty evaluator is not asymptotically leakier than itself with respect to \( \implies^{ub} \): \( \neg \text{asymleak}(\mathbb{B}, \mathbb{B}, \implies^{ub}) \). For any \( u(n) \) we can find a constant which multiplies its size to be greater than the size of \( b(n) \).

The discussion of \( \text{lazy} \iff \text{ebv} \) in Example 7.2 also shows that these evaluators are asymptotically leakier than each other: the example programs have unbounded space usage on the leaky evaluator and bounded space usage on the safe evaluator in both comparisons. So input is not necessary to find an asymptotic leak. \( \square \)

Apart from the inclusion of input and output streams, the main difference between \( \text{asymleak} \) and \( \preceq \) is that the bounding constant \( k \) is program-dependent (but input-independent) in the definition of asymptotic space usage. This means that \( \preceq \) is larger than \( \text{asymleak} \) as shown in Proposition 7.4.

**Proposition 7.4 (An asymptotic leak guarantees \( \preceq \))**

\( \text{asymleak} \subseteq \preceq \)

**Proof**

To compare the definitions we assume that \( \mathbb{A} \) and \( \mathbb{B} \) ignore any input they are given, so the definition of \( \text{asymleak}(\mathbb{A}, \mathbb{B}, \implies) \) becomes:

\[ \exists G, H \cdot G \implies H \land \forall k \cdot \text{space} \ G \mathbb{A} > k \times \text{space} \ H \mathbb{B} \]

that is: \( \text{space} \ G \mathbb{A} = \infty \) and \( \text{space} \ H \mathbb{B} \in \mathbb{N} \). Clearly, this implies \( \preceq \) because we can use the same \( G \) and \( H \) for every \( k \) in its definition, \( \forall k \in \mathbb{N}: \exists G, H \cdot G \implies H \land \text{space} \ G \mathbb{A} > k \times \text{space} \ H \mathbb{B} \).

If there is a leak in the definition of \( \preceq \), then for any \( k \) there are graphs \( G \) and \( H \) such that \( \text{space} \ G \mathbb{A} > k \times \text{space} \ H \mathbb{B} \), but this does not guarantee that \( \text{space} \ G \mathbb{A} > (k + 1) \times \text{space} \ H \mathbb{B} \). So some leaks are not asymptotic leaks. \( \square \)

Having established that \( \preceq \) includes all asymptotic space leaks, we ignore IO from now on. IO is important but it complicates the discussion too much. It might seem reasonable that we could ignore IO altogether. To justify that position we would need to prove that whenever there is an asymptotic leak caused by a program that does
IO, there is also a leak caused by programs that do no IO. This sounds plausible — we did not need to consider IO to show that lazy evaluation and call-by-value are asymptotically leakier than each other — but it is not true in general. Consider comparing an evaluator that buffers all its input with one that does not. The buffering evaluator has an asymptotic leak if there is any program that reads its input which the non-buffering evaluator can run in constant space. But if we ignore IO operations then there is no leak because the two evaluators behave identically.

### 7.2.2 Space-fault criteria

**Definition 7.5 (Absolute worsening)**

\[
\text{absolute}(A, B, \iff) = \exists G, H \cdot G \iff H \land \text{space } G \ A > \text{space } H \ B
\]

The harshest definition we could give of a space fault is **absolute**. This summarises the compiler writer’s aim — to prevent absolute worsening in space usage — but an absolute worsening in a GraphKit evaluator does not say anything meaningful about any implementation of that evaluator because our space model is not accurate enough. Clearly \(\equiv \subset \text{absolute}\), and this observation is the basis of Gustavsson and Sands proof technique [GS00]. They define **strong improvement** as the negation of **absolute**, which turns out to be easier to reason with than the weaker idea of asymptotic improvement.

If we are interested in a particular implementation then it may be appropriate to define a space-fault criterion as worsening by more than some fixed constant factor. More abstract study requires a criterion that defines a space fault as worsening by more than any constant factor. Several variants are possible depending on where the constant is quantified and whether it is related to program size. The three possibilities discussed by Minamide [Min01] are re-cast in our framework by Definition 7.6.

**Definition 7.6 (Inefficient, weakly inefficient, unsafe)**

\[
\text{inefficient}(A, B, \iff) = \forall k \cdot \exists G, H \cdot G \iff H \land \text{space } G \ A > k \times \text{space } H \ B
\]

\[
\text{strongly inefficient}(A, B, \iff)
\]

\[
= \forall k \cdot \exists G, H \cdot G \iff H \land \text{space } G \ A > k \times \text{size } H \times \text{space } H \ B
\]

\[
\text{unsafe}(A, B, \iff)
\]

\[
= \exists G, H \cdot G \iff H \land \forall k \cdot \text{space } G \ A > k \times \text{space } H \ B
\]

\[
= \exists G, H \cdot G \iff H \land \text{space } G \ A = \infty \land \text{space } H \ B \in \mathbb{N}
\]

The **inefficient** criterion is the same as \(A \equiv B\) wrt \(\iff\). It is used by Minamide in a proof that the CPS transformation for call-by-value programs is space efficient with the constant \(k = 3\) in his size model [Min00]. In that proof the measure of state size is closer to our **size** measure than our node-count measure. In general, a version of each space-fault criterion can be defined for any state-size measure, possibly leading to different conclusions about whether an evaluator comparison is leaky.

The **strongly inefficient** criterion allows the program size to be taken into account before we consider efficiency. This is a useful intermediate definition which could help to overcome problems caused by our abstraction from **size usage** to **node usage**. Intuitively, if \(k\) nodes in \(G\) serve the same purpose as a large term in one node in \(H\) then we should allow \(G\) to use \(k\) times as much space as \(H\). Minamide uses this criterion (or rather its negation, **weak efficiency**) to show that a model of an interpreter and a model of a compiler for call-by-value evaluation are not leakier than each other.
The unsafe criterion is the asymptotic-leakiness criterion asymleak specialised to the case where there is no input. Its negation is the standard described by Appel [Appel92] as space-safety.

**Proposition 7.5** (Criteria ordering)

\( \text{unsafe} \subseteq \text{stronglyinefficient} \subseteq \text{inefficient} \)

**Proof**

\( \text{unsafe}(A, B, \iff) \Rightarrow \text{stronglyinefficient}(A, B, \iff) \)

\( \text{unsafe}(A, B, \iff) \not\Rightarrow \text{stronglyinefficient}(A, B, \iff) \)

Similar to Proposition 7.4.

\( \text{stronglyinefficient}(A, B, \iff) \Rightarrow \text{inefficient}(A, B, \iff) \)

\( \text{stronglyinefficient}(A, B, \iff) \not\Rightarrow \text{inefficient}(A, B, \iff) \)

as size \( H \times \text{space} \ H B \geq \text{space} \ H B \).

Finding the smallest criterion \( C \) such that \( C(A, B, \iff) \) tells us how much worse \( A \) is than \( B \) for evaluating graphs in \( \iff \). The criteria in Definition 7.6 can be re-formulated to allow worsening by up to some function of space \( H B \). This way we could distinguish the more serious exponential inefficiency leaks from quadratic inefficiency leaks, for example.

### 7.2.3 Node usage versus size usage

The definition of \( \equiv \) is based on the node-usage definition of space. We claim that the real space usage of an implementation should fall somewhere between our node-usage and size-usage models of space. This section looks at what kind of worsening \( \equiv \) guarantees in the size-usage model. If it guarantees at least asymptotically worse size usage then we can proceed safe in the knowledge that when we claim one of our models is leakier than another, the fault will be present in their implementations. For this discussion we define size-usage analogues of the criteria in Definition 7.6.

**Definition 7.7** (Size-usage leak criteria)

\[ \text{sizeleak}(A, B, \iff) = \forall k \cdot \exists G, \ H \cdot G \iff H \land \text{sizeUsage} \ G A > k \times \text{sizeUsage} \ H B \]

\[ \text{sizestrongleak}(A, B, \iff) = \forall k \cdot \exists G, \ H \cdot G \iff H \land \text{sizeUsage} \ G A > k \times \text{size} \ H \times \text{sizeUsage} \ H B \]

\[ \text{sizesunsafe}(A, B, \iff) = \exists G, \ H \cdot G \iff H \land \forall k \cdot \text{sizeUsage} \ G A > k \times \text{sizeUsage} \ H B \]

\[ \text{sizesunsafe}(A, B, \iff) = \exists G, \ H \cdot G \iff H \land \text{sizeUsage} \ G A = \infty \land \text{sizeUsage} \ H B \in \mathbb{N} \]

The first observation is that if the evaluators are not \textit{accurate} then \( \equiv \) says nothing about size usage. If \( A \) constructs a chain of \( n \) nodes while \( B \) constructs a term of size \( n \), and there is no upper bound on \( n \), then we can have \( \equiv \) but not \textit{sizeleak}.

But assuming the evaluators are \textit{accurate} then \( \equiv \) must correspond to some degree of worsening in size usage. With \textit{uniform accuracy} the node-usage leak classes are identical to their size-usage analogues.

**Proposition 7.6** (Uniform accuracy equates node and size usage leaks)

1. If \( B \) is \textit{uniformly accurate
then \( \text{sizeleak}(\mathbb{A}, \mathbb{B}, \iff) \) whenever \( \mathbb{A} \equiv \mathbb{B} \) wrt \( \iff \).

2. If \( \mathbb{A} \) is uniformly accurate
then \( \mathbb{A} \equiv \mathbb{B} \) wrt whenever \( \text{sizeleak}(\mathbb{A}, \mathbb{B}, \iff) \).

Similarly for the stronglyinefficient and sizestrongleak criteria and for the unsafe and sizeunsafe criteria.

**Proof**

1. Let \( k_B \) be the program-independent accuracy constant (see Definition 5.4) such that \( k_B \times \text{space } H \mathbb{B} \geq \text{sizeUsage } H \mathbb{B} \).

\[ \forall k \cdot \exists G, H \cdot G \iff H \land \text{space } G \mathbb{A} > k \times \text{space } H \mathbb{B} \]

\[ \Rightarrow \forall k \cdot \exists G, H \cdot G \iff H \land \text{sizeUsage } G \mathbb{A} > k \times \text{sizeUsage } H \mathbb{B} \]

\[ \Rightarrow \forall k \cdot \exists G, H \cdot G \iff H \land \text{sizeUsage } G \mathbb{A} > (k \div k_B) \times \text{sizeUsage } H \mathbb{B} \]

2. And the other criteria comparisons follow similarly. \( \square \)

The call-by-need variant evaluators are accurate but not uniformly accurate. In this case we can still show that the smaller asynmleak criteria and its size-usage analogue sizeunsafe are equivalent.

**Proposition 7.7 (Accuracy equates unsafe node and size usage leaks)**

1. If \( \mathbb{B} \) is accurate then \( \text{unsafe}(\mathbb{A}, \mathbb{B}, \iff) \Rightarrow \text{sizeunsafe}(\mathbb{A}, \mathbb{B}, \iff) \).

2. If \( \mathbb{A} \) is accurate then \( \text{sizeunsafe}(\mathbb{A}, \mathbb{B}, \iff) \Rightarrow \text{unsafe}(\mathbb{A}, \mathbb{B}, \iff) \).

**Proof**

1. Accuracy (Definition 5.4) says there is a \( k_B \) such that

\[ k_B \times \text{size } H \times \text{space } H \mathbb{B} \geq \text{sizeUsage } H \mathbb{B} \]

\[ \exists G, H \cdot G \iff H \land \forall k \cdot \text{space } G \mathbb{A} > k \times \text{space } H \mathbb{B} \] by Definition 7.6

\[ \Rightarrow \exists G, H \cdot G \iff H \land \forall k \cdot \text{sizeUsage } G \mathbb{A} > (k \div k_B) \times \text{sizeUsage } H \mathbb{B} \]

\[ \Rightarrow \exists G, H \cdot G \iff H \land \forall k \cdot \text{sizeUsage } G \mathbb{A} > k \times \text{sizeUsage } H \mathbb{B} \]

2. Similar. \( \square \)

So if an evaluator is unsafe in the node-usage model then it is unsafe in the size-usage model. We can do slightly better than this. The next proposition shows that stronglyinefficient \( \subseteq \) sizeleak and \( \equiv \supseteq \) sizestrongleak. These observations add credence to Minamide’s suggestion of the strong inefficiency criteria as a useful standard, and support our claim earlier that strong inefficiency plays a similar role to accuracy in allowing initial program size to be ignored. The relationships between the size-usage and space-usage leak criteria are summarised in Figure 7.1.

**Proposition 7.8 (Relating the node and size usage of accurate evaluators)**

If the evaluators under comparison are accurate then:

1. stronglyinefficient \( \subseteq \) sizeleak

2. \( \equiv \supseteq \) sizestrongleak

3. \( \equiv \neq \) sizeleak

4. stronglyinefficient \( \neq \) sizestrongleak

**Proof**

1. \[ \forall k \exists G, H \cdot G \iff H \land \text{space } G \mathbb{A} > k \times \text{size } H \times \text{space } H \mathbb{B} \]

\[ \Rightarrow \forall k \exists G, H \cdot G \iff H \land \text{sizeUsage } G \mathbb{A} > (k \div k_B) \times \text{size } H \times \text{sizeUsage } H \mathbb{B} \]

Where \( k_B \) is the accuracy constant of \( \mathbb{B} \). As \( (k \div k_B) \times \text{size } H \geq k \):

\[ \Rightarrow \forall k \exists G, H \cdot G \iff H \land \text{sizeUsage } G \mathbb{A} > k \times \text{sizeUsage } H \mathbb{B} \]

stronglyinefficient \( \nsubseteq \) sizeleak is similar.
2. Similar.
3. \( \forall k \exists G, H \cdot G \implies H \land \text{space } G \overset{\text{A}}{\gg} k \times \text{space } H \overset{\text{B}}{\gg} \)
\( \implies \forall k \exists G, H \cdot G \implies H \land \text{sizeUsage } G \overset{\text{A}}{\gg} (k \div k_B) \times (\text{sizeUsage } H \overset{\text{B}}{\gg} \div \text{size } H) \)
Therefore \( \not\models \subseteq \text{sizeleak} \). Similarly, \( \text{sizeleak} \not\subseteq \not\models \).
4. Similar. \( \square \)

Proving strong inefficiency in either the size-usage or node-usage models guarantees some kind of worsening. However, without uniform accuracy, proving \( \not\models \not\models \) does not necessarily guarantee worsening in the size-usage model and a \text{sizeleak} does not guarantee worsening in the node-usage model.

**Example 7.4 (Leakier but not a size leak)**
Consider comparing \( \square \) with itself with respect to the translation \( \models_{\text{W}} \) between unary numbers represented as a chain of \text{succ} nodes and unary numbers represented as a term in a single node defined inductively below.

\[
\begin{align*}
\{a \mapsto 0\}a & \models_{\text{W}} \{b \mapsto 0\}b \\
Gx & \models_{\text{W}} \{b \mapsto Y\}b \\
G\{a \mapsto \text{succ } x\}a & \models_{\text{W}} \{b \mapsto S(Y)\}b
\end{align*}
\]

We have \( \square \not\models \square \models_{\text{W}} \) because for any \( k \) the number of nodes in the chain representing \( k \) is \( k + 1 \). But \( \neg\text{sizeleak}(\square, \square, \models_{\text{W}}) \) because the size of the chain representing \( k \) is \( 2 \times k + 1 \) whereas the single-term representation of \( k \) has size \( k + 1 \). \( \square \)

This result seems to invalidate the definition of \( \not\models \) in favour of the strongly inefficient criterion. However, \( \not\models \) is arguably still the most suitable criteria for comparing the call-by-need evaluators because: 1. the size model of a real implementation, which represents expression nodes as a code pointer with an environment, is closer to the node-usage model: environment size depends on the number of free variables in an expression, which is kept small (though admittedly it is not a constant); 2. the translation relation in our evaluator comparisons is usually an identity, or a close variant on identity, so we are not prone to the kind of problem encountered in Example 7.4.

### 7.3 Summary

A **relative space leak** is a ternary relation on two evaluators and a translation relation. A translation is needed because the two evaluators might use different languages. We say that \( \overset{\text{A}}{\models} \overset{\text{B}}{\models} \) \( \models_{\text{W}} \) if for any \( k \) we can find graphs \( G \models H \) such that the space usage on \( \overset{\text{A}}{\models} \) is more than \( k \) times the space usage of its translation \( H \) on \( \overset{\text{B}}{\models} \). Leaks can often be shown indirectly thanks to the transitivity-like properties of \( \not\models \). No method for deciding \( \not\models \) exists.

The strength of a space fault can be explained by defining other space-fault criteria so that the smallest criterion which recognises the fault indicates its severity. Comparing the size-usage and node-usage models for measuring space shows that if the evaluators under comparison are accurate then \textit{unsafe} worsening in both models is equivalent; \textit{strongly-inefficient} worsening in one model corresponds to some kind of worsening in the other; and \textit{inefficiency} in one model may or may not mean worsening in the other.
Figure 7.1: Lattice of leak criteria
Part III

Relatively No Leaks
Chapter 8

Space Relations

To prepare the ground for proving that $[A]$ is not leakier than $[B]$ with respect to $\iff$ in Chapter 9, this chapter looks at how the translation relation $\iff$ should be specified. A basic prerequisite for the no-leak proof is that $G \iff H$ implies that $\#G$ bounds $\#H$. We call a relation with this space-bound property a space relation. This chapter looks at how to specify a space relation that the properties needed for the no-leak proof, and how to use the definition of $\iff$ to show $G \iff H$.

Section 8.1 develops a framework for specifying space relations as rewrite systems. Section 8.2 explores this framework, showing that we can think of space relations as garbage collectors. Section 8.3 looks at the properties space relations should have, to motivate the definition of a proportional space relation (PSR). Section 8.4 defines PSRs, the restriction of the general framework that we use for no-leak proofs. Section 8.5 explains the properties of PSRs.

8.1 From translation relations to space relations

The first part of the no-leak problem is how to define a translation relation in such a way that it says something about the cardinalities of the graphs it relates. This section develops a simple framework in the form of a rewrite system which can be used to guarantee the space-bound property we seek.

8.1.1 Bisimulation relations

Translation relations are similar to graph bisimilarity relations. A bisimilarity on graphs [AK96] relates a graph to other graphs which have the same meaning — essentially the role of a translation. In graph theory, the meaning of a graph is given by unwinding it into a (possibly infinite) term. Therefore we begin by considering Blom’s definition of bisimilar graphs.

Definition 8.1 (Bisimilar graphs [Blo01])
Term graphs $G$ and $H$ are bisimilar, $G \equiv H$, if there is a correspondence relation $\phi \subseteq \text{var } G \times \text{var } H$ such that if $(x, y) \in \phi$ then the function symbol at node $x$ in $G$ is the same as the function symbol at node $y$ in $H$ and the addresses of corresponding arguments are in $\phi$. Corresponding root variables of $G$ and $H$ must be paired in $\phi$. □
Example 8.1 (Bisimulation relation)
The following graphs are bisimilar despite having slightly different sharing. The correspondence relation required, $\phi$, is attached to the $\leftrightarrow$ symbol, the dot notation is shorthand for $\{(a, b), (a, c)\}$.

$$\{a \mapsto F a\}a \leftrightarrow_{\{a, b, c\}} \{b \mapsto F c, c \mapsto F b\}b$$

With Blom’s definition, the graphs $\{a \mapsto F (F a)\}a$ and $\{b \mapsto F c, c \mapsto F b\}b$ are not bisimilar because the inner term $F a$ cannot be related to $c$ — think of the left graph as shorthand for $\{a \mapsto F i, i \mapsto F a\}a$ where $i$ is an internal node. For graph translation we may want to decompose such nodes, perhaps translating these internal nodes to separated external nodes. □

Details aside, this way of specifying bisimilarity — or in general, any translation — by defining it wherever some correspondence relation $\phi$ between variables exists is a suitable approach. The translation process can be thought of as a traversal of the graph which applies rules to translate a few nodes at a time, these little sub-graphs being stitched into a translated graph through the information held in $\phi$. Something needs changing though, because $\leftrightarrow$ is designed to have the opposite space property to the one we want — it ignores the sharing in graphs.

8.1.2 Exponential space relations

We need to prevent the correspondence relation from pairing a variable with an unlimited number of partners. A simple way to achieve this is to remove nodes from the graphs as they are related.

Example 8.2 (Relating binary trees)
The rules (Branches) and (Leaves) below define a translation $\approx$ between graphs written in the binary-tree grammar of Example 5.7. The definition of $\approx$ works in the same way as $\leftrightarrow$ except that nodes are removed as they are traversed. We define the translation $\approx$ as $G \approx H$ if $G \approx \{\}$ $H$. The axiom (Cycles) allows cyclic trees to be translated.

$$G b \approx_{\phi \cup \{a, d\}} H e \quad G c \approx_{\phi \cup \{a, d\}} H f \quad G \{a \mapsto (\} \}a \approx_{\phi} H \{d \mapsto (\} \} \}b \quad \text{(Branches)}$$

$$G a \approx_{\phi \cup \{a, b\}} H b \quad \text{(Cycles)}$$

Now the depth of a tree is guaranteed to be the same as the depth of any of its translations. For example:

$$\{a \mapsto (b, c), b \mapsto (a, b), c \mapsto (a, c)\}a \approx \{d \mapsto (e, e), e \mapsto (d, e)\}d$$

Unlike bisimilarity, $\approx$ does not relate trees of different depths. For $n > 1$:

$$\{a \mapsto (a, a)\}a \not\approx \left(\{a_n \mapsto (a_1, a_1)\} \cup \{a_i \mapsto (a_{i+1}, a_{i+1})\}_{i=1}^n\right)a_1.$$ □

If we followed this style of definition then translations could pair each graph node with a different translated graph node in each branch of the proof tree. So $G \approx H \Rightarrow b^{\#G} \geq \#H \geq \log_b(\#G)$ where $b$ is the maximum number of premises of the rules defining $\approx$. An obvious advantage of this definition style is that cycles in the graph are only followed once; the translation proof tree for finite graphs is finite. The correspondence relation $\phi$ records the information which ensures that cyclic references are matched properly.
8.1.3 Space relations

**Definition 8.2 (Space relation)**

\[ \iff \text{ is a space relation if } \forall k \in \mathbb{N} \exists G, H \cdot (G \iff H) \Rightarrow (k \times \#G \geq \#H). \]

To tie in with the definition of a graph, a translation relation \( \iff \) must be a *space relation* (Definition 8.2). We achieve this by restricting the exponential space relation so that the proof tree for any pair of related graphs never branches; all the rules defining the relation have at most one premise. The problem then is how to define a translation for graphs that branch. Our solution works by defining translations as configuration-rewrite systems, rather than using the inference-rule notation.

**Definition 8.3 (\( G_1 - G_2 \) configuration-rewrite system)**

A \( G_1 - G_2 \) configuration (or just configuration for short) is a quadruple \( \langle \theta, \phi, G, H \rangle \) where \( \theta \) and \( \phi \) are relations, \( G \) is an unrooted \( G_1 \) graph and \( H \) is an unrooted \( G_2 \) graph.

A configuration-rewrite rule consists of a left configuration pattern and a right configuration pattern \( \langle \Theta, \Phi, P, Q \rangle \rightarrow \langle \Theta', \Phi', P', Q' \rangle \) where \( \Theta \) and \( \Phi \) are relations and \( P \) and \( Q \) are unrooted graph patterns. The graph-pattern nodes are always distinct, ensured by adding constraint sets as usual.

A configuration context \( C \) is a function mapping a configuration pattern to a configuration by applying a term context \( T \) to its components and optionally adding extra variable pairs to the relation components and extra nodes to the graph components:

\[ C(\langle \Theta, \Phi, P, Q \rangle) = \langle \theta \cup T(\Theta), \phi \cup T(\Phi), G \cup T(P), H \cup T(Q) \rangle. \]

A set of configuration-rewrite rules define a rewrite relation \( \rightarrow \) where \( \Delta \rightarrow \Delta' \) if there is a configuration context \( C \) and a rule \( L \rightarrow R \) such that \( C(L) \equiv \Delta \) and \( C(R) \equiv \Delta' \).

A start function maps a pair of rooted graphs to an initial configuration.

The predicate *end* \( \langle \theta, \phi, G, H \rangle \) is true if \( \theta \subseteq \phi \).

A configuration-rewrite system \( \iff \) is a start function paired with a finite set of configuration-rewrite rules: \( \iff = \langle \text{start}, \{ L_i \rightarrow R_i \}_{i=1}^n \rangle \).

The translation relation \( \iff \) defined by a configuration-rewrite system (overloading the \( \iff \) notation) is \( \iff = \{ (G, H) | \exists \Omega \cdot \text{start} (G, H) \rightarrow^* \Omega \land \text{end} \Omega \} \)

Configuration-rewrite rules may allocate, deallocate or change the various parts of a configuration. The exact role of the different components will become apparent, but roughly: \( G \) and \( H \) are the parts of the graphs yet to be translated, \( \theta \) is a goal relation and \( \phi \) is the correspondence relation which records pairs of variables as it did in \( \iff \) and \( \approx \).

Space relations will be defined as configuration-rewrite systems. With this operational view of the relation we talk about the *proof trace* — rather than the proof tree — of a pair of related graphs. Related graphs may have several traces.

**Example 8.3 (Relating binary trees)**

The connection between configuration rewriting and the inference-rule style used to define \( \approx \) is straightforward. In Example 8.2, we constructed a proof tree with nodes of the form \( Ga \approx_b Hb \). This corresponds to a configuration \( \{ a \cdot b \}, \phi, G, H \). The main difference is that configurations have a set of root addresses instead of just a pair. This generalisation to sets allows a rewrite system to traverse all the branches of a graph. The predicate *end* is a universal replacement for the \( (Cycles) \) axiom.
\[ \Longrightarrow = (\text{start}_\text{tree}, \{(\text{Branches}), (\text{Leaves})\}) \]

\[
\text{start}_\text{tree}(Ga, Hb) = \langle \{a \cdot b\}, \{\}, G, H \rangle
\]

\[
\langle \{\}, \{\}, \{a \mapsto (b, c), \{d \mapsto (e, f)\} \rangle \quad \Longrightarrow \quad \langle \{b \cdot c \cdot f\}, \{a \cdot d\}, \{\}, \{\} \rangle \quad (\text{Branches})
\]

\[
\langle \{\}, \{\}, \{a \mapsto ()\}, \{d \mapsto ()\} \rangle \quad \Longrightarrow \quad \langle \{\}, \{a \cdot d\}, \{\}, \{\} \rangle \quad (\text{Leaves})
\]

The configuration-rewrite rules \((\text{Branches})\) and \((\text{Leaves})\) define a translation \(\Longrightarrow\) between isomorphic binary trees (a much smaller relation than \(\approx\) in Example 8.2). Leaves whose addresses are paired in the goal relation are simply removed from the graphs and their addresses are added to the correspondence relation. Branches at addresses paired in the goal relation are also removed from the graphs; their addresses are added to the correspondence relation; the addresses of their left sub-trees are paired and added to the goal relation, similarly for the right sub-trees.

So two tree graphs are isomorphic if there is some proof trace which constructs an isomorphism (the correspondence relation in the final configuration of their proof trace) between their reachable nodes. The example trace below demonstrates the idea.

\[
\text{start}_\text{tree}(\{a \mapsto (b, b), b \mapsto (a, c), c \mapsto ()\} a, \{d \mapsto (e, e), e \mapsto (d, f), f \mapsto (), g \mapsto ()\} d)
\]

\[
= \langle \{a \cdot d\}, \{\}, \{a \mapsto (b, b), b \mapsto (a, c), c \mapsto ()\}, \{d \mapsto (e, e), e \mapsto (d, f), f \mapsto (), g \mapsto ()\} \rangle
\]

\[
\rightarrow \langle \{b \cdot e\}, \{a \cdot d\}, \{b \mapsto (a, c), c \mapsto ()\}, \{e \mapsto (d, f), f \mapsto (), g \mapsto ()\} \rangle \quad (\text{Branches})
\]

\[
\rightarrow \langle \{a \cdot d, b \cdot e\}, \{a \cdot d, c \cdot f\}, \{b \mapsto (a, c), \}, \{e \mapsto (d, f), f \mapsto (), g \mapsto ()\} \rangle \quad (\text{Leaves})
\]

\[
\rightarrow \langle \{a \cdot d, c \cdot f, b \cdot e\}, \{a \cdot d, c \cdot f, b \cdot e\}, \{\}, \{g \mapsto ()\} \rangle \quad (\text{Branches})
\]

The rewrite steps in this trace could take place in any order. Not all of our space relations will have this property but often they do. Even if we rooted the rules (by making the goal relation in the rule left patterns \(\{a \cdot d\}\)) there would still be many possible proof traces because the branches of related trees could be followed in any order. □

A configuration-rewrite trace only says something about the size of the parts of the graph it traverses. To make sure that \(\Longrightarrow\) is a space relation we need to ensure that the entire non-garbage portion of \(H\) is traversed in the proof trace of \(G \Longrightarrow H\); and that the nodes covered in \(G\) are not garbage. Section 8.4 presents our PSR framework for defining configuration-rewrite systems that are guaranteed to have this property. Next we explore configuration rewriting and space relations.

### 8.2 Garbage collection by space relation

One way to think of a space relation \(\Longrightarrow\) is to say if \(G \Longrightarrow H\) then \(H\) is the garbage collection of \(G\) — a space-relation definition can be seen as a high-level operational specification of a garbage collector. The analogy with copying garbage collectors is quite strong: in the configuration \((\theta, \phi, G, H)\), \(G\) is the from space, \(H\) is the to space, \(\phi\) is the forwarding set (it maps addresses already copied from \(G\) to their new location in \(H\)) and \(\theta\) is the frontier (the root set from which all reachable nodes in \(G\) must be copied to \(H\)). With this re-interpretation we can define garbage collection by space relation.
Definition 8.4 (Space-relation garbage collection)

\[ gc \models (G) = H_1 \text{ where } \{H_i\}_{i=1}^n = \{H | G \iff H, \text{size } H = \min \{\text{size } H' | G \iff H'\} \}
\]

The \( \iff \)-collection of \( G \) is a graph \( H_1 \) chosen from the set of the \( n \) smallest translations of \( G \). It is undefined if \( G \) has no translation. Evaluation with evaluator \( A \) and maximal \( \iff \)-collection is the composition formed by following every evaluation step with a \( \iff \)-collection. As \( \iff \) may be partial evaluation can fail if a graph has no collection. □

Of course it would be hideously inefficient to implement a collector this way because configuration rewriting is highly non-deterministic. But for specifying garbage collectors concisely, space relations seem to be an ideal tool. GraphKit allows the specification of non-standard collectors and evaluation with non-standard collectors by this method (see Appendix A).

Example 8.4 (Space relation collector, identity space relation)

The standard reachability-based garbage collector \( gc \) is just the identity space relation on all possible function symbols, \( \iff^{id} \). The identity space relation for a function symbol \( F \) which binds \( b \) variables and takes \( a \) arguments, \((F \ c_1 \cdots c_b.A_1 \cdots A_a)\), is defined by \((Id,F)\) below. The set \( TA \) contains the indexes of all the term arguments of \( F \); these arguments are copied into new nodes by \((Id,F)\) and occurrences of any bound variables are replaced by new variables via the substitutions. Corresponding variable arguments and the addresses of corresponding term arguments are added to the goal relation. The substitutes for corresponding bound variables are added to the correspondence relation.

\[
\langle \{\}, \{\}, \{n \mapsto F \ x_1 \cdots x_b. X_1 \cdots X_a\}, \{m \mapsto F \ y_1 \cdots y_b. Y_1 \cdots Y_a\} \rangle \rightarrow \langle \{u_i. v_i\}_{i=1}^a, \{n.m\} \{w_i. z_i\}_{i=1}^b, \{u_i \mapsto X_i \theta | i \in TA\}, \{v_i \mapsto Y_i \theta' | i \in TA\} \rangle
\]

\((Id,F)\)

where

\( TA = \{i \exists x \cdot A_i = \text{ Tar } c \} \)

\( u_i. v_i = X_i. Y_i, \text{ if } i \notin TA \)

\( \theta = [w_i / x_i]_{i=1}^b \)

\( \theta' = [z_i / y_i]_{i=1}^b \)

The identity translation relation is defined as the identity on all possible function symbols and the identity start rule.

\[
\iff = (\text{start}, \iff^{id})
\]

\[
\iff = \bigcup \{(Id,F) | F \in \text{FSymbols} \}
\]

\( \text{start}(G x_1, \ldots, x_n, H y_1, \ldots, y_n) = \langle \{x_i. y_i\}_{i=1}^n, \{\}, G, H \rangle \)

Identity rules are needed quite often — not just for garbage collection, (Branches) and (Leaves) in Example 8.3 are identity rules. To avoid defining them repeatedly we just take the instance we need. The example below shows the rule denoted by \((Id.Apply)\) in the \( \lambda \)-calculus grammar of Example 4.1.

\[
\langle \{\}, \{\}, \{n \mapsto X \ u\}, \{m \mapsto Y \ v\} \rangle \rightarrow \langle \{x. y, u. v\}, \{n. m\}, \{x \mapsto X\}, \{y \mapsto Y\} \rangle (Id.Apply)
\]

Intuitively, choosing the smallest \( H \) such that \( G \iff^{id} H \) has the same effect as garbage collecting \( G \) because \( H \) is isomorphic to some sub-part of \( G \). That sub-part
must include all root-reachable nodes to satisfy the end predicate but it does not have to include any unreachable nodes. For a more formal argument, \((Id.F)\) satisfies the Proportional Space Relation constraints introduced in Section 8.4 which guarantee these properties.

Defining exactly what a garbage collector does is a tricky problem. It is well known that garbage collectors do not simply remove all the garbage in a graph as this is impossible (see Proposition 2.1). They cannot remove all nodes not needed in some future step; but some collectors do remove or change nodes that would otherwise be used. Examples include shorting out indirection chains to live values [Tur79] or evaluating projections [Wad87a].

In the light of this view of a garbage collector as an evaluator’s assistant, which goes round the graph sweeping up the rubbish and performing menial tasks, we characterise collectors as follows.

- Garbage collectors are global rewrite systems that do not affect the outcome of the computation (they may do something to reduce or increase the number of steps needed; they may turn a non-terminating state into an error state or vice-versa).

- Garbage collectors do not increase graph size (by more than a constant factor!).

Papers which formalise particular garbage collectors also seem to follow this view, defining them as some abstract machine [MFH95, Wal00]. Other work aimed at higher-level presentations of operational behaviour (including the standard GraphKit collector) tends to settle for a much more limited definition of what a garbage collector is, based on reachability (e.g. [Lau93, App92]). Rose [Ros96] uses a mix of term-rewrite rules and the abstract concept of contextual closure to define a collector.

Garbage collectors defined as configuration-rewrite systems are more concise and abstract than a deterministic, and maybe collection-method-specific, abstract machine. They offer a flexible formal framework which could be used for proofs about the capabilities of different collectors.

Clearly configuration rewriting is a global graph-rewrite system. Whether a particular space relation preserves the meaning of a program is something that classic proof techniques can demonstrate — a user-defined space relation may or may not. Not increasing graph size by more than a constant factor is exactly the guarantee of a space relation. So it is in this sense that space relations are garbage collectors. The GraphKit framework for space-relation collectors is not defined formally here, but we look at some examples which are used in some of the leakiness proofs in Part IV.

**Example 8.5 (More garbage collectors)**

As a very simple example, consider implementing the (dead-binding-elimination) transformation, discussed in Example 3.6, as a garbage collector: extend the collector to remove any let-bound expression not mentioned in the let-body. This is expressed very concisely by the space relation \(let\_gc\).

\[
let\_gc = (start, \overset{id}{\rightarrow} \cup \{(Dead\_Let)\})
\]

\[
\langle \{\}, \{\}, \{a \mapsto x = A \text{ in } B\} \rangle \rightarrow \langle \{b \cdot c\}, \{a \cdot c\}, \{b \mapsto B\}, \{\} \rangle \quad (Dead\_Let)
\]

If \((Dead\_Let)\) is applied and \(x\) turns out to be reachable in the body \(B\) then \(x\) will be added to the goal relation by some step in the trace and overall the trace will not
reach an end state. A successful garbage-collection trace has to use \((Id,let)\) instead. If \(x\) is not reachable from \(B\) then \((DeadLet)\) can be used and the trace still reaches an end state. Because removing the let gives a smaller graph, the graph after garbage collection will not contain any dead let-bindings; \(letgc\) is a garbage collector.

We can use the same idea to short-cut indirection nodes with \(Igc\).

\[
Igc = (\text{start}, \overset{id}\rightarrow \cup \{(\text{DeadInd})\})
\langle \{\}, \{\}, \{a \mapsto I b\}, \{\} \rangle \rightarrow \langle \{b \cdot c\}, \{a \cdot c\}, \{\}, \{\} \rangle \quad \text{(DeadInd)}
\]

The idea of a projection-shorting garbage collector [Wad87a] was mentioned in Section 6.2.5. Suppose the tail projection of a cons has already been computed. The head projection still points to the whole cons node. \(\text{(ProjHead)}\) shortens the projection in this case. Similarly, the tail projection can be shortened by \(\text{(ProjTail)}\). A third possibility is that both projections point to a cons \(\text{(ProjCons)}\), but this situation can be avoided by static analysis.

\[
\pi gc = (\text{start}, \overset{id}\rightarrow \cup \{(\text{ProjHead}), (\text{ProjTail}), (\text{ProjCons})\})
\langle \{\}, \{\}, \{a \mapsto \pi h b, b \mapsto (h : t)\}, \{c \mapsto d\} \rightarrow \langle \{h \cdot d\}, \{a \cdot c\}, \{\}, \{\} \rangle \quad \text{(ProjHead)}
\langle \{\}, \{\}, \{a \mapsto \pi t b, b \mapsto (h : t)\}, \{c \mapsto l\} \rightarrow \langle \{t \cdot l\}, \{a \cdot c\}, \{\}, \{\} \rangle \quad \text{(ProjTail)}
\langle \{\}, \{\}, \{a \mapsto \pi h c, b \mapsto \pi t c, c \mapsto (h : t)\}, \{e \mapsto d, f \mapsto l\} \rightarrow \langle \{h \cdot d, t \cdot l\}, \{a \cdot e, b \cdot f\}, \{\}, \{\} \rangle \quad \text{(ProjCons)}
\]

Non-standard garbage collection is used in recent Haskell compilers to implement another space-saving idea. Sometimes variable-update markers are dead: if a variable is only reachable from its update marker then both update marker and black-holed variable can be removed [GS00]. Again, we can define this idea as a space relation \((DeadUdm)\). This collector can be used with the evaluator \(#gc\) (see Appendix A), but it cannot be formulated so easily for \([\text{lazy}]\).

\[
# gc = (\text{start}, \overset{id}\rightarrow \cup \{(\text{DeadUdm})\})
\langle \{\}, \{\}, \{s \mapsto \#x t, x \mapsto \bot\}, \{\} \rangle \rightarrow \langle \{s \cdot u\}, \{t \cdot u\}, \{\}, \{\} \rangle \quad \text{(DeadUdm)}
\]

\[\square\]

### 8.3 Desired properties of space relations

Guaranteeing that a configuration-rewrite system defines a space relation imposes restrictions on its definition. But there are other properties that it would be desirable for space relations to have too. This section explains the other properties that influence our PSR rule restrictions defined in the next section.

#### 8.3.1 Garbage collection property

To help make configuration rewriting suitable for reasoning about space the following \(gc\) rule should be valid. Although configuration-rewrite rules need not be guided explicitly by roots, the intuition is that the goal relation of a configuration provides a set of roots and the proof trace should traverse and remove everything reachable from it. If the intuition holds true then we can garbage collect a configuration by applying \(gc\); translation only needs to consider reachable nodes.
Definition 8.5 (Configuration garbage collection)

\[ gc(\theta, \phi, G, H) = \langle \theta - \phi, \phi \cap (var G' \times var H'), G', H' \rangle \]

where \( G' = G \mid \text{reach}(G, \text{dom} \theta) \)

\[ H' = H \mid \text{reach}(H, \text{rng} \theta) \]

The \( gc \) rule simplifies a configuration by: 1. removing any pairs from its goal relation which are already in its correspondence relation; 2. restricting the correspondence relation to variables in the graphs which are reachable from the goal relation; 3. removing nodes from \( G \) that are unreachable from the goal; 4. removing nodes from \( H \) that are unreachable from the goal.

Guaranteeing the applicability of \( gc \) imposes some reachability conditions on the configuration-rewrite rules defining a space relation: totally unconnected patterns cannot be allowed as they could match garbage nodes in one graph with reachable nodes in the other graph. Making sure that \( gc \) holds means that the space-bound guarantee of a relation ignores any garbage.

8.3.2 Local reasoning properties

There is a tradeoff between making \( \Rightarrow \) proof traces easy to construct and making the \( \Rightarrow \) proofs easy to construct. The issue here is whether the configuration-rewrite rules should be rooted or not. We put forward unrooted rules but noted that they have a poor complexity in Example 8.3: there are a lot of possible proof traces to try when we ask if a space relation relates \( G \) to \( H \). So rooting the rules would be helpful. For the \( \Rightarrow \) proofs, the important properties are \( SubTrace \) and \( GeTrace \).

Definition 8.6 (Sub-trace and garbage collection properties)

\[ SubTrace \Leftarrow= \exists \Omega \cdot (\theta \cup \theta', \phi, G, H) \Rightarrow^* \Omega \land end \Omega \Rightarrow \exists \Omega' \cdot (\theta, \phi, G, H) \Rightarrow^* \Omega' \land end \Omega' \]

\[ GeTrace \Leftarrow= \exists \Omega \cdot \Delta \Rightarrow^* \Omega \land end \Omega \Rightarrow \exists \Omega' \cdot gc \Delta \Rightarrow^* \Omega' \land end \Omega' \]

The \( SubTrace \) property holds if whenever a configuration rewrites to an \( end \)-state we can take a subset of its goal relation and the resulting configuration still rewrites to an \( end \)-state. Similarly, \( GeTrace \) says that whenever a configuration rewrites to an \( end \)-state, its garbage collection also rewrites to an \( end \)-state.

In combination these properties ensure that the proof trace of a configuration represents the proofs of all its possible \textit{sub-configurations}: if we do not know whether the variable pair \( x \cdot y \) are garbage or not, the proof trace with this pair in the initial goal relation represents the cases where they are garbage and where they are not garbage. A good intuition is that if \( a \cdot b \) is in the correspondence relation in the end configuration of a trace then \( a \) is garbage exactly if \( b \) is garbage. For example, a proof trace of \( \{ \{ a \cdot c, \Omega \cdot d \}, \{ a \mapsto (b, b), b \mapsto (b, b) \}, \{ c \mapsto (d, d), d \mapsto (d, d) \} \} \) represents proofs of both \( \{ a \mapsto (b, b), b \mapsto (b, b) \} a \Rightarrow^* \{ c \mapsto (d, d), d \mapsto (d, d) \} \) and \( \{ b \mapsto (b, b) \} b \Rightarrow^* \{ d \mapsto (d, d) \} \).

If all \( \Rightarrow \) rule left-pattern nodes are root-reachable then \( GeTrace \Rightarrow^* \) follows immediately. Then we would need to show that a proof trace can be re-ordered to get \( SubTrace \Rightarrow^* \). With unrooted \( \Rightarrow \) rules, \( SubTrace \Rightarrow^* \) is immediate — because removing goals does not affect the applicability of any rules — but we must impose reachability conditions on \( \Rightarrow \) rule construction to get \( GeTrace \Rightarrow^* \).
Unfortunately, rooted rules cannot always guarantee \textit{SubTrace}. We cannot always reorder steps in a rewrite trace as desired. Even if we flatten the graphs first so as to give an addresses to every function symbol, there may not be a proof trace for the sub-configuration.

**Proposition 8.1** (Rooted rewrite rules do not guarantee \textit{SubTrace})

Defining $\rightarrow$ by rooted configuration-rewrite rules does not imply \textit{SubTrace} $\rightarrow$.

**Proof**

Consider the following simple graph grammar and example graph.

\[
\begin{align*}
\text{ROOT} & := \langle t, t \rangle \\
T & := (T, T') \mid F t \mid G t
\end{align*}
\]

\[
\{x \mapsto (F y, F y), y \mapsto G x\}x, y
\]

(8.1)

Now consider the space relation $\overset{\text{subid}}{\longrightarrow}$ defined below. It is a subset of the identity which includes pairs whose elements are equal, F symbols that point to G symbols, and any individual G symbols.

\[
\overset{\text{subid}}{\longrightarrow} = (\text{start}, \{(\text{PairAs}), (\text{FGs}), (\text{Id}, G)\})
\]

\[
\langle \{\}, \{\}, \{a \mapsto (A, A), b \mapsto (A, A)\} \rightarrow \langle \{a \cdot b\}, \{a \mapsto A\}, \{d \mapsto A\} \rangle \rangle \quad \text{(PairAs)}
\]

\[
\langle \{\}, \{\}, \{a \mapsto F b, b \mapsto G c\}, \{d \mapsto F e, c \mapsto G f\} \rightarrow \langle \{a \cdot d, b \cdot e\}, \{\}, \{\} \rangle \rangle \quad \text{(FGs)}
\]

\[
\langle \{\}, \{\}, \{b \mapsto G c\}, \{c \mapsto G f\} \rightarrow \langle \{a \cdot f\}, \{b \cdot e\}, \{\}, \{\} \rangle \rangle \quad \text{(Id, G)}
\]

The following trace shows that (8.1) $\overset{\text{subid}}{\longrightarrow}$ (8.1). We can reduce the goal relation of the initial configuration by removing $x \cdot x$, $y \cdot y$, or both, then apply gc and the resulting configuration still has a proof trace.

\[
\langle \{x \cdot x, y \cdot y\}, \{\}, \{x \mapsto (F y, F y), y \mapsto G x\}, \{x \mapsto (F y, F y), y \mapsto G x\} \rangle
\]

\[
\rightarrow \langle \{a \cdot b, y \cdot y\}, \{x \cdot x\}, \{y \mapsto G x, a \mapsto F y\}, \{y \mapsto G x, b \mapsto F y\} \rangle \quad \text{(PairAs)}
\]

\[
\rightarrow \langle \{\}, \{x \cdot x, a \cdot b, y \cdot y\}, \{\}, \{\} \rangle \quad \text{(FGs)}
\]

If we root the rewrite rules in the obvious way, so for example (Id, G) becomes:

\[
\langle \{b \cdot e\}, \{\}, \{b \mapsto G c\}, \{e \mapsto G f\} \rangle \rightarrow \langle \{c \cdot f\}, \{b \cdot e\}, \{\}, \{\} \rangle
\]

There is still a translation trace if the goal relation is $\{x \cdot x, y \cdot y\}$, $\{x \cdot x\}$ or $\emptyset$, but not if it is $\{y \cdot y\}$. \qed

One problem is that $\rightarrow$-rules may decompose the terms in graph nodes. So to reorder a proof trace to get \textit{SubTrace} $\rightarrow$ with rooted rules we would need to flatten the graphs first. A second problem is that rewrite rules could have non-linear left patterns, like (PairAs) in Proposition 8.1 and this places restrictions on the order in which rules can be applied. For this reason we will define our space relations by unrooted rules: it seems that an unrooted system with the GcTrace property is more flexible and easier to construct than a rooted system with the SubTrace property.
8.3.3 Higher-order graph translation

Care is required when translating higher-order graphs. The problem is the potential for name clashes between variables bound in different graph nodes during translation. Definition 4.5 ensures that no variable is re-bound in any sub-term in its scope. But it is possible for the same variable to be bound in two different nodes or in two separate sub-terms.

Configuration-rewrite rules construct a correspondence between variable names, so if bound variables are not renamed as they are translated name clashes can occur, causing the translation relation to be larger than intended.

Example 8.6 (Bound variable name clash problem)
Consider adding the rule \((\lambda \lambda)\) to the usual identity translation as shown below. It is like \((Id.\lambda)\) except that the bound variables \(x\) and \(y\) are not replaced by new variables. The example trace shows how with this omission to the \(\lambda\)-encoding of true can translate a \(\lambda\)-encoding of false.

\[
\begin{align*}
\text{bad} & \rightarrow \text{id} \cup \{(\lambda \lambda)\} \\
& \langle \{\}, \{a \mapsto \lambda x.E\}, \{b \mapsto \lambda y.F\} \rangle \rightarrow \langle \{e \cdot f\}, \{a \cdot b, x \cdot y\}, \{e \mapsto E\}, \{f \mapsto F\} \rangle (\lambda \lambda) \\
& \langle \{c \cdot d\}, \{\}, \{a \mapsto \lambda x.\lambda y.x, c \mapsto \lambda x.a x\}, \{b \mapsto \lambda w.\lambda z.\lambda z, d \mapsto \lambda z.b z\} \rangle \\
& \text{bad} \rightarrow \langle \{c \cdot d, e \cdot f\}, \{a \cdot b\}, \{e \mapsto \lambda y.x, c \mapsto \lambda x.a x\}, \{f \mapsto \lambda z.z, d \mapsto \lambda z.b z\} \rangle (\lambda \lambda) \\
& \text{bad} \rightarrow \langle \{c \cdot d, x \cdot z\}, \{a \cdot b\}, \{c \mapsto \lambda x.a x\}, \{d \mapsto \lambda z.b z\} \rangle (\lambda \lambda), (\text{Id.Var}) \\
& \text{bad} \rightarrow \langle \{g \cdot h\}, \{a \cdot b, x \cdot z\}, \{g \mapsto a x\}, \{h \mapsto b z\} \rangle (\lambda \lambda) \\
& \text{bad} \rightarrow \langle \{\}, \{\}, \{\}, \{\} \rangle (\text{Id.Apply}), (\text{Id.Var})
\end{align*}
\]

The terms at \(c\) and \(d\) are isomorphic, assuming that the terms at \(a\) and \(b\) are isomorphic. The terms at \(a\) and \(b\) are not isomorphic because \(x\) and \(y\) are bound at different places but the translation trace still reaches an end state because the variable pair \(x \cdot z\) is added to the correspondence relation when \(c\) and \(d\) are traversed. So \(\text{bad} \rightarrow \) is more than an identity.

Therefore our configuration-rewrite rule framework must include the facility to replace bound variables (by substitution) during translation so that the relations they define are not accidentally too large.

8.4 Proportional space relations

A Proportional Space Relation is a configuration-rewrite system whose definition satisfies the following restrictions.

Definition 8.7 (PSR rule)
A PSR rule is a configuration-rewrite rule which satisfies the restrictions in Figure 8.1.
A PSR rule \( \{ \}, \{ \}, P, Q \rightarrow (\Theta, \Phi, P', Q') \) is a \( G_1 - G_2 \) configuration-rewrite rule where:

1. \( G_1 \vdash \text{UnrootedPat } P : E \) \( G_2 \vdash \text{UnrootedPat } Q : F \)
2. \(#P \geq 1 \) \( #Q \geq 1 \)
3. \((a \mapsto T) \in P \Rightarrow T \notin \text{Holes} \) \((a \mapsto T) \in Q \Rightarrow T \notin \text{Holes} \)
4. \((h \mapsto (C, \beta, \theta)) \in E \Rightarrow \theta = [\] \) \((h \mapsto (C, \beta, \theta)) \in F \Rightarrow \theta = [\] \)
5. \( G_1 \vdash \text{UnrootedPat } P' : E' \) \( G_2 \vdash \text{UnrootedPat } Q' : F' \)
6. \( P' \subseteq \{ a \mapsto h\theta \mid h \in \text{holes } P, \)
   \( a \in \text{Fresh}, \)
   \( \theta : cv h E \rightarrow \text{Fresh} \}\)
   \( Q' \subseteq \{ a \mapsto h\theta \mid h \in \text{holes } Q, \)
   \( a \in \text{Fresh}, \)
   \( \theta : cv h F \rightarrow \text{Fresh} \}\)

where \( \text{Fresh} = \text{Var} - \text{var } PQ \)

7. \( \text{holes } P' \subseteq \{ h \in \text{holes } P \} \)
   \( \text{holes } Q' = \{ h \in \text{holes } Q \} \)
8. \( \Phi \subseteq \{ \text{dom } P \cup \{ \text{rng } \theta \mid (a \mapsto h\theta) \in P' \} \} \times \{ \text{dom } Q \cup \{ \text{rng } \theta \mid (a \mapsto h\theta) \in Q' \} \} \)
9. \( \text{dom } P \subseteq \text{reach}(P, \text{dom } \Phi) \)
   \( \text{dom } Q \subseteq \text{reach}(Q, \text{rng } \Phi) \)
10. \( \text{rng } \Theta \subseteq \text{fv } P \cup \text{dom } P' \)
   \( \text{dom } \Theta = \text{fv } Q \cup \text{dom } Q' \)
11. \( #P' \geq #P \Rightarrow #Q' - #Q = #P' - #P \)

1. \( P \) and \( Q \) are unrooted graph patterns. We do not write the required disequity constraints, they are added implicitly.
2. \( P \) and \( Q \) are non-empty.
3. Nodes in \( P \) and \( Q \) cannot just contain a hole.
4. Holes in \( P \) and \( Q \) do not have substitutions.
5. \( P' \) and \( Q' \) are unrooted graph patterns.
6. Nodes in \( P' \) and \( Q' \) may only contain holes. Each capture variable of a hole is substituted for. The substitutes and the domain of these patterns are fresh variables which are allocated when the rule is applied. The substitutes and domain do not overlap but the same variable may be substituted for several capture variables.
7. \( P' \) is linear in holes, all of which occurred in \( P \) and \( Q' \) contains one instance of each hole in \( Q \).
8. The correspondence relation \( \Phi \) contains pairs of variables whose binding occurrence is removed by this rule. These can be node addresses in \( P \) or \( Q \), or the variables substituted in \( P' \) or \( Q' \).
9. The correspondence relation \( \Phi \) is effectively the root of the pattern: all nodes in \( P \) and \( Q \) must be reachable from \( \Phi \).
10. The goal relation is \( \Theta \). Its domain can contain free variables in \( P \) and variables allocated by \( P' \); its range must include all of the free variables in \( Q \) and all of the variables allocated by \( Q' \) to guarantee that a successful proof trace covers the entire reachable part of graphs matching \( Q \).
11. If a rule increases the node count of the graph matching \( P \) then the node count of the graph matching \( Q \) increases by the same amount. This makes PSRs space relations.

Figure 8.1: Proportional Space Relation rule restrictions.
Example 8.7 (PSR rules)

Of the rules we have considered so far, (Branches), (Leaves), (Id.F), (ProjHead), (ProjTail), (ProjCons), (PairAs) and (FGs) are all PSR rules; (DeadLet), (DeadInd) and (DeadUdm) are not because their Q-patterns are empty; (λλ) is not because it does not substitute for the bound variables x and y. \[\Box\]

A Proportional Space Relation is a configuration-rewrite system whose rewrite rules are all PSR rules and whose start rule is the identity start rule (see Example 8.4). If a different start rule is needed, the properties of PSRs in Section 8.5 must still hold for our proof method to work.

The various restrictions in Definition 8.7 guarantee that a PSR is a terminating rewrite system and a space relation. Intuitively, each rewrite in a PSR proof trace showing \(G \Rightarrow H\) decreases the size of the graphs in a configuration: a successful proof trace covers all the reachable nodes in \(H\) and only reachable nodes in \(G\). Example 8.8 shows how PSR rewriting works in practice. A good intuition is to think of the proof trace as labelling all the subexpressions in the graph components of a configuration as it rewrites them, and pairing up addresses in such a way that each address only occurs in a bounded number of pairs.

Example 8.8 (Constructor translation)

We can relate the behaviour of \([\text{if}]\) and \([\text{lazy}]\) by specifying a PSR \(\xrightarrow{\text{IfApps}}\) which translates boolean constructors and if-expressions into untyped λ-calculus terms. We extend the identity on expressions with two rules to convert boolean values into the appropriate combinators and one to convert an if-expression into function applications.

\[
\xrightarrow{\text{IfApps}} = (\text{start}, \{(\text{Id}\lambda), (\text{Id} \text{Apply}), (\text{Id} \text{Var}), (\text{Id} \text{let}), (\text{Id} \perp), (\text{True} \lambda), (\text{False} \lambda), (\text{IfApps})\})
\]

\[
\begin{align*}
\langle \{\}, \{\}, \{a \mapsto \text{True}\}, \{b \mapsto \text{λ}t.\text{λ}f.t\} \rangle &\Rightarrow \langle \{\}, \{a \cdot b\}, \{\}, \{\} \rangle & (\text{True} \lambda) \\
\langle \{\}, \{\}, \{a \mapsto \text{False}\}, \{b \mapsto \text{λ}t.\text{λ}f.t\} \rangle &\Rightarrow \langle \{\}, \{a \cdot b\}, \{\}, \{\} \rangle & (\text{False} \lambda) \\
\langle \{\}, \{\}, \{a \mapsto \text{if} X y z\}, \{b \mapsto (U v) w\} \rangle &\Rightarrow \langle \{x \mapsto X, y \mapsto U, z \mapsto w\}, \{a \cdot b\}, \{\}, \{\} \rangle & (\text{IfApps})
\end{align*}
\]

A translation trace showing that the initial graph in Example 4.12 translates to

The PSR rule restrictions can be generalised in various ways without losing the properties of interest. However, for simplicity, and because they are sufficient for the proofs presented here, we prefer the restrictions of Definition 8.7. As an example of a safe generalisation, consider the rule (8.2) for translating between different representations of a list of boolean values (this is similar to an example we used in \([\text{BR00a}]\)).

\[
\begin{align*}
\langle \{\}, \{\}, \{x \mapsto \text{True} : y\}, \{a \mapsto b : c, b \mapsto \text{True}\} \rangle &\Rightarrow \langle \{y \cdot c\}, \{x \cdot a\}, \{\}, \{b \mapsto \text{True}\} \rangle & (8.2)
\end{align*}
\]

Using (8.2), the boolean value at node b may be shared by many elements of a list, but this rule still defines a space relation because the number of elements of each list must be the same in both graphs. There is no asymptotic gain in sharing the list elements.
\[ \text{start } \{ a \mapsto \text{let } true = \text{True in (let } false = \text{False in } (\lambda x. \text{if } x \text{ false true}) \text{ true}) \} a, c \}
\{ a \mapsto \text{let } true = \lambda x. \lambda y. x \text{ in } (\text{let } false = \lambda x. \lambda y. y \text{ in } (\lambda x. \text{false true}) \text{ true}) \} a, c \}
\]
\[
= \begin{cases}
\{ a \mapsto \text{let } true = \text{True in (let } false = \text{False in } (\lambda x. \text{if } x \text{ false true}) \text{ true}) \}, \\
\{ a \mapsto \text{let true } = \lambda x. \lambda y. x \text{ in } (\text{let } false = \lambda x. \lambda y. y \text{ in } (\lambda x. \text{false true}) \text{ true}) \}, \\
\{ b \mapsto e \cdot e, \{ b \cdot f \}, \\
\{ e \mapsto \text{let } true = \text{True in (let } false = \text{False in } (\lambda x. \text{if } x \text{ false true}) \text{ true}) \}, \\
\{ e \mapsto \lambda x. \lambda y. x \text{ in } (\text{let } false = \lambda x. \lambda y. y \text{ in } (\lambda x. \text{false true}) \text{ true}) \}, \\
\{ b \cdot g, e \cdot e, \{ c \cdot f \}, \\
\{ d \mapsto \text{let } false = \text{False in } (\lambda x. \text{if } x \text{ false c}) \text{ c} \}, \\
\{ e \mapsto \text{let } true = \text{True in (let } false = \text{False in } (\lambda x. \text{if } x \text{ false true}) \text{ true}) \}, \\
\{ b \mapsto True, d \mapsto \text{let } false = \text{False in } (\lambda x. \text{if } x \text{ false true}) \text{ true}) \}, \\
\{ e \mapsto \lambda x. \lambda y. x \text{ in } (\text{let } false = \lambda x. \lambda y. y \text{ in } (\lambda x. \text{false true}) \text{ true}) \}, \\
\{ b \mapsto True, d \mapsto \text{let } false = \text{False in } (\lambda x. \text{if } x \text{ false true}) \text{ true}) \}, \\
\{ e \mapsto \lambda x. \lambda y. x \text{ in } (\text{let } false = \lambda x. \lambda y. y \text{ in } (\lambda x. \text{false true}) \text{ true}) \}, \\
\{ e \mapsto \text{let } false = \text{False in } (\lambda x. \text{if } x \text{ false true}) \text{ true}) \}, \\
\{ c \cdot f, b \cdot i \}, \\
\{ e \mapsto \lambda x. \lambda y. x \text{ in } (\text{let } false = \lambda x. \lambda y. y \text{ in } (\lambda x. \text{false true}) \text{ true}) \}, \\
\{ e \mapsto \text{let } false = \text{False in } (\lambda x. \text{if } x \text{ false true}) \text{ true}) \}, \\
\{ c \cdot f, b \cdot i \}, \\
\{ e \mapsto \text{let } false = \text{False in } (\lambda x. \text{if } x \text{ false true}) \text{ true}) \}, \\
\{ e \mapsto (\lambda x. \text{if } x \text{ b c}) \text{ c} \}, \\
\{ j \mapsto (\lambda x. \text{i f}) \text{ f} \}, \\
\{ a \mapsto e \cdot e, \{ e \cdot f, b \cdot i \}, \\
\{ e \mapsto \lambda x. \lambda y. x \text{ in } (\text{let } false = \lambda x. \lambda y. y \text{ in } (\lambda x. \text{false true}) \text{ true}) \}, \\
\{ b \cdot h, e \cdot e, \{ c \cdot f, b \cdot i, g \cdot j \}, \\
\{ e \mapsto \text{if } g \text{ b c} \}, \\
\{ h \mapsto j i f \}, \\
\{ a \mapsto \text{if } g \text{ b c} \}, \\
\{ h \mapsto j i f \}, \\
\{ a \mapsto g \}, \\
\{ d \mapsto j \}, \\
\{ c \cdot e \}, \{ e \cdot e \}, \{ g \cdot j \}, \\
\{ } \}
\}
\]

Figure 8.2: An example PSR rewrite trace. It begins by applying start to re-encode two graphs as a configuration. Eight PSR rewrite steps follow. These could occur in several different orders with the same result, for example, the (True \lambda) step could exchange places with any later step with the same result. After each rewrite step the configuration garbage collector gc is applied, so a \cdot a is removed after the first step because there are no references to a left in the graphs. The final configuration does not satisfy end because it still contains e \cdot e so the trace does not verify that the two original graphs are translations of each other.
To obtain the $GcTrace$ property requires a further restriction. Not on individual rules, but on the rules defining a PSR as a whole. A self-consistent PSR guarantees that if any step in a trace involves graph nodes that $gc$ can remove, then there is a rule that can be applied which only involves the nodes left by the collector, and therefore the garbage-collected configuration has a proof trace. This is explained in more detail in Proposition 8.3.

**Definition 8.8 (Self-consistent PSR)**
A PSR $\Rightarrow$ is self-consistent if for every rule $\langle \{\}, \{\}, P, Q \rangle \Rightarrow \langle \Theta, \Phi, P', Q' \rangle$ in $\Rightarrow$, for every $\Phi' = \Phi - \{xy\}$ where $x \in dom P$ and $y \in dom Q$ and $\Phi' \cap dom P \times dom Q \neq \emptyset$, the sub-rule $\langle \{\}, \{\}, P_s, Q_s \rangle \Rightarrow \langle \Theta_s, \Phi_s, P'_s, Q'_s \rangle$ is also part of the definition of $\Rightarrow$ where:

\[
\begin{align*}
P_s &= P|\text{reach}(P, \text{dom } \Phi') \\
Q_s &= Q|\text{reach}(Q, \text{rng } \Phi') \\
\Theta_s &= \{w \cdot z \in \Theta \mid z \in \text{var } (Q_s \cup Q'_s)\} \\
\Phi_s &= \{x \cdot y \in \Phi \mid x \in \text{var } (P_s \cup P'_s), y \in \text{var } (Q_s \cup Q'_s)\} \\
P'_s &= \{\{a \mapsto h\theta\} \in P'|h \in \text{holes } P_s\} \\
Q'_s &= \{\{a \mapsto h\theta\} \in Q'|h \in \text{holes } Q_s\}
\end{align*}
\]

\[\square\]

**Example 8.9 (Self-consistent PSR)**
The PSR $\Rightarrow$ of Proposition 8.1 is self-consistent. For example:

$\{x \mapsto F y, y \mapsto G y\} y, y \Rightarrow \{x \mapsto F y, y \mapsto G y\} y, y.$

And thanks to self-consistency:

$gc \{x \mapsto F y, y \mapsto G y\} y, y \Rightarrow gc \{x \mapsto F y, y \mapsto G y\} y, y.$

That is, $\{y \mapsto G y\} y, y \Rightarrow gc \{x \mapsto F y, y \mapsto G y\} y, y$. Clearly this would not work if we destroyed self-consistency by removing $(Id, G)$ from $\Rightarrow$.

An extension of GraphKit reads and checks PSR rule definitions (see Appendix A). The user-defined PSR does not have to be self-consistent: any additional rules needed are derived automatically.

### 8.4.1 Translating null terms

To translate null terms and variables we define the following rules which are implicitly included in every PSR.

**Definition 8.9 (Null translation rules)**

$\langle \emptyset \{\cdot \cdot \cdot \}, \phi, G, H \rangle \Rightarrow \langle \emptyset, \phi, G, H \rangle$ (Trans$\emptyset$)

$\langle \emptyset, \phi, G \{a \mapsto \mathcal{E}\}, H \{b \mapsto \mathcal{E}\} \rangle \Rightarrow \langle \emptyset, \phi \{a \cdot b\}, G, H \rangle$ (Trans$\mathcal{E}$)

The (Trans$\emptyset$) rule removes pairs of null variables that find their way into the goal relation (they cannot ever occur in the correspondence relation). Null terms are removed by (Trans$\mathcal{E}$); its definition is the same as $(Id, \mathcal{E})$.

\[\square\]

**Example 8.10 (Translating graphs with nulls)**

With the null translation rule, the final configuration in Figure 8.2 can be rewritten to $\langle \{\}, \{\}, \{\} \rangle$ which satisfies end, so now we have a proof of:
\{a \mapsto \text{let } true = True \text{ in } (\text{let } false = False \text{ in } (\lambda x. \text{if } x \text{ false } true) \text{ true})\}a, \epsilon

\vartriangleright

\{a \mapsto \text{let } true = \lambda x. \lambda y. x \text{ in } (\text{let } false = \lambda x. \lambda y. y \text{ in } (\lambda x. \text{false } true) \text{ true})\}a, \epsilon

\square

8.5 Properties of Proportional Space Relations

PSRs are designed to have the SubTrace and GcTrace properties discussed in Section 8.3.

Proposition 8.2 (PSRs rules guarantee SubTrace )

If a configuration-rewrite relation \( \rightarrow \) is defined by PSR rules then SubTrace \( \rightarrow \epsilon \):

\[ \exists \Omega \cdot \langle \theta \cup \theta', \phi, G, H \rangle \rightarrow^* \Omega' \Rightarrow \exists \Omega' \cdot \langle \theta, \phi, G, H \rangle \rightarrow^* \Omega' \]

Proof

Both traces involve the same steps in the same order. Let the left trace including \( \theta' \) be \( \langle \theta_0, \phi_0, G_0, H_0 \rangle \rightarrow r_1 \cdots \rightarrow r_n \langle \theta_n, \phi_n, G_n, H_n \rangle \). Map this trace onto the right trace by removing any pairs \( x \cdot y \in \theta' \) from all \( \theta_i \). If some step \( j \) re-introduces \( x \cdot y \), that is, \( r_j \) is defined \( \Delta \rightarrow \langle \Theta, \Phi, P', Q' \rangle \) and \( C(\langle \Theta, \Phi, P', Q' \rangle) = \langle \theta_j, \phi_j, G_j, H_j \rangle \) where the context \( C \) is defined \( C(\langle \theta, \phi, p, q \rangle) = \langle \theta' \cup T(\theta), \phi' \cup T(\phi), p \cup T(p), q \cup T(q) \rangle \) and \( x \cdot y \in T(\Phi) \) then just remove \( x \cdot y \) from all \( \theta_i \) where \( i < j \). \( \square \)

Proposition 8.3 (Self-consistent PSRs rules guarantee GcTrace)

If a configuration-rewrite relation \( \rightarrow \) is defined by PSR rules then GcTrace \( \rightarrow \epsilon \):

\[ \exists \Omega \cdot \Delta \rightarrow^* \Omega \Rightarrow \exists \Omega' \cdot gc \Delta \rightarrow^* \Omega' \]

Proof

Map the trace of a configuration \( \Delta_n \) onto a new trace of \( gc \Delta_n \) as explained below. All nodes involved in the rewrite at any step of the new trace are reachable, therefore the new trace is also a valid trace for the garbage-collected configuration (note that \( \Delta_i \) is short for \( \langle \theta_i, \phi_i, P_i, Q_i \rangle \)).

Begin with the trace of a configuration \( \Delta_n \): \( \{ \Delta_i \xrightarrow{r_i} \}_{i=1}^n \Delta_0 \). Let \( T = \{ \Delta_0 \} \) be the first approximation to the new trace.

1. For an approximate trace \( T = \{ \Delta_i \xrightarrow{r_i} \}_{i \in I} \Delta_0 \) where \( I \subseteq \{1, \ldots, n\} \): if end \( \Delta_0 \) then \( T \) is the new trace, otherwise generate the next approximation to the new trace as follows.

2. Take any \( x \cdot y \in \theta'_0 - \phi'_0 \) and find the step \( j \) which removes \( x \cdot y \) in the original trace: \( x \cdot y \in \phi_{j-1} - \phi_j \).

3. Add step \( j \) to the new trace \( T \):

i. If \( j \notin I \) then we add a new step: If \( r_j \) is defined \( \Delta \rightarrow \langle \Theta, \Phi, P', Q' \rangle \) and there is a context \( C \) such that \( C(\Delta) = \Delta_j \) then the new step uses \( r'_j \) which is defined as the sub-rule of \( r_j \) which just includes those nodes reachable from \( x \cdot y \). This rule exists by self-consistency (Definition 8.8). So let \( r'_j \) be defined \( I' \rightarrow R' \). Update \( T \) to
\( \{ \Delta'_i \xrightarrow{r'_j} | i \in I', i > j \} \{ \Delta''_i \xrightarrow{r'_j} | i \in I', i \leq j \} \Delta''_m \) where \( I' = I \cup \{ j \} \), and \( \Delta''_m = \Delta'_m \) where \( m = \max \{ i \in I' | j \leq i \} \). Configurations \( \Delta''_m \) where \( i < j \) are obtained by applying the new rewrite step \( r'_j \) to \( \Delta'_i \): \( \Delta''_m = \Delta'_m \land C(I') \cup C(R') \).

ii. If \( j \in I \) then a step \( j \) already exists. We extend it to use a new \( r'_j \) which is a sub-rule of \( r_j \) including the old \( r'_j \) and the \( r'_j \) described above. The next approximation to \( T \) is also obtained as above.

Every variable in \( \Delta'_0, \theta \) is reachable from \( \Delta_0, \theta \) because the sub-rules in the new trace \( T \) only add variables to \( \Delta'_0, \theta \) that are reachable from variables already in \( \Delta'_0, \theta \). Therefore the new trace can also be applied to \( gc \Delta_0 \). Further, by the PSR rule definition, no reachable nodes in \( \Delta_0.H \) are left in \( \Delta'_0.H \).

PSRs define space relations. Before the main proof we give a couple of key properties of the \( \text{start} \) rule and the configuration garbage collector: The \( \text{start} \) rule does not discard any reachable nodes when it turns two graphs into a configuration (Proposition 8.4) and PSR rules are not garbage-generating in the fourth element of configurations (Proposition 8.5). Therefore a PSR rewrite trace for \( G \implies H \), which leads to an \( \text{end} \) state, traverses the entire live part of \( H \). The trace length bounds the size of the live part of \( H \). Proposition 8.6 uses these facts to show that PSRs are space relations.

**Proposition 8.4 (Distributivity of \( gc \) over \( \text{start} \))**

\[ gc(\text{start}(G, H)) = \text{start}(gc G, gc H) \]

**Proof**

From the definitions of \( gc \) for configurations and graphs:

\[ gc(\text{start}(Gx_1, \ldots , x_n, Hy_1, \ldots , y_n)) = gc(\{x_i \mid i \leq n\}, G, H) \]

\[ = (\{x_i \mid i \leq n\}, G[\text{reach}(G, \{x_i \mid i \leq n\}), H[\text{reach}(H, \{y_i \mid i \leq n\})]) \]

\[ = \text{start}(G[\text{reach}(G, \{x_i \mid i \leq n\}), H[\text{reach}(H, \{y_i \mid i \leq n\}))x_1, \ldots , x_n, (H[\text{reach}(H, \{y_i \mid i \leq n\}))y_1, \ldots , y_n) \]

\[ = \text{start}(gc Gx_1, \ldots , x_n, gc Hy_1, \ldots , y_n) \]

**Proposition 8.5 (PSR garbage-generation property)**

If \( r \) is a PSR rule and \( (gc \Delta) \xrightarrow{r} (\theta, \phi, G, H) \) then \( H = H[\text{reach}(H, \text{rng}\theta)] \)

**Proof**

Let \( \Delta = (\theta, \phi, G', H') \) and \( r \) be the rule \( (\theta, \phi, G', H') \).

We know \( H' = H'[\text{reach}(H', \text{rng}\theta)] \) and there is a graph context \( \mathbb{H}(Q') \) such that \( \mathbb{H}(Q) \equiv H' \) and \( \mathbb{H}(Q') \equiv H \). By Definition 8.7, \( f v \ Q = f v \ Q' \) and \( \text{holes} \ Q = \text{holes} \ Q' \) and \( Q' \) is reachable from \( \text{rng} \ \theta \). Therefore \( H = H[\text{reach}(H, \text{rng}\theta)] \).

**Proposition 8.6 (PSRs are space relations)**

If \( \implies \) is a PSR then \( \exists \forall G, \ H \cdot G \implies H \Rightarrow k \times \# gc \ G \geq \# gc \ H \).

**Proof**

Let \( k = \max \{ 1 + (\# Q - \# Q') \div (\# P - \# P') \} \implies (\text{start}, \xrightarrow{\phi}, \{\}, \{\}, P, Q) \xrightarrow{\rightarrow} (\theta, \phi, P', Q') \in \rightarrow \)

If \( G \implies H \) then \( \text{start}(G, H) \xrightarrow{\phi} \Omega \land \text{end} \ \Omega \). \( \#(gc \ G) = \# G' \) and \( \#(gc \ H) = \# H' \) where \( \langle \theta, \phi, G', H' \rangle = gc(\text{start}(G, H)) \) by Proposition 8.4.

We require \( \langle \theta, \phi, G, H \rangle = gc \Delta \land \Delta \xrightarrow{\phi} \Omega \land \text{end} \ \Omega \Rightarrow k \times \# G \geq \# H \).
1. If $\text{end } \langle \theta, \phi, G, H \rangle$ then $\#H = 0$ by the definition of end and gc, therefore $k \times \#G \geq \#H$.

2. Otherwise: $\langle \theta, \phi, G, H \rangle \xrightarrow{r} \Delta$ and $\langle \theta', \phi', G', H' \rangle = gc \Delta$ where the rule $r \in \Im$ is defined $\langle \{\}, \{\}, P, Q \rangle \rightarrow \langle \Theta, \Phi, P', Q' \rangle$. By Definition 8.7 and Proposition 8.5: $\#G \geq \#G' + \#P - \#P'$ and $\#H = \#H' + \#Q - \#Q'$. By hypothesis $k \times \#G' \geq \#H'$ and by Definition 8.7 $k \times (\#P - \#P') \geq (\#Q - \#Q')$. Therefore $k \times \#G \geq \#H$ as required. 

\[\square\]

Example 8.11 (Calculating the space bound)

For the translation $\xrightarrow{\mathcal{A}}$ we find $k = 2$ in Proposition 8.6. In this case the addition of 1 in the calculation of $k$ (to avoid the rounding down in the definition of $\div$ making $k$ too small) is unnecessary — the exact value of $k$ is 1 so $\xrightarrow{\mathcal{A}}$ preserves graph cardinality. 

\[\square\]

PSR configuration rewriting is guaranteed to terminate because each step reduces the size of the graph components of a configuration.

Proposition 8.7 (Termination of PSR rewriting)

If $G$ and $H$ are finite graphs then all proof traces of $\text{start } (G, H)$ are finite.

\[\text{Proof}\]

If $\langle \theta, \phi, G, H \rangle \rightarrow \langle \theta', \phi', G', H' \rangle$ then $\text{size } G' < \text{size } G \land \text{size } H' < \text{size } H$ by Definition 8.7. 

\[\square\]

The proof traces showing $G \implies H$ may all terminate but finding them all can be a lengthy process. In the worst case, where the size of $G$ and $H$ are $O(n)$ and the PSR has $r$ rules, there may be $O(r \times n^{2 \times n})$ traces because any of the rules might be applied to pair any node in $G$ with any node in $H$ giving $r \times n^2$ possible first moves. In fact there may be even more if any PSR rule has more than one node in its left patterns, but we can think of this as a sequence of several one-node rules for the complexity analysis.

Increasing the number of rules in a PSR definition increases the complexity of PSR proofs (and the no-leak proofs in following chapters). It is advantageous to keep the number of rules and the rule definitions as small as possible. As well as checking for equality modulo renaming, we can exploit the following addition property.

Proposition 8.8 (PSR rule addition)

If $L \rightarrow R$ and $L' \rightarrow R'$ are PSR rules and the variables in $LR$ are disjoint from those in $L'R'$ then $LL' \rightarrow RR'$ is also a PSR rule where juxtaposition denotes component-wise union of configuration patterns.

\[\text{Proof}\]

Follows from the PSR rule definition. 

\[\square\]

The significance is that if $L \rightarrow R$, $L' \rightarrow R'$ and $LL' \rightarrow RR'$ are all part of a PSR definition we can eliminate $LL' \rightarrow RR'$ without shrinking the PSR. This kind of redundancy is quite likely after any extra rules needed for self-consistency have been derived, particularly where the PSR rules are inferred as in Chapter 10.
8.6 Summary

A space relation is a translation which does not increase graph size by more than some fixed constant factor. The Proportional Space Relation framework is designed for the specification of space relations. We have shown that there is a strong correspondence between the idea of a space relation and copying garbage collection; GraphKit allows non-standard collectors to be specified in this way. PSRs allow space relations to be specified in a local and compositional manner: a graph is translated piece by piece instead of having to capture the whole relationship in a single equation. PSRs are also equipped with several important properties which will simplify the no-leak proofs.
Chapter 9

Checking No Leak

This chapter describes how to check that an evaluator $\text{A}$ is not leakier than another evaluator $\text{B}$ with respect to a translation relation $\iff$. A simulation-like correctness proof technique is adapted so that it also guarantees the required space bound. The main restriction is that $\iff$ must be defined as a PSR so we can exploit its properties to complete the proof.

Section 9.1 introduces the restricted form of simulation we use for automated checking. Section 9.2 explains how we present a proof as a collection of simulation squares. Section 9.3 describes how a PSR is used to translate a graph or a graph pattern and hence how the top of a square is constructed. Section 9.4 describes the information other than the evaluator and translation definitions which is needed to complete a square. The main running example is a proof of $\text{badbb} \iff \text{lazy}$.

9.1 Simulation proofs

A simulation proof is the basic technique in structural semantics for showing an operational semantics is correct relative to another: $\text{A}$ simulates $\text{B}$ if whenever $\text{B}$ evaluates a program to a value, $\text{A}$ also evaluates that program to the same value. Including the idea of a translation gives us Definition 9.1.

**Definition 9.1 (Simulation, mutual simulation)**

$\text{A}$ simulates $\text{B}$ with respect to translation $\iff$ if:

$$\forall G, H, H' \cdot G \iff H \land H \not\xrightarrow{\text{B}} \not\xrightarrow{\text{B}} \implies \exists G' \cdot G \xrightarrow{\text{B}} G' \land G' \iff H'.$$

We may call $\iff$ a simulation.

If $\text{B}$ also simulates $\text{A}$ with respect to $\iff^{-1}$ then $\iff$ is a mutual simulation. □

The empty translation is the simplest example of a simulation. When simulation is used to define operational equality, $\iff$ must be the largest possible relation. Abramsky [Abr90] defines bisimilar λ-calculus terms like this as the greatest fix-point of a certain equation. When simulation is used for correctness proofs, $\iff$ must include enough programs. At the very least, all possible initial program states should be included. For our call-by-need evaluators based on $\text{lazy}$, we define the initial states by the translation $\text{idX}$.\[\text{Definition 9.2 (idX initial states for call-by-need evaluators)}

The valid initial states of call-by-need evaluators are defined by the translation $\text{idX}$,
specified by the following identity PSR rules. It includes any graph constructed only from \( \bot \)-less expression terms.

\[
\begin{align*}
\text{id} & \iff (\text{start}, \emptyset) \\
\text{id} & \rightarrow \{(\text{Id.Var}), (\text{Id.\lambda}), (\text{Id.Apply}), (\text{Id.let})\}
\end{align*}
\]

Therefore, to be convincing, the translation used in a call-by-need evaluator comparison should be a superset of \( \rightarrow \). Any control-structure terms can be included as necessary. For example, in principle we might expect \( \text{ind} \) — an evaluator which uses indirections — to be leakier than \( \text{lazy} \) because we could define a graph containing a long chain of indirections. By excluding such graphs from the translation we may still be able to argue that \( \text{ind} \Rightarrow \text{lazy} \).

### 9.1.1 Lockstep simulations

A translation \( \iff \) that is also a space relation guarantees that \( \#G \) bounds \( \#H \) whenever \( G \iff H \). Proving that \( \iff \) is a simulation would tell us that the size of the normal form of \( G \) bounds the size of the normal form of \( H \). This is not quite enough to prove that there is no space leak. A possible scenario is that \( G \) runs in constant space while \( H \) continues to grow as it evaluates but near the end of its evaluation a large part of it becomes garbage and the final graphs have the same size. In the case where \( G \) and \( H \) are non-terminating, the growth of \( H \) could continue indefinitely.

We can be sure that each step in the evaluation of \( H \) allocates no more than some constant amount of space \( k \). Therefore we can choose some \( s \) and be sure that any sequence of up to \( s \) evaluation steps increases graph size by no more than \( k \times s \). This observation motivates the \( s \)-lockstep simulation of Definition 9.3 which can be used to prove no leak by Proposition 9.1.

**Definition 9.3 (s-lockstep simulation)**

\[\Box \] \( s \)-lockstep simulates \( \Box \) with respect to translation \( \iff \) if:

\[
\forall G, H, H^s \cdot G \iff H \Rightarrow H \Rightarrow H^n_g \Rightarrow 1 \leq n \leq s
\]

Simulation is restricted by \( s \)-lockstep simulation as follows. If \( H \) is the translation of \( G \) and \( H \) is a redex then there must be a graph \( H' \) in the evaluation trace of \( H \) which is the translation of some graph \( G' \) in the evaluation trace of \( G \). The graph \( G' \) may occur any number of evaluation steps after \( G \) including 0; \( H' \) must be at least one evaluation step after \( H \) otherwise every translation would trivially satisfy the definition; \( H' \) must occur after no more than \( s \) steps to guarantee that the space bound is preserved.

**Proposition 9.1 (s-lockstep space-relation simulations guarantee no leak)**

If \( \iff \) is a space relation and there is an \( s \in \mathbb{N} \) such that \( \Box \) \( s \)-lockstep simulates \( \Box \) with respect to \( \iff \) then \( \Box \Rightarrow \Box \) wrt \( \iff \).

**Proof**

We require \( \exists k \cdot G \iff H \Rightarrow k \times \text{space} G \Rightarrow \text{space} H \). We know there is a \( k \) such
that $G \implies H \Rightarrow k_t \times \#G \geq \#H$ and a $k_a = \max\{\#R - \#L|L \rightarrow R| \in \mathcal{I}\}$.
Let $k = k_t \times (1 + k_a \times s)$. If $H \not\rightarrow_B$ it follows that $k \times \text{space } G \mathcal{A} \geq \text{space } H \mathcal{B}$. Otherwise $\exists H^n \cdot H \not\rightarrow_{g^B} H^n$ and by Definition 9.3:

$$\exists G', H' \cdot G \not\rightarrow_{g^A} G' \land H \not\rightarrow_{g^B} H' \land G' \implies H' \land 1 \leq n \leq s$$

From the definition of evaluation, $\text{space } G \mathcal{A} \geq \max\{\#G, \text{space } G' \mathcal{A}\}$ and:

$$\forall H^n \cdot H \not\rightarrow_{g^B} H^n \not\rightarrow_{g^B} H' \Rightarrow \#H^n \leq \#H + k_a \times s \text{ where } 0 \leq n \leq m$$

So $\text{space } H \mathcal{B} \leq \max\{\#H + k_a \times s, \text{space } H' \mathcal{B}\}$. Also, $k \times \#G \geq (\#H + k_a \times s)$ and by hypothesis, $k \times \text{space } G' \mathcal{A} \geq \text{space } H' \mathcal{B}$. Therefore $k \times \text{space } G \mathcal{A} \geq \text{space } H \mathcal{B}$ as required. \hfill \square

With suitable insights about the evaluators under comparison we can often construct $s$-lockstep proofs by hand. But we aim to find such a proof automatically and clearly this is not possible without some restrictions.

To make the search for a proof terminate, we restrict the number of steps $\mathcal{A}$ and $\mathcal{B}$ may take between pairs of graphs in the simulation. In the simplest cases the number of steps can be restricted to one. Given an $s$-lockstep simulation, we can always convert it into a 1-lockstep simulation. So restricting $\mathcal{B}$ is no restriction in theory, but in practice this conversion may not always be very convenient. This gives us the lockstep simulation of Definition 9.4. Relaxing the lockstep restriction is investigated in Section 10.3.

**Definition 9.4 (Lockstep simulation)**

\begin{itemize}
    
    $\mathcal{A}$ lockstep simulates $\mathcal{B}$ with respect to translation $\implies$ if:

    $\forall G, H, H' \cdot G \implies H \land H \not\rightarrow_{g^B} H' \Rightarrow \exists G' \cdot G \not\rightarrow_{g^A} G' \land G' \implies H'$. \hfill \square

    Lockstep simulation restricts 1-lockstep simulation so that if $G$ translates to $H$, the evaluation trace of $G$ is at least as long as the evaluation trace of $H$ and the $i$th graph in the trace of $H$ is the translation of the $i$th graph in the trace of $G$.

\end{itemize}

**9.2 Simulation proof squares**

We present a lockstep simulation proof as a set of **proof squares**. Conceptually there is a square for each $G, H$ and $H'$ such that $G \implies H$ and $H \not\rightarrow_{g^B} H'$. The square contains the information needed to show that the lockstep simulation equation (Definition 9.4) is satisfied for these graphs. The general form of a proof square is shown below.

$$
\begin{align*}
    \Diamond & \quad \rightarrow^* \quad \Diamond^* \\
    (r) & \downarrow (r') \\
    \Downarrow & \quad \Downarrow
\end{align*}
$$

As PSRs are defined by configuration rewrite systems, each corner is a configuration. The sides are the transitions involved in the lockstep-simulation equation.

- The configuration $\Diamond \ast \Diamond$ is $\text{start}(G, H)$.
- The top side of the square represents the proof of $G \implies H$ and it can be any one of the many possible configuration rewrite traces.

- The configuration $\Delta^*$ is the result of this rewrite trace, an end state.

- The left side is the evaluation step (with garbage collection) in which $(r)$ is the $[\mathcal{A}]$-rule which rewrites $G$ to $G'$ and $(r')$ is the $[\mathcal{B}]$-rule which rewrites $H$ to $H'$. Because our evaluators are deterministic there is only one possible $G'$ and $H'$. In general there might be many and we would need a separate square for each combination of possible $G'$ and $H'$ graphs.

- The configuration $\triangledown$ is $\text{start} (G', H')$.

- The bottom of the square is the proof of $G' \implies H'$, another configuration rewrite trace. Again, only one trace is needed.

- The configuration $\triangledown \ast$ is the result of this rewrite trace.

Constructing proofs in exactly this form could give an unlimited number of squares because $\implies$ is not usually finite. So instead, the corners of our proof squares are configuration patterns and we construct a (hopefully small) set of squares for each rule $(L \rightarrow R) \in \mathcal{M}$ which represent the squares for all instances of $L$. The steps in proof construction are as follows.

1. Find a finite representation of the square tops for $L$. That is, $\{\Delta_i \rightarrow^* \Delta^*\}_{i=1}^n$ such that if $G \implies H$ and $H$ is an instance of $L$, then $\text{start} (G, H)$ is an instance of some $\Delta_i$. Now $\Delta^*$ is not an end state, it is just a configuration in a trace beginning at $\Delta_i$. Square-top construction (alias pattern translation) is explained in Section 9.3.

2. For each square top, evaluate its top-left configuration to find the bottom-left configuration as described before: let $(G, H) = \text{start}^{-1}(\Delta_i)$, then $\triangledown_i = \text{start} (G', H')$ where $G \rightarrow \mathcal{A} G'$ and $H \rightarrow \mathcal{B} H'$. The order of variable pairs in a configuration goal relation is significant for these operations. If $G$ is not an $\mathcal{A}$-redex the whole proof fails because $\implies$ cannot be a lockstep simulation. Do not garbage collect $\triangledown$ because the SubTrace and GcTrace properties of PSRs guarantee that the proof without any garbage collection represents the proofs of all its possible instances with garbage collection.

3. We require that an instance of $\triangledown$ rewrites to an end state whenever the same instance of $\Delta^*$ rewrites to an end state. This is formalised by Definition 9.5. And it is the arrow that forms the right side of a proof square. In the simplest cases $\triangledown_i$ can be rewritten to $\Delta_i^*$ with PSR rules and the square right side is trivial. This rewrite trace forms the bottom of the square and gives us $\triangledown_i$. In more complex cases $\Delta^*$ and $\triangledown_i$ are different and a special rule is needed to complete a square. We often need some extra rules to complete squares; technically these rules are part of the configuration implication arrow in the right side, but we include them in the rewrite on the bottom to make the squares easier to read. Such rules and the square right-side rule are explained in Section 9.4.
Definition 9.5 (Configuration implication)
\[ \Delta \Rightarrow \Delta' \]
if \( C(\Delta) \rightarrow^* \Omega \land \text{end } \Omega \Rightarrow \exists \Omega' \cdot C(\Delta') \rightarrow^* \Omega' \land \text{end } \Omega' \)

Configuration \( \Delta \) implies configuration \( \Delta' \) if whenever an instance of \( \Delta \) rewrites to an end state, the same instance of \( \Delta' \) rewrites to an end state. \( \square \)

Proposition 9.2 (Black holing does not cause a leak)
Through this chapter the examples prove \( \textbf{badbh} \) \( \not\equiv \textbf{lazy} \) : black-holing variables cannot introduce a space leak. The proof shows that the translation \( \textbf{badbh} \) defined below is a lockstep simulation.

\[
\textbf{badbh} \mapsto (\text{start}, \text{idX} \cup \{ (\text{Id} .) , (\text{Id} .) , (\text{Var} \perp) \}) \]

\[
\langle \{ \}, \{ \}, \{ a \mapsto x \} , \{ b \mapsto \perp \} \rangle \mapsto \langle \{ \} , \{ \} , \{ a \cdot b \} , \{ \} \rangle \quad (\text{Var} \perp)
\]

The difference between the two evaluators is that \( \textbf{badbh} \) does not overwrite a variable with \( \perp \) while its value is found. Therefore \( \mapsto \) is defined as an identity on the function symbols of the grammar except for \( \perp \) which translates to any variable by \( (\text{Var} \perp) \). The space-bound constant factor \( k \) associated with \( \textbf{badbh} \) is 1.

The square (9.2) for the (Lookup) redexes of \( \textbf{lazy} \) is straightforward. The top uses \( (\text{Id} . \text{Var}) \) to translate the left pattern of (Lookup) to a renaming of itself. Applying \( \text{start}^{-1} \) to the top left configuration gives the patterns \( \{ c \mapsto d \} , c , \{ a \mapsto x \} , a , s \) which are rewritten by the evaluators to form the bottom-left configuration. The bottom-left configuration rewrites to the top-right configuration in two steps, making the square right side trivial. Therefore all \( \textbf{lazy} \) (Lookup) redexes translate to \( \textbf{badbh} \) (LookupBad) redexes and the graphs generated by evaluating each redex one step are in \( \textbf{badbh} \).

\[
\begin{align*}
\langle \{ c \mapsto d \} , \{ a \mapsto x \} \rangle & \mapsto (\text{Id} . \text{Var}) \langle \{ \} , \{ \} , \{ b \cdot s , d \cdot x \} , \{ c \cdot a \} \rangle \\
(\text{LookupBad}) \downarrow (\text{Lookup}) & \quad \uparrow
\end{align*}
\]

(9.2)

\[
\begin{align*}
\langle \{ d \cdot x , a \cdot b \} , \{ \} \rangle & \mapsto (\text{Id} . \text{#} , \text{Var} \perp) \langle \{ b \cdot s , d \cdot x \} , \{ c \cdot a \} \rangle \\
\langle \{ c \mapsto d , a \mapsto \# c b \} , \{ a \mapsto \perp , b \mapsto \# a s \} \rangle & \mapsto (\text{Id} . \text{#} , \text{Var} \perp) \langle \{ b \cdot s , d \cdot x \} , \{ c \cdot a \} \rangle
\end{align*}
\]

The squares for the other \( \textbf{badbh} \) redexes are given in Section 9.4 as the rules necessary to complete them are introduced. \( \square \)

9.3 Graph-pattern translation

The set of square tops for the redexes of a rule \( L \rightarrow R \in \textbf{B} \) are found by translating \( L \). Section 9.3.1 explains how graphs can be translated with a PSR. Section 9.3.2 extends this to graph-pattern translation, showing how the PSR framework ensures that a pattern has finitely many translations and therefore the number of squares in a simulation proof is finite. Section 9.3.3 extends configuration garbage collection to configurations of graph patterns to simplify the presentation of the proof squares.
9.3.1 Graph translation

Instead of starting with a translation \( \iff \) and graphs \( G \) and \( H \) and rewriting \( \text{start}(G, H) \) to decide if \( G \iff H \), we want to start with \( H \) and find every \( G \) such that \( G \iff H \). Section 12.4 uses this translation mechanism directly for translating graphs and Section 8.2 uses it for doing garbage collection by space relation.

Translation is an adaption of the idea of forward closures from term rewriting [Der87]. We begin with a graph \( H \) and an unknown \( G \) and apply PSR rules as normal except \( G \) may be extended to allow a rule to apply. This creates every possible PSR rewrite trace beginning at \( H \) and some other graph \( G \). The traces that reach an \( \text{end} \) state give a translation of \( H \).

**Definition 9.6 (Graph translation)**

\[
\text{translate } \{L_i \xrightarrow{r_i} R_i\}_{i=1}^n H
\]
\[
= \{ G \iff \Delta \iff \Delta' \in \text{FCS} \{ \text{start}(\{ a, s \}, H) \xrightarrow{r} \text{start}(\{ a, s \}, H) \}
\]
\[
(G, H) = \text{start}^{-1}(\Delta)
\]

To translate \( H \) using a PSR defined by \( n \) rules: build an empty pattern \( \{ a, s \} \) where \( a \) and \( s \) are fresh variables; convert it and \( H \) to a configuration; find all traces \( \Delta \iff \Delta' \) in the forward closure set \( \text{FCS} \) of the empty trace which begins and ends with this configuration; convert \( \Delta \) back into graphs to get a translation. The forward closures of a set of traces are found by the following rules.

\[
\text{FCS} (\{\theta, \phi, P, H\} \xrightarrow{r'} \{\theta', \phi', P', H'\}) \cup \text{TS} = S \cup \text{FCS} \text{TS}
\]

where

\[
S = \{\theta, \phi, P, H\} \xrightarrow{r'} \{\theta', \phi', P', H'\}, \text{ if end } \{\theta', \phi', P', H'\}
\]

\[
S = \text{FCS} \{C(\theta, \phi, P, H)) \xrightarrow{r+r_i} \text{D}(R_i)
\]
\[
\mid C(\theta', \phi', P', H')) \equiv \text{D}(L_i) \equiv
\]
\[
\{\{ x \neq y \mid x \in \text{dom } P', \text{ dom } P \},
\}
\[
\{ y \in \text{dom } \theta \cup \text{dom } \theta' \}
\]
\[
S = \text{FCS} \{a \mapsto b \} (\theta, \phi, P, H) \xrightarrow{r'} \{\theta', \phi', P', H'\} | a \cdot x \in \theta, b \cdot x \in \phi'
\]

1. If a trace ends in an \( \text{end} \) state it is a forward closure; \( P \) is a translation of \( H \).

2. Translating function symbols with PSR rules: if a trace can be extended by rewriting its end with a PSR rule \( L_i \rightarrow R_i \), without adding anything to \( H' \), then this rewrite step translates part of \( H' \). Pattern \( P \) is extended through the context \( C \) and the extended trace end is found by specialising \( R_i \). This finds every way in which the existing trace can be followed by \( L_i \rightarrow R_i \) to form a new trace. The merging of trace and PSR rule is complicated slightly because we need to prevent any nodes allocated by the trace from being equated with free variables in the configuration at the start of the trace. This is arranged by adding the constraint sets indicated. Further constraints (not shown) also ensure that the nodes in the translated graph are distinct. Null terms and variables translate using the same process by the rules \text{Transc} and \text{TransE}. 
3. Translating arcs: if the goal relation includes \( a \cdot x \) but \( x \) has been removed from \( H \) and paired with \( b \) then any translation must map \( a \) to \( b \). This step adds any sharing or cyclic references needed in the translated graph.

\[
\text{translate} \ (\#gc^{-1}) \ \{ a \mapsto \bot, h \mapsto \#a, c \mapsto \lambda x.\lambda y.y \}\{c, h\}
\]

Defining \( \Delta = \text{start} (\{ \{ \} \} c', h'), \{ a \mapsto \bot, h \mapsto \#a, c \mapsto \lambda x.\lambda y.y \}\{c, h\} \), we compute a two-element forward closure set (the configuration garbage collector is used here to simplify the trace end configurations).

\[
\begin{align*}
\text{FCS} \ \{ \Delta \xrightarrow{\text{FCS}} \Delta \} \\
= \left\{ \langle \{ c', c, h' \cdot h \}, \{ \} \rangle, \langle a' \mapsto \bot, h' \mapsto \#a', c' \mapsto \lambda x'.\lambda y'.y' \rangle, \langle a \mapsto \bot, h \mapsto \#a, c \mapsto \lambda x.\lambda y.y \rangle \right\} & \xrightarrow{(Id.\lambda, Id.\lambda, Id.\text{Var}, Id.\#, Id.\bot, \text{Trans} \epsilon)} \left\{ \langle \{ \} \rangle, \langle \{ \} \rangle \right\} \\
= \left\{ \langle \{ c', c, h \}, \{ \} \rangle, \langle c' \mapsto \lambda x'.\lambda y'.y' \rangle, \langle a \mapsto \bot, h \mapsto \#a, c \mapsto \lambda x.\lambda y.y \rangle \right\} & \xrightarrow{(Id.\lambda, Id.\lambda, Id.\text{Var}, \text{DeadUdm, Trans} \epsilon)} \left\{ \langle \{ \} \rangle, \langle \{ \} \rangle \right\}
\end{align*}
\]

In this case, as well as the obvious forward closure obtained from applying identity PSR rules, we also get another in which the \( \text{(DeadUdm)} \) rule can be applied (because the only reference to \( a \) is its update marker), resulting in a smaller garbage-collected graph \( \{ c' \mapsto \lambda x'.\lambda y'.y' \}\) \( c' \), \( \epsilon \) and eliminating the final evaluation step from the trace in Example 4.1. □

### 9.3.2 Square-top construction

To translate a graph pattern, and hence to find its square tops, the rules of Definition 9.6 are adapted slightly.

**Definition 9.7** (Graph-pattern translation)

\[
\begin{align*}
sqTops \ \{ \{ \}, \{ \}, P_i, Q_i \} \xrightarrow{\text{FCS}} \{ \Theta_i, \Phi_i, P'_i, Q'_i \}_{i=1}^n Q \\
= \text{FFCS} \ \{ \text{start} (\{ \{ \} a, s \}, Q) \xrightarrow{\text{FCS}} \text{start} (\{ \{ \} a, s \}, Q) \}
\end{align*}
\]

To translate pattern \( Q \) using a PSR defined by \( n \) rules: build an empty pattern \( \{ \} a, s \) where \( a \) and \( s \) are fresh variables; convert it and \( Q \) to a configuration; find all traces in the finite forward closure set \( \text{FFCS} \) (defined by the modified \( \text{FCS} \) below) of the empty trace which begins and ends with this configuration. These are the square tops for \( Q \).
\[ \text{FFCS} (\{ \langle \theta, \phi, P, Q \rangle \rightarrow \langle \theta', \phi', P', Q' \rangle \}) = S \cup \text{FFCS} \text{ TS} \]

where
\[ S = \{ \langle \theta, \phi, P, Q \rangle \rightarrow \langle \theta', \phi', P', Q' \rangle \}, \text{ if symbols } Q' = \{ \} \]
\[ S = \text{FFCS} \{ \mathcal{C}(\langle \theta, \phi, P, Q \rangle) \rightarrow \mathcal{D}(\langle \Theta_i, \Phi_i, P_i, Q'_i \rangle) \mid \mathcal{C}(\langle \theta', \phi', P', Q' \rangle) \equiv \mathcal{D}(\{ \}, \{ \}, P_i, Q_i) \equiv \} \]
\[ \mathcal{T}(Q') \subseteq Q'' \subseteq \mathcal{T}(Q_i), \]
\[ \text{symbols } Q'' \neq \{ \} \}
\]

\[ S = \text{FFCS} \{ \{ a \mapsto b \}(\langle \theta, \phi, P, Q \rangle \rightarrow \langle \theta', \phi', P', Q' \rangle)|a \cdot x \in \theta', b \cdot x \in \phi' \}
\]
\[ S = \text{FFCS} \{ \{ a \mapsto a' \}(\langle \theta, \phi, P, Q \rangle \rightarrow \langle \theta', \phi', P', Q' \rangle)|a \cdot b \in \theta', a \in \text{dom } P \}
\]
\[ \cup \{ \{ a \mapsto a' \}(\langle \theta, \phi, P\{a \mapsto h' \}, Q \rangle \rightarrow \langle \theta', \phi', P'\{a \mapsto h' \}, Q' \rangle)|a \notin \text{dom } P', h' \text{ fresh} \}
\]

where \{ \{ b \mapsto h \} \subseteq Q', a \cdot b \in \theta' \}

1. Now traces are accepted when the graphs in their final configuration contain only holes, rather than when they lead to an \textit{end} state.

2. The function-symbol translation rule is modified. When a trace is extended by adding rule \( r_i \) to its end, the pattern \( Q' \) can be specialised by general unification rather than matching. To guarantee termination there must be a non-trivial overlap \( Q'' \) between function symbols in \( Q' \) and \( Q_i \).

3. Arc translation is unchanged from FCS.

4. A new rule translates any holes left at the end of a trace. We need this because the end of a trace is not an \textit{end} state. If a hole at node \( b \) is to be paired with \textit{something} at node \( a \) then either \( a \) is a node in the pattern \( P \) or else \( P \) is extended with a new node \( a \) containing a new hole \( h' \).

\[ \square \]

\textbf{Example 9.2 (Square-top construction)}

The set of square tops for our example proof is constructed by applying the function \((s\text{qTops } \rightarrow k)\) to each left pattern of \text{[lazy]}\#. One of the finite forward closure constructions for the left pattern of \textit{(Update)} is shown next (other forward closures generated from different rule orders produce the same square top modulo renaming). In our example there happens to be one finite forward closure for each of the rules; in general we might not be so lucky.

All three steps in this FFCS trace use rule (2) of Definition 9.7. Rule (1) applies to the final set in the trace because the pattern \{ \{ j \mapsto B \} \} contains only holes. For an example use of rule (4) see square (10.16). Evaluator left patterns are linear in variable references so rule (3) is never needed for square-top construction.
\[ sqTops \equiv \{ g \mapsto \lambda x.B, m \mapsto \bot, h \mapsto \#m \, t \} g, h \]

\[
\begin{align*}
\text{FFCS} & \equiv \exists \text{FFCS} \\
\{ e \cdot g, a \cdot h \}, \quad \{ \}, \quad \{ \} & \mapsto \{ e \cdot g, a \cdot h \}, \quad \{ \}, \quad \{ \} \\
\{ g \mapsto \lambda x.B, m \mapsto \bot, h \mapsto \#m \, t \} & \mapsto \{ g \mapsto \lambda x.B, m \mapsto \bot, h \mapsto \#m \, t \}
\end{align*}
\]

\[
\begin{align*}
\text{FFCS} & \equiv \exists \text{FFCS} \\
\{ e \cdot g, a \cdot h \}, \quad \{ \}, \quad \{ e \mapsto \lambda f.A \}, \quad \{ g \mapsto \lambda x.B, m \mapsto \bot, h \mapsto \#m \, t \} & \mapsto \{ i \mapsto A[z/f] \}, \quad \{ j \mapsto B[y/x] \}, \quad \{ \} \\
\{ i \mapsto A[z/f] \}, \quad \{ j \mapsto B[y/x] \} & \mapsto \{ i \mapsto A[z/f] \}, \quad \{ j \mapsto B[y/x] \}
\end{align*}
\]

\[ \Box \]

**Proposition 9.3 (The square tops set is finite)**

For any PSR \( \implies \) and pattern \( L \), the calculation of \( sqTops \implies L \) terminates.

**Proof**

The length of each trace in the finite forward closure set is limited by the number of function symbols in \( L \) because each rewrite step removes a function symbol and right-patterns in PSR rules do not introduce any new function symbols. \( \Box \)

**Proposition 9.4 (The square tops set is complete)**

1. If \( G \iff H \land H \rightarrow B \, H' \) then there is a rule \( (L \rightarrow R) \in \mathbb{B} \) and a square top \( \Delta \vdash \Delta' \in (sqTops \iff L) \) such that \( \text{start}(G, H) \) is an instance of \( \Delta \).
2. Further, the left configuration of every such square top has an instance \( \text{start}(G, H) \) such that \( H \) is a \( \mathbb{B} \)-redex.

**Proof**

1. (Outline) Begin with the configurations \( \text{start}(G, H) \) and \( \Delta = \text{start}(\{ \} a, b, L') \) where \( L \) is the left pattern of the \( \mathbb{B} \) rule which matches \( H \) and \( L' \) is a renaming of \( L \) such that \( \text{dom} \, L' \subseteq \text{dom} \, H \).

Map the proof trace of \( \text{start}(G, H) \) to a finite forward closure of \( \Delta \): when a rewrite step involves nodes in \( L' \), use step (3) in the definition of FFCS, otherwise leave the forward closure untouched. At the end of the trace all root-reachable nodes have been removed from \( H \) and therefore the trace satisfies condition (1) in FFCS.

2. Let \( (P, Q) = \text{start}^{-1}(\Delta) \). The graph \( \mathcal{E}(Q) \) is a \( \mathbb{B} \)-redex where \( \mathcal{E} \) is the null context. \( \Box \)
9.3.3 Garbage collection for pattern configurations

The configuration garbage collector cannot be applied to pattern configurations because it might remove unreachable nodes that become reachable when the pattern is specialised by some context. But we can define a safe approximate garbage collector to apply to the end pattern of a forward closure.

Definition 9.8 (Pattern-configuration garbage collection)
The pattern-configuration garbage collector takes two configurations as arguments: the start and end of a forward closure. The result is a new end configuration.

\[
gc(\theta’, \phi’, P’, Q’) \ (\theta, \phi, P, Q) = \begin{cases} \theta - \phi, \\ \{x : C \cdot y : C’ \in \phi \mid x \in \text{var} P'' \land (x \in \text{dom} P’ \land \neg \text{stackCat} C), \\ y \in \text{var} Q’’ \land (y \in \text{dom} Q’ \land \neg \text{stackCat} C’) \} \end{cases} \tag{1} \]

\[
\begin{align*} 
P'' &= P|_{\text{reach}(P, \text{dom} \theta \cup \text{dom} P')} \\
Q'' &= Q|_{\text{reach}(Q, \text{rng} \theta \cup \text{dom} Q')} \tag{2} \end{align*} \tag{3}
\]

where

1. The goal relation \(\theta\) is simplified as normal by removing any pairs already in \(\phi\).

2. Pairs \(x \cdot y\) in \(\phi\) are kept only if instances of both variables can occur in any instance of the configuration. This can happen if the variable occurs in the garbage collection of \(P\) or \(Q\); or if the variable occurs in the domain of \(P'\) or \(Q'\) so it may be reachable from other nodes in an instance of the start configuration, however, if it belongs to a stack category then the only reference to it must have been removed.

3. Nodes in \(P\) might be reachable if they are reachable from \(\theta\) or from any node in pattern \(P'\); similarly for \(Q\).

\(\square\)

Example 9.3 (Pattern translation with garbage collection)
Applying gc to the pattern translation in Example 9.2 simplifies its result as shown below. Three pairs are removed from the goal and the pair \(a \cdot h\) is removed from the correspondence relation; both \(a\) and \(h\) belong to stack categories so there can be no other occurrences of either variable in any instance of the configuration.

\[
\left\{ \begin{array}{l} \{e \cdot g, a \cdot h\}, \\
\{\}, \\
\{e \mapsto \lambda f. A, k \mapsto \perp, a \mapsto \# k d\}, \\
g \mapsto \lambda x. B, m \mapsto \perp, h \mapsto \# m t \end{array} \right\} \xrightarrow{(\text{Id}_x, \lambda, \# \text{Var} \perp)} \left\{ \begin{array}{l} \{i \cdot j, d \cdot t\}, \\
\{e \cdot g, z \cdot y, k \cdot m\}, \\
i \mapsto A[y/x], \\
j \mapsto B[y/x] \end{array} \right\} \xrightarrow{\text{\text{\text{\sim}}}} \right.
\]

\(\square\)

Proposition 9.5 (Safety of pattern garbage collection)

\(\star \xrightarrow{\text{\text{\text{\sim}}} \triangle} \triangle’ \in (sq \text{Top} \iff \text{L}) \land \text{start}(G, H) = C(\star) \Rightarrow gc \ C(\triangle) = gc \ C(gc \cdot \triangle \triangle’)\)

Pattern-configuration garbage collection does not remove anything that could be needed in the garbage collection of any instance, assuming there is no conflict between variable names in \(C\) and variables allocated by the square-top trace.

Proof
Let the context be defined \(C((\theta, \phi, p, q)) = (T(\theta), T(\phi), G \cup T(p), H \cup T(q))\) and
the top-left configuration be $\delta = (\Theta', \{\}, P', Q')$ and the top-right configuration be $\delta' = (\Theta, \Phi, P, Q)$. The pattern garbage collection is as shown in Definition 9.8. The garbage collection of the instance is:

$$gc(\delta) = (T(\Theta - \Phi), T(\Phi \cap (\text{var } G' \times \text{var } H')), G', H')$$

where $G' = G \cup T(P)\mid_{\text{reach}(G \cup \Phi \cup P, \text{dom } \Downarrow(\Theta))}$

$$H' = H \cup T(H)\mid_{\text{reachable}(H \cup \Phi \cup Q, \text{rng } \Downarrow(\Theta))}$$

The goal relation is the same in $gc(\delta \cdot \delta')$. For the graph components we consider the subset of $G'$ in the garbage collection minus $G$:

$$T(P)\mid_{\text{reachable}(G \cup \Phi \cup P, \text{dom } \Downarrow(\Theta))} \subseteq T(P)\mid_{\text{reachable}(\Phi \cup P, \text{dom } \Downarrow(\Theta))} \subseteq T(P)\mid_{\text{reachable}(\Phi \cup P, \text{dom } \Downarrow(\Theta) \cup \text{dom } \Downarrow(P')) = T(P'')$$

And similarly for $Q$.

By a similar chain of reasoning for the correspondence set we have:

$$T(\Phi \cap (\text{var } G' \times \text{var } H')) \subseteq T(\Phi \cap ((\text{var } P'' \times \text{dom } P') \times (\text{var } Q'' \times \text{dom } Q')))$$

As there is only one arc to any stack-category node and $L$ is fully-connected, any such variables in $\Phi$ cannot occur in the context, hence the extra restriction in Definition 9.8.

\[\Box\]

### 9.4 Configuration simplification

Sometimes square completion is easy. In the very simplest cases, like Square (9.2) in Proposition 9.2, the bottom-left configuration can be rewritten with PSR rules in the normal way until a bottom-right configuration is found which is identical to the top-right configuration, trivially satisfying the definition of $\Rightarrow$.

In general this does not happen for a number of reasons. The following sub-sections introduce some proof rules which can be applied in addition to PSR rewriting to make a square complete. We also introduce a general test for $\Rightarrow$ to enable the automatic completion of more squares.

#### 9.4.1 Renaming nodes

When an evaluator rule moves a hole, its proof square can fail to complete.

**Example 9.4 (Moving hole prevents proof)**

Continuing the proof of $\text{badhole} \not\Rightarrow \text{lazy}$, consider the following square for the $(\text{Push})$ rule. It does not complete properly — the missing $\downarrow$ arrow from its right side is deliberate — because the holes $A$ and $B$ have moved to different addresses.

$$\langle \{ e \cdot a, b \cdot s \}, \{ \} \rangle \quad (\text{Id, Apply}) \quad \rightarrow \quad \langle \{ b \cdot s, i \cdot j, f \cdot x \}, \{ e \cdot a \} \rangle$$

$$(\text{Push}) \downarrow (\text{Push})$$

$$\langle \{ e \cdot a, a \cdot b \}, \{ \} \rangle \quad (\text{Id, :}) \quad \rightarrow \quad \langle \{ e \cdot a, f \cdot x, b \cdot s \}, \{ \} \rangle$$

$$\langle \{ e \mapsto A, a \mapsto f : b \}, \{ a \mapsto B, b \mapsto x : s \} \rangle \rightarrow \quad \langle \{ e \mapsto A \}, \{ a \mapsto B \} \rangle$$

\[9.3\]

\[\Box\]
To handle situations like Square (9.3) we need a safe renaming principle. Intuitively, we can rename a variable to a fresh variable without affecting the meaning of a configuration. This extends to variables in a configuration pattern \( \Delta \) because if we rename it by applying a simple context: \( \{ a \mapsto b \} \{ \Delta \} \) then we can recover any instance of \( \Delta \) if the renaming can be reversed. To reverse the renaming, \( b \) must not occur in \( \Delta \). In practice, we find that \( b \) does occur in \( \Delta \) sometimes so our renaming rule works by swapping two variables.

**Proposition 9.6 (Renaming rule)**

\[ \Delta \Rightarrow \{ a \mapsto b, b \mapsto a \}(\Delta) \]

**Proof**

A slight abuse of the definition of \( \Rightarrow \): for any instance of a configuration pattern, \( \mathbb{C}(\Delta) \), there is a context \( \mathbb{D} = \mathbb{C} \circ \{ a \mapsto b, b \mapsto a \} \) which produces the same configuration \( \mathbb{C}(\Delta) \equiv \mathbb{D}(\{ a \mapsto b, b \mapsto a \}(\Delta)) \) and therefore the meaning of a configuration pattern is unchanged by renaming. \( \square \)

In practice we only use renaming in situations like Square (9.3) to rename the addresses of nodes containing holes in a bottom-right configuration to make them the same as their top-right configuration occurrences. It is often also necessary to use a garbage-introduction step to extend the correspondence relation when renaming.

**Definition 9.9 (Node-renaming rule (Ren))**

The following renaming rule (9.4) may be applied to a square-bottom configuration if the top-right configuration is an instance of \( \langle \{ x \cdot y \}, \{ \}, \{ x \mapsto h \}, \{ y \mapsto h' \} \rangle \). The renaming context \( \mathbb{C} \) is defined \( \{ a \mapsto x, x \mapsto a, b \mapsto y, y \mapsto b \} \).

\[ \langle \theta, \phi, P\{ a \mapsto h \}, Q\{ b \mapsto h' \} \rangle \xrightarrow{(Ren)} \mathbb{C}(\langle \theta, \phi, P\{ a \mapsto h \}, Q\{ b \mapsto h' \} \rangle) \]  \hspace{1cm} (9.4)

If \( x \) and \( y \) do not occur in the top-left or bottom-left configurations of the square the rule is modified to (9.5). This introduces a garbage pair \( x \cdot y \) to \( \phi \), which is renamed to \( a \cdot b \).

\[ \langle \theta, \phi, P\{ a \mapsto h \}, Q\{ b \mapsto h' \} \rangle \xrightarrow{(Ren)} \mathbb{C}(\langle \theta, \phi\{ x \cdot y \}, P\{ a \mapsto h \}, Q\{ b \mapsto h' \} \rangle) \]  \hspace{1cm} (9.5)

\( \square \)

**Example 9.5 (Renaming completes proof square)**

Returning to the problem of Example 9.4, applying (Ren) to exchange \( e \) with \( i \) and \( a \) with \( j \), the (Push) square now completes properly as shown below. Before the rename step, we add \( i \cdot j \) to the correspondence relation as in Definition 9.9. This tweak makes the top-right and bottom-right configurations identical, without it the square right side would not satisfy \( \Rightarrow \) because the relationship between nodes \( e \) and \( a \) would be lost.
\[
\begin{align*}
\left(\{e \cdot a, b \cdot s\}, \emptyset\right), & \quad (Id\_Apply) & \left(\{b \cdot s, i \cdot j, f \cdot x\}, \{e \cdot a\}\right), \\
\left(\{e \mapsto A f\}, \emptyset\right), & \quad \left(\{i \mapsto A\}, \emptyset\right) & \left(\{j \mapsto B\}, \emptyset\right)
\end{align*}
\]
(Push) \downarrow (Push) \hspace{1cm} \Downarrow
(9.6)

\[
\begin{align*}
\left(\{e \cdot a, a \cdot b\}, \emptyset\right), & \quad (Ren, Id\_\downarrow): & \left(\{b \cdot s, i \cdot j, f \cdot x\}, \{e \cdot a\}\right), \\
\left(\{e \mapsto A, a \mapsto f : b\}, \emptyset\right) & \quad \left(\{i \mapsto A\}, \emptyset\right) & \left(\{j \mapsto B\}, \emptyset\right)
\end{align*}
\]

\[
\Box
\]

### 9.4.2 Square-right-side rule

Many of our example proof squares (all of them, eventually, for the \texttt{badbbl} \ll \texttt{lazyl} running example) have identical top-right and bottom-right configurations. This is not true in general so we need a more sophisticated test for the right-side arrow \(\Rightarrow\) than configuration equality.

**Proposition 9.7 (Configuration implication test)**

In the following proof square:

\[
\begin{align*}
\bullet \Rightarrow & \ast \quad \langle \theta, \phi, P, Q \rangle \\
\downarrow & \quad \smile \left\downarrow \ast \quad \langle \theta', \phi', P', Q' \rangle
\end{align*}
\]

If \(\theta' \subseteq \theta\) and \((\theta'', \phi'', P'', Q'') = \text{gc} \bullet \Delta \langle \theta'', \phi, P, Q \rangle\) and \(\phi'' \subseteq \phi' \wedge P'' \subseteq P' \wedge Q'' \subseteq Q'\) then \(\langle \theta, \phi, P, Q \rangle \Rightarrow \langle \theta', \phi', P', Q' \rangle\).

**Proof**

If the top-right configuration goal \(\theta\) is a super set of the bottom-right configuration goal \(\theta'\) then we can reduce the top-right pattern goal to \(\theta'\) and garbage collect the resulting configuration to give a configuration which satisfies Definition 9.5 because PSRs have the \textit{SubTrace} and \textit{GcTrace} properties. The bottom-right configuration may contain extra nodes in \(P'\) or \(Q'\), or elements in \(\phi'\), which are not needed during the proof trace. \(\Box\)

This test is incomplete, but it is sufficient for the majority of our examples.

**Example 9.6** (\(\mathcal{H} \ll \mathcal{H}\) wrt \(\text{id}\))

To show \(\mathcal{H} \models \mathcal{H}\) requires Square (9.7) for (IfTrue)-redexes. This evaluator rule selects the then-branch of an if-expression, so after its application the holes \(H\) and \(F\) for the else-branches disappear and the top-right and bottom-right configurations are different. The square bottom uses the \(\text{(Ren)}\) rule to give a bottom-right configuration which is a sub-part of the top-right configuration. Using Proposition 9.7, we replace the goal relation in the top-right configuration with \(\{i k, f l\}\) and apply the pattern garbage collector. This produces the bottom-right configuration so the square is complete. Note that the \(\Rightarrow\) arrow does not hold in the other direction.
\[
\begin{align*}
\left<\{c\cdot d, b\cdot g\}, \emptyset, \{c \mapsto \text{True}, b \mapsto \{E; F\} : f\}, \{d \mapsto \text{True}, g \mapsto \{G; H\} : t\}\right>& \quad \text{Id.}_c \mapsto \left<\{i \mapsto E, j \mapsto F\}, \{k \mapsto G, l \mapsto H\}\right> \\
(\text{IfTrue}) \downarrow (\text{IfTrue}) & \quad \Downarrow \quad \text{(9.7)}
\end{align*}
\]

9.4.3 Renaming hole substitutions

The variables substituted into a hole in the top-right and bottom-right configurations of a square can be named differently, preventing completion of the square.

Example 9.7 (Hole substitutions prevent proof)
The (\text{Reduce})-redex square (9.8) in our \text{bad}[bh] \text{lazy} proof does not complete because the substitute for \( f \) in holes \( A \) is new variable \( j \) in the top-right configuration and free variable \( d \) in the bottom-right configuration; similarly for \( y \) in hole \( E \).

\[
\begin{align*}
\left<\{e\cdot a, b\cdot s\}, \emptyset, \{e \mapsto \lambda f.A, b \mapsto d : g\}, \{a \mapsto \lambda y.E, s \mapsto x : t\}\right>& \quad \text{Id.}_c \mapsto \left<\{i \mapsto A[j/f], \{e\cdot a, j\cdot l\}\}, \{k \mapsto E[l/y]\}\right> \\
(\text{Reduce}) \downarrow (\text{Reduce}) & \quad \Downarrow \quad \text{(9.8)}
\end{align*}
\]

We introduce another rule to safely rename variables in hole substitutions, to handle the situations that arise in our examples: hole substitutions in one graph need to be paired with hole substitutions in the other graph.

Proposition 9.8 (Hole-substitution-elimination rule (\text{HRen}))
\[
\begin{align*}
\left<\{x'\cdot y\}, \{x''\cdot y''\}, \{a \mapsto h\theta[x'/x]\}, \{b \mapsto h'\theta'[y''/y]\}\right>& \Rightarrow \left<\{\}, \{x''\cdot y''\}, \{a \mapsto h\theta[x'/x]\}, \{b \mapsto h'\theta'[y''/y]\}\right>
\end{align*}
\]

A pair of substituted variables \( x''\cdot y'' \) occurring in the correspondence relation can be replaced by a pair of variables \( x'\cdot y' \) in the goal relation and \( x'\cdot y' \) removed from the goal relation, \( x''\cdot y'' \) may then be removed from the correspondence relation by pattern garbage collection. Reversing this implication we get a rewrite rule \( \Rightarrow \) to use for simplifying configurations.

Proof
Map the trace of the LHS in any context leading to an \textit{end} state to the trace of the
RHS as follows. If some rewrite step adds $x''y''$ to the goal in the LHS trace, the RHS trace adds $x'y'$ instead. If the pair $x''y''$ is never added to the goal in the LHS trace, the removal of $x'y'$ from the RHS goal is safe by the SubTrace property.

\[\]

**Example 9.8 ((HRen) completes proof squares)**

Using the (HRen) rule completes the (Reduce)-redex square (9.8) as shown by (9.9). Two applications of (HRen) also complete the (Let)-redex square (9.10).

\[
\begin{align*}
\{e \cdot a, b \cdot s\}, \{\}, \\
\{e \mapsto \lambda.f.A, b \mapsto d : g\}, \\
\{a \mapsto \lambda y.E, s \mapsto x : t\} & \xrightarrow{(\text{Id.} \cdot \lambda, \text{Id.} :)} \{i \cdot k, d \cdot x, g \cdot t\}, \{e \cdot a, j \cdot l\}, \\
\{i \mapsto A[j/f]\}, \\
\{k \mapsto E[l/y]\} \\
\text{(Reduce)} & \xrightarrow{\downarrow} \text{(Reduce)} \\
\{e \cdot a, g \cdot t\}, \{\}, \\
\{e \mapsto A[d/f]\}, \\
\{a \mapsto E[x/y]\} & \xrightarrow{(\text{Ren}, \text{HRen})} \{d \cdot x, i \cdot k, g \cdot t\}, \{j \cdot l, e \cdot a\}, \\
\{i \mapsto A[j/f]\}, \\
\{k \mapsto E[l/y]\} \\
\{g \cdot a, b \cdot s\}, \{\}, \\
\{g \mapsto \text{let } h = A \text{ in } C\}, \\
\{a \mapsto \text{let } y = E \text{ in } X\} & \xrightarrow{(\text{Id.Let})} \{b \cdot s, k \cdot n, m \cdot p\}, \{g \cdot a, l \cdot o\}, \\
\{k \mapsto A[l/h]\}, \{m \mapsto C[l/h]\}, \\
\{n \mapsto E[o/y]\}, \{p \mapsto X[o/y]\} \\
\text{(Let)} & \xrightarrow{\downarrow} \text{(Let)} \\
\{g \cdot a, b \cdot s\}, \{\}, \\
\{g \mapsto C[a/h], a \mapsto A[a/h]\}, \\
\{a \mapsto X[b/y], b \mapsto E[b/y]\} & \xrightarrow{(\text{Ren}, \text{Ren}, \text{HRen}, \text{HRen})} \{k \cdot n, m \cdot p, b \cdot s\}, \{l \cdot o, g \cdot a\}, \\
\{k \mapsto A[l/h]\}, \{m \mapsto C[l/h]\}, \\
\{n \mapsto E[o/y]\}, \{p \mapsto X[o/y]\} \\
\end{align*}
\]

\[\]

**9.4.4 Copying and sharing**

The final hurdle for our proof of \texttt{badbb} $\equiv \text{lazy}$ is the (Update)-redex square.

**Example 9.9 (Copying prevents proof)**

The (Update) rule and its \texttt{badbb} counterpart duplicate a $\lambda$-value. So the bottom-right configuration of (9.11) has two copies of the holes $A$ and $E$.

\[
\begin{align*}
\{e \cdot a, b \cdot s\}, \{\}, \\
\{e \mapsto \lambda.f.A, m \mapsto d, b \mapsto \#m h\}, \\
\{a \mapsto \lambda x.E, y \mapsto \bot, s \mapsto \#y t\} & \xrightarrow{(\text{Var.}, \text{Id.} \cdot \lambda, \text{Id.} \#)} \{i \cdot k, h \cdot t\}, \{m \cdot y, e \cdot a, j \cdot l\}, \\
\{i \mapsto A[j/f]\}, \\
\{k \mapsto E[l/x]\} \\
\text{(UpdateBad)} & \xrightarrow{\downarrow} \text{(Update)} \\
\{m \cdot y, h \cdot t\}, \{\}, \\
\{m \mapsto \lambda.f.A, e \mapsto \lambda.f.A\}, \\
\{y \mapsto \lambda x.E, a \mapsto \lambda x.E\} & \xrightarrow{(\text{Id.} \cdot \lambda, \text{Ren}, \text{HRen}, \text{Id.} \lambda)} \{h \cdot t, i \cdot k, b \cdot d\}, \{j \cdot l, m \cdot y, e \cdot a, c \cdot g\}, \\
\{i \mapsto A[j/f]\}, \{b \mapsto A[c/f]\}, \\
\{k \mapsto E[l/x]\}, \{d \mapsto E[g/x]\} \\
\end{align*}
\]

\[\]
Perhaps not surprisingly, a proof system designed for reasoning about space properties has problems when there is some copying. We cannot introduce a new PSR rule such as \((\lambda^2)\) below (this rule can be inferred by the process described in Section 10.1) to circumvent the problem because that would introduce a new \((\text{Update})\)-redex square for duplicated values — and a new \((\text{Reduce})\)-redex square — then we would need a PSR rule \((\lambda^3)\), and so on.

\[
\langle \{ \}, \{ \}, \{ \lambda f . A, e \mapsto \lambda f . A \}, \{ y \mapsto \lambda x . E, a \mapsto \lambda x . E \} \rangle \\
\rightarrow \langle \{ i \cdot k \}, \{ m \cdot y, e \cdot a, j \cdot i \}, \{ i \mapsto A[j/f], \{ k \mapsto E[l/x] \} \rangle
\]

\((\lambda^2)\)

Another solution is to introduce a rule. Definition 9.10 states the rule we want.

**Definition 9.10 (Duplicate-elimination rule \((\text{Share})\))**

\[
\langle \{ a \cdot b \}, \{ a \mapsto A \}, \{ b \mapsto B \} \rangle \\
\Rightarrow \langle \{ a \cdot b \}, \{ a \mapsto A, c \mapsto A \}, \{ b \mapsto B, d \mapsto B \} \rangle
\]

Reversing this implication we get a rewrite rule \(\overset{\text{Share}}{\Rightarrow}\) to use for simplifying configurations. Note that \(A\) and \(B\) are arbitrary term patterns, not necessarily just holes.

\(\square\)

The \((\text{Share})\) rule appears to be safe in our current example, but it is not applicable in general. Whether it is safe to duplicate two nodes in the goal relation amounts to another restriction on the translation. If the part of the translation proof trace that pairs \(a\) with \(b\) (in an instance of Definition 9.10) does not involve any nodes in the rest of the graph then they can be duplicated as indicated. A very simple way to arrange this is to restrict PSR rules so their left patterns have only one node in each graph — the \(\overset{\text{id}}{\Rightarrow}\) rules all have this property and so do the \(\overset{\text{badh}}{\Rightarrow}\) rules. But we choose a more general solution. A sharable PSR rule can translate many nodes providing that the translation includes sub-rules so that duplicating nodes is safe. This technique is similar to the way we guarantee \(Gc\text{Trace}\) by only accepting self-consistent PSRs.

**Proposition 9.9 (Sharable translation)**

The \((\text{Share})\) rule may be applied in a translation \(\overset{\Rightarrow}{\Rightarrow}\) rewrite trace if: for all PSR rules \(\langle \{ \}, \{ \}, P \{ a \mapsto N \}, Q \{ b \mapsto M \} \rangle \overset{\Theta, \Phi \{ \text{a \cdot b} \}, P', Q'}{\Rightarrow} \langle \Theta_s, \Phi_s, P'_s, Q'_s \rangle\) in the translation \(\overset{\Rightarrow}{\Rightarrow}\), the sub-rule \(\langle \{ \}, \{ \}, \{ a \mapsto N \}, \{ b \mapsto M \} \rangle \overset{\Theta_s, \Phi_s, P'_s, Q'_s}{\Rightarrow} \langle \Theta_s, \Phi_s, P'_s, Q'_s \rangle\) is also part of \(\overset{\Rightarrow}{\Rightarrow}\) where:

- \(\Theta_s = \Theta_b \cup \{ x \cdot y \in \Theta | x \in \text{fv} \{ a \mapsto N \}, y \in \text{fv} \{ b \mapsto M \} \}\)
- \(\Theta_b = \{ x \cdot y \in \Theta | x \in \text{dom} \ P'_b \}, y \in \text{dom} \ Q'_b \}\)
- \(\Phi_s = \{ x \cdot y \in \Phi | x \in \text{fv} \{ b \mapsto M \} \}\)
- \(P'_s = \{ \{ a \mapsto h \} \in P' | h \in \text{holes} \ N \}\)
- \(Q'_s = \{ \{ a \mapsto h \} \in Q' | h \in \text{holes} \ M \}\)

The side-conditions attached to the construction of \(\Theta_b\) ensure that any nodes introduced by the sub-rule duplicate nodes introduced by the full rule.

**Proof**

First generalise the definition of \((\text{Share})\) to:

\[
\langle \{ a_i \cdot b_i \}_{i=1}^n, \{ \}, \{ a_i \mapsto A_i \}_{i=1}^n, \{ b \mapsto B_i \}_{i=1}^n \rangle \\
\Rightarrow \langle \{ a_i \cdot b_i \}_{i=1}^n, \{ \}, \{ a_i \mapsto A_i, c_i \mapsto A_i \}_{i=1}^n, \{ b_i \mapsto B_i, d_i \mapsto B_i \}_{i=1}^n \rangle
\]

\((LHS)\)
so the initial configuration contains \( n \) duplicated pairs of nodes.

Map the translation trace of any instance of LHS to the same instance of RHS as follows. Steps not involving duplicated nodes are unchanged.

If a step uses PSR rule \( r \) to remove \( m \) of the duplicated nodes and introduce \( p \) new nodes:

\[
\begin{align*}
\text{LHS} \xrightarrow{r} & \langle \theta\{a_i \cdot b_i\}_{i=1}^n, \phi, G'G_i\{a_i \mapsto A_i\}_{i=1}^n, H'H_i\{b_i \mapsto B_i\}_{i=1}^n \rangle \\
& \xrightarrow{r} \langle \theta'\{a_i \cdot b_i\}_{i=1}^n \{a_i' \cdot b_i'\}_{i=1}^p, \phi\{a_i \cdot b_i\}_{i=1}^m \{a_i' \cdot b_i'\}_{i=1}^p, G''G_i\{a_i' \mapsto A_i\}_{i=1}^p, H''H_i\{b_i' \mapsto B_i\}_{i=1}^p \rangle
\end{align*}
\]

where \( G, H \) are unaffected by \( r \); \( G', H' \) are other nodes removed by \( r \); \( G'', H'' \) are nodes introduced by \( r \) containing sub-terms of \( G', H' \) which we are not interested in; \( \{a_i' \cdot b_i'\}_{i=1}^p \) are new nodes which contain sub-terms of the \( m \) nodes we are interested in.

Let \( s \) be the \( m \) sub-rules of \( r \) for translating the duplicated pairs \( \{a_i \cdot b_i\}_{i=1}^m \). Then \( r \) in the left trace corresponds to \( r \) in the right trace which removes \( m \) duplicated nodes and introduces \( p \) duplicated nodes. By induction, all the remaining duplicated nodes are removed by further duplicated steps.

\[
\begin{align*}
\text{RHS} \xrightarrow{r} & \langle \theta\{a_i \cdot b_i\}_{i=1}^n, \phi, G'G_i\{a_i \mapsto A_i\}_{i=1}^n, H'H_i\{b_i \mapsto B_i\}_{i=1}^n \rangle \\
& \xrightarrow{rr} \langle \theta'\{a_i \cdot b_i\}_{i=1}^n \{a_i' \cdot b_i'\}_{i=1}^p, \phi\{a_i \cdot b_i\}_{i=1}^m \{a_i' \cdot b_i'\}_{i=1}^p, G''G_i\{a_i' \mapsto A_i\}_{i=1}^p, H''H_i\{b_i' \mapsto B_i\}_{i=1}^p \rangle
\end{align*}
\]

\( \square \)

Another application for (Share) is to shorten configuration-rewrite traces. It might be necessary to apply (Share) during square-top construction too, if a pattern is non-linear in holes.

**Example 9.10** **((Share) completes a proof)**

The translation \( \text{badlh} \Rightarrow \) is sharable because all of its rules translate a single node. The (Update)-redex square can be completed now (9.12), and the proof of \( \text{badlh} \not\Rightarrow \text{lazy} \) is complete too.

\[
\begin{align*}
\langle \{e \cdot a, b \cdot s\}, \{\} \rangle, \\
\langle \{e \mapsto \lambda f.A, m \mapsto d, b \mapsto \#m h\} \rangle, & \quad (\text{Var} \perp, \quad Id.\lambda, \text{Id.}\#) \\
\langle a \mapsto \lambda x.E, y \mapsto \perp, s \mapsto \#y t\} \rangle & \quad \langle i \mapsto A[j/f] \rangle, \quad \langle k \mapsto E[l/x] \rangle \\
(\text{UpdateBad}) \downarrow (\text{Update}) \quad \blacktriangleleft & \quad \quad (9.12)
\end{align*}
\]

\( \square \)

The sharable restriction is quite severe and we will see examples in Chapter 10 where other methods are needed to show that duplication is safe. The sharable test can be extended to use information about the categories of graph nodes. For example, a PSR might be sharable for category X nodes but not category S nodes.
9.5 Summary

We have defined and illustrated an automated procedure for checking a proof of \( \mathbb{M} \Rightarrow \mathbb{M} \) wrt \( \Leftrightarrow \). The basic method is a simple simulation proof. The proof is presented as a collection of squares, each of which shows that evaluation of a subset of the \( \mathbb{M} \)-redexes preserves the translation \( \Leftrightarrow \). The properties of PSR translations discussed in Chapter 8 are crucial for the success of this proof. Termination conditions guarantee a finite square set. The \( \text{SubTrace} \) and \( \text{GcTrace} \) properties allow us to ignore garbage collection after an evaluation step, localising the reasoning and making the squares very simple. And the space-bound property is built into PSRs.

To generate the proof squares we showed how a PSR can be used to translate a graph or a pattern. This has other applications such as garbage collecting a graph by finding its smallest translation. We also defined a number of rules to complete the simulation proof squares. The \( \text{(Share)} \) rule that deals with copying in rewrite steps amounts to another restriction on the construction of PSRs.

So the limitations of the basic proof method are defined by the restriction to lockstep simulation and the various PSR properties that allow this local reasoning technique to succeed. With these limitations we cannot compare many of our call-by-need evaluators. However, in addition to the proof that \( \text{badbh} \Rightarrow \text{laz} \), three other proofs that we have completed successfully with this method are given in Appendix B.1. The next chapter looks at how we can get around some of the restrictions.
Chapter 10

Advanced No Leak Proofs

The lockstep-simulation proof checker of Chapter 9 has many shortcomings which prevent us comparing many graph evaluators. This chapter introduces a series of extensions to the basic method.

Section 10.1: In the call-by-need evaluator comparisons we usually want to prove $A \not\subset B$ with respect to a translation between the initial states, the program states, graphs containing only expression nodes and no control structures. The basic lockstep-simulation proof checker needs to be told the full translation between all redex states, including control structures. We investigate the use of completion to allow the simulation proof to begin with an initial translation and infer the translation between control structures as needed.

Section 10.2: The local reasoning used in our simulation proofs seems to be inadequate for some evaluator comparisons. Sometimes a translation cannot be determined locally by breaking a graph into small pieces, translating the pieces and then building them back into a translated graph, in the way that PSR translation does. Some simulation proofs need a more global translation relation.

Section 10.3: The lockstep-simulation method is fundamentally inadequate for some proofs. We look at how non-lockstep behaviour can be reasoned about, including $k$-lockstep and many-to-one step examples.

Section 10.4 concludes Part III by showing the relationship between several call-by-need evaluators.

10.1 Completion of Proportional Space Relations

In the simulation proofs presented in Chapter 9 we had to state the full translation between all possible states. Really, we are only interested in a simulation proof for possible valid initial states. That is, for graphs in an initial translation like $idX$.

Unfortunately, checking a simulation proof with such a translation will soon fail because the translation knows nothing about control structures.

Example 10.1 (Incomplete translation makes simulation fail)

Approaching the proof of $\texttt{badbl} \not\subset \texttt{lazy}$ in this way, using the translation $idX$ generates the squares shown below. For (Let)-redexes everything is fine (10.1); for (Push)-redexes the square (10.2) fails because $idX$ does not know how to translate pushed arguments; similarly, the (Lookup)-redexes square (10.3) fails. The (Reduce) and (Update) redexes are not included in $idX$. 

155
\[
\begin{align*}
\{g \cdot a, b \cdot s\}, \{\}, & \quad \text{(Id, let)} & \{b \cdot s, k \cdot n, m \cdot p\}, \{g \cdot a, t \cdot o\}, \\
g \mapsto & \text{let } h = A \text{ in } C, & k \mapsto A[l/h], m \mapsto C[l/h], \\
a \mapsto & \text{let } y = E \text{ in } X, & n \mapsto E[o/y], p \mapsto X[o/y].
\end{align*}
\]

(\text{Let} \downarrow) \uparrow (\text{Let})

\begin{align*}
\{g \cdot a, b \cdot s\}, \{\}, & \quad \text{(Ren, Ren, HRen, HRen)} & \{b \cdot s, k \cdot n, m \cdot p\}, \{g \cdot a, t \cdot o\}, \\
g \mapsto & C[a/h], a \mapsto A[a/h], & k \mapsto A[l/h], m \mapsto C[l/h], \\
a \mapsto & X[b/y], b \mapsto E[b/y], & n \mapsto E[o/y], p \mapsto X[o/y].
\end{align*}

\[\text{(Push)} \downarrow \text{(Push)}\]

\begin{align*}
\{e \cdot a, a \cdot b\}, \{\}, & \quad \text{(Ren)} & \{i \cdot j, a \cdot b\}, \{e \cdot a\}, \\
e \mapsto & A, a \mapsto f : b, & i \mapsto A, a \mapsto f : b, \\
a \mapsto & F, b \mapsto x : s, & j \mapsto F, b \mapsto x : s.
\end{align*}

\[\text{(LookupBad)} \downarrow \text{(Lookup)}\]

\begin{align*}
\{c \cdot a, b \cdot s\}, \{\}, & \quad \text{(Id, Var)} & \{b \cdot s, d \cdot x\}, \{c \cdot a\}, \\
c \mapsto & d, & \{\}, \\
a \mapsto & x.
\end{align*}

\[\text{(Lookup)}\]

\begin{align*}
\{d \cdot x, a \cdot b\}, \{\}, & \quad () & \{d \cdot x, a \cdot b\}, \{\}, \\
c \mapsto & d, a \mapsto \# c \ b, & c \mapsto d, a \mapsto \# c \ b, \\
a \mapsto & \bot, b \mapsto \# a \ s, & a \mapsto \bot, b \mapsto \# a \ s.
\end{align*}

\[\square\]

We could try to complete the translation by hand, introducing suitable new PSR rules to complete the failed squares and then checking whether the extended translation is a simulation. The idea is similar to Knuth-Bendix completion [KB70], used to obtain term rewriting systems that are confluent and terminating.

Like Knuth-Bendix completion, we add a PSR completion loop to our proof checker which infers new PSR rules until a complete simulation proof is found. The process is not entirely automatic however. This completion loop could repeat endlessly if we make a bad choice of new rules, or if a lockstep simulation proof is not possible and nothing else happens to prevent the introduction of new rules. There can be many different new PSRs which could be used to complete a particular square and whichever is chosen the completion loop is not guaranteed to terminate. So we propose to use a proof editor which infers possible new rules and asks the user to choose one.

The general form of an incomplete proof square is as follows.
\[ \cdot \Delta \rightarrow^* \cdot \Delta^* \]
\[
\downarrow
\{ \cdot \nabla \rightarrow^* \cdot \nabla_{i} \}_{i=1}^{n}
\]

Several different candidate square bottoms may be generated using different PSR rules, all of which have in common that \( \Delta^* \not\rightarrow \cdot \nabla_{i} \). To complete the square, all we need is a new rule, or perhaps some new rules, to rewrite one \( \cdot \nabla_{i} \) to a configuration \( \Delta^* \Rightarrow \Delta \). Or maybe the new rule should not rewrite \( \cdot \nabla_{i} \), but some other configuration occurring in one of the bottom traces. So there are two problems: how to infer a suitable new rule (Section 10.1.1) and which new rule to choose (Section 10.1.2).

### 10.1.1 PSR rule inference

A new rule can be inferred for each \( \cdot \nabla_{i} \) to rewrite it to \( \cdot \Delta^* \) by \textit{inferPSR} (Definition 10.1). The correctness condition for this inferred rule (Proposition 10.1) is that applying it to \( \cdot \nabla_{i} \) produces a new bottom-right configuration which is implied by \( \cdot \Delta^* \), so the square completes.

**Definition 10.1 (PSR rule inference)**

\[
\text{inferPSR } \langle \Theta, \Phi, P, Q \rangle \langle \Theta', \Phi', P', Q' \rangle
= \langle \{ \}, \{ \}, P - P', Q - Q' \rangle \rightarrow \langle \{ x : y \in \Theta' \cup \Phi' \mid x \in \text{fv } P - P' \cup \text{dom } P' - P, \]
\[
y \in \text{fv } Q - Q' \cup \text{dom } Q' - Q \}, \Phi' \cup \Theta - \Theta', P' - P, Q' - Q \rangle
\]

\textit{inferPSR} introduces a rule \( \text{(new)} \) which rewrites its first argument to a configuration implied by its second argument. Essentially, it subtracts the configuration components to form a rule which bridges the gap. That is, it removes the nodes in \( P \) and \( Q \) not in \( P' \) and \( Q' \) and adds those in \( P' \) and \( Q' \) not in \( P \) and \( Q \). The correspondence pattern in \( \text{(new)} \) is formed from any pairs in \( \Phi' \) not already in \( \Phi \). It also includes any pairs in \( \Theta - \Theta' \) because the configuration garbage collector may have removed such pairs from \( \Phi' \) which are necessary to make \( \text{(new)} \) a PSR rule. A suitable goal pattern is formed by taking pairs from \( \Theta' \) and \( \Phi' \) whose elements are free in \( P \) and \( Q \) or bound in \( P' \) and \( Q' \).

In addition to the inferred rule \( \text{(new)} \), any rules required to make the PSR self-consistent are also introduced. There is no guarantee that the result is a valid set of PSR rules, so they are checked before being presented to the user.

**Proposition 10.1 (Correctness of PSR rule inference)**

If \( r = \text{inferPSR } \cdot \nabla_{i} \Rightarrow \cdot \Delta^* \) and \( \cdot \nabla_{i} \rightarrow^r \cdot \nabla_{i} \) then \( \cdot \Delta^* \Rightarrow \cdot \nabla_{i} \).

**Proof**

Let \( r \), \( \cdot \nabla_{i} \) and \( \cdot \Delta^* \) be defined as in Definition 10.1. It follows that \( \cdot \nabla_{i} \rightarrow^r \cdot \nabla_{i}' \) and \( \cdot \Delta^* \Rightarrow gc \cdot \Delta \Rightarrow \cdot \nabla_{i} \).

**Example 10.2 (PSR rule inference)**

The following new PSR rule, renamed \((Id.:)\) here, is inferred for the incomplete \((Push)\) square in Example 10.1. Adding this rule to the translation \( \equiv \) not only completes
the \((\text{Push})\) square — which looks exactly like it did in our discussion of proof checking in Example 9.5, square (9.6) — but it also adds the \((\text{Reduce})\) square (9.9) to the proof.

\[
\langle \{\}, \{\}, \{a \mapsto y : t\}, \{b \mapsto x : s\} \rangle \rightarrow \langle \{t \cdot s, y \cdot x\}, \{a \cdot b\}, \{\}, \{\} \rangle \quad (\text{Id. :})
\]

PSR inference as described here always adds just one new PSR (and any sub-rules needed for self consistency) to complete a square. This is probably the right thing to do because it finds the least general translation needed to complete the simulation proof. It can be surprising sometimes though. A more general translation which completes the square in more steps may appear more natural.

**Example 10.3 (Inferred translation is more specific)**

The PSR rules inferred for the incomplete \((\text{Lookup})\) square in Example 10.1 are shown here. The \((\text{Var} \# \perp \#)\) rule translates an update marker with a variable \(d\) to an update marker with a black hole. For self-consistency, the rule \((\text{Var} \perp \#)\) is introduced.

\[
\langle \{\}, \{\}, \{a \mapsto \# c, b \mapsto \# d\}, \{a \mapsto \perp, b \mapsto \# a, s\} \rangle
\rightarrow\langle \{b \cdot s\}, \{c \cdot a, a \cdot b\}, \{\}, \{\} \rangle \quad (\text{Var} \# \perp \#)
\]

\[
\langle \{\}, \{\}, \{c \mapsto d\}, \{a \mapsto \perp\} \rangle \rightarrow \langle \{\}, \{c \cdot a\}, \{\}, \{\} \rangle \quad (\text{Var} \perp \#)
\]

This is less general than the translation \(\text{badh} \mapsto \text{badbh} \leftrightarrow \text{lazy}\) throughout Chapter 9 because \((\text{Var} \# \perp \#)\) recognises that update markers and black holes occur together, instead of translating them separately with \((\text{Id. } \#)\) and \((\text{Var} \perp \#)\). This makes the \((\text{Lookup})\) square (10.5) slightly simpler than (9.2). The \((\text{Var} \# \perp \#)\) rule also adds an \((\text{Update})\)-redex square (10.6) to the simulation proof, which now completes as shown below. Again, it is slightly simpler than (9.12).

\[
\langle \{c \cdot a, b \cdot s\}, \{\}, \{c \mapsto d\}, \{a \mapsto \perp\} \rangle \quad (\text{Id. Var}) \quad \langle \{b \cdot s, d \cdot x\}, \{c \cdot a\}, \{\}, \{\} \rangle
\]

\[
(\text{LookupBad}) \downarrow (\text{Lookup}) \quad \uparrow \quad \tag{10.5}
\]

\[
\langle \{d \cdot x, a \cdot b\}, \{\}, \{c \mapsto d, a \mapsto \# c, b\}, \{a \mapsto \perp, b \mapsto \# a, s\} \rangle \quad (\text{Var} \# \perp \#) \quad \langle \{b \cdot s, d \cdot x\}, \{c \cdot a\}, \{\}, \{\} \rangle
\]

\[
(\text{UpdateBad}) \downarrow (\text{Update}) \quad \uparrow \quad \tag{10.6}
\]

\[
\langle \{f \cdot y, g \cdot t\}, \{\}, \{f \mapsto \lambda j. A, i \mapsto \lambda j. A\}, \{y \mapsto \lambda x. E, a \mapsto \lambda x. E\} \rangle \quad (\text{Share, Id. } \lambda, \text{ Ren, HRen}) \quad \langle \{g \cdot t, m \cdot o\}, \{f \cdot y, i \cdot a, n \cdot p\}, \{m \mapsto A[n/j]\}, \{o \mapsto E[p/x]\} \rangle
\]

\[
\langle \{g \cdot t, m \cdot o\}, \{f \cdot y, i \cdot a, n \cdot p\}, \{m \mapsto A[n/j]\}, \{o \mapsto E[p/x]\} \rangle \quad \uparrow \quad \tag{10.6}
\]
10.1.2 Controlling PSR completion

Returning to the general form of an incomplete simulation square (10.4), there may be several bottom-right configurations to infer a new rule from, so there could be several possible inferred translations. It is also possible for some $\nabla^+_i$ — or all of them — to fail to give a valid PSR. In this case we go back to the previous configuration in the square bottom trace $\nabla \rightarrow^* \nabla^+_i$ and try to infer a PSR from there. The incomplete square bottoms (10.4) are presented as a set, but really they are a tree and each trace in our set is a path from the root of the tree $\nabla$ to one of the leaves $\nabla^+_i$. Suppose that one of the square bottom traces is $\nabla \rightarrow^* \Delta \rightarrow \nabla^+_i$, we cannot infer a PSR from $\nabla_i$ and we can infer a PSR from $\Delta$. We risk introducing a more complicated translation than necessary if $\Delta$ is on another bottom trace which does result in a valid PSR. In such cases the inferred rule is omitted from the solution set.

Definition 10.2 (Inferred rule selection)

$\nabla \rightarrow^* \Delta^* \nabla^+_i$

\[
\text{choices} = \left\{ \begin{array}{l}
\text{for } \nabla \rightarrow^* \nabla^+_i \\
\quad \text{where newRules = \{inferPSR }\nabla \Delta \Delta^*, \Delta, \{\Delta' | \nabla \rightarrow^* \Delta' \rightarrow^* \Delta\} | \nabla \rightarrow^* \Delta \rightarrow^* \nabla^+_i]\end{array} \right\}
\]

When a proof square does not complete, the set newRules is found by using inferPSR (Definition 10.1) to infer a new rule from each configuration in the set of failed bottom traces. Each new rule is recorded along with the bottom trace configuration used, $\Delta$, and the set of configurations in the trace from $\nabla$ up to but not including $\Delta$.

The set of possible new rules presented to the user is given by choices. A rule is excluded from choices if it is inferred from $\Delta$ and $\Delta$ occurs in a square bottom trace leading to $\Delta'$ and a rule $r$ can be inferred from $\Delta'$.

The choices set is a compromise between introducing the most general or the least general new PSR. Choosing to complete a square by selecting the rule inferPSR $\nabla \Delta \Delta^*$ would give a most specific solution. Some disadvantage of this choice are:

1. It would introduce many extra rules when extended for self-consistency.
2. It is less likely to be sharable.
3. Its patterns are large, so translation may become slow.

Hence choices prefers to introduce rules from configurations as near as possible to the right side of the square. But this is still not necessarily the most general or most natural solution, as we saw in Example 10.3.

Adding PSR completion to the proof checker we can discover simulation proofs for the four examples that we were able to check. These are [bad] and [laz] and the other three summarised in Appendix B.2. Comparing evaluators with themselves using the initial translation $\text{bad}$ does not always yield a complete proof. This seems to be because configuration-pattern garbage collection can remove information which is needed to build a valid PSR rule.
10.2 Global reasoning

The PSR simulation proofs presented so far work locally. They consider only the changes that happen at the root of a graph when it is rewritten by an evaluation step. The PSR properties ensure that any changes in the rest of the graph (the generation of garbage) are covered. Attempted proofs of evaluators often fail because this local reasoning technique is inadequate in some way.

There are two ways to deal with this problem. The PSR notation can be extended to permit more global reasoning, either by enriching the PSR configuration rewrite rule notation or by making a PSR start rule do more work. Or extra properties about the evaluators under comparison can be proved to show that a seemingly incomplete PSR simulation proof suffices. Both of these ideas are used here.

10.2.1 Enriched space relations

**Example 10.4 (Inadequacy of local reasoning)**

To show $\text{ind} \not\equiv \text{lazy} \; \text{wrt} \; \frac{\text{id}_X}{\text{id}}$ we might try letting the proof editor described in the previous section complete the simulation proof. It soon becomes apparent that the completion will not terminate. First the (Lookup)-redex square is completed by inferring a rule $(\# \not\equiv \# \not\equiv)$. This adds the incomplete (Update)-redex square (10.7) to the proof. Note that there are two failed square-bottom traces.

\[
\begin{align*}
\langle \{g \cdot a, e \cdot s\}, \{\} \rangle & \rightarrow \langle \{f \cdot t, k \cdot m\}, \{d \cdot y, g \cdot a, l \cdot n\} \rangle, \\
\langle d \rightarrow \bot, e \rightarrow \#d, f \rightarrow \lambda h. A\rangle, \\
\{a \rightarrow \lambda x. E, y \rightarrow \bot, s \rightarrow \#y. t\} & \rightarrow \langle k \rightarrow A[l/h], \{m \rightarrow E[n/x]\} \rangle \\
(\text{VUpdate}) \downarrow (\text{Update})
\end{align*}
\]

\[\text{(10.7)}\]

The PSR rule $(\lambda \lambda \lambda \lambda)$ is inferred from the bottom-left configuration, with $(\text{Id} \cdot \lambda)$ and $(\lambda \lambda \lambda)$ added for self-consistency (Inferred rules from either bottom-right configuration of (10.7) are not valid PSR rules, so the proof editor only offers this one choice of inferred PSR). These rules recognise that an indirection to a value in $\text{ind}$ corresponds to a duplicated value in $\text{lazy}$.

\[
\begin{align*}
\langle \{\}, \{\} \rangle, \langle d \rightarrow I \cdot g, g \rightarrow \lambda h. A\rangle, \langle y \rightarrow \lambda x. E, a \rightarrow \lambda x. E\rangle & \rightarrow \langle \{k \cdot m\}, \{d \cdot y, g \cdot a, l \cdot n\}, \{k \rightarrow A[l/h]\}, \{m \rightarrow E[n/x]\} \rangle, \\
(I\lambda\lambda\lambda)
\end{align*}
\]

\[
\begin{align*}
\langle \{\}, \{\} \rangle, \langle d \rightarrow I \cdot g, g \rightarrow \lambda h. A\rangle, \langle y \rightarrow \lambda x. E\rangle & \rightarrow \langle \{k \cdot m\}, \{d \cdot y\}, \{k \rightarrow A[l/h]\}, \{m \rightarrow E[n/x]\} \rangle, \\
(I\lambda\lambda)
\end{align*}
\]

Accepting these rules completes Square (10.7) but it introduces the new square
(10.8) because there is a new way to translate instances of the *(Update)* rule left pattern.

\[
\begin{align*}
\{b \cdot a, e \cdot s\}, \{\}, & \quad \{f \cdot t, o \cdot p\}, \\
\langle d \mapsto \bot, e \mapsto \# f, & \quad \langle c \mapsto \text{I} c, c \mapsto \lambda i.B\rangle, \\
b \mapsto I c, c \mapsto \lambda i.B\rangle, & \quad (\# \bot \# \bot, \quad I \lambda \lambda\lambda) \quad \{d \cdot y, b \cdot a, c \cdot n, q \cdot u\}, \\
\{y \mapsto \bot, s \mapsto \# y t, & \quad \{o \mapsto B[q/i]\}, \\
a \mapsto \lambda x.E, n \mapsto \lambda x.E\} & \quad \{p \mapsto E[u/x]\}
\end{align*}
\]

\[(IUpdate) \downarrow (Update) \tag{10.8}\]

The PSR rule \((II \lambda \lambda \lambda)\) is inferred from the bottom-left configuration. Like (10.7), this square has other bottom traces too. The rules \((I \lambda \lambda \lambda), (Id.\lambda), (I \lambda \lambda)\) and \((II \lambda \lambda \lambda)\) are added for self-consistency. These rules recognise that two indirections to a value in \(\text{Ind}\) correspond to a tripped value in \(\text{Lazy}\).

\[
\langle \{\}, \{\}, \{c \mapsto \lambda i.B, d \mapsto \text{I} c, b \mapsto \text{I} c\}, \{n \mapsto \lambda x.E, y \mapsto \lambda x.E, a \mapsto \lambda x.E\}\rangle
\]
\[
\mapsto \langle \{o \cdot p\}, \{c \cdot n, d \cdot y, b \cdot a, q \cdot u\}, \{o \mapsto B[q/i]\}, \{p \mapsto E[u/x]\}\rangle
\]
\[(II \lambda \lambda \lambda)\]

\[
\langle \{\}, \{\}, \{c \mapsto \lambda i.B, d \mapsto \text{I} c, b \mapsto \text{I} c\}, \{n \mapsto \lambda x.E, a \mapsto \lambda x.E\}\rangle
\]
\[
\mapsto \langle \{o \cdot p\}, \{d \cdot y, b \cdot a, q \cdot u\}, \{o \mapsto B[q/i]\}, \{p \mapsto E[u/x]\}\rangle
\]
\[(II \lambda \lambda)\]

The completion loop continues like this, inferring \((III \lambda \lambda \lambda \lambda), (III \lambda \lambda \lambda \lambda)\), and so on until the user of the proof editor has enough déjà-vu.

To make the completion loop in Example 10.4 terminate we need to translate \(n\) indirection nodes into \(n\) copies of a \(\lambda\)-value, for any \(n\). A finite set of PSR rules is never going to be enough. This translation could be expressed inductively (10.9), but this breaks the PSR restrictions which ensure that pattern translation terminates.

\[
\langle \{\}, \{\}, \{a \mapsto \text{I} b, b \mapsto \lambda x.E\}, \{c \mapsto \lambda y.F, d \mapsto \lambda y.F\}\rangle
\]
\[
\mapsto \langle \{b \cdot d\}, \{a \cdot b\}, \{b \mapsto \lambda x.E\}, \{d \mapsto \lambda y.F\}\rangle
\]
\[(10.9)\]

A different way of extending the PSR framework to express this relationship is to employ an indexed notation. We introduce one meta-PSR rule indexed by \(n\) which stands for an infinite family of rules, all of which are PSR rules individually. This is similar to a technique which makes Knuth-Bendix completion more likely to terminate [TW93].
Example 10.5 (Indexed PSR rules)

\[
\begin{aligned}
\text{start} &\xrightarrow{\text{Id.Apply}} \{ (\text{Id.Apply}), (\text{Id.Var}), (\text{Id.let}), (\text{Id.}), (\text{Id.}), (I^n \lambda \lambda^n \lambda), (I^n \lambda \lambda^n) \} \\
\langle \{ \}, \{ \bar{a} \mapsto I b, b \mapsto \lambda x. E \}, \{ a \mapsto \lambda x'. E', \bar{a}' \mapsto \lambda x'. E' \} \rangle &\xrightarrow{\text{Id.}: I, \text{Id.} \#} \langle \{ e \mapsto E[y/x], e' \mapsto E'[y'/x'] \}, (I^n \lambda \lambda^n) \lambda \rangle \\
\langle \{ \}, \{ \bar{a} \mapsto I b, b \mapsto \lambda x. E \}, \{ a \mapsto \lambda x'. E' \} \rangle &\xrightarrow{\{ e \mapsto E[y/x], e' \mapsto E'[y'/x'] \}, (I^n \lambda \lambda^n \lambda) \rangle
\end{aligned}
\]

The rule \((I^n \lambda \lambda^n \lambda)\) translates all \(n\) indirections to the same \(\lambda\)-value into \(n+1\) copies of a \(\lambda\)-value. For any \(n \in \mathbb{N}\) this meta-rule yields a valid PSR rule. For self-consistency we need the second meta-rule \((I^n \lambda \lambda^n)\) which translates \(n\) indirections into \(n\) copies of the value for the situations where node \(b'\) in \((I^n \lambda \lambda^n \lambda)\) is garbage. Note that \((I^0 \lambda \lambda^0 \lambda)\) is the same as \((\text{Id.}\lambda)\). These meta-rules use a shorthand vector notation: \(\overline{a'} \mapsto I b\) in the left pattern stands for \(a_1 \mapsto I b, \ldots, a_n \mapsto I b\) and \(\overline{a' a'}\) in the right pattern stands for \(a_1 a_1, \ldots, a_n a_n\).

The simulation proof now proceeds as normal.

Example 10.6 (Simulation with indexed PSR rules)

There are three squares for (Update) redexes: (10.10) below is for the case where \texttt{laz\_update} updates by copying a \(\lambda\)-value and \texttt{ind\_update} updates by creating an indirection to the value. The other two squares are in Appendix B.5: (B.23) and (B.24) are both for indirection updates. In (B.23) the value pointed to by the indirections is reachable in \texttt{laz\_update}, in (B.24) it is not. Note that these squares complete without needing the (Share) rule that we used for the (Update)-redex squares in other proofs.

\[
\begin{bmatrix}
\{ a \cdot a', s \cdot s' \}, \{ \}, \\
\langle \bar{b} \mapsto I a, a \mapsto \lambda x. E, \\
y \mapsto \perp, s \mapsto \# y t \rangle, \\
\langle \bar{b} \mapsto \lambda x'. E', a' \mapsto \lambda x'. E', \\
y' \mapsto \perp, s' \mapsto \# y' t' \rangle &\quad (I^n \lambda \lambda^n \lambda, \text{Id.}, \text{Id.} \#) &\quad \langle \{ e \mapsto E[z/x], \}, \{ e' \mapsto E'[z'/x'] \}, \bar{b}', a \cdot a', y \cdot y', z \cdot z' \rangle \\
\langle \bar{b} \mapsto I a, a \mapsto \lambda x. E, \\
y \mapsto I a \rangle, \\
\langle \bar{b} \mapsto \lambda x'. E', a' \mapsto \lambda x'. E', \\
y' \mapsto \lambda x'. E' \rangle &\quad \langle \{ e \mapsto E[z/x], \}, \{ e' \mapsto E'[z'/x'] \}, \bar{b}', a \cdot a', y \cdot y', z \cdot z' \rangle
\end{bmatrix}
\]

\(\downarrow\) (Update)

\[
\begin{bmatrix}
\{ y \cdot y', t \cdot t' \}, \{ \}, \\
\langle \bar{b} \mapsto I a, a \mapsto \lambda x. E, \\
y \mapsto I a \rangle, \\
\langle \bar{b} \mapsto \lambda x'. E', a' \mapsto \lambda x'. E', \\
y' \mapsto \lambda x'. E' \rangle &\quad (I^{n+1} \lambda \lambda^{n+1} \lambda) &\quad \langle \{ e \mapsto E[z/x], \}, \{ e' \mapsto E'[z'/x'] \}, \bar{b}', a \cdot a', y \cdot y', z \cdot z' \rangle \\
\langle \bar{b} \mapsto I a, a \mapsto \lambda x. E, \\
y \mapsto \perp, s \mapsto \# y t \rangle, \\
\langle \bar{b} \mapsto \lambda x'. E', a' \mapsto \lambda x'. E', \\
y' \mapsto \perp, s' \mapsto \# y' t' \rangle &\quad \langle \{ e \mapsto E[z/x], \}, \{ e' \mapsto E'[z'/x'] \}, \bar{b}', a \cdot a', y \cdot y', z \cdot z' \rangle
\end{bmatrix}
\]

The squares for rules not involving \(\lambda\)-values complete as normal: we get (B.3) for (Lookup), (B.5) for (Push) and (9.10) for (Let). The proof does not quite succeed though as one of the (Reduce) squares present a new problem.

When a \texttt{laz\_reduce} (Reduce) step corresponds to an \texttt{ind\_reduce} (IReduce) step there is no problem, we get squares (B.25) and (B.26). However, as (IReduce) copies the value before reducing it, we do need the (Share) rule.
The problem case occurs where an `\text{lazy}` (Reduce) step corresponds to an `\text{ind}` (IReduce) step. We get the following incomplete square.

\[
\begin{align*}
\{a \cdot a', s \cdot s'\}, \{\}\quad & \quad \emptyset
\\
\langle \overline{b} \mapsto I a, a \mapsto \lambda y. E, \quad s \mapsto x : t \rangle
\\
\{\overline{b} \mapsto \lambda y'. E', a' \mapsto \lambda y'. E', \quad s' \mapsto x' : t' \rangle
\end{align*}
\]

(\text{Reduce}) \downarrow (\text{Reduce})

\begin{align*}
\{a \cdot a', t \cdot t'\}, \{\}\quad & \quad \emptyset
\\
\langle \overline{b} \mapsto I a, a \mapsto E[x/y], \quad s \mapsto \lambda y'. E' \rangle
\\
\{\overline{b} \mapsto \lambda y'. E', a' \mapsto E'[x'/y'] \} & \quad \mapsto \quad \emptyset
\\
\end{align*}

\begin{align*}
\{e \cdot e', x \cdot x', t \cdot t'\}
\\
\{b \mapsto a \cdot a', z \cdot z'\}
\\
\{e \mapsto E[z/x]\}
\\
\{a \mapsto a', z \mapsto z'\}
\\
\{b \mapsto I a, e \mapsto E[z/x]\}
\\
\{\overline{b} \mapsto \lambda y'. E', \quad \overline{e} \mapsto E'[x'/y'] \}
\end{align*}

(10.11)

If the situation described by (10.11) happened during evaluation, `\text{ind}` would not only fail to bound the space usage of `\text{lazy}`, it would fail to produce the right result sometimes. If there are indirections to a `\lambda`-value and that `\lambda`-value is specialised in-place then the values of all the indirections get changed too. This would be wrong; so it is not true that `\text{ind} \models \text{laazy}` with respect to the full translation defined by `\mapsto`. \hfill \Box

10.2.2 Simulation-supporting lemmas

The disastrous failed proof of `\text{ind} \models \text{laazy}` in Example 10.6 can be fixed by showing that the circumstances that cause Square (10.11) to fail never occur if all initial states are in the `\mapsto` translation. That is, we need to show that any value to which there is an indirection is never reduced. Instead of trying to extend the PSR framework to encode this property we just prove a lemma (Proposition 10.2).

**Proposition 10.2 (Values with indirections are never reduced)**

\[ G \mapsto X H \land G \rightarrow^* \text{ind} G'a, s \mapsto (a' \mapsto I a) \in G' \implies \exists x, E, y, t \cdot \{a \mapsto \lambda x. E, s \mapsto \#y : t\} \subset G' \]

If a graph \( G' \) in the evaluation trace of \( G \) on `\text{ind}` contains an indirection to the root expression then the root expression is a `\lambda`-value and the root stack node is an update marker.

**Proof**

Graph \( G \) contains no indirections Initially if it is in dom `\mapsto X`. The result follows by induction on the evaluation trace. \hfill \Box

Now the proof of `\text{ind} \models \text{laazy}` wrt `\mapsto X` can be completed.

**Example 10.7 (Restricting translation with a lemma)**

With the knowledge that values about to be reduced are not referred to by any indirections, the (Reduce)-redex square can be completed (10.12). In fact, it is just the same as Square (9.9) in the `\text{badbh} \models \text{laazy}` proof.
Proposition 10.3 (Indirections do not leak or repair)

\[ \text{\texttt{ind} } \not\equiv \text{\texttt{lazy}} \text{ wrt } \not\vdash^X \]

\[ \text{Proof} \]

The squares for this proof are the same as they were for \texttt{ind} \texttt{\not\equiv lazy} but with the direction of the translation reversed. See the squares \texttt{Lookup(B.3), Push(B.5), Let(9.10), VUpdate(10.10), IUpdate(B.23,B.24), Reduce(10.12), IReduce(B.25,B.26).} \]

Two more call-by-need evaluator comparisons are proved in Appendix B.3 using this supporting lemma technique, including \texttt{host} \texttt{\not\equiv lazy} which unlike its inverse \texttt{host} \texttt{\equiv lazy} fails using the basic PSR simulation-checking method.

10.3 Non-lockstep reasoning

Sometimes a simulation proof fails because the evaluators are not lockstep. In the simplest cases this means that a \texttt{B}-redex translates to an \texttt{A}-irreducible graph. In more complicated cases, the most natural translation between steps is not a lockstep simulation.

10.3.1 Non-lockstep termination

A simulation proof of \texttt{A} \texttt{\equiv B} can fail if \texttt{A} stops before \texttt{B}.

Example 10.8 (Premature termination prevents proof)

Evaluator \texttt{nobh} does not black hole in its \texttt{(LookupNh)} rule, so after the rewrite the black-holed node in \texttt{lazy} could correspond to any expression except \texttt{⊥} on \texttt{nobh}.

Therefore, to show \texttt{nobh} \texttt{\equiv lazy} we define the translation \( \not\vdash^X \) as follows — there are three new rules, \( \text{(Var\,\bot)} \) was defined in Example 10.3.

\[ \text{\texttt{nobh} } \not\vdash^X = \text{\texttt{(start, } \not\vdash \cup \{ (Id:\,.), (Id:\#), (Var:\bot), (\lambda \bot), (Apply:\bot), (let:\bot) \})} \]

\[ \langle \{\}, \{\}, \{a \mapsto \lambda x.E\}, \{b \mapsto \bot\} \rangle \not\vdash^X \langle \{\}, \{a \cdot b\}, \{\}, \{\} \rangle \quad (\lambda \bot) \]

\[ \langle \{\}, \{\}, \{a \mapsto E\ x\}, \{b \mapsto \bot\} \rangle \not\vdash^X \langle \{\}, \{a \cdot b\}, \{\}, \{\} \rangle \quad (Apply \bot) \]

\[ \langle \{\}, \{\}, \{a \mapsto \text{let}\ x = E\ in\ B\}, \{b \mapsto \bot\} \rangle \not\vdash^X \langle \{\}, \{a \cdot b\}, \{\}, \{\} \rangle \quad (let \bot) \]

The square top (10.13) is found by translating the left pattern of the \texttt{(Lookup)} rule.
\[
\left\{ a \cdot a', s \cdot s' \right\}, \{\}, \left\{ a \mapsto x \right\}, \left\{ a' \mapsto x' \right\} \right) 
\xrightarrow{(Id \cdot Var)} 
\left\{ x \cdot x', s \cdot s' \right\}, \{ a \cdot a' \}, \left\{ \right\}, \left\{ \right\} 
\tag{10.13}
\]

Not every instance of \( \{ a \mapsto x \} a, s \) is a \textsf{nobh} redex, so the lockstep simulation proof fails.

In cases like this we need to extend the definition of lockstep simulation to include the possibility of uneven termination. Proposition 10.4 is sufficient for the examples we deal with here.

**Proposition 10.4 (Lockstep or bounded termination)**

If \( G \ intends H \land H \xrightarrow{\#e} B H' \Rightarrow (\exists G' \cdot G \xrightarrow{\#e} A G' \land G' \xrightarrow{\#e} H') \land (H' \nrightarrow B \land G' \nrightarrow A) \) then \( \textsf{nobh} \Rightarrow \textsf{lazy} \) wrt \( \xrightarrow{\Longrightarrow} \).

**Proof**

A simple extension of the argument in Proposition 9.1.

**Example 10.9 (Simulation with bounded termination)**

Returning to the square top (10.13) of Example 10.8, we proceed by cases. If \( x' = e \), \( x' = a' \) or \( x' \) is free then \textsf{lazy} terminates after the (Lookup). Otherwise the (Lookup) redex is an instance of \( \{ a' \mapsto x', x' \mapsto E' \} a, s \). If \( E' = \bot \) or \( E' = \mathcal{E} \) then \( \textsf{lazy} \) terminates after the (Lookup). Otherwise \( E' \) is an expression. If it is a variable \( y' \) we get Square (10.14). For other expressions the squares (B.20), (B.21) and (B.22) in Appendix B.5 are generated.

\[
\left\{ a \cdot a', s \cdot s' \right\}, \{\}, \left\{ a \mapsto x, x \mapsto y \right\}, \left\{ a' \mapsto x', x' \mapsto y' \right\} \right) \xrightarrow{(Id \cdot Var, \text{Id \cdot Var})} \left\{ y \cdot y', s \cdot s' \right\}, \{a \cdot a', x \cdot x'\}, \left\{ \right\}, \left\{ \right\} 
\tag{10.14}
\]

\( \text{(LookupNbh)} \downarrow \text{(Lookup)} \)

The rest of the \textsf{nobh} \( \Rightarrow \textsf{lazy} \) proof completes in lockstep without any difficulties. The squares are: \textit{Let}(9.10), \textit{Reduce}(9.9), \textit{Push}(9.6), \textit{Update}(B.12, B.13, B.14, B.15).

### 10.3.2 Non-lockstep and global reasoning

In Section 6.2.2 we suggested that our version of Sestoft’s semantics \textsf{sest} and our standard evaluator \textsf{lazy} have exactly the same space behaviour. To prove this we need to combine two extensions to the basic proof method.

1. Non-lockstep termination. Much like \( \text{(LookupNbh)} \) in \textsf{nobh}, the rule \( \text{(ScLookup)} \) in \textsf{sest} needs any variable it looks up to be defined somewhere in the graph.
2. Global reasoning. When \texttt{sest} looks up a variable \( x \) it copies the definition of \( x \) into the root expression node. When \texttt{lazy} looks up a variable \( x \) it moves the root expression address to \( x \). So after a sequence of several lookups the translation between the \texttt{sest} state and the \texttt{lazy} state cannot easily be defined locally. It certainly cannot be defined by a finite set of PSR rules.

We choose to give a \textit{global} definition of the translation of \texttt{sest} stack nodes into \texttt{lazy} stack nodes. The PSR framework does not directly allow this, but we can accommodate it by extending the \textit{start} rule definition. This is a more apt solution than having a family of stack-translation PSR rules because a graph only contains one stack structure.

\textbf{Example 10.10 (Global translation by extending \textit{start})}

The translation \( \sest \rightarrow \textit{start}_\sest \) from \texttt{sest} states to their \texttt{lazy} equivalents uses \textit{start}_\texttt{sest} to translate all stack nodes and black holes simultaneously. Then expressions translate locally using PSR rules as normal.

\[
\sest \rightarrow \textit{start}_\sest, \{(\text{Id.Var}), (\text{Id.Lambda}), (\text{Id.Apply}), (\text{Id.let})\}
\]

\[
\textit{start}_\sest (s_i \mapsto \#a_i, s_{i-1}, a_i \mapsto \bot | i \in U) \quad \left\{ \begin{array}{l}
   s'_i \mapsto \#a'_i, s'_{i-1}, a'_i \mapsto \bot | i \in U \\
   s'_i \mapsto b'_i : s'_{i-1} | i \in P \\
   a'_m \mapsto E' \quad H(a'_m, s'_n)
\end{array} \right.
\]

\[
\langle \{a \cdot a' \} | b_i, b'_i | i \in P \}, \{a_i, a'_i | i \in \{0\} U \}, G(a \mapsto E), H(a' \mapsto E') \rangle
\]

where

\[
n \in \mathbb{N}, U P = \{1, \ldots, n\}, s_0 = \epsilon, m = \max U, u_i = \max(\{0\} U - \{i, \ldots, n\})
\]

The rule \textit{start}_\texttt{sest} is actually a family of rules. It has an instance for each number \( n \) of stack nodes in the graph being translated. For each \( n \), any subset \( U \) of the stack nodes may be update markers and the rest are pushed arguments in \( P \). Pushed arguments in the \texttt{sest} graph translate unchanged to the \texttt{lazy} graph. If the \( i \)th stack node in the \texttt{sest} graph is an update marker for the black-holed node \( a_i \) then it translates to an update marker for \( a'_u \). The node \( a'_m \) is also black holed; it was the control-expression address immediately before the update-marker node \( s_i \) was created. So for the first update marker at \( s_j, u_j = 0 \); for the next update marker \( u_k = j \); and so on. The control expressions \( E \) and \( E' \) are moved to new nodes by \textit{start}_\texttt{sest} because references to \( a_0 \) in the rest of the \texttt{sest} graph translate to references to \( a'_0 \), not references to \( a'_m \).

\( \square \)

An alternative \textit{start} rule does not have to follow the restrictions of PSR rules needed to guarantee termination of pattern translation. Nor does it need to satisfy \textit{SubTrace} or be \textit{sharable}, as these conditions relate to configuration rewriting. But any \textit{start} rule must be an invertible function and it must preserve the guarantee that PSRs are space relations (Definition 8.2). It must also preserve \( G \textit{Trace} \), \( G \implies H \Rightarrow gcG \implies gcH \), so that garbage does not need to be considered in the simulation proof.

\textbf{Example 10.11 (Safety of extended \textit{start})}

The translation \( \sest \rightarrow \textit{start}_\sest \) of Example 10.10 is a space relation because the new \textit{start} rule \textit{start}_\texttt{sest} removes the same number of nodes from each graph. Further, all these nodes are root-reachable so they are not affected by garbage collection.

\( \square \)
Constructing a simulation proof with an alternative start rule proceeds as normal.

**Proposition 10.5** \( \text{\textbf{ses}\text{\textbf{t}} \Leftrightarrow \text{\textbf{laz}} \text{\textbf{y}} \text{\textbf{y}} \text{\textbf{y}} \text{\textbf{y}} \text{wrt} \text{\textbf{ses}} \text{\textbf{t}}} \)

First \( \text{\textbf{ses}\text{\textbf{t}}} \Leftrightarrow \text{\textbf{laz}} \text{\textbf{y}} \). For the (Lookups)-redexes, the square tops are found by extending \( \text{\textbf{start}}_{\text{\textbf{ses}\text{\textbf{t}}} \text{\textbf{t}}} \) to patterns. The vector notation abbreviates the collection of pairs put into the initial configuration by \( \text{\textbf{start}}_{\text{\textbf{ses}\text{\textbf{t}}} \text{\textbf{t}}} \) and lets us ignore the question of exactly what stack nodes a (Lookups)-redex might contain.

\[
\text{\textbf{start}}_{\text{\textbf{ses}\text{\textbf{t}}} \text{\textbf{t}}}((\{a_0, s_n\}, \{a'_m \mapsto x'\} a_m, s_n)) = (\{ a \cdot a', b_i \cdot b'_i \}, \{ a_i \cdot a'_i \}, \{ \}, \{ a' \mapsto x' \})
\]

Pattern translation using \( \text{\textbf{Id}}. \text{\textbf{Var}} \) produces the square top (10.15).

\[
\begin{align*}
\langle & a \cdot a', b_i \cdot b'_i \rangle, \{ a_i \cdot a'_i \}, \{ a \mapsto x \}, \{ a' \mapsto x' \} \\
\text{(Id.Var)} \quad \rightarrow \quad & \langle x \cdot x', b_i \cdot b'_i \rangle, \{ a \cdot a', a_i \cdot a'_i \}, \{ x \mapsto E \}, \{ x' \mapsto E' \} \\
\text{(ScLookups) \ down \ (Lookups)} \quad \downarrow \\
\langle & x \cdot x', a_i \cdot a'_i \rangle, \{ x \mapsto E \}, \{ x' \mapsto E' \}
\end{align*}
\] (10.16)

The other proof squares are constructed similarly. They appear in Appendix B.5: Let(B.27), Push(B.28), Reduce(B.29) and Update(B.30). To show \( \text{\textbf{ses}} \text{\textbf{t}} \Leftrightarrow \text{\textbf{laz}} \text{\textbf{y}} \text{\textbf{y}} \text{\textbf{y}} \text{\textbf{y}} \text{wrt} \text{\textbf{ses}} \text{\textbf{t}}} \) the same squares are required. \( \square \)

### 10.3.3 \( k \)-steps to 1-step simulation

Sometimes a simulation proof needs to use a more general form of 1-lockstep simulation where \( \text{\textbf{A}} \) may take up to \( k \) steps to every one taken by \( \text{\textbf{B}} \) in the comparison \( \text{\textbf{A}} \Leftrightarrow \text{\textbf{B}} \). If the user supplies the value of \( k \), extending the proof-checking method to cover these cases is straightforward. After constructing a simulation square top, a bottom-left configuration is constructed for each number of steps in \( \text{\textbf{A}} \) from 0 to \( k \). For proof completion this may extend the number of choices presented to the user.
Example 10.12 (Three-to-one step simulation proof)

We claimed that $\text{prsh}$ improves the space usage of the basic lists-and-projections evaluator $\text{proj}$ by shortcutting both projections to a cons when one of them is requested. Here we prove that it does not make things worse, $\text{proj} \equiv_{idX} \text{prsh}$. The bulk of this proof is presented in Appendix B.4. The translation $\equiv_{idX}$ defined there uses an extended $\text{start}$ rule to translate most of the stack structures, and other nodes are translated by PSR rules. A local translation using only PSR rules turned out not to be possible.

Essentially, these evaluators proceed in lockstep, looking up variables, reducing $\lambda$-bodies and case-expressions and if-expressions and so on. The squares for all these cases are simple and are omitted to save space. However, they need the support of a lemma (Proposition B.13) to support the uses of the $(\text{Share})$ rule.

When $(\text{LetCons'})$ in $\text{prsh}$ allocates a pattern binding $(h : t) = E$, introducing two projection nodes which point to $E$ as well as each other, the simulation square (B.33) uses the translation rule $(\pi h t h t)$ defined in Appendix B.4 to take account of the differences between the two evaluators.

If a head projection is requested by $(\text{PihPush})$ the evaluators remain in lockstep as shown by squares (B.34) and (B.35) — depending on whether the tail projection is reachable in the $\text{proj}$ graph. The different stack structures are accounted for by $\text{start}_\pi$. Squares (B.36) and (B.37) cover the analogous cases where the first projection to be requested is the tail.

Once the list constructor value is found, the projection updates also proceed in lockstep. These are described by squares (B.38), (B.39), (B.40) and (B.41) depending on which projection is being updated and whether the other projection is reachable or not. The translation required after the first update turns the other projection in the $\text{proj}$ graph — the one that has not yet been requested — into a variable in the $\text{prsh}$ graph where it is updated at the same time as the first requested projection. This translation is supplied by the rules $(\pi t : t :)$, $(\pi t : t)$, $(\pi h : h :)$ and $(\pi h : h)$ in Appendix B.4.

When the second projection is requested the evaluators are not quite lockstep anymore: $\text{prsh}$ has already updated the projection so it just needs to look up a variable while $\text{proj}$ has to find the address of the projection from its constructor first. The square (10.17) below illustrates this for the case where the head is requested after the tail. It is generated simply by translating the left pattern of the $(\text{Lookup})$ rule with the $(\pi h : h)$ rule.

\[
\begin{align*}
&\left\{a \cdot a', s_i \cdot s_i', \{a_i \cdot a_i', \}ight\},
&\{a \mapsto \pi h, c \mapsto h : t\},
\{a' \mapsto h'^*\},
\begin{array}{c}
(\pi h : h) \\
\rightarrow \\
\{h \cdot h', s_i \cdot s_i'\},
\{a \cdot a', a_i \cdot a_i', \}
\end{array}
\end{align*}
\]

\[
(P\text{ihLook}, P\text{ihUpdate}, \text{Lookup}) \downarrow (\text{Lookup}) \quad (10.17)
\]

\[
\begin{align*}
&\left\{h \cdot h', s_i \cdot s_i', \{a \cdot a', a_i \cdot a_i', \}ight\},
&\{c \mapsto h : t\},
\{\}
\begin{array}{c}
\rightarrow \\
\{h \cdot h', s_i \cdot s_i'\},
\{a \cdot a', a_i \cdot a_i', \}
\end{array}
\end{align*}
\]

Of course, there is a similar square (B.44) for the three steps to one involved in looking up the tail projection after the head. The translation turns out to be slightly
more general than we need, generating two more similar squares for these cases —

squares (B.42) and (B.43) — but where the constructor remains reachable after the

(PiReduce). We could use a global lemma to argue that this situation never arises if

the initial graphs do not contain projections.

\[ \square \]

10.3.4 Unlimited-steps to 1-step simulation

Sometimes a simulation proof needs the most general kind of 1-lockstep simulation

where \( \bar{A} \) may take any number of steps to every one taken by \( \bar{B} \) in the comparison

\( \bar{A} \equiv \bar{B} \).

Example 10.13 (many to one step simulation proof)

To show \( \bar{bhr} \equiv \bar{laz} \) we define the translation \( \bar{bhr} \rightarrow \), a superset of \( \bar{idX} \rightarrow \). This has a

global start rule which maps the \( \bar{bhr} \) stack structure, including its Ret terms, to the

\( \bar{laz} \) stack structure which does not have any Ret terms. The \( \bar{bhr} \) evaluator introduces a Ret after each (RetReduce) step so \( \text{start}_{\bar{bhr}} \) allows any number of them to appear between other stack terms.

\[ \bar{bhr} \rightarrow = (\text{start}_{\bar{bhr}}, \{(Id.\ Var), (Id.\lambda), (Id.\ Apply), (Id.\; let)\}) \]

\[ \text{start}_{\bar{bhr}} \]

\[
\begin{align*}
\{ s_i \mapsto \#a_i \; s_{i-1}, a_i \mapsto \bot \; |i \in U \} \\
\{ s_i \mapsto b_i : s_{i-1} \; |i \in P \} \\
\{ s_i \mapsto \text{Ret} \; s_{i-1} \; |i \in R \} \\
Ga, s_n
\end{align*}
\]

\[
\begin{align*}
\{ s'_i \mapsto \#a'_i \; p_i, a'_i \mapsto \bot \; |i \in U \} \\
\{ s'_i \mapsto b'_i : p_i |i \in P \} \\
Ha', p_{n+1}
\end{align*}
\]

where

\[ n \in \mathbb{N}, UPR = \{1, \ldots, n\}, s_0 = \epsilon, p_i = s'_{\max \{0\} UP \{-i, \ldots, n\}} \]

To show that \( \bar{bhr} \rightarrow \) is a 1-lockstep simulation requires special treatment for the \( \bar{laz} \)

rules (Update) and (Reduce) which destroy the root stack term. According to \( \bar{bhr} \rightarrow \) there may be any number of Ret terms above the variable update marker or pushed function argument corresponding to the top stack term on \( \bar{laz} \).

The (Reduce) square (10.18) shows one of these cases. Applying \( \text{start}_{\bar{bhr}} \) to its
top-left configuration tells us:

\[ G \{ a \mapsto \lambda x. E, s_n \mapsto \text{Ret} \; s_{n-1}, \ldots, s_{m+1} \mapsto \text{Ret} \; s_m, s_m \mapsto y : s_{m-1} \} a, s_n \]

\[ \bar{bhr} \rightarrow \]

\[ H \{ a' \mapsto \lambda x'. E', s'_m \mapsto y' : s'_{m-1} \} a', s'_m \]

Evaluating the right graph by one step corresponds to evaluating the left graph by

\( n - m + 1 \) steps. This produces the updated graphs:

\[ G \{ a \mapsto E[y/x], t \mapsto \text{Ret} \; s_{m-1} \} a, s \]

\[ \bar{bhr} \rightarrow \]

\[ H \{ a' \mapsto E'[y'/x'] \} a', s'_{m-1} \]

Applying \( \text{start}_{\bar{bhr}} \) to these graphs gives the bottom-left configuration of (10.18).
\begin{align*}
\langle a \cdot a', y \cdot y', b_i \cdot b_i' \rangle, \langle a_i \cdot a_i' \rangle, \\
\langle a \mapsto \lambda x. E \rangle, \\
\langle a' \mapsto \lambda x'. E' \rangle & \xrightarrow{(Id, \lambda)} \\
\langle e \cdot e', y \cdot y', b_i \cdot b_i' \rangle, \\
\langle a \cdot a', z \cdot z', a_i \cdot a_i' \rangle, \\
\langle e \mapsto E[z/x], \{e' \mapsto E'[z'/x']\} \rangle
\end{align*}

(RetReturn*, RetReduce) \downarrow (Reduce)

\begin{align*}
\langle a \cdot a', b_i \cdot b_i', \langle a_i \cdot a_i' \rangle, \\
\langle a \mapsto E[y/x] \rangle, \\
\langle a' \mapsto E'[y'/x'] \rangle & \xrightarrow{(H \text{ Ren}, \text{ Ren})} \\
\langle e \cdot e', y \cdot y', b_i \cdot b_i' \rangle, \\
\langle a \cdot a', z \cdot z', a_i \cdot a_i' \rangle, \\
\langle e \mapsto E[z/x], \{e' \mapsto E'[z'/x']\} \rangle
\end{align*}

The rest of the squares for this proof appear in Appendix B.5: Let(B.27), Push(B.28), Lookup(B.32) and Update(B.31).

Despite using a translation \(\overset{\text{bbtr}}{\Rightarrow}\) which maps a potentially unbounded number of nodes to one, Example 10.13 is not a proof of \(\overset{\text{bbtr}}{\Rightarrow} \overset{\text{lazy}}{\Rightarrow} \overset{\text{bbtr}}{\Rightarrow}\). Even though \(\overset{\text{bbtr}}{\Rightarrow}\) can translate graphs of unbounded size to graphs of bounded size such graphs might not arise during evaluation. Showing that there is a leak is the subject of Part IV.

### 10.4 Summary

This chapter has shown how to extend the basic PSR simulation proof method to make more complicated evaluator comparisons.

Using PSR inference and completion to discover a simulation proof, rather than merely checking it, has been successful for a range of simple examples. But it still needs the guidance of a user, and it is not clear how to decide which rule to introduce.

The inability of the PSR framework to incorporate global reasoning led us to enrich the PSR rule notation and to allow extensions of the \texttt{start} rule so that complex translations can be defined. It is not yet clear if these two techniques have equivalent proof power. Certainly the \texttt{start} extension technique is often simpler to formulate and reduces the amount of work needed to complete the proof squares. Disappointingly, we still often need to support this global reasoning with additional lemmas to express facts about graphs that cannot be encoded easily in the PSR rule framework.

Reasoning about non-lockstep behaviour has turned out to be quite straightforward — assuming that we have a clear understanding of the way the evaluators under comparison work. It is not so clear how a general mechanism for indicating where non-lockstep behaviour arises, and what to do about it, could be added to the proof method.

Figure 10.1 concludes this part of the thesis by showing the \(\overset{\text{ns}}{\Rightarrow}\) relationships we have established between different call-by-need evaluators.
Figure 10.1: This diagram relates 12 call-by-need variant evaluators. Some of them are combined with a space-relation garbage collector. In principle all the comparisons are with respect to the translation \( \overset{idX}{\Rightarrow} \) but larger relations are needed to complete all of the proofs. The translations and proof methods used are summarised in Table 10.1. Comparison 8 is actually \( \overset{\text{ses}}{\Rightarrow} \overset{\text{lazy}}{\Rightarrow} \) (but \( \overset{\text{ses}}{\Rightarrow} \) is transitive, hence the inner rectangle). In comparisons 11 the left evaluator is \( \overset{\text{proj}}{\Rightarrow} \). The evaluator \( \overset{\text{#ind}}{\Rightarrow} \) by itself cannot be compared with any of the others.

We conjecture that \( \overset{\text{proj}}{\Rightarrow} \) with the projection garbage collector \( \pi gc \) has the same space behaviour as \( \overset{\text{prsh}}{\Rightarrow} \), that is: \( \overset{\text{proj}}{\Rightarrow} ; \pi gc \overset{idX}{\Rightarrow} \overset{\text{prsh}}{\Rightarrow} \) wrt \( \overset{idX}{\Rightarrow} \). It also seems to be the case that update-marker collection in \( \overset{\text{ses}}{\Rightarrow} \) produces the same space behaviour as indirection collection in \( \overset{\text{#ind}}{\Rightarrow} \). We conjecture: \( \overset{\text{lazy}}{\Rightarrow} \overset{\text{ses}}{\Rightarrow} ; \# gc \overset{idX}{\Rightarrow} \overset{\text{#ind}}{\Rightarrow} ; I gc \) wrt \( \overset{idX}{\Rightarrow} \).

<table>
<thead>
<tr>
<th>Comparisons</th>
<th>Trans.</th>
<th>Proof method</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,5</td>
<td>( \overset{\text{nobh}}{\Rightarrow} )</td>
<td>non-lockstep termination</td>
<td>10.3, B.4</td>
</tr>
<tr>
<td>2</td>
<td>( \overset{\text{noah}}{\Rightarrow} )</td>
<td>lockstep, local property lemma</td>
<td>B.3</td>
</tr>
<tr>
<td>3,4,6,9</td>
<td>( \overset{idX}{\Rightarrow} )</td>
<td>automated completion</td>
<td>10.1, B.2</td>
</tr>
<tr>
<td>7</td>
<td>( \overset{\text{bhtr}}{\Rightarrow} )</td>
<td>many-to-1 step simulation, global translation</td>
<td>10.3</td>
</tr>
<tr>
<td>8</td>
<td>( \overset{\text{ses}}{\Rightarrow} )</td>
<td>non-lockstep termination, global translation</td>
<td>10.3</td>
</tr>
<tr>
<td>9</td>
<td>( \overset{idX}{\Rightarrow} )</td>
<td>lockstep, global property lemma</td>
<td>B.3</td>
</tr>
<tr>
<td>10</td>
<td>( \overset{\text{ind}}{\Rightarrow} )</td>
<td>enriched translation, global property lemma</td>
<td>10.2</td>
</tr>
<tr>
<td>11</td>
<td>( \overset{idX}{\Rightarrow} )</td>
<td>k-to-1 step simulation, global translation</td>
<td>B.4</td>
</tr>
</tbody>
</table>
Part IV

Looking for Leaks
Chapter 11

Classification of Space Leaks

Detecting a space leak requires an existence proof — witness graphs to show that \( A \) has worse space complexity than \( B \). This chapter investigates how we can recognise leak-witness programs and how we can characterise different kinds of leak.

Section 11.1 looks at the possibility that a translation is inherently inefficient. Under some circumstances this observation can lead to an easy proof of a space leak. Another potential aid in the search for a proof of \( A \sim B \) is that we may already have a proof of \( A \equiv B \). Section 11.2 looks briefly at whether the simulation proof and simulation completion procedure say anything about the presence of a leak. Section 11.3 investigates the behaviour of leaks caused by inefficient evaluation, developing equations which can be used to prove that a graph is a leak witness. Section 11.4 goes on to classify inefficient evaluation-leak witnesses according to their behaviour.

11.1 Leaky translation

The first distinction we can make is between leaks caused by an inefficient encoding and leaks caused by an inefficient evaluation strategy. An inefficient encoding means that the size of a graph cannot be bounded by the size of its translation.

**Definition 11.1 (Inefficient translation)**
Translation \( \iff \) is inefficient if \( \forall k \exists G, H \cdot G \iff H \land \#G > k \times \#H \).

**Example 11.1 (Inefficient number translation)**
The translation \( \overset{ub}{\rightarrow} \) defined in Example 7.3 is inefficient since for any constant factor \( k, \#u(n) > k \times \#b(n) \) where \( k = n \div \log_2 n \).

**Example 11.2 (Inefficient PSR)**
The PSR \( \overset{badh}{\rightarrow} \) used in Proposition 9.2 for the proof of \( \overset{badh}{\rightarrow} \iff lazy \) is inefficient. Its \( \overset{Var \perp}{\rightarrow} \) rule translates \( \{ a \mapsto a_1 \} a, \epsilon \) into \( \{ b \mapsto \perp \} b, \epsilon \). As \( a_1 \) is a free variable, for any \( k \) we have \( \{ a_i \mapsto a_{i+1} \}_{i=1}^k \{ a \mapsto a_1 \} a, \epsilon \overset{badh}{\rightarrow} \{ b \mapsto \perp \} b, \epsilon \).

**Example 11.3 (Inefficient space relation)**
The space relation \( \overset{bhr}{\rightarrow} \) used in the proof of \( \overset{bhr}{\rightarrow} \iff lazy \) in Example 10.13 is another inefficient translation. The evaluator \( \overset{bhr}{\rightarrow} \) can build up chains of Ret terms on its stack which correspond to nothing in \( lazy \) so for any \( k \) we have:
\( \{ s_i \mapsto Ret \ s_{i-1} \}_{i=1}^k \{ s_i \mapsto Ret \ \epsilon \} \epsilon, s_k \overset{bhr}{\rightarrow} \{ \} \epsilon, \epsilon \).

175
Inefficient translation by itself is not enough to guarantee a space leak, but it can be a useful starting point, and we can easily devise tests to decide whether PSRs are inefficient. For instance, in Example 11.2 all we had to do was find a PSR rule with a free variable in its left pattern which is not added to the goal set (the variable \( a_1 \)) then construct a graph that can be replicated any number of times to form a chain of arbitrary length. Attaching this chain to \( a_1 \) produces the example of inefficient translation.

In general we need to show that the inefficiency is not rectified by evaluation. Then we can say there is a translation leak. Example 11.1 would not be a translation leak if the unary and binary number evaluators both converted the number \( n \) into a chain of \( n \) zeros, for instance.

**Definition 11.2 (Translation leak)**

\[
\text{translation leak} \left( \mathcal{A}, \mathcal{B}, \xrightarrow{\text{ub}} \right) = \forall k, k' \exists G, H \cdot G \xrightarrow{\text{ub}} H \land \#G > k \times \#H \land \text{space } G \mathcal{A} \geq k' \times \text{space } H \mathcal{B} \quad \square
\]

Devising a general test for a translation leak is a difficult problem. A simulation argument is needed to show that the inefficiency of translation is preserved by evaluation. A simpler method is to find an inefficient translation between terminal states.

**Example 11.4 (Translation leaks)**

Consider the unary-binary translation \( \xrightarrow{\text{ub}} \) of Example 11.1. If evaluation does not increase graph size we have a translation leak. For example, translation leak \( (\square, \square, \xrightarrow{\text{ub}}) \).

The \( \xrightarrow{\text{badbh}} \) inefficient-translation witness in Example 11.2 exposes a translation leak because the graph \( \{ b \rightarrow \bot \} \) is not a lazy redex. Similarly, the \( \xrightarrow{\text{bbly}} \) inefficient translation witness in Example 11.3 is not a redex so translation leak \( (\text{bbly}, \text{lazy}, \xrightarrow{\text{bbly}}) \). \( \square \)

### 11.2 Space leaks and PSR completion

One problem with the idea of a translation leak is that some of the example leak witnesses suggest that they are not very convincing. The non-redex examples which suggested that \( \xrightarrow{\text{badbh}} \text{ lazy} \) and \( \xrightarrow{\text{bbly}} \text{ lazy} \) in Example 11.4 are not meaningful terminal states in the sense defined for Sestoft’s abstract machines (see Example 3.3). Neither are they reachable by evaluation from the initial states defined by the translation \( \xrightarrow{\text{idbs}} \).

The problem is that translations like \( \xrightarrow{\text{badbh}} \) and \( \xrightarrow{\text{bbly}} \) which are used to complete simulation proofs include not only initial graphs and those derived from them by evaluation, but also other meaningless graphs whose evaluation is space-safe.

So the successful proof of \( \mathcal{A} \xrightarrow{\text{idbs}} \mathcal{B} \) by PSR completion may or may not help us to prove \( \mathcal{A} \approx \mathcal{B} \) by leaky translation. Any leak-witness graph found by this method must be shown to be reachable by evaluation from an initial state.

Something else that might help the search for a proof of \( \mathcal{A} \approx \mathcal{B} \) wrt \( \xrightarrow{\text{idbs}} \) is the fact that attempting to prove \( \mathcal{A} \not\approx \mathcal{B} \) wrt \( \xrightarrow{\text{idbs}} \) must fail. This could happen for several reasons.

- The translation \( \xrightarrow{\text{idbs}} \) might not be a PSR — it might not even be a space relation.

This suggests the possibility of a translation leak.
The evaluators might not be lockstep. This may also help us to construct a leak witness: find $G$ and $H$ such that $G \implies H$, $H$ is not a redex and evaluating $G$ on $\mathcal{A}$ can be shown to require unbounded space usage.

The simulation-proof completion loop could fail. This is interesting if completion enters a loop which introduces a new PSR rule at each iteration, where successive rules translate the same subgraph to ever larger subgraphs. If we can detect this behaviour then the looping completion may suggest how to construct graphs to witness the leak. However, this does not happen very often. It is much more likely that completion will fail because none of the inferred translation rules satisfy the PSR restrictions; not a very reliable way of detecting leaks.

### 11.3 Leaky evaluation

As translation leaks can be of questionable value and we may not even have a proof of $\mathcal{A} \not\subseteq \mathcal{B}$ to start from, we now move on to look at leaks caused by inefficient evaluation, rather than inefficient translation. We define *evaluation leaks* as all the space leaks which are not translation leaks. Inefficient translation leaks can involve a combination of inefficient translation and evaluation but inefficient evaluation leaks are the sole responsibility of the evaluators.

**Definition 11.3 (Evaluation leak)**

$\text{evalleak}(\mathcal{A}, \mathcal{B}, \implies) = \mathcal{A} \not\subseteq \mathcal{B}$ wrt $\implies \land \neg \text{translationleak}(\mathcal{A}, \mathcal{B}, \implies) = \exists k \cdot \forall k' \cdot \exists G, H \cdot G \implies H \land \#G \leq k \times \#H \land \text{space } G \mathcal{A} > k' \times \text{space } H \mathcal{B}$

We begin the analysis of these leaks by looking at the sub-class of unsafe evaluation leaks defined by the *unsafe* criterion in Definition 7.6. These leaks are exposed by a single witness program, rather than the family of witnesses needed to show $\not\subseteq$.

Unsafe evaluation leaks can be proved by witness pairs $G \implies H$ where $G$ uses unbounded space and $H$ does not. Given that $G$ is finite and each step of its evaluation can only increase its size by a finite amount, $G$ must be a non-terminating program. Therefore we start to characterise these leak witnesses by giving an equation which describes the set of non-terminating and infinitely growing graphs on evaluator $\mathcal{A}$.

**Definition 11.4 (Infinite space programs)**

$\text{InfSpace} \mathcal{A} = \{G \mid \text{space } G \mathcal{A} = \infty, \text{time } G \mathcal{A} = \infty\}$

Equivalently, these programs can be defined co-inductively as a greatest fix point. Using a similar method to Gordon’s definition of all non-terminating programs [Gor94] we can write $\text{InfSpace} \mathcal{A} = \nu X. \{G \mid G \not\subseteq \mathcal{A} G' \land \#G > \#G, G' \in X\}$. This highlights the obvious characteristic behaviour of these programs: they grow after some number of evaluation steps (with garbage collection) and this behaviour repeats without end.

If $\mathcal{A}$ has an unsafe evaluation leak relative to $\mathcal{B}$, witnessed by $G \implies H$, then $G$ must be an element of $\text{InfSpace} \mathcal{A}$. If we can find such a $G$ then we have a *candidate leak witness*. To prove that $G \implies H$ is a leak witness we need to show that the space usage of $G$ is not bounded by the space usage of $H$. The *Infin* leaks are a sub-class of the unsafe evaluation leaks which have this property.
Definition 11.5 (Infin space leak)
\[ \text{Infin}(\mathbb{A}, \mathbb{B}, \Rightarrow) = \exists G, H \cdot G \Rightarrow H \land \#G \in \mathbb{N} \land G \in \text{Infspace } \mathbb{A} \land \#H \in \mathbb{N} \land H \notin \text{Infspace } \mathbb{B} \]

\[ \square \]

Proposition 11.1 (Infin leaks are unsafe evaluation leaks)
\[ \text{Infin}(\mathbb{A}, \mathbb{B}, \Rightarrow) = \text{evaluation leak}(\mathbb{A}, \mathbb{B}, \Rightarrow) \cap \text{unsafe}(\mathbb{A}, \mathbb{B}, \Rightarrow) \]

Proof
Follows from Definitions 7.6, 11.3 and 11.5

\[ \square \]

Infin excludes translation leaks such as unsafe(\mathbb{E}, \mathbb{E}, \{(a_i \mapsto a_{i+1}) | i \in \mathbb{N} \}a_0, \{\epsilon\})

Example 11.5 (An Infin leak)
Consider a simple graph language of variable indirections defined by the grammar \( X ::= I x \) with \( \text{ROOT} ::= \langle x \rangle \). After garbage collection, a graph can only be a chain of nodes, which could be a cycle. Now consider the following two evaluators. Both follow the chain to its end. However, \( \text{bad} \) inserts an extra link before moving on whereas \( \text{good} \) does not.

\[ \text{good} = \{a \mapsto I b\}a \rightarrow \{a \mapsto I b\}b \quad \text{(Indirect)} \]

\[ \text{bad} = \{a \mapsto I b\}a \rightarrow \{a \mapsto I c, c \mapsto I b\}b \quad \text{(xIndirect)} \]

Consider the family of graphs defined by Chain.

\[ \text{Chain}(n) = \{a_i \mapsto I a_{i+1}\}_{i=1}^{n-1} \{a_n \mapsto I a_1\}a_1 \]

Because \( \text{Chain}(n) \rightarrow_{\text{bad}} \text{Chain}(n+1) \) and \( \#\text{Chain}(n+1) > \#\text{Chain}(n) \) it follows that \( \text{Chain}(1) \in \text{Infspace } \text{[bad]} \). Similarly, \( \text{Chain}(1) \notin \text{Infspace } \text{[good]} \). Therefore \( \text{Infin(\text{[bad]}, \text{[good]}, \Rightarrow)} \).

\[ \square \]

Example 11.6 (Indirections that leak)
Consider the conjecture \( \text{#ind} \rightleftharpoons \text{lazy} \) wrt \( \Rightarrow \).

\[ \{f \mapsto \lambda x. \text{let } y = x \text{ in } y, y \mapsto f, a \mapsto g g\}a, \epsilon \]  (11.1)

\[ \Rightarrow_{\text{idX}} \]

\[ \{f \mapsto \lambda x. \text{let } y = x \text{ in } y, y \mapsto f, a \mapsto g g\}a, \epsilon \]  (11.2)

In the example graph (11.1) and its translation (11.2), the function \( f \) allocates a new variable \( y \) defined as a pointer to its argument then applies \( y \) to itself. Another function \( g \) is defined as a pointer to \( f \) and the root expression applies \( g \) to itself. Inspection shows that this is a non-terminating program because every application of \( f \) results in a new application of \( f \). It is not so clear whether the program should be in Infspace or not. After a few evaluation steps we reach (11.3) or (11.4).

\[ \{a \mapsto h h, h \mapsto g, g \mapsto I f, f \mapsto \cdots\}a, \epsilon \]

(\text{Push}), (\text{Lookup },:), (\text{Lookup#}), (\text{Update}), (\text{Reduce}), (\text{Let})  (11.3)
\[(11.2) \quad \longrightarrow_{\text{nay}} \{a \mapsto h \, h, h \mapsto g, g \mapsto \cdots\} a, \epsilon \\
(Push), (Lookup), (Update), (Reduce), (Let) \]  
(11.4)

The definition of \(f\) does not change so we abbreviate it as \(\cdots\). The evaluator \(\text{eval}_{\text{Ind}}\) updates \(g\) with an indirection to \(f\) \((11.3)\) whereas \(\text{lazy}_{\text{Ind}}\) copies the definition of \(f\) into \(g\) \((11.4)\). Both allocate a new node \(h\) which is defined as \(g\). Graph \((11.4)\) is identical to \((11.2)\) modulo renaming. It follows that \((11.2) \in \text{InfTime}_{\text{Lazy}}\) and \((11.2) \notin \text{InfSpace}_{\text{Lazy}}\).

\[(11.3) \quad \longrightarrow_{\text{Ind}}^n \{a \mapsto i, i \mapsto h, h \mapsto I \, g, g \mapsto I \, f, f \mapsto \cdots\} a, \epsilon \\
(Push), (Lookup \_), (Lookup \#), (Indirect\_), (Update\_), (Reduce\_), (Let) \]  
(11.5)

\[ICh(n) = \{a \mapsto b, b \mapsto c_1\} \{c_i \mapsto I \, c_{i+1}\}_{i=1}^n \{c_{n+1} \mapsto \cdots\} a, \epsilon \]  
(11.6)

After another 7 evaluation steps, \((11.3)\) becomes \((11.5)\). In general, if we define the family of graphs \(ICh(n)\) \((11.6)\) then \(ICh(n) \longrightarrow_{\text{Ind}}^n ICh(n + 1)\) using the rule sequence \((Push), (Lookup \_), (Lookup \#), (Indirect\_), (Update\_), (Reduce\_), (Let)\). Therefore \(ICh(0) \in \text{InfSpace}_{\text{Ind}}\) and we have \(\text{Inf}\text{fin}(\text{Ind}, \text{Lazy}, \text{Id}_{\text{A}})\). \(\Box\)

So the definition of \(\text{Inf}\text{fin}\) not only denotes a particular class of leaks, but it also gives us a way to prove that a pair of graphs constitute a leak witness by specialising the definition of \(\text{InfSpace}\). The non-terminating nature of the \(\text{Inf}\text{fin}\) leak witnesses means we cannot hope to provide a complete general method for finding or checking them, but as Example 11.5 showed, these leaks can occur in practice and the witnesses may not be very complicated so it is worth pursuing this approach further.

This leads us to define a simpler class of leaks which we can test for. A graph begins a self-feeding loop if its evaluation trace uses the same sequence of \(n\) steps repeatedly and without terminating. An expanding self-feeding loop also grows with each iteration of the \(n\) steps.

**Definition 11.6 (Expanding self-feeding loops)**

\[\text{sfl} = \{G | \exists G, n \cdot G \longrightarrow_{\text{Ind}}^n G(G)\} \]

\[\text{xsf} = \{G | \exists G, n \cdot G \longrightarrow_{\text{Ind}}^n G(G), \#G(G) > \#G\} \]  
\(\Box\)

The repetitive behaviour follows from the fact that if \(G \longrightarrow G'\) then \(G(G) \longrightarrow G(G')\). Now we define the class of leaks witnessed by graphs which begin expanding self-feeding loops, the \(\text{sfl}\) leaks.

**Definition 11.7 (Expanding self-feeding loop leaks)**

\[\text{sfl}(\text{Lazy} , \text{Id}_{\text{Lazy}}) = \exists G, H \cdot G \longrightarrow H \wedge \#G \in \mathbb{N} \wedge \#H \in \mathbb{N} \]

\[\wedge G \in \text{xsf} \wedge H \notin \text{InfSpace}_{\text{Lazy}} \]  
\(\Box\)

**Example 11.7 (An \(\text{sfl}\) leak)**

Returning to the leak witnesses of Example 11.5, we have \(\text{sfl}(\text{Lazy} , \text{Id}_{\text{Lazy}})\) as \(\text{Chain}(n) \in \text{xsf} \text{Lazy} \text{Ind}_{\text{Ind}}\). We did not show \(\text{xsf}(\text{Ind}_{\text{Ind}} , \text{Lazy} , \text{Id}_{\text{Lazy}})\) because although the evaluation of \(ICh(n)\) is a loop it is not regular, so it is not in \(\text{sflos}_{\text{Ind}}\). \(\Box\)
11.4 Leaky evaluation sequences

The Infin leak class makes an interesting lateral jump in the definition of a space leak: the witness can be seen as a sequence of evaluation rules, rather than a program. We could say that such witnesses are specified 
*imperatively* — by the sequence of instructions which evaluate them — rather than 
*declaratively* as graph programs. This view helps to disentangle another perceived difference between space leaks.

All evaluation-leak witnesses must repeatedly allocate and keep live new nodes. But there are two different ways this behaviour can arise.

- **Active growth**: \( \texttt{A} \) allocates some growing structure. This structure is used in some way in future steps, but the purpose it serves has no counterpart in \( \texttt{B} \). Possible examples from our selection of lazy evaluators include indirection nodes that form chains which may be traversed in \( \texttt{Find} \), or the \( \texttt{Ret} \) stack terms introduced by \( \texttt{Push} \) which are deleted when a function returns. Active leaks can be corrected by modifying the evaluator to manage without the superfluous growing structure. Equivalently, it may be more efficient to modify the garbage collector to remove the structure as part of its global rewrite of the graph. Examples of this might include the update marker collector \( \#gc \) and the indirection collector \( Igc \).

- **Passive growth**: \( \texttt{A} \) allocates some structure but that is kept live after it has ceased to be useful by some unnecessary reference in the context. This structure has a counterpart on \( \texttt{B} \), but \( \texttt{B} \) does not have the extra reference so it can often garbage collect the structure sooner. Possible examples include failing to trim an environment properly, not black-holing variables or not short-circuiting projections to data structures. Passive leaks can be corrected by modifying the evaluator to trim away graph arcs that are known to be useless. Equivalently, the garbage collector may be modified to do this trimming. For example, we can use an evaluator like \( \texttt{Push} \) to short cut both projections of a cons term simultaneously, or we can employ a garbage collector like \( \pi gc \) to short cut projections. Either approach may lead to a large structure referred to only by one of the projections being garbage collected.

The key to distinguishing between the more obviously wasteful active leaks and the subtly careless passive leaks is to look at the role of the context. The active leak is context independent; the passive leak is not. For the Infin leaks there is quite a simple way to formalise this distinction. Given a leak-witness pair \( G \Rightarrow H \) such that \( G \in \text{InfSpace}\ \texttt{A} \), we generalise \( G \) to find the sub-graph \( G' \) which is directly responsible for the leak: there is some context \( \Sigma \) such that \( G \equiv C(G') \) and \( G' \in \text{InfSpace}\ \texttt{A} \). Any sub-graph of \( G' \) is not a leak witness, so for all \( D(G'') \equiv G' \) where \( G'' \neq G' \), \( G'' \notin \text{InfSpace}\ \texttt{A} \). Reducing \( G \) to \( G' \) removes all the parts of the context which are not necessary for its infinite evaluation trace to cause infinite growth. Now, if there is a sub-graph of \( G' \) which uses infinite time but not infinite space then the leak is passive, otherwise it is active.

**Example 11.8 (Active and passive Infin leaks)**

Returning to the proof of \( \texttt{Bad} \Rightarrow \texttt{Good} \) in Example 11.5, the leak witness \( \text{Chain}(1) \) has no sub-graph which uses infinite space as its only sub-graphs are \( \{ a_1 \mapsto I \ x \} a_1 \), \( \{ a_1 \mapsto X \} a_1 \) and \( \{ \} a_1 \) which all terminate. Therefore it is an active leak.
Less trivially, the proof of \( \vdash \text{find} \text{ lazy} \) is also an active leak. If we replace any occurrence of \( f, x, y, g \) or \( a \) with a free variable, or replace any sub-expression with a hole, or remove any node in the graph \( \{ f \mapsto \lambda x. \text{let } y = x \text{ in } y \} \), \( a \mapsto g \), \( \epsilon \) then it terminates.

For a passive-leak witness, recall Example 6.4 where we defined a graph to compute \( \text{last zeros} + (\text{let } (h : t) = \text{zeros in } h) \) where \( \text{zeros} \) is an infinite list that unwinds as we try to find its last element. This program is in InfSpace \( \text{pint} \) and therefore a candidate leak witness. We can replace this expression with the more general \( \text{last zeros} + X \) where \( X \) is just a hole. Now it is still in InfTime \( \text{pint} \) because we can never reach the last element of \( \text{zeros} \), but it is no longer in in InfSpace \( \text{pint} \) because we have eliminated the duplicate reference to \( \text{zeros} \).

In the wider class of all space leaks, where leak witnesses do not have to be non-terminating programs, we can still make the distinction between active and passive leaks. The important observation is that when we reduce a graph \( G \) to one of its sub-graphs \( G' \), the rule sequence that forms the evaluation trace of \( G' \) is a sub-sequence of the evaluation trace of \( G \).

**Definition 11.9 (Evaluation trace)**

\[
\text{trace}_A = \nu T. \{ G \mapsto \langle \rangle | G \not\longrightarrow_A \} \\
\cup \{ G \mapsto \langle r \rangle | r \in \mathcal{A}, \mathcal{B} = \{ r \}, G \longrightarrow_r G', t = T(G') \}
\]

We define active space faults as those for which there is a witness whose \( \text{trace} \) is responsible for the fault. The trace of a passive space-fault witness needs to begin with particular graphs for the leak to arise; the trace alone does not expose the fault.

**Definition 11.10 (Active and passive space-fault witnesses)**

Suppose \( G \rightleftharpoons H \) is a space-fault witness by criterion \( C \), so \( C(\mathcal{A}, \mathcal{B}, \mapsto) \). Let \( \bar{r} = \text{trace}_A(G) \) and let \( G' \) be the most general graph pattern such that:

\[
\exists C \cdot C(G') \equiv G \text{ and } \text{trace}_A(G') = \bar{r}
\]

If \( C(\mathcal{A}, \mathcal{B}, \{(G, H)\}) \) then the witness is active, otherwise it is passive.

**Definition 11.11 (Active and passive space faults)**

A space fault (by any criterion) is active if at least one of its witnesses is active, otherwise it is passive.

**Example 11.9 (Passive space fault)**

Returning to the \( \text{last zeros} + (\text{let } (h : t) = \text{zeros in } h) \) program of Example 6.4, we can use this to show that \( \text{pint} \equiv \text{pint} \) wrt \( \equiv \text{com} \), where we define \( \equiv \text{com} = \equiv \text{id} \cup \{(\text{ComPlus})\} \).

\[
\langle \{ \}, \{ x \mapsto A + B \}, \{ y \mapsto D + C \} \\
\rightarrow \{ \{a \cdot c, b \cdot d\}, \{ x \mapsto A, y \mapsto B \}, \{ c \mapsto C, d \mapsto D \} \} \text{ (ComPlus)}
\]

Example 11.8 showed that this program is a passive-leak witness. It seems reasonable to conjecture that \( \text{pint} \equiv \text{pint} \) wrt \( \equiv \text{com} \) is a passive leak because the unwinding list \( \text{zeros} \) remains reachable only when it might be used after the computation of \( \text{last zeros} \). This lingering reference to \( \text{zeros} \) cannot be involved in any evaluation step in the trace of the leak witness because \( \text{last zeros} \) never terminates, so it must be part of the context instead.

\[\square\]
Figure 11.1: Venn diagram illustrating the relationship between different classes of space leak. *Infin* leaks (Definition 11.5) are *unsafe* (Definition 7.6) and *evaluation* (Definition 11.3) leaks; they may be either *active* or *passive* (Definition 11.10). *Xsfl* leaks (Definition 11.7) are all *Infin*.

The active-passive distinction may appear to be merely an interesting observation at this stage. But the important thing is that it precisely characterises which leaks are the result of a faulty sequence of evaluation rules, and which are caused by a combination of evaluation rules and the initial graph. If we can regard the evaluation trace as the leak witness, rather than the initial graph, then we can test and even search for leak witnesses much more reliably. This checking and search is the subject of chapters 12 and 13.

### 11.5 Summary

Space faults may be caused by an inefficient translation scheme, indicating a fundamentally inefficient graph encoding. Or they may be the result of an inefficient evaluation strategy. We have separated these two ideas and looked at further ways in which space faults can be classified. Apart from the idea of different strengths of leak, which was discussed in Section 7.2, we have shown how certain leaks can be characterised by general equations, such as the *Infin* class. Then we showed how to separate the idea of an active, useless growth from a passive growth by looking at the contribution the evaluation trace of a leak witness makes towards its behaviour. These ideas are summarised by Figure 11.1.
Chapter 12

Checking Leak Witnesses

This chapter develops the definition of the active $X_{sl}$ leak class into a technique for checking candidate leak witnesses automatically. The witness-checking tool needs to be given two evaluators $\mathbb{A}$ and $\mathbb{B}$, a translation $\equiv$ and a trace $\bar{r} \in \mathbb{A}^n$. It tests our supposition that there are graphs $G$ and $H$ such that $G \equiv H$, the evaluation trace of $G$ on $\mathbb{A}$ repeats the rule sequence $\bar{r}$, graph $G$ expands without limit and graph $H$ runs in constant space on $\mathbb{B}$. If the test succeeds the tool displays witness graphs and a summary of their behaviours.

Section 12.1 describes an analysis to show that $\bar{r}$ forms a self-feeding loop, generating a graph whose evaluation trace follows the loop. Self-feeding loop are found by constructing super-rules of the rule sequence $\bar{r}$ and then solving the self-feeding loop equation. Section 12.2 explains the main component of this analysis: a unification algorithm for arbitrary graph patterns. Section 12.3 forms candidate leak witnesses from self-feeding loop graphs by selecting those which expand after garbage collection. Section 12.4 confirms a candidate is a leak witness by testing the behaviour of its translations on $\mathbb{B}$. Section 12.5 shows how this method can be adapted to check passive $X_{sl}$ leak witnesses. The running examples are $\text{bhr} \Rightarrow \text{lazy}$ and $\text{badbhr} \Rightarrow \text{lazy}$.

12.1 Self-feeding loop construction

We begin with a rule sequence $\bar{r} \in \mathbb{A}$ and which we believe can repeat indefinitely, forming the evaluation trace of a candidate $X_{sl}$ leak witness. Such a sequence might be found by observing the evaluation of a graph which seems to have wasteful space behaviour.

Example 12.1 (Finding candidate witness rule sequences)
The evaluator $\text{bhr}$ adds a Ret term to its stack every time a $\lambda$-value is reduced to an instance of its body. The Ret is not removed until the value of its body is found. This observation suggests that $\text{bhr}$ is leakier than $\text{lazy}$ when evaluating a function that recursively calls itself an unbounded number of times ($\text{bhr}$ is designed to circumvent the tail-recursion optimisation that the $\text{lazy}$ provide for free). Observing the evaluation trace of a simple program with this behaviour shown next suggests that the rule sequence $(\text{Push}), (\text{Lookup}), (\text{Update}), (\text{RetReduce})$ may form an active leak witness proving $\text{bhr} \Rightarrow \text{lazy}$. 

183
\{a \mapsto \text{let } f = \lambda x.f \ x \text{ in } f \ e\} a, \ e

\Rightarrow_{\text{bhtr}} \{a \mapsto b \ e, b \mapsto \lambda x.b \ x\} a, \ e  

\Rightarrow_{\text{bhtr}}^4 \{a \mapsto b \ e, b \mapsto \lambda x.b \ x, c \mapsto \text{Ret } \ e\} a, \ c

\Rightarrow_{\text{bhtr}}^4 \{a \mapsto b \ e, b \mapsto \lambda x.b \ x, d \mapsto \text{Ret } c, c \mapsto \text{Ret } \ e\} a, \ d

(Let)  
(Push), (Lookup), (Update), (RetReduce)  
(Push), (Lookup), (Update), (RetReduce)

\hfill \square

12.1.1 Super-rules

If the leak is active then any graph whose trace repeats the rule sequence \(\bar{r}^*\) is a candidate leak witness. Any such graph is an instance of the most general redex of \(\bar{r}\). Therefore we find the most general redex and result of \(\bar{r}\), patterns \(L\) and \(R\) such that \(L \Rightarrow_{\text{A}}^* R\) and we call this a super-rule.

Constructing a set of super-rules is another variant of the idea of a forward closure that we used for graph translation in Definition 9.6.

**Definition 12.1 (Super-rules of a rule sequence)**

\[
\text{super} (L \rightarrow R) (L' \rightarrow R') = \{C(L) \rightarrow D'(R') \mid C(R) \equiv D(L'),
\]

\[
\left\{ x \neq y | x \in \text{dom } D(R') - \text{dom } C(L), y \in \text{fv } C(L) \right\}
\]

\[
\text{superRule } (r) = \{r\}
\]

\[
\text{superRule } (\bar{r} + \bar{s}) = \cup \{\text{superRule } r' | r' \in \text{superRule } \bar{s}\}
\]

The function \text{super} coalesces two rules by unifying the right pattern of the first rule with the left pattern of the second rule and using these unifiers to construct all possible most general redex and result pairs of the two rules. As in forward closures for translation (Definition 9.6), constraints are added to ensure that allocated nodes are not referred to by the new left pattern. The super-rules of a rule sequence, superRule \(\bar{r}\), are found by coalescing the first rule with each super-rule of the rest of the sequence.

\hfill \square

**Example 12.2 (A super-rule)**

The rule sequence (\text{Push}), (\text{Lookup}), (\text{Update}), (\text{RetReduce}) \in \text{bhtr}^4 identified in Example 12.1 has the following super-rule (\text{Push}, \text{Lookup}, \text{Update}, \text{RetReduce}).

\[
\{a \mapsto g \ c, g \mapsto \lambda h.A\} a, \ b \ \rightarrow \ \{a \mapsto A[d/h], g \mapsto \lambda h.A, d \mapsto \text{Ret } b\} a, \ d
\]

This rule replaces the application \(g \ c\) with the specialised function body and adds a \text{Ret} to the stack. In this case there is only one super-rule.

\hfill \square

The main technical component of super-rule construction is a unification algorithm for arbitrary graph patterns which may include holes with substitutions. We must also take care with the disequality constraints. For example, if a rule sequence looks up a function, specialises its body and then looks up another function, the first and second functions could be the same or they could be distinct; both possibilities should be covered by the super-rules of the sequence. These issues are addressed in Section 12.2. For now we just note that \text{superRule} may have many solutions when the basic unifier of Definition 5.12 is extended suitably.
Proposition 12.1 (Completeness of \textit{superRule})

\[ G \xrightarrow{\bar{r}} H \iff \exists r \in \text{superRule} \bar{r} \cdot G \xrightarrow{r} H \]

\textbf{Proof}

By induction on the definition of \textit{superRule}. \qed

So the super-rules of \( \bar{r} \) are an exact replacement for the rule sequence \( \bar{r} \). Another application of super-rules is to coalesce evaluator rules which might form a more efficient evaluator in some cases. For example, we could obtain the (VPush) rule of \texttt{[lazy]} defined in Example 5.8 by calculating the super-rule of (Push) and (Lookup).

Another consequence of Proposition 12.1 is that we can construct the super-rules of a rule sequence in any order and the resulting set is still complete. Therefore if we are constructing several super-rules some of the work can be shared.

\textbf{Example 12.3 (Associativity of \textit{super})}

\textit{superRule (Push), (Lookup), (Update)} can be constructed left-to-right or right-to-left.

\[
\begin{align*}
\text{super (super (Push) (Lookup)) (Update)} & = \text{super (} \{a \mapsto g \cdot c\} a, b \mapsto \{a \mapsto \bot, h \mapsto \# a, d \mapsto c : b\} g, h \} \text{(Update)} \\
& = \{j \mapsto c \cdot d, c \mapsto \lambda i.A\} j, b \mapsto \{j \mapsto \lambda i.A, c \mapsto \lambda i.A, k \mapsto d : b\} j, k \\
\text{super (Push) (super (Lookup) (Update))} & = \text{super (} \{h \mapsto c, c \mapsto \lambda g.A\} h, i \mapsto \{h \mapsto \lambda g.A, c \mapsto \lambda g.A\} h, i \} \\
& = \{a \mapsto g \cdot c, g \mapsto \lambda h.B\} a, b \mapsto \{a \mapsto \lambda h.B, g \mapsto \lambda h.B, d \mapsto c : b\} a, d
\end{align*}
\]

The two versions of (Push, Lookup, Update) generated are identical up to renaming. \qed

\section{12.1.2 \textbf{Self-feeding loops}}

To find the most general graph pattern whose evaluation trace repeats \( s \) indefinitely, where \( s \) is a super-rule of \( \bar{r} \), amounts to another unification problem. We need a solution to the self-feeding loop equation in Definition 11.6. The problem is that the most general graph of the super-rule may not be self-feeding, so we need to find a most general instance of the super-rule \texttt{LHS} that self-feeds. We need to find a unifier and a matcher at the same time.

\textbf{Proposition 12.2 (Finding self-feeding loops)}

\texttt{sloops} \( \{L \rightarrow R\} = \{C(L)\} \exists \mathbb{D} \cdot \mathbb{D}(C(L)) \equiv C(R) \}.

If there is a graph context \( C \) (a unifier) which specialises \( L \) such that \( C(L) \) self-feeds and there is a graph context \( \mathbb{D} \) (a matcher) which specialises \( C(L) \) to \( C(R) \) then \( C(L) \) begins a self-feeding loop.

\textbf{Proof}

From Definition 11.6. \qed

\textbf{Example 12.4 (A self-feeding loop)}

The super-rule (\texttt{Push, Lookup, Update, RetReduce}) of Example 12.2 has the following self-feeding loop.

\[
\begin{align*}
\{a \mapsto g \cdot c, g \mapsto \lambda h.g \cdot c\} a, b \quad \rightarrow \quad \{a \mapsto g \cdot c, g \mapsto \lambda h.g \cdot c, d \mapsto \text{Ret} b\} a, d
\end{align*}
\]

We obtain this from the super-rule of Example 12.2 by letting \( C = \{A \mapsto g \cdot c\} \) and \( \mathbb{D} = \{\} \) in Definition 12.2. \qed
Example 12.5 (Another self-feeding loop)
The rule \( (\text{Indirect}) \in \text{good} \), defined \( \{a \mapsto I \ b\}a \longrightarrow \{a \mapsto I \ b\}b \), has the following self-feeding loop: \( \{a \mapsto I \ a\}a \longrightarrow \{a \mapsto I \ a\}a \). We obtain this by letting \( \mathcal{C} = \{b \mapsto a\} \) and \( \mathcal{D} = \{\} \) in Definition 12.2. It cannot form a self-feeding loop if \( \mathcal{C} = \{\} \). \( \square \)

Example 12.6 (Inaccurate self-feeding loop)
The inaccurate rewrite rule \( \{a \mapsto G\ h\}a \longrightarrow \{a \mapsto G\ (G\ h)\}a \) is a self-feeding loop. We just need to let \( \mathcal{C} = \{\} \) and \( \mathcal{D} = \{h \mapsto G\ h\} \) in Definition 12.2 to show this. \( \square \)

To extend graph unification to find self-feeding loops we need to construct the contexts \( \mathcal{C} \) and \( \mathcal{D} \) in Definition 12.2 simultaneously. We achieve this by generating multiple solutions. The function \( \text{sflUnify} \) (Definition 12.2) is a variant on unification which finds a matcher \( \mathcal{D} \) and a unifier \( \mathcal{C} \) so \( (\mathcal{D}, \mathcal{C}) \in \text{sflUnify} \text{ LR} \Rightarrow \mathcal{D}(\mathcal{C}(L)) \subseteq \mathcal{C}(R) \). For simplicity, the matchers produced by \( \text{sflUnify} \) only contain mappings for variables, none for holes. This incompleteness means that \( \text{sflUnify} \) cannot recognise the self-feeding loop in Example 12.6 for instance. It often seems that if \( \mathcal{D} \) includes a hole mapping then the self-feeding loop is inaccurate. However this is not always true, as the examples in Section 12.5 demonstrate.

Definition 12.2 (Self-feeding loop detection)
The \( \text{sflUnify} \) rules define a self-feeding-loop detection algorithm. They produce a set of matcher-unifier pair solutions. The main difference from plain unification is in the rule for equating variables: either \( x \) and \( y \) are unified as normal by mapping \( x \) to \( y \) in the unifier, or \( x \) can be mapped to \( y \) in the matcher. The rest of the rules proceed inductively on the graphs (rules for null patterns are not shown), creating matcher-unifiers for sub-parts and composing them. The last rule covers the cases where \( P \) or \( Q \) is a hole: these unifiers are generated as described in Section 12.2 and the corresponding matcher is an identity on all variables in the unified patterns.

\[
\begin{align*}
\text{sflUnify} \quad & (Lv, \bar{v}) \quad (Ru, \bar{u}) = \text{sflUnify} v u \odot \text{sflUnify} L\bar{v}R\bar{u} \\
\text{sflUnify} \quad & \{\} \quad R = \{(\{\}, \{\})\} \\
\text{sflUnify} \quad & L \quad \{\} = \{\} \\
\text{sflUnify} \quad & \{a \mapsto P\}L \quad R = \left\{ \begin{array}{l}
\text{sflUnify} a b \\
\odot \text{sflUnify} P Q \\
\odot \text{sflUnify} L R'
\end{array} \right\} \quad R' = R - \{b \mapsto Q\} \\
\text{sflUnify} \quad & x \quad y = \{(\{y \mapsto y\}, \{x \mapsto y\}), \{x \mapsto y\}, \{\}\} \\
\text{sflUnify} \quad & F \quad F = \{\{\}, \{\}\} \\
\text{sflUnify} \quad & (F \ P \bar{P}) \quad (F \ Q \bar{Q}) = \text{sflUnify} P Q \odot \text{sflUnify} (F \bar{P}) (F \bar{Q}) \\
\text{sflUnify} \quad & (F \ x \bar{x}.\bar{P}) \quad (F \ y \bar{y}.\bar{Q}) = \text{sflUnify} x y \odot \text{sflUnify} (F \bar{x}.\bar{P}) (F \bar{y}.\bar{Q}) \\
\text{sflUnify} \quad & P \quad Q = \{(\{x \mapsto x\}, \{x \in \text{var}\ P(C)\}, \mathcal{C}) \mid \mathcal{C} \in \text{unify} P Q\}
\end{align*}
\]

To complete the algorithm we need the matcher-unifier composition rule \( \odot \) defined next. This composes each \( (\mathcal{D}', \mathcal{C}') \) in one solution set with each \( (\mathcal{D}, \mathcal{C}) \) in another solution set to form a new set of solutions. In each new solution \( (\mathcal{D}'', \mathcal{C}'') \), the matcher \( \mathcal{D}'' \) is a function and \( \mathcal{C}'' \) is a unifier; if \( \mathcal{D}(\mathcal{C}(x)) = \mathcal{C}(y) \) then \( \mathcal{D}'(\mathcal{C}'(x)) = \mathcal{C}''(y) \) in addition to the usual \( \mathcal{C}(x) = \mathcal{C}(y) \Rightarrow \mathcal{C}''(x) = \mathcal{C}''(y) \), similarly for \( (\mathcal{D}', \mathcal{C}') \). The definition of \( \odot \) obtains suitable composed unifiers \( \mathcal{C}'' \) by the unifier composition of Definition 5.11 then tries to construct a \( \mathcal{D}'' \) which meets the requirements (this can fail).
\[
dcs \otimes dcs' = \left\{ (D', C') \mid (D, C) \in dcs, (D', C') \in dcs', C' \subset C \odot C, \\
D' = \{ C'(x) \mapsto C'(y) \mid w \mapsto z \in D, C'(y) = z, C(x) = w \} \\
\cup \{ C'(x) \mapsto C'(y) \mid w \mapsto z \in D', C'(y) = z, C'(x) = w \}, \\
function D' \right\}
\]

**Example 12.7 (Self-feeding loop detection)**

The super-rule (Push, Lookup, Update, RetReduce) has the four self-feeding loops shown below. They are all essentially the same, generated by unifying the left and right patterns with the context \( C = \{ A \mapsto x, y, b \mapsto d \}. \) There are four solutions because \( x \) may be equated with \( h \) or \( g \), and \( y \) may be equated with \( h \) or it can be free.

\[
\begin{align*}
\{ a \mapsto g, c, g \mapsto \lambda h. g c \} a, b & \rightarrow \{ a \mapsto g, c, g \mapsto \lambda h. g c, d \mapsto \text{Ret b} \} a, d \\
\{ a \mapsto c, c \mapsto \lambda h. h c \} a, b & \rightarrow \{ a \mapsto c, c \mapsto \lambda h. h c, d \mapsto \text{Ret b} \} a, d \\
\{ a \mapsto g, c, g \mapsto \lambda h. g h \} a, b & \rightarrow \{ a \mapsto g, c, g \mapsto \lambda h. g h, d \mapsto \text{Ret b} \} a, d \\
\{ a \mapsto g, g \mapsto \lambda h. h h \} a, b & \rightarrow \{ a \mapsto g, g \mapsto \lambda h. h h, d \mapsto \text{Ret b} \} a, d
\end{align*}
\]

For a more detailed example, consider the rule LookupBad in \( \text{badbh} \) (the self-feeding rule in Example 12.5 is very similar). \( sflUnify \) proceeds as shown below.

\[
sflUnify \{ a \mapsto x \} a, s \{ a \mapsto x, t \mapsto \#a s \} x, t \\
= \{ \{(a \mapsto x), \}\}, \{(x \mapsto x), \{a \mapsto x\}\}\) \tag{1}
\]
\[
\otimes sflUnify \{ a \mapsto x \} s \{ a \mapsto x, t \mapsto \#a s \} t \\
= \{ \{(s \mapsto t), \}\}, \{(t \mapsto t), \{s \mapsto t\}\}\) \tag{2}
\]
\[
\otimes sflUnify \{ a \mapsto x \} \{ a \mapsto x, t \mapsto \#a s \} \\
= \{ \{(a \mapsto a), \}\}\) \tag{3}
\]
\[
\otimes sflUnify xx \\
= \{(\{x \mapsto x\}, \}\}\) \tag{4}
\]
\[
= \{(\{a \mapsto a, x \mapsto x\}, \}\}\) \tag{5}
\]
\[
= \{(\{a \mapsto a, x \mapsto x, s \mapsto t\}, \}\}, \{(\{a \mapsto a, x \mapsto x, t \mapsto t\}, \{s \mapsto t\}\}\}\) \tag{6}
\]
\[
= \{(\{x \mapsto x, s \mapsto t\}, \{a \mapsto x\} ), \{(x \mapsto x, t \mapsto t\}, \{s \mapsto t\}\}\}\) \tag{7}
\]

The algorithm generates all possible matcher-unifier sets for the sub-parts of the graphs: (1) for the expression-root variables; (2) for the stack roots; (3) for the node addresses (matching \( a \) in the left pattern with \( t \) in the right pattern is also considered but rejected because they have different categories); (4) for the terms at address \( a \). The matcher-unifiers (6) are the compositions of (2), (3) and (4). There are two solutions as either the matcher or the unifier can equate \( s \) and \( t \). The final composition (7) yields only two solutions. Of the possible four combinations of (1) with (6), two fail because the matcher is not a function as it contains \( \{ a \mapsto x, a \mapsto a \} \). The surviving two solutions tell us that \( \{ x \mapsto x \} x, s \) and \( \{ x \mapsto x \} x, t \) begin self-feeding loops. \( \square \)

As noted earlier, \( sflUnify \) is an incomplete solution to the self-feeding loop equation. It cannot handle cases like Example 12.6 where hole mappings are needed in the matcher rather than the unifier. The soundness of the algorithm is asserted by Proposition 12.3 and supported by the results of our self-feeding loop tests and searches.
Proposition 12.3 (Soundness of $sflUnify$)

$$(\mathcal{D}, \mathcal{C}) \in sflUnify \ L \rightarrow \mathcal{D}(\mathcal{C}(L)) \equiv \mathcal{C}(R)$$

**Proof**

By induction on the definition of $sflUnify$.

This assumes that the matcher-unifier composition has the following property. If
$$(\mathcal{D}, \mathcal{C}) \in dcs \text{ and } (\mathcal{D}^\prime, \mathcal{C}^\prime) \in dcs \odot dcs^\prime \text{ and } \mathcal{D}(\mathcal{C}(x)) = \mathcal{C}(y) \text{ then } \mathcal{D}^\prime(\mathcal{C}^\prime(x)) = \mathcal{C}^\prime(y).$$

Similarly when $(\mathcal{D}^\prime, \mathcal{C}) \in dcs^\prime \text{ and } (\mathcal{D}, \mathcal{C}^\prime) \in dcs \odot dcs^\prime \text{ and } \mathcal{D}(\mathcal{C}(x)) = \mathcal{C}(y)$.

Let $\mathcal{C}^\prime \subseteq \mathcal{C} \odot \mathcal{C}$. Then if $(w \rightarrow z) \in \mathcal{D}$ and $\mathcal{C}(x) = w$ and $\mathcal{C}(y) = z$ we require $(\mathcal{C}^\prime(x) \rightarrow \mathcal{C}^\prime(y)) \in \mathcal{D}^\prime$. Therefore $(\mathcal{D}^\prime, \mathcal{C}^\prime)$ is correct by construction. \qed

# 12.2 Graph pattern unification

Super-rule construction and self-feeding loop construction are both based on the unification of arbitrary graph patterns. Super-rule construction preserves certain properties of rules: holes in left patterns are always substitution-free and left patterns are connected. However, the left-pattern unifier is clearly not sufficient for unifying right and left patterns. Holes with substitutions need to be considered or solutions will be lost. Also, the disequality constraints on patterns need to be preserved properly so that we can measure pattern cardinality by counting nodes and so that $sflUnify$, which assumes that the domains of graph patterns contain distinct nodes, will work properly.

## 12.2.1 More specific unifiers

The first problem is that using a most general unifier can make super-rule construction go wrong.

**Example 12.8 (Good unifier, bad super-rule)**

Using the left-pattern unifier to create the super-rule of a variant on (Push) and a variant on (LookupBad) produces a solution in which $a = f$ or $a \neq f$ is left unspecified:

$$
\text{super } \{ \{a \mapsto A b\}a, s\} \{\} \rightarrow \{ a \mapsto \bot, b, c \mapsto A, t \mapsto a : s\}c, t\} \{ a \neq c\}
$$

$$(\{c \mapsto f, f \mapsto X\}c, t\} \{ c \neq f\} \rightarrow \{ c \mapsto f, f \mapsto X, u \mapsto \# c t\}f, u\} \{ c \neq f\})
$$

$$= \{ a \mapsto f b, f \mapsto X\}a, s\} \{\}
$$

$$\rightarrow \{ a \mapsto \bot, b, c \mapsto f, f \mapsto X, u \mapsto \# c t, t \mapsto a : s\}f, u\} \{ a \neq c, c \neq f, u \neq t\}$$

The problem is that this solution does not represent both possibilities properly. If $a = f$ then $X = (f b)$ but if we apply this specialisation to the right pattern then two nodes with address $a$ map to different terms. But, of course, it is possible to apply this two-rule sequence to graphs like \{a \mapsto a \epsilon\}a, \epsilon. \qed

So instead of a single most general unifier, we generate multiple solutions to ensure that different variables in the domain of a pattern are distinct.

**Definition 12.3 (Full unification)**

$$
\text{unify}P = \left\{ \mathcal{T} \in \mathcal{T} \odot \mathcal{T} \mid \mathcal{T} = \left\{ \{x \mapsto y\} 1 \leq i \leq n, \{x, y\} \subseteq X_i \right\}, \mathcal{T} = \left\{ \{x \neq y\} 1 \leq j \leq n, x \in X_i, y \in X_j\right\} \right\}
$$
The rule above is added to the algorithm of Definition 5.12. It finds all ways of partitioning the domain of pattern $P$ into $n$ non-empty disjoint sets $X_i$. For each partition, the variables in $X_i$ are equated and variables in different sets are disequated. A solution is attempted for each partition.

Example 12.9 (Good unifier, good super-rules)
Returning to Example 12.8, we apply full disunification. The category $S$ and category $X$ nodes cannot be equated and we already have $a \neq c$ from the right pattern of the first rule and $c \neq f$ from the left pattern of the second rule, so there are only two solutions.

$$\{a \mapsto f b, f \mapsto X\} a, s || \{a \neq f\}$$
$$\rightarrow \{a \mapsto \bot b, c \mapsto f, f \mapsto X, u \mapsto \# c t, t \mapsto a : s\} f, u || \{a \neq c, c \neq f, a \neq f, u \neq t\}$$

$$\{a \mapsto a b\} a, s || \{\}$$
$$\rightarrow \{a \mapsto \bot b, c \mapsto a, u \mapsto \# c t, t \mapsto a : s\} a, u || \{a \neq c, u \neq t\}$$

Now the $a = f$ solution right-pattern has only one node with address $a$. We can specialise $X$ to $(a b)$ in the $a \neq f$ solution, and it is still correct, but we cannot have $a = f$ as well.

The problem is solved but now the number of unifiers is exponential in the number of nodes in the graphs being unified. For small patterns this is acceptable as the patterns already have disequality constraints on their nodes and equating nodes belonging to different categories will fail. But as the rule sequences become longer, the increase in the number of solutions may become unreasonable. So it would be useful to find cases where a more general solution does not upset the super-rule and self-feeding loop construction procedures.

12.2.2 Holes with substitutions

Super-rule construction and the self-feed test involve unifying patterns that contain holes with substitutions. This means that we need to solve equations like $h \theta = P$ where $P$ could contain holes with substitutions. We cannot ignore the higher-order nature of graph grammars because this would lead to incompleteness — sometimes we need to use the bound variables in a super-rule to make it self feed.

Example 12.10 (Ignoring substitutions ignores self-feeding loops)
Consider the following higher-order graph grammar and the evaluator $\text{horder}$.

$$\text{ROOT} ::= \langle x \rangle$$

$$X ::= H x.X \mid F x$$

$$\text{horder} = \{(FFHreduce)\}$$

$$\{a \mapsto F b, b \mapsto F c, c \mapsto H x.A\} a \rightarrow \{a \mapsto A[b/x], b \mapsto A[c/x]\} a \quad (FFHreduce)$$

The graph $\{a \mapsto F b, b \mapsto F c, c \mapsto H x.(F x)\} a$ repeatedly evaluates to itself on $\text{horder}$ but $sfl\text{Unify}$ is unable to verify that $(FFHreduce)$ is self-feeding with the
disunification algorithm defined up to now. Ignoring hole substitutions and unifying the left and right patterns gives a unifier \( \{ A \mapsto Fb, A \mapsto Fc \} \) which fails because of the constraint \( b \neq c \) implicit in the graph patterns. To detect the self-feeding behaviour we need to consider hole substitutions.

Again, our solution to this problem is to generate multiple solutions with the help of the disequality constraints. We build up to the full rule of Definition 12.4 by considering some restricted forms of the general equation.

\[
  h[x/y] = (F z) \tag{12.1}
\]

In (12.1) the unifier must map \( h \) to \((F v)\) where \( v \) is some variable. There are two possibilities: if \( v \) is affected by the substitution then \( h = F y, h[x/y] = F x \) and \( x = z \). Otherwise, the substitution causes no change so \( h = F z \) and \( z \neq y \).

Example 12.11 (Considering substitutions recognises self-feeding loops)

This simple extension is enough to solve the problem in Example 12.10. Solving \( A[b/x] = Fb \) gives the unifiers \( \{ A \mapsto F x, b \mapsto b \} \) and \( \{ A \mapsto F b, b \neq x \} \). Similarly, solving \( A[c/x] = F c \) gives the unifiers \( \{ A \mapsto F x, c \mapsto c \} \) and \( \{ A \mapsto F c, c \neq x \} \). Of the four possible unifier compositions three fail leaving \( \{ A \mapsto F x, b \mapsto b, c \mapsto c \} \). This produces the expected self-feeding loop:

\[
  \{ a \mapsto F b, b \mapsto F c, c \mapsto H x, (F x) \} a \rightarrow \{ a \mapsto F b, b \mapsto F c, c \mapsto H x, (F x) \} a
\]

\[ \square \]

Now we consider more complex instances of the general hole unification equation.

\[
  (h[x_i/y_i]_{i=1}^n \{ y_i \neq y_j | 1 \leq i < j \leq n \}) = (F z) \tag{12.2}
\]

In (12.2) \( h \) has a substitution for many variables. Again \( h \) maps to \((F v)\) but now \( v \) may be affected by any one element of the substitution or it may remain unchanged. There are \( n + 1 \) possibilities: for each \( i \) we could have \( h = F y_i, h[x_i/y_i] = F x_i \) and \( x_i = z \) where the \( i \)th element of the substitution affects the variable. Alternatively the substitution has no effect so \( h = F z \) and \( \{ z \neq y_i \}_{i=1}^n \)

\[
  (h[x_i/y_i]_{i=1}^n \{ y_i \neq y_j | 1 \leq i < j \leq n \}) = P \text{ where holes } P = \langle \rangle \tag{12.3}
\]

In (12.3) we are unifying the hole with an arbitrary pattern which may have many free variables but no holes. Each \( x \in fv P \) may or may not be affected by some element of the substitution. First we find \( p\circ P \) which is a set containing all the ways of partitioning \( fv P \) into a sequence of \( n + 1 \) sets. Let \( \langle V_0, \ldots, V_n \rangle \) be an element of \( p\circ P \) in which \( V_i \) are the variables that are affected by the \( i \)th element of the substitution. \( V_0 \) contains variables affected by none of the substitutions. For each partition make a unifier as follows:

\[
  \{ h \mapsto [y_i/z] | z \in V_i]_{i=1}^n (P) \} \cup \{ z \mapsto x_i | z \in V_i]_{i=1}^n \{ z \neq y_i | i \neq j, z \in V_j \}_{i=0}^n \}
\]

\[
  (h[x_i/y_i]_{i=1}^n \{ y_i \neq y_j | 1 \leq i < j \leq n \}) = P
\]

where the holes in \( P \) have no substitutions

\[ \square \]
In (12.4) the substitution should be added to all holes occurring in P, so we add \( h' \mapsto h'[x_i/y_i]_{i=1}^{n} | h' \in \text{holes P} \) to each unifier generated as for (12.3). This does not cause an infinite unification or make the occur-check fail.

\[
h[x_i/y_i]_{i=1}^{n} = h'[x'_i/y'_i]_{i=1}^{n}
\]

(12.5)

To solve (12.5) let \( \theta = \{ x_i/y_i \}_{i=1}^{n} \cup \{ x'_i/y'_i \}_{i=1}^{m} \). The unifier is \( \{ h \mapsto h\theta, h' \mapsto h\theta \} \). The only problem is that the domain of \( \theta \) should be disjoint. But we cannot just enforce \( \{ y_i \neq y'_j \} \) because that could lose some solutions. For example, consider solving the following equation:

\[
\{ a \mapsto \lambda x. h, b \mapsto h[a/x] \} b, s = \{ a' \mapsto \lambda x'. h', b' \mapsto h'[a'/x'] \} b', s'
\]

If we generate just one unifier \( \{ h' \mapsto h[a/x, a'/x'] \} \) the overall unification fails. Instead, we need to generate a separate solution for each combination of variables in the domain of \( \theta \), some of which may result in failure when combined with unifiers for other parts of the patterns.

\[
U = \left\{ \begin{array}{l}
\{ h \mapsto h\theta, h' \mapsto h\theta \} \cup \{ a \mapsto b, c \mapsto d | \{ a/c, b/d \} \subseteq S_i \}
\\
\emptyset \left| \begin{array}{l}
i \neq j \Rightarrow S_i \neq S_j
\end{array} \right.
\end{array} \right\}
\]

Therefore the unifiers are defined by the set \( U \). Find each way of partitioning \( \theta \) into \( n \) non-empty disjoint subsets. Substitution elements in the same subset \( S_i \) are equated and substitution elements in distinct subsets are disequated. For our simple example this produces the unifiers \( \{ h' \mapsto h[a/x, a'/x'], h \mapsto h[a/x, a'/x'] \} \) which fails and \( \{ h' \mapsto h[a/x], h \mapsto h[a/x], a' \mapsto a, x' \mapsto x \} \) which succeeds.

**Definition 12.4 (Hole-pattern unification)**

\[
\text{unify } h[x_i/y_i]_{i=1}^{n} P \quad \text{unify } h[x_i/y_i]_{i=1}^{n} P
\]

\[
= \{(E \cup E' \cup \{ h' \mapsto h'[x_i/y_i]_{i=1}^{n} | h' \in \text{holes P} \} \cup \{ h \mapsto \phi(P) \} | D \cup D',
\\
\bigcup_{i=0}^{n} V_i = \text{fu}_P, i \neq j \Rightarrow V_i \neq V_j,
\\
E = \{ v \mapsto x_i | 1 \leq i \leq n, v \in V_i \},
\\
\phi = \{ v \mapsto y_i | 1 \leq i \leq n, v \in V_i \},
\\
D = \{ v \neq y_j | 0 \leq i \leq n, v \in V_i, 1 \leq j < i \},
\\
\{ h'_i \theta'_i \}_{i=1}^{m} = \text{the holes occurring in } P,
\\
\theta''_i = [x_i/y_i]_{i=1}^{n} \cup \theta'_i,
\\
\bigcup_{k=1}^{j} S_{i,k} = \theta''_i, k \neq l \Rightarrow S_{i,k} \neq S_{i,l}, 1 \leq k \leq j_i \Rightarrow S_{i,k} \neq \{ \},
\\
E' = \{ a \mapsto b, c \mapsto d | 1 \leq i \leq m, 1 \leq k \leq j_i, \{ a/c, b/d \} \subseteq S_{i,k} \},
\\
D' = \{ c \neq d | 1 \leq i \leq m, 1 \leq k \leq j_i, \{ a/c \} \subseteq S_{i,k}, k \neq l, \{ b/d \} \subseteq S_{i,l} \}
\}
\]

The hole-unification rule merges the method for solving (12.3) to handle free variables in \( P \) and the method for solving (12.5) to handle holes in \( P \). The free variables of \( P \) are partitioned into \( n + 1 \) sets \( (n \) is the size of the substitution on \( h \)). The variables in the \( i \)th set are affected by the \( i \)th element of the substitution as described for (12.3). This gives us the equations \( E \), the disequations \( D \) and the specialisation of \( h \) towards the unifier.

The \( m \) holes in \( P \) are found and any substitutions they have are merged with \( [x_i/y_i]_{i=1}^{n} \) to form the substitutions \( \theta''_i \). Every way is found of partitioning \( \theta''_i \) into \( j_i \) distinct, non-empty sets \( S_{i,k} \). An equality set \( E' \) and a disequality set \( D' \) is generated
for each partition as described for (12.5). The holes in \( P \) are mapped to themselves with the substitution \([x_i/y_i]_i^n\) as described for (12.4). This substitution is merged with their own by the equations in \( E' \) and the disequations in \( D' \) so the substitutions in the unified pattern have distinct domains. \( \square \)

Hole-pattern unification can give a solution-set size exponential in the number of free variables of the patterns unified with holes and the size of the hole substitutions in the patterns. Again, more general solutions that do not compromise the correctness of super-rule and self-feeding loop construction would be very useful.

**Example 12.12 (Hole-pattern unification)**
In Example 12.7, to find the self-feeding loops of \( (\text{Push, Lookup, Update, RetReduce}) \) we needed to solve \( g c = A[c/h] \). This is done by finding all partitions of the free variables of \( g c \) into two sets: \( \{\{g, c\}, \{\}\}, \{\{c\}, \{g\}\}, \{\}, \{g, c\}\} \). For each partition we build a solution yielding \( \{A \mapsto g c\}, \{A \mapsto h c\}, \{A \mapsto g h\}, \{A \mapsto h h\} \). All of these unifiers give rise to a matcher-unifier in the \texttt{sflUnify} algorithm and hence the four solutions listed in Example 12.7. \( \square \)

### 12.3 Candidate leak witness construction

Given the self-feeding loops of a rule sequence, candidate leak witnesses are generated by selecting those that form expanding self-feeding loops. To decide whether any allocation in the evaluation loop causes expansion, the loop is garbage-collected after one iteration. Unfortunately, the loops are graph patterns so they can be *garbage generating*, meaning that expansion does not guarantee repeating expansion. Therefore we select the loops that are both expanding and non-garbage-generating (using the approximation of Proposition 5.4).

**Proposition 12.4 (Finding expanding self-feeding loops)**

\[
xsfloops r \supseteq \{L | L \rightarrow R \in \sfl loops r, R' = gc R, \#R' > \#L, \neg ggen(L \rightarrow R') \}.
\]

**Proof**

From the definition of \( \text{xsfloops} \) and \( ggen \). \( \square \)

**Example 12.13 (Expanding self-feeding loop)**

The self-feeding loops in Example 12.7 expand. The first one is repeated below. The right pattern of this rule is unchanged by garbage collection; it expands and it generates no garbage.

\[
\{a \mapsto g c, g \mapsto \lambda h.g c\} a, b \rightarrow \{a \mapsto g c, g \mapsto \lambda h.g c, d \mapsto \text{Ret } b\} a, d
\]

\( \square \)

**Example 12.14 (Non-expanding self-feeding loop)**

The self-feeding loop \( (\text{Push, Lookup, Update, Reduce, Let}) \) shown below allocates a new node \( z' \) which is reachable. However, this allocation does not cause expansion because the loop is garbage generating: \( y \), the original argument to \( f \), confined to the substitution on \( A \) after the rewrite, is not reachable in any context in which \( x \) is not a free variable of the term substituted for \( A \). So this loop is garbage generating and it is not an active leak witness.
\{a \mapsto f \, y, f \mapsto \lambda x. \text{let } z = A \text{ in } f \, z \} a, s \\
\quad \longrightarrow \{a \mapsto f \, z', z' \mapsto A[y/x], f \mapsto \lambda x. \text{let } z = A \text{ in } f \, z \} a, s

However, this loop is the basis of a passive leak witness. Substitute \( x \) for the hole \( A \) and the loop is not garbage generating; the expansion will repeat causing a chain of nodes to build up in the graph. This suggests a method for verifying, or even searching for, passive \( \text{Xsfl} \) leaks.

In GraphKit, an evaluator can have a non-standard garbage collector defined by a space relation (see Section 8.2). In this case we replace the \( gc \) in Definition 12.4 with the space-relation collector. A problem with this scheme is that we currently have no way of telling whether a rule is garbage generating with an arbitrary space relation serving as garbage collector. Another problem is that garbage collecting a graph \textit{pattern} with a space relation (using the pattern translator of Definition 9.7) might not terminate.

We circumvent these difficulties by grounding the candidate-witness graph pattern to a graph. Restricting our attention to ground instances is acceptable because any instance of a candidate witness pattern has unbounded space usage. A potential disadvantage is that we might not be able to translate a particular ground instance, or we might not be able to prove anything about its behaviour.

We could try to construct a proper graph from the witness-graph pattern, by filling its holes with small terms and its free variables with arcs to appropriate nodes. But it is sufficient to ground the pattern by putting it in the \textit{null context}. Then applying a space relation garbage collector to the ground right pattern will terminate.

**Definition 12.5 (Ground \textit{Xsfl} candidate witnesses)**

\begin{align*}
\text{candidate } g \, \bar{r} \,
\mathbb{H} = \{L!\mid L \longrightarrow R \in \text{superRule} \bar{r}, G(L) \in \text{sfloops} \{L \longrightarrow R\}, \\
L! = \mathbb{H}(G(L)), R' = g(\mathbb{H}(G(R))), \#R' > \#L!\} \\
\mathbb{E}(G) = \{x \mapsto \epsilon\mid x \in f_0 \, G\} \cup \{h \mapsto \mathcal{E}\mid h \in \text{Holes}\})(G)
\end{align*}

To generate \( \text{Xsfl} \) candidate witness graphs of \( \bar{r} \) with garbage collector \( g \) use

candidate \( g \, \bar{r} \, \mathbb{E} \) where \( \mathbb{E} \) is the null context.

**Example 12.15 (Self-feeding loop expansion cured by garbage collector)**

Rule sequence \((\text{Push}), (\text{ScLookup}), (\text{ScUpdate}), (\text{Reduce}), (\text{Let}), (\text{ScLookup}) \in \text{sect}^6\) has the following ground \( \text{Xsfl} \) candidate leak witness.

\begin{align*}
\{a \mapsto f \, \epsilon, f \mapsto \lambda x. \text{let } y = f \, \epsilon \text{ in } y\} a, \epsilon \\
\quad \longrightarrow \{a \mapsto f \, \epsilon, f \mapsto \lambda x. \text{let } y = f \, \epsilon \text{ in } y, b \mapsto \perp, s \mapsto \#b : \epsilon\} a, s
\end{align*}

Applying the garbage collector \( \#gc \) to the right graph removes nodes \( b \) and \( s \) and replaces the stack-root variable with \( \epsilon \), resulting in the left graph and making the loop non-expanding.

So a non-standard garbage collector can prevent some potential candidates from expanding. However, it could allow expansion with the first \( n \) applications of a self-feeding loop and then collapse away the growth. This means that \textit{candidate} can be misleading: it is possible for a candidate witness produced by \textit{candidate } \( g \, \bar{r} \, \mathbb{E} \) to run in constant space if \( g \) is a non-standard collector.
12.4 Checking candidate leak witnesses

Given a candidate leak witness $G$, we translate it: $G \iff H$, then determine the behaviour of $H$ on $\mathbb{N}$. We can say $\mathbb{N} \cong \mathbb{N}$ if we can prove that $H$ terminates, or that it evaluates perpetually in constant space. The completeness of our proof checker for active Xsfl leaks is determined by the power of our termination and non-termination analyses.

12.4.1 Candidate witness translation

Translations are found by the procedure described in Definition 9.6. If there are no translations then the candidate cannot be a leak witness. Sometimes there might be a translation if the user presents candidate checker with certain rotations of a rule sequence but not with others. To solve this problem we generate candidates for all permutation loops of the rule sequence.

Example 12.16 (Candidate translation)

Taking the candidate witness $\{a \mapsto g \varepsilon, g \mapsto \lambda h.g \varepsilon\}a, \varepsilon$ produced from the rule sequence (Push), (Lookup), (Update), (RetReduce) on \texttt{hi} in Example 12.13, we can translate it to itself using \texttt{idX} for testing on \texttt{lazy}.

Had we started with the candidate witness $\{i \mapsto c, c \mapsto \lambda g.c \varepsilon, j \mapsto \varepsilon : c\}i, j$ produced from the rule sequence (Lookup), (Update), (RetReduce), (Push) there are no translations because the witness involves the stack function symbol ($\lambda$) which is not included in \texttt{idX}. Therefore we rotate the rule sequence until a translation is found. The only rotation with a translation is (Push), (Lookup), (Update), (RetReduce).  

12.4.2 Termination analysis

The termination analysis takes a very simple approach to the classic undecidable problem. We just evaluate the translation by at most $n$ steps on $\mathbb{N}$ to test whether it terminates. We expect all our call-by-need evaluators to take roughly the same number of steps to evaluate a given graph, since mostly they are lockstep, or nearly, as shown by the proofs in Part III. So if the translation is going to loop then it should be repeating rules after about $l$ steps where $l$ is the length of the repeating rule sequence on $\mathbb{N}$. To allow for minor differences in evaluators we test for termination within $2 \times l$ steps.

Example 12.17 (Termination analysis by evaluation)

The candidate graph $\{a \mapsto g \varepsilon, g \mapsto \lambda h.g \varepsilon\}a, \varepsilon$ for the \texttt{hi} = \texttt{lazy} comparison from Example 12.16 does not terminate within 8 evaluation steps, so the comparison is unproved: it could terminate later, non-terminate in constant space or non-terminate in unbounded space.

The candidate graph $\{a \mapsto a\}a, \varepsilon$ for the \texttt{bad} = \texttt{lazy} comparison terminates on \texttt{lazy}. After one step it reaches the irreducible state $\{a \mapsto \bot, s \mapsto \#a : \varepsilon\}a, s$. Therefore it is a leak witness and \texttt{bad} = \texttt{lazy} wrt \texttt{idX}. In this case the leak witness clearly highlights the source of the space fault: \texttt{bad} leaks on cyclic programs through repeated use of its (LookupBad) rule.
12.4.3 Non-termination analysis

If the translated graph $H$ cannot be shown to terminate, it may be possible to show that it non-terminates in constant space. This analysis adapts our Xsfl analysis to look for non-expanding self-feeding loops. To make this analysis terminate we only look for self-feeding loops beginning at $H$ with a length less than $n$ where $n$ is twice the loop length on $A$, as for the termination analysis.

A problem with this approach is that there are not just $n$ possible loops to try because the loop might not start immediately. To improve the analysis we could look for loops beginning after $i$ steps, where $i < n$.

A useful aid is that we already have the first $n$ steps of the evaluation trace of $H$ from the failed termination analysis. If the evaluation of $H$ loops then in this sequence $\langle r_i \rangle_{i=1}^n$, with a loop length $l \leq n$, we must have $r_i = r_{i+1}$, so we do not have to construct many of the super-rules for each of the $n$ sequence lengths (or $n^2$ if we allow the loop to start at the $i$th step). We can also make savings by starting with the smallest rule sequence and building the super-rules for $\langle r_i \rangle_{i=1}^n$ out of the super-rules for $\langle r_i \rangle_{i=1}^{n-1}$. This analysis completes the leak witness checker.

**Definition 12.6 (Leak-witness checker)**

$$
\text{witness } A \quad gc_A \quad gc_B \quad \Rightarrow \bar{r}
$$

$$
= \{ G \mid G \in \text{candidate } gc_A \quad \bar{r} \quad E, \quad G \Rightarrow H, \quad \exists \bar{H}^\prime, n < 2 \times \#\bar{r} \cdot \quad H \Rightarrow^{n} \bar{H}^\prime \land

\bar{H}^\prime \not\Rightarrow B

\lor \langle r_i \rangle_{i=1}^n + \bar{r} = \text{trace}_B(H) \land

\exists L \in \text{selfloops (superRule } \langle r_i \rangle_{i=1}^n), \exists G \cdot \quad H \equiv G(L) \land

\#gc_B(H^\prime) = \#H

\}$$

To check that a rule sequence $\bar{r}$ forms the trace of an active Xsfl leak witness: find its candidate witness graphs $G$; translate $G$ to $H$; if $H$ terminates in less than $2 \times \#\bar{r}$ steps or starts a non-expanding self-feeding loop of length less than $2 \times \#\bar{r}$ then $G$ is a witness. Either evaluator may have a non-standard garbage collector.

**Example 12.18 (Witness checking)**

The candidate witness graph $\{a \mapsto g \epsilon, g \mapsto \lambda h.g \epsilon \}a, \epsilon$ for the $\text{bfr} \approx \text{lazy}$ comparison translates to itself under $\Rightarrow^{idX}$. It does not terminate but its evaluation follows a self feeding loop $(\text{Push, Lookup, Update, Reduce})$ which does not cause expansion. Therefore the graph is a leak witnesses and $\text{bfr} \approx \text{lazy}$ wrt $\Rightarrow^{idX}$.  

12.5 Passive-leak witness checking

The active-leak checking function described in the preceding sections can easily be adapted to check passive Xsfl leak witnesses. First we need to find a suitable candidate by hand.
Example 12.19 (Finding a passive candidate witness)

\[
\begin{align*}
\{ & \text{main} \mapsto \text{finull}, \\
& \text{finull} \mapsto \text{finite nulls}, \\
& \text{nulls} \mapsto \text{listof } \epsilon, \\
& \text{finite} \mapsto \lambda l. \text{case } l \text{ of } \{ [] \mapsto \text{True}; (h : t) \mapsto \text{finite } t \}, \\
& \text{listof } \mapsto \lambda x. \text{let } \text{lof } = \text{listof } x \text{ in } x : \text{lof} \} \text{main}, \epsilon \\
\end{align*}
\]

To show \[\text{nobh} \equiv \text{[lazy]}\] \(\text{wrt } \text{idX}\) we need to expose the inefficiency of \((\text{LookupNh})\), which keeps a reachable copy of a variable's definition while the value of that definition is being found. The graph (12.6) is a suitable program involving list constructors and case-expressions. A simple variant of the program discussed in Example 6.3: it looks up \text{finull} which tests whether the infinitely unwinding list \text{nulls} is finite. After the initial \((\text{LookupNh})\), the evaluation trace follows the 13-rule sequence \((\text{Push}), (\text{LookupNh}), (\text{UpdateNh}), (\text{Reduce}), (\text{LPush}), (\text{LookupNh}), (\text{Push}), (\text{LookupNh}), (\text{UpdateNh}), (\text{Reduce}), (\text{Let}), (\text{UpdateNh}), (\text{LReduce : })\) repeatedly. The list grows by one element at each iteration suggesting that we can use this program as a candidate leak witness. To simplify things a little we convert the example into (12.7) which is a \(\lambda\)-calculus version of the same program: the list constructor becomes a \(\lambda\)-value and case becomes function application.

\[
\begin{align*}
\{ & a \mapsto b, \\
& b \mapsto \text{fin } l, \\
& l \mapsto \text{lof } \epsilon, \\
& \text{fin} \mapsto \lambda z. z \text{ fin}, \\
& \text{lof} \mapsto \lambda x. \text{let } y = \text{lof } x \text{ in } \lambda w. w y \} a, \epsilon
\end{align*}
\]

The evaluation trace of (12.7) follows the same pattern but the list push and reduce rules \((\text{LPush})\) and \((\text{LReduce : })\) are replaced by \(\lambda\)-value versions \((\text{Push})\) and \((\text{Reduce}).\)

Next the super-rules of the looping rule sequence are generated just as they are for an active leak. Then self-feeding loops can be generated also as before. Then we need to supply a graph context in place of \(\mathbb{E}\) in Definition 12.6 to make the passive candidate.

Example 12.20 (Generating a passive candidate witness)
One of the four super-rules of the 13-rule sequence is shown next (12.8). Its variables are named to make its relation to (12.7) apparent.

\[
\begin{align*}
\{ & b \mapsto \text{fin } l, \\
& l \mapsto \text{lof } \epsilon, \\
& \text{fin} \mapsto \lambda z. z \text{ d}, \\
& \text{lof} \mapsto \lambda x. \text{let } y = B \text{ in } \lambda w. C \} b, c \longrightarrow \{ \\
& b \mapsto C[d/w, l'/y, e/x], \\
& l \mapsto \lambda w. C[l'/y, e/x], \\
& \text{fin} \mapsto \lambda z. z \text{ d}, \\
& \text{lof} \mapsto \lambda x. \text{let } y = B \text{ in } \lambda w. C \} b, c
\end{align*}
\]

Clearly (12.7) is an instance of the left pattern of (12.8). Unfortunately, the incompleteness of \(sfl\text{Unify}\) means that it cannot find any self-feeding loops of (12.8). However, if we specialise the super-rule to (12.9) by hand using the context
\{ B \mapsto \text{lof } e, C \mapsto w\ y, d \mapsto \text{fin} \} \), then \text{selfUnify} \ is able to confirm this is a self-feeding loop. To find this context we had to solve the self-feeding loop equation by hand, using our conjectured leak witness (12.7) as a guide.

\[
\begin{align*}
\{ \text{fin } l, &\quad \rightarrow \quad \{ \text{fin } l', \\
\text{lof } e, &\quad \text{lof } e', \\
\text{fin } \lambda z.\ z \text{ fin}, &\quad \lambda z.\ z \text{ fin}, \\
\text{lof } \lambda x.\ \text{let } y = \text{lof } e \in \lambda w.\ w \ y \} b, c &\quad \lambda x.\ \text{let } y = \text{lof } e \in \lambda w.\ w \ y \} b, c
\end{align*}
\]

(12.9)

This loop is garbage generating: the standard garbage collector removes node \( l \) from the right side if we put it in the null context \( E \) (as in Definition 12.6). So we need to specialise the loop again to make sure that \( l \) remains reachable in our context-sensitive witness. We choose the context \( C \), using the conjectured witness (12.7) to help us work out a suitable way to prevent the loop generating garbage.

\[
C(g) = (\{ a \mapsto \text{fin } l, s \mapsto \#a \ e \} \cup \{ c \mapsto s, e \mapsto \epsilon \}(g))
\]

Calculating candidate gc (12.9) \( C \) gives almost the witness (12.7). The difference is that it is evaluated one step — as if the (LookupNh) rule has added an update marker for \( a \) and copied the contents of \( b \) into \( a \) — and the argument of \( \text{lof } \) in the definition of \( y \) is \( \epsilon \) instead of \( x \); see left-hand side of (12.10). Now the whole right pattern is reachable — the self-feeding loop causes expansion in context \( C \) — and we have a candidate witness.

\[ \square \]

It just remains to check the witness using Definition 12.6 as normal.

**Example 12.21 (Checking a passive witness)**

The candidate is translated by \( \text{nabh} \) (used for the proof of \( \text{nhbh} \equiv \text{lazy} \) in Section 10.3) as shown in (12.10). The only change caused by translation is in the definition of \( a \) which gets black-holed. The translated graph does not terminate, but its evaluation trace also forms a 13-rule self-feeding loop which does not expand. The witness is confirmed.

\[
\begin{align*}
\{ a \mapsto \text{fin } l, &\quad \text{nabh} \quad \{ a \mapsto \bot, \\
\quad s \mapsto \#a \ e, &\quad s \mapsto \#a \ e, \\
\quad b \mapsto \text{fin } l, &\quad b \mapsto \text{fin } l, \\
\quad l \mapsto \text{lof } e, &\quad l \mapsto \text{lof } e, \\
\quad \text{fin } \mapsto \lambda z.\ z \text{ fin}, &\quad \text{fin } \mapsto \lambda z.\ z \text{ fin}, \\
\quad \text{lof } \mapsto \lambda x.\ \text{let } y = \text{lof } e \in \lambda w.\ w \ y \} b, s &\quad \lambda x.\ \text{let } y = \text{lof } e \in \lambda w.\ w \ y \} b, s
\end{align*}
\]

(12.10)

\[ \square \]

### 12.6 Summary

Leak-witness graphs are checked as follows. A procedure generates all the *super-rules* of a rule sequence — single rules which are exact replacements for the rule sequence. A sequence may have many super-rules to satisfy the need for disjoint domains of graph patterns and unification of higher-order patterns involving hole substitutions.
Unification is extended to include these features. Self-feeding loop construction offers an incomplete but practical solution to the self-feeding loop equation. Applying it to a super-rule finds graph patterns whose evaluation trace repeats the same rule sequence indefinitely. This is the basis of the active-leak witnesses checker. A graph is generated from a self-feeding loop by putting it in the null context. If it expands it is a candidate witness. Translating the candidate and applying approximate termination or non-termination analyses can confirm that the candidate is a witness, and therefore that an evaluator has an active leak. Passive witnesses can also be checked if additional information in the form of a graph context for the self-feeding loop is supplied.
Chapter 13

Searching for Leak Witnesses

We can adapt the active Xsfl leak-witness checker of Chapter 12 to search for leak witnesses automatically. This chapter presents the results of two experiments searching for leaks.

The witness trials take a call-by-need evaluator, search its rule sequences for candidate leak witnesses and test whether those candidates witness a leak relative to another call-by-need evaluator. The aim is to find out which evaluator pairs have an active Xsfl leak and in each case to find the smallest witness, to indicate the source of the leak.

The candidate trials look for candidate leak witnesses on the call-by-need evaluator [lazy]. The aim is to find programs which indicate possible space inefficiency in the standard lazy evaluation strategy.

Section 13.1 begins with a straightforward search of all possible rule sequences. Section 13.2 to Section 13.6 introduce some tactics to restrict the search space without losing any solutions. Section 13.7 summarises the results and more details are given in Appendix C.

13.1 Searching for a witness

The simplest search builds all possible rule sequences of length \( n \), generates the set of candidate leak witnesses for each sequence and tests each candidate as described in Chapter 12.

**Definition 13.1 (All sequences witness and candidate search)**

\[ \text{allSeqs } [A] n = [A]^n \]

\[ \text{witnesses } S [A] [B] \iff = \{ w | n \in \mathbb{N}^+, \tilde{r} \in S [A] n, w \in \text{witness } [A] \text{ gc } [B] \text{ gc } \iff \tilde{r} \} \]

\[ \text{candidates } S [A] = \{ w | n \in \mathbb{N}^+, \tilde{r} \in S [A] n, w \in \text{candidate gc } \tilde{r} \text{ E} \} \]

All rule sequences for a given evaluator and length are generated by the allSeqs function. The function witnesses takes a sequence generator \( S \), two evaluators and a translation. It applies the witness checker to each rule sequence generated for each length (shortest first). The function candidates is similar, finding all the candidate graphs produced from the set of sequences generated by \( S \). Non-standard garbage collectors could also be used if required.

**Example 13.1 (Witness trials)**

The search witnesses allSeqs \([\text{badbh} \ [\text{nobh}]) \iff \text{soon discovers that the one-rule sequence (LookupBad) proves } [\text{badbh} \ \rightarrow [\text{nobh}] \text{ — it happens to be the third sequence}

199
Table 13.1: All sequences witness search to length 3.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>First witness</th>
<th>Search statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>allSeqs tried</td>
</tr>
<tr>
<td>badbh</td>
<td>nobh</td>
<td>(13.1)</td>
</tr>
<tr>
<td>badbh</td>
<td>bhr</td>
<td>(13.2)</td>
</tr>
<tr>
<td>badbh</td>
<td>lazy</td>
<td>(13.3)</td>
</tr>
</tbody>
</table>

tried. The witness is summarised by (13.1) which shows the expanding self-feeding loop of (LookupBad) and the terminating evaluation trace of its translation on nobh.

\[
\begin{align*}
\{ x \mapsto x \} x, \epsilon \rightarrow \text{id}_\text{badbh} & \quad \{ x \mapsto x, a \mapsto \# x \} x, a \ (\text{LookupBad}) \\
\{ x \mapsto x \} x, \epsilon \rightarrow_{\text{nobh}} & \quad \{ x \mapsto x \} x, \epsilon \not\rightarrow_{\text{nobh}} \\
\{ x \mapsto x \} x, \epsilon \rightarrow \text{id}_\text{badbh} & \quad \{ x \mapsto x, a \mapsto \# x \} x, a \ (\text{LookupBad}) \\
\{ x \mapsto x \} x, \epsilon \rightarrow_{\text{bhr}} & \quad \{ x \mapsto \bot, a \mapsto \# x \} x, a \not\rightarrow_{\text{bhr}} \ (\text{Lookup}) \\
\{ x \mapsto x \} x, \epsilon \rightarrow \text{id}_\text{badbh} & \quad \{ x \mapsto x, a \mapsto \# x \} x, a \ (\text{LookupBad}) \\
\{ x \mapsto x \} x, \epsilon \rightarrow_{\text{lazy}} & \quad \{ x \mapsto \bot, a \mapsto \# x \} x, a \not\rightarrow_{\text{lazy}} \ (\text{Lookup})
\end{align*}
\]

Similar witnesses (13.2) and (13.3) are discovered for comparisons of badbh with other evaluators.

Table 13.1 summarises the amount of search work needed to discover these witnesses. It shows the number of sequences generated by the allSeqs tactic before a witness was found; the number of super-rules generated from those sequences; the number of super-rules that form self-feeding loops and the number of those that produce Xsfl witnesses.

All of these comparisons produce the same candidate which causes a leak on badbh by building an unlimited chain of update markers in the stack. The evaluators lazy and bhr prevent this from happening through black-holing (13.2) and (13.3); nobh prevents it by not having any rule to lookup a variable that refers to itself (13.1). So our call-by-need evaluators separate the issue of not detecting direct self-dependency, as modelled by badbh, from not black-holing, as modelled by nobh.

Many other evaluator comparisons were tried, examining sequences up to length 3, but only those involving badbh produced a witness within this limit. □

Example 13.2 (Candidate trials)

The results of the search candidates allSeqs lazy up to sequence length 5 are summarised in Table 13.2 (all the lazy candidate leak witnesses found are collected in Table C.1). This search produces a first candidate for inefficient space usage: the program let \(fx = f \epsilon \epsilon \) in \(f \epsilon \) whose evaluation repeatedly applies the function \(f\) to two arguments. This causes a chain of unused arguments to build up in the stack because \(f\) only takes one argument. The program is ill-typed and we conclude that a type
checkers is a useful tool for preventing some space leaks! Such candidates could be eliminated by restricting translation to well-typed graphs. It might even be possible to use type checking to reduce the search space by eliminating sequences that result in ill-typed programs.

Blindly searching all sequences generates a huge search space. The complexity of the search is roughly the number of sequences considered multiplied by the complexity of super-rule construction for the maximum sequence length considered. The following sections introduce a series of optimisations to cut this space safely by eliminating sequences before the witness checker is applied to them. This idea of introducing tactics for guiding a search without compromising its completeness is known as proof planning in the literature [Bun88].

13.2 Looking for differences

When $A$ and $B$ are close variants and the translation is an identity — as in the witness trials — many rule sequences are not going to produce witnesses because they have the same behaviour on both evaluators. We can restrict the search space on $A$ to the distinct-rule sequences: those including a rule not in $B$.

**Definition 13.2 (Distinct rule sequences)**

$$
\text{distinctSeqs}(B, A, n) = \bigcup_{i=0}^{n-1} \left\{ \text{for } r \in (A \cap B)^i, r \in (A - B), \vec{r} \in A^{(n-i-1)} \right\}
$$

For efficiency, the distinctSeqs tactic generates the set of distinct-rule sequences directly instead of selecting them from allSegs. For sequences of length $n$ it takes $i$ rules common to $A$ and $B$, then a rule in $A$ only, then $(n - i - 1)$ rules from $A$.

**Example 13.3 (Witness trials)**

The distinct-rule sequences tactic reduces the witness-trial search spaces as shown in Table 13.3. Searches such as witnesses (distinctSeqs lazy) badbh lazy $\overset{\text{id}X}{\Rightarrow}$ which produced results much sooner as many rules sequences are eliminated from the search. Extending the search length to 4 we obtain results for searches such as witnesses (distinctSeqs lazy) lhtl lazy $\overset{\text{id}X}{\Rightarrow}$ which finds the witness (13.6) used in the examples throughout Chapter 12. Fortunately, all these witnesses happen to be well-typed, so we can regard them as valid smallest proofs of leakiness. The dots in (13.4), (13.5) and (13.6) indicate that the translated witnesses are non-terminating.
Table 13.3: Distinct sequences witness search to length 4.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>First witness</th>
<th>Search statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>distinctSeqs tried</td>
</tr>
<tr>
<td><code>badbh</code></td>
<td><code>nobh</code></td>
<td>(13.1)</td>
</tr>
<tr>
<td><code>badbh</code></td>
<td><code>bhrtr</code></td>
<td>(13.2)</td>
</tr>
<tr>
<td><code>badbh</code></td>
<td><code>lazy</code></td>
<td>(13.3)</td>
</tr>
<tr>
<td><code>notr</code></td>
<td><code>noabh</code></td>
<td>(13.4)</td>
</tr>
<tr>
<td><code>bhrtr</code></td>
<td><code>badbh</code></td>
<td>(13.5)</td>
</tr>
<tr>
<td><code>bhrtr</code></td>
<td><code>lazy</code></td>
<td>(13.6)</td>
</tr>
</tbody>
</table>

\[
\{ a \mapsto g \epsilon, g \mapsto \lambda h.g \epsilon \} a, \epsilon \\
\quad \rightarrow_{\text{idX}}^{4} \{ a \mapsto g \epsilon, g \mapsto \lambda h.g \epsilon, d \mapsto \text{Env } \epsilon \} a, d \\
\quad \quad \quad \text{(Push, Lookup, Update, EnvReduce)}
\]

\[
\{ a \mapsto g \epsilon, g \mapsto \lambda h.g \epsilon \} a, \epsilon \\
\quad \rightarrow_{\text{notr}}^{4} \{ a \mapsto g \epsilon, g \mapsto \lambda h.g \epsilon \} a, \epsilon \cdots \\
\quad \text{(PushNah, LookupNbh, UpdateNbh, ReduceNah)}
\]

\[
\{ a \mapsto g \epsilon, g \mapsto \lambda h.g \epsilon \} a, \epsilon \\
\quad \rightarrow_{\text{bhrtr}}^{4} \{ a \mapsto g \epsilon, g \mapsto \lambda h.g \epsilon, d \mapsto \text{Ret } \epsilon \} a, d \\
\quad \text{(Push, Lookup, Update, RetReduce)}
\]

\[
\{ a \mapsto g \epsilon, g \mapsto \lambda h.g \epsilon \} a, \epsilon \\
\quad \rightarrow_{\text{badbh}}^{4} \{ a \mapsto g \epsilon, g \mapsto \lambda h.g \epsilon \} a, \epsilon \cdots \\
\quad \text{(Push, LookupBad, UpdateBad, Reduce)}
\]

\[
\{ a \mapsto g \epsilon, g \mapsto \lambda h.g \epsilon \} a, \epsilon \\
\quad \rightarrow_{\text{bhrtr}}^{4} \{ a \mapsto g \epsilon, g \mapsto \lambda h.g \epsilon, d \mapsto \text{Ret } \epsilon \} a, d \\
\quad \text{(Push, Lookup, Update, RetReduce)}
\]

\[
\{ a \mapsto g \epsilon, g \mapsto \lambda h.g \epsilon \} a, \epsilon \\
\quad \rightarrow_{\text{lazy}}^{4} \{ a \mapsto g \epsilon, g \mapsto \lambda h.g \epsilon \} a, \epsilon \cdots \\
\quad \text{(Push, Lookup, Update, Reduce)}
\]

The value of distinctSeqs reduces as the sequence length increases. In the best case \[A\] has \(r\) rules of which one is not part of \[B\]. Searching rule sequences of length \(n\), the total number of sequences is \(r^n\) but the number ruled out by the distinct-sequence optimisation is only \((r - 1)^n\). For example, if we are searching for a proof of \[\text{bhrtr} \rightleftharpoons \text{lazy}\] with a sequence length up to 4, the total number of sequences is \(5^2 + 5^3 + 5^4 = 780\). The number excluded by distinctSeqs is \(4 + 4^2 + 4^3 + 4^4 = 340\), a worthwhile saving of around 43\%. With a length of 10 only around 11\% of sequences are eliminated. If the evaluators have more different rules the savings are reduced even more. Another problem is that distinct sequences are no help in the candidate trials.
### 13.3 Next-rule analysis

Some rule sequences are impossible because rule \( a \) can never follow rule \( b \). Clearly these sequences will not have any super-rules. To eliminate them we generate the *next-rule table* of an evaluator.

**Definition 13.3 (Next-rule table)**

\[
nrt[L \rightarrow R][L' \rightarrow R'] = \exists C', D \cdot C(R) \equiv D(L')
\]

A next-rule table is a boolean matrix with a column and a row for each rule. Entry \([i][j]\) is true iff rule \( j \) may follow rule \( i \) in some evaluation trace.

**Example 13.4 (Next-rule table of \textit{lazy})**

Table 13.4 is the next-rule table of \textit{lazy}. It is quite dense, 20 out of a possible 25 rule sequences may occur.

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{First Rule} & \text{Let} & \text{Push} & \text{Reduce} & \text{Lookup} & \text{Update} \\
\hline
\text{Let} & \text{True} & \text{True} & \text{True} & \text{True} & \text{True} \\
\text{Push} & \text{True} & \text{True} & \text{True} & \text{True} & \text{False} \\
\text{Reduce} & \text{True} & \text{True} & \text{True} & \text{True} & \text{True} \\
\text{Lookup} & \text{True} & \text{True} & \text{False} & \text{True} & \text{False} \\
\text{Update} & \text{False} & \text{False} & \text{True} & \text{False} & \text{True} \\
\hline
\end{array}
\]

In general, we expect an \( nrt \) to be quite dense — a sparse table would suggest that the evaluator should be reformulated by coalescing some of its rules. In general, further processing of the \( nrt \) may be appropriate to leave only strongly-connected cycles, since we are only interested in rules that are reachable from themselves. None of the call-by-need evaluators need this optimisation. The \( nrt \) sequences are those permitted by the next-rule table.

**Definition 13.4 (Next-rule-table sequences)**

\[
nrtSeqs \nexists n = \{ \langle r_1, \ldots, r_i \rangle \in \text{allSeqs} \nexists n | \wedge_{j=1}^{i} \nrt[r_j][r_{j+1}] \land \nrt[r_i][r_1] \}
\]

All adjacent rule pairs in \( nrtSeqs \) are possible and the first rule may follow the last.

The set \( nrtSeqs \nexists n \) can be generated more directly for efficiency. Rather than selecting from \( \text{allSeqs} \nexists n \), we trim \( \nexists \) for each of its rules \( r \) so that after rule \( r \) the next rule in a sequence must come from the trimmed evaluator \( \nrt[r] \). Then only those sequences for which \( \neg \nrt[r_n][r_1] \) need to be removed. The first rule \( r_1 \) in a sequence could be recorded as the rest of the sequence is generated with the final rule \( r_i \) selected from a three-dimensional table to eliminate all filtering. We can also combine this with the \textit{distinctSeqs} tactic of Section 13.2 if appropriate.

**Example 13.5 (Witness trials)**

Extending the search to length 5 and using the \( nrtSeqs \) tactic we do not find any new leaks among the call-by-need evaluators but the size of the search space is reduced significantly as shown by Table 13.5.
Table 13.5: Next-rule-table sequences witness search to length 5.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>First witness</th>
<th>Search statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>nrtSeqs tried</td>
</tr>
<tr>
<td>badbh</td>
<td>nobby</td>
<td>(13.1)</td>
</tr>
<tr>
<td>badbh</td>
<td>bht</td>
<td>(13.2)</td>
</tr>
<tr>
<td>badbh</td>
<td>lazy</td>
<td>(13.3)</td>
</tr>
<tr>
<td>notr</td>
<td>noabh</td>
<td>(13.4)</td>
</tr>
<tr>
<td>bht</td>
<td>badbh</td>
<td>(13.5)</td>
</tr>
<tr>
<td>bht</td>
<td>lazy</td>
<td>(13.6)</td>
</tr>
</tbody>
</table>

Table 13.6: Next-rule-table sequences candidate search to length 6.

<table>
<thead>
<tr>
<th>Seq len</th>
<th>#allSeqs</th>
<th>#nrtSeqs</th>
<th>Srs</th>
<th>Sfls</th>
<th>Xsfls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>17</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>125</td>
<td>65</td>
<td>104</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>625</td>
<td>257</td>
<td>602</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3125</td>
<td>1025</td>
<td>3725</td>
<td>52</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>15625</td>
<td>4097</td>
<td>24406</td>
<td>399</td>
<td>159</td>
</tr>
</tbody>
</table>

Example 13.6 (Candidate trials)

Table 13.6 shows that the number of sequences generated by nrtSeqs lazy 3 is 65. They have 104 super-rules. This is less than the 128 super-rules generated when all sequences were searched (Table 13.2) because it excludes super-rules which cannot form a loop, such as (Lookup, Lookup, Update).

The nrtSeqs tactic gets better as the sequences lengthen. At length 12, there are only 16,777,217 nrtSeqs compared to 244,140,625 allSeqs. The size of nrtSeqs lazy n is roughly 4^n whereas the size of allSeqs lazy n is 5^n.

Taking the lazy candidate search to length 6 gives the first well-typed example of inefficient space usage (and more ill-typed examples, see Appendix C). The witness program is let \( f x = \text{let } y = f \epsilon \text{ in } y \) \( \text{in } f \epsilon \). Example 12.15 showed that this graph can be made to run in constant space by using the update-marker garbage collector #gc. So we have not unearthed a new leak here; there is a known cure for the leaky programs represented by this witness.

For searches of longer sequences it may be appropriate to create 3 or 4 dimensional next-rule tables for eliminating longer rule sequences which are impossible, such as (Push), (Push), (Reduce), (Update) lazy 4. A similar idea is to use more sharing by building super-rules from compound rules, so sequences of length 7 might be generated by taking combinations of 3 rules from A lazy 2 and one from A which are permitted by the appropriate next-rule table. Our experiments in this area show that further speedups can be obtained but the choice of data structure for the evaluator and the next-rule tables becomes much more important to avoid long access times. The amount of sharing and the size of the tables should also be chosen carefully to avoid memory problems.
Table 13.7: Permutation-loop-free sequences witness search to length 6.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>First witness</th>
<th>Search statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>badbh</td>
<td>nobh</td>
<td>13.1</td>
</tr>
<tr>
<td>badbh</td>
<td>bhtr</td>
<td>13.2</td>
</tr>
<tr>
<td>badbh</td>
<td>lazy</td>
<td>13.3</td>
</tr>
<tr>
<td>notr</td>
<td>hoabb</td>
<td>13.4</td>
</tr>
<tr>
<td>bhtr</td>
<td>badbh</td>
<td>13.5</td>
</tr>
<tr>
<td>bhtr</td>
<td>lazy</td>
<td>13.6</td>
</tr>
<tr>
<td>lazy</td>
<td>sest ; #gc</td>
<td>13.7</td>
</tr>
<tr>
<td>lazy</td>
<td>#ind</td>
<td>13.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>plfSeqs tried</th>
<th>Srs</th>
<th>Sfls</th>
<th>Xsfls</th>
</tr>
</thead>
<tbody>
<tr>
<td>badbh</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>badbh</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>badbh</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>notr</td>
<td>30</td>
<td>41</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bhtr</td>
<td>9</td>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bhtr</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>lazy</td>
<td>443</td>
<td>3369</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>lazy</td>
<td>372</td>
<td>3238</td>
<td>16</td>
<td>5</td>
</tr>
</tbody>
</table>

13.4 Permutation-loop removal

So far the search generates all permutation loops of a rule sequence: as well as \( \langle a, b, c \rangle \) we get \( \langle b, c, a \rangle \) and \( \langle c, a, b \rangle \). If one of these forms a leak witness, so will the others, so we can safely eliminate loops at the sequence generation stage.

**Definition 13.5 (Permutation-loop-free sequences)**

\[
\text{plfSeqs} \equiv n \wedge \{ \langle r_i \rangle_{i=1}^n \in \text{nrtSeqs} \mid n \wedge \{ \langle r_i \rangle_{i=1}^n < \langle r_j, \ldots, r_n \rangle + \langle r_1, \ldots, r_{j-1} \rangle | 1 < j \leq i \} \}
\]

We define a total ordering \(<\) on the rules in an evaluator, \(\text{plfSeqs}\) selects only those sequences which are the least of all their permutation loops according to \(<\).

A potential problem with the \(\text{plfSeqs}\) tactic is that the least permutation loop might not produce the most readable witness of all the permutation loops. Worse, we lose a solution if the least permutation-loop candidate is not translatable but another permutation-loop candidate has a translation. Fortunately, we have already arranged that the witness checker tries all permutation loops of a candidate (see Example 12.16) so this is not a problem.

**Example 13.7 (Witness trials)**

The search witnesses \(\text{plfSeqs}\) \(\text{bhtr}\) \(\text{lasy}\) \(\text{idX}\) produces a result after trying only eight sequences (see Table 13.7). Increasing the search to sequences of length 6 also discovers the first witness that \(\text{lasy}\) is leakier than \(\text{sset}\) with the \(#gc\) collector (13.7). This is the program discussed in Example 12.15 whose expansion can be cured by the update-marker garbage collector.

\[
\begin{align*}
\{ a \mapsto g \epsilon, g \mapsto \lambda h.\text{let } i = g \epsilon \text{ in } i \} a, \epsilon \\
\Rightarrow^6_{\text{lazy}} \{ a \mapsto \perp, e \mapsto g \epsilon, g \mapsto \lambda h.\text{let } i = g \epsilon \text{ in } i, d \mapsto \#a \epsilon \} e, d \\
(\text{Push, Lookup, Update, Reduce, Let, Lookup}) \\
\rightarrow^\text{idX} \\
\{ a \mapsto g \epsilon, g \mapsto \lambda h.\text{let } i = g \epsilon \text{ in } i \} a, \epsilon \\
\Rightarrow^6_{\text{sset}; \#gc} \{ f \mapsto g \epsilon, g \mapsto \lambda h.\text{let } i = g \epsilon \text{ in } i \} f, \epsilon \\
(\text{Push, ScLookup, ScUpdate, Reduce, Let, ScLookup})
\end{align*}
\]
Table 13.8: Permutation-loop-free sequences candidate search to length 7.

<table>
<thead>
<tr>
<th>Seq length</th>
<th>#allSeqs</th>
<th>#plfSeqs</th>
<th>Srs</th>
<th>Sfls</th>
<th>Xsfls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>11</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>125</td>
<td>25</td>
<td>44</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>625</td>
<td>71</td>
<td>210</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3125</td>
<td>209</td>
<td>1114</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>15625</td>
<td>701</td>
<td>6778</td>
<td>48</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>78125</td>
<td>2345</td>
<td>43203</td>
<td>271</td>
<td>170</td>
</tr>
</tbody>
</table>

\[
\{a \mapsto g \epsilon, g \mapsto \lambda h. \text{let } i = g \epsilon \text{ in } i \} a, \epsilon \\
\xrightarrow{6}_{\text{lazy}} \{a \mapsto \bot, e \mapsto g \epsilon, g \mapsto \lambda h. \text{let } i = g \epsilon \text{ in } i, d \mapsto \# a \epsilon \} e, d \\
\quad (\text{Push, Lookup, Update, Reduce, Let, Lookup})
\]

\[
\sum_{\text{idX} \#} \xrightarrow{\mathbb{I}} \{a \mapsto g \epsilon, g \mapsto \lambda h. \text{let } i = g \epsilon \text{ in } i, b \mapsto \# \epsilon \} a, b \\
\xrightarrow{6}_{\text{ind}} \{f \mapsto g \epsilon, g \mapsto \lambda h. \text{let } i = g \epsilon \text{ in } i, b \mapsto \# \epsilon \} f, b \cdots \\
\quad (\text{Push, Lookup \#}, \text{Update, Reduce, Let, Lookup\#})
\]

The same witness (13.8) shows \[\text{lazy} \equiv \text{ind}\] but in order to show that its translation runs in constant space we use the translation \[\sum_{\text{idX} \#}\] defined as follows.

\[
\sum_{\text{idX} \#} = (\text{start \#}, \{(\text{Id.} \lambda), (\text{Id. Var}), (\text{Id.} \text{Apply}), (\text{Id.} \text{let})\})
\]

\[
\text{start \#}(\{ \text{Ga, } \epsilon \}, (H \{ t \mapsto \# \epsilon \} b, t)) = (\{ a \cdot b \}, \{ \}, G, H)
\]

Translating with \[\sum_{\text{idX}}\], our simple non-termination analysis cannot detect that it runs in constant space. The trace is basically a six-rule self-feeding loop but the repetitive behaviour does not begin until the seventh step of the evaluation trace because the first variable lookup must use \((\text{Lookup } \epsilon)\) so that the variable lookups in the repeating loop can use the non-allocating rule \((\text{Lookup } \#)\).

\[\square\]

Example 13.8 (Candidate trials)
Table 13.8 shows a substantial reduction in the search space resulting from the \(\text{plfSeqs}\) tactic. Nearly \((n - 1)/n\) of sequences of length \(n\) are eliminated before super-rule generation begins. So the search space at length 5 is down from \(5^5\) to \(5^4\), at length 10 it is reduced from \(5^{10}\) to \(5^8\). In combination with \(\text{nrtSeqs}\) the exponent of the search-space size is reduced by around 40%.

\[\square\]

The main drawback of \(\text{plfSeqs}\) is that its implementation draws from \(\text{nrtSeqs}\) as suggested in Definition 13.5. Defining a direct generator for \(\text{plfSeqs}\) seems to be a difficult problem, even more so in combination with the other tactics. We can define approximate versions, such as \(\text{lfp paths}\) (13.9), which rule out some permutation loops by arranging that the first rule of all the sequences it generates is never greater than any other rules in the sequence by the rule ordering \(>\).
**Table 13.9: Expanding sequences witness search to length 7.**

<table>
<thead>
<tr>
<th>Comparison</th>
<th>First witness</th>
<th>Search statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>badbh &amp; badbh</td>
<td>nobh</td>
<td>( (13.1) )</td>
</tr>
<tr>
<td>badbh &amp; badbh</td>
<td>bhtr</td>
<td>( (13.2) )</td>
</tr>
<tr>
<td>badbh &amp; badbh</td>
<td>lazy</td>
<td>( (13.3) )</td>
</tr>
<tr>
<td>noth &amp; noth</td>
<td>hoabdh</td>
<td>( (13.4) )</td>
</tr>
<tr>
<td>bhtr &amp; badbh</td>
<td>badbh</td>
<td>( (13.5) )</td>
</tr>
<tr>
<td>bhtr &amp; badbh</td>
<td>lazy</td>
<td>( (13.6) )</td>
</tr>
<tr>
<td>lazy &amp; sest ; #gc</td>
<td>lazy</td>
<td>( (13.7) )</td>
</tr>
<tr>
<td>lazy &amp; lazy</td>
<td>#ind</td>
<td>( (13.8) )</td>
</tr>
</tbody>
</table>

\[
\text{Ifpaths} \left( \{r_i\}_{i=1}^m \right) n = \bigcup_{i=1}^m \left\{ \langle r_i \rangle + S \mid S \in \left( \{r_j\}_{j=1}^m \right)^{n-1} \right\} 
\]  
\( (13.9) \)

So Ifpaths \( \{a, b\} 2 = \{\langle a, a\rangle, \langle a, b\rangle, \langle b, b\rangle\} \) is permutation-loop free but longer sequences are not: Ifpaths \( \{a, b\} 3 = \{\langle a, a, a\rangle, \langle a, a, b\rangle, \langle a, b, b\rangle, \langle a, b, a\rangle, \langle b, b, b\rangle, \langle b, b, a\rangle\} \). Difficulties aside, permutation-loop elimination is very useful in cutting the search space.

### 13.5 Expanding sequences

All candidate leak witnesses are expanding self-feeding loops. Therefore any rule sequence which cannot cause expansion can be eliminated before its super-rules are generated.

**Definition 13.6 (Expanding sequences)**

\( \text{expSeqs} ^A n = \{ \langle L_i \rightarrow R_i \rangle_{i=1}^n \in \text{plfSeqs} ^A n \mid \sum_{i=1}^n (\#R_i - \#L_i) > 0 \} \)

All permutation-loop-free rule sequences up to length \( n \) which might cause their redex to expand are included in \( \text{expSeqs} ^A n \).

This tactic does not interfere with any others but, again, a generator for \( \text{expSeqs} \) is not easily combined with them to form a more efficient sequence generator. A straightforward implementation selects from \( \text{plfSeqs} \) as suggested in Definition 13.6. A simple way to improve the search for a given sequence length is to order the rules from most allocating to most deallocating so that sequences produced earlier are more likely to cause expansion. For \[\text{lazy}\], the alphabetical ordering of rules \[\text{Let, Lookup, Push, Reduce, Update}\] has this property.

**Example 13.9 (Witness trials)**

Now we extend the search length to 7. Some of the length 4 witnesses and all the length 1 witnesses are discovered instantly (see Table 13.9). However, no new witnesses are found and although reduced, the search space size is still large.
Table 13.10: Expanding sequences candidate search to length 7.

<table>
<thead>
<tr>
<th>Seq length</th>
<th># allSeqs</th>
<th># expSeqs</th>
<th>Srs</th>
<th>Sfls</th>
<th>Xsfls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>6</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>125</td>
<td>17</td>
<td>36</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>625</td>
<td>42</td>
<td>167</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3125</td>
<td>153</td>
<td>1019</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>15625</td>
<td>457</td>
<td>5940</td>
<td>29</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>78125</td>
<td>1797</td>
<td>40904</td>
<td>271</td>
<td>170</td>
</tr>
</tbody>
</table>

Example 13.10 (Candidate trials)
Table 13.10 shows that removing non-expanding sequences typically eliminates 25% of the permutation-loop-free sequences. This figure does not appear to change as the search depth is increased. As lazy has three expanding rules and two contracting rules we might expect around 40% of sequences to be non-expanding. The discrepancy may be a result of the next-rule-table analysis which makes some rules more likely to occur in a sequence than others.

13.6 Approximate candidate sequences

The exact super-rule construction algorithm can generate a lot of solutions for certain combinations of rules, particularly where many of the right-patterns have hole substitutions or disconnected nodes. An approximate version of the disunification algorithm, which generalises at the points where the exact version produces multiple solutions, gives only one super-rule for each rule sequence.

Definition 13.7 (Approximate pattern unification)
\[ \text{apxUnify } h[x_i/y_i]_{i=1}^n P = \{ h \mapsto \{ v \mapsto z \mid v \in \text{fv}(P), z \text{ fresh} \}(P) \} \]

Approximate unification is defined by the usual \text{unify} rules except for term patterns when one pattern is a hole with a substitution. Then we generalise by mapping the free variables in \( P \) to fresh variables, instead of generating a unifier for each way the substitution can affect the free variables as in Definition 12.4. Any hole substitutions in \( P \) are ignored. The multiple solutions of Definition 12.3 are not generated either.

The approximate version of \text{superRule} forms a powerful sequence filter. An approximate unifier can always be specialised to any of the exact unifiers. So we should not lose any solutions this way. This is an unproven conjecture but it has never been observed to fail.

Definition 13.8 (Approximate sequences)
\[ \text{apxSeqs } n = \{ \bar{r} \in \text{expSeqs } n \mid (L \rightarrow R) \in \text{apxSuperRule } \bar{r}, \text{sflUnify } L R \neq \{ \} \} \]

The \text{apxSeqs} tactic generates all \text{expSeqs} with a self-feeding approximate super-rule. The approximate super-rule is generated for each sequence \( \bar{r} \). If it has any self-feeding loops then \( \bar{r} \) might produce candidate witnesses.
Table 13.11: Approximate candidate sequences witness search to length 8.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>First witness</th>
<th>Search statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>badbh</td>
<td>nobh</td>
<td>$(13.1)$</td>
</tr>
<tr>
<td>badbh</td>
<td>bhtr</td>
<td>$(13.2)$</td>
</tr>
<tr>
<td>badbh</td>
<td>lazy</td>
<td>$(13.3)$</td>
</tr>
<tr>
<td>hotr</td>
<td>noabnh</td>
<td>$(13.4)$</td>
</tr>
<tr>
<td>bhtr</td>
<td>badbh</td>
<td>$(13.5)$</td>
</tr>
<tr>
<td>bhtr</td>
<td>lazy</td>
<td>$(13.6)$</td>
</tr>
<tr>
<td>lazy</td>
<td>sess ; #gc</td>
<td>$(13.7)$</td>
</tr>
<tr>
<td>lazy</td>
<td>#incl</td>
<td>$(13.8)$</td>
</tr>
</tbody>
</table>

Table 13.12: Approximate candidate sequences candidate search to length 8.

<table>
<thead>
<tr>
<th>Seq length</th>
<th>#allSeqs</th>
<th>#apxSeqs</th>
<th>Srs</th>
<th>Slf</th>
<th>XSlf</th>
<th>Distinct XSlf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>125</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>625</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3125</td>
<td>3</td>
<td>13</td>
<td>8</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>15625</td>
<td>8</td>
<td>67</td>
<td>29</td>
<td>22</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>78125</td>
<td>39</td>
<td>624</td>
<td>271</td>
<td>170</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>390625</td>
<td>128</td>
<td>4092</td>
<td>1375</td>
<td>1093</td>
<td>94</td>
</tr>
</tbody>
</table>

Example 13.11 (Witness trials)
Searching longer sequences does not reveal any new leak witnesses but the search space is reduced dramatically as shown in Table 13.11. Given time, larger evaluators can be searched. However, the implementation is still not really fast enough to find the 6-rule self-feeding loop witnessing that prsh is leakier than sess with #gc. Just searching the 15-rule evaluator prsh to length 5 (this takes about a day) produces 134,004 distinct nrt sequences, of which 22,527 are permutation-loop-free and might expand, giving 8 candidate witnesses, none of which run in bounded space on sess with #gc.

So we are still not able to search very long rule sequences or very large evaluators, and the search is only a semi-decision procedure. Table 13.11 is far from proof that there are no more active leaks among our call-by-need evaluators. But the lack of new leak witnesses suggests that it might be worth attempting proofs that the other evaluator comparisons are passive leaks.

Example 13.12 (Candidate trials)
Table 13.12 shows the power of the apxSeqs tactic in reducing the search space of previous searches and allowing searches of longer sequences. The number of sequences which are put through the full super-rule and self-feeding loop tests is reduced by an order of magnitude for these searches of short sequences. The consistency of the
numbers in Table 13.12 with previous candidate trials support the claim that the \textit{apxSeqs} tactic does not lose any solutions.

\section*{13.7 Summary}

Figure 13.1 summarises the results of the leak search and checking experiments conducted throughout Part IV. Searching all possible rule sequences for active leak witnesses is a successful way to find a smallest proof-by-example of leakiness. However, the search space resulting from considering all possible sequences is exponential in the sequence length. We introduced a series of techniques which reduce this space dramatically. Eliminating impossible sequences at the generation stage reduces the base of the search-space-size equation; removing redundant permutations reduces the exponent; removing non-expanding sequences gives a constant factor improvement and an approximate witness checker concentrates the complexity of the full checker onto a few sequences of which most turn out to be genuine witnesses.

Appendix A gives more information on the implementation and Appendix C summarises the graphs found by the candidate trials.
Figure 13.1: Hierarchy of leakage for call-by-need evaluators. Each comparison is labelled with a letter. For the \( \Rightarrow \) comparisons the left letter labels the comparison above-evaluator \( \Rightarrow \) below-evaluator. All comparisons are with respect to the translation \( id \) except \( r \) and \( s \) which use \( id \) to include lists and projections. In comparisons \( r \) and \( s \) the leakier evaluator is \( \text{proj} \). The \( \equiv \) comparisons are carried over from Figure 10.1. Table 13.13 summarises the proof and classification of each leak in the hierarchy.

Table 13.13: Details of proofs in leak hierarchy. In the witness column, \( G \) stands for the graph \( \{ \text{fin} \mapsto \lambda z.z \text{fin}, l \mapsto \text{lof} \epsilon, \text{lof} \mapsto \lambda x. \text{let } y = \text{lof } x \text{ in } \lambda z.y \} \). All the active \( Xsfl \) leaks are discovered by the search procedure described in this chapter. The rest are hand-made, but all the \( Xsfl \) witnesses can be checked automatically as described in Chapter 12.

<table>
<thead>
<tr>
<th>Comparisons</th>
<th>Witness</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>a,d,f,l</td>
<td>( G{ a \mapsto (\text{fin } l) \epsilon } a, \epsilon )</td>
<td>passive ( Xsfl )</td>
</tr>
<tr>
<td>b,k,o</td>
<td>( { a \mapsto f \epsilon, f \mapsto \lambda x.f \epsilon } a, \epsilon )</td>
<td>active ( Xsfl )</td>
</tr>
<tr>
<td>c,g,h,m</td>
<td>( G{ a \mapsto b, b \mapsto \text{fin } l } a, \epsilon )</td>
<td>passive ( Xsfl )</td>
</tr>
<tr>
<td>e</td>
<td>( G{ a \mapsto f l, f \mapsto \lambda x.\text{fin } l } a, \epsilon )</td>
<td>passive ( Xsfl )</td>
</tr>
<tr>
<td>i,j,n</td>
<td>( { a \mapsto a } a, \epsilon )</td>
<td>active ( Xsfl )</td>
</tr>
<tr>
<td>p,t,u,v</td>
<td>( { a \mapsto f \epsilon, f \mapsto \lambda x.\text{let } y = f \epsilon \text{ in } y } a, \epsilon )</td>
<td>active ( Xsfl )</td>
</tr>
<tr>
<td>q</td>
<td>( { a \mapsto g g, g \mapsto f, f \mapsto \lambda x.\text{let } y = x \text{ in } y } a, \epsilon )</td>
<td>passive ( Inffin )</td>
</tr>
<tr>
<td>r,s,w</td>
<td>( G{ a \mapsto \text{let } (h:t) = \epsilon : l \text{ in } (\text{fin } t) h } a, \epsilon )</td>
<td>passive ( Xsfl )</td>
</tr>
</tbody>
</table>
Conclusions
Chapter 14

Assessment

Preceding chapters used equational term-graph evaluators to describe space semantics; formalised space leaks as ternary relations on two evaluators and a translation; developed techniques for showing no-leak by simulation with localised reasoning where possible; used translation as a way to specify garbage collection and proved leaks by checking witnesses or searching for witness examples.

Section 14.1 lists the main contributions of this thesis. Section 14.2 lists ideas for improvement of the current work and for further work. Section 14.3 concludes.

14.1 Summary of contributions

14.1.1 Term graphs for space semantics

Graph models of abstract-machine configurations (Sections 4.1 and 4.2) The GraphKit equational term-graph model provides a powerful and expressive model of state. It generalises both higher-order terms and graphs so that structures such as graphs of λ-terms are easily defined as well as simpler graph structures to model features like run-time stacks. The system of categories and the guarantee of well-formedness help to restrict this powerful model so that only those graphs which have some sensible interpretation can be written. The system only seems to be too restrictive when we want to describe unusual binding structures, such as in environment-based evaluation.

GraphKit evaluators model abstract machines (Sections 4.3 and 4.4) An abstract machine is modelled in GraphKit as an evaluator: a set of transformation rules which are guaranteed to preserve the category structure and well-formedness of graphs. Rules may copy or substitute into whole terms which enables a high-level description of an evaluator compared with systems that require explicit substitution. Rules may also read or write terms via a simple IO mechanism. The process of matching a graph and a rule pattern is formalised by defining graph contexts. Contexts fit well with our operational description and are also useful for defining an order of generality on graph patterns. Null terms and variables are also included in the theory so that an instance of any graph pattern can easily be found.

GraphKit evaluators are space semantics (Section 5.1) GraphKit includes by default maximal reachability-based garbage collection and a graph size model which
make it suitable for reasoning about space complexity. The accuracy restriction makes evaluators a stronger resource-usage model with an accurate time measure and an accurate space usage measure based on node counting. In practice, most of the evaluation strategies we considered could easily be modelled as accurate GraphKit evaluators.

**Limited safe static memory management (Sections 5.2 and 8.2)** GraphKit uses maximal garbage collection but it also provides an optimisation for explicit deallocation where parts of a graph are being used in a stack-like manner. There is also the facility for defining non-standard garbage-collection strategies by specifying a collector as a *space relation*.

**GraphKit models of lazy evaluation and Haskell (Chapter 6)** We produced GraphKit evaluators which model the behaviour of several core parts of Haskell and several well-known lazy evaluation techniques. There is some question of how close these models are to reality, it is easy to make small optimisations in the graph models which make them less like the machines they model. As an example there are some small differences between our standard lazy evaluator and the popular Sestoft machine, which we encode more directly as [sest]. Of course, the proof techniques we developed can often show that such optimisations are safe (as well as space-safe). The natural level of abstraction of GraphKit evaluators sometimes makes modelling more difficult — the exact correspondence between our model of trimming-free evaluation [not] and a trimming-free abstract machine is more difficult to visualise — nevertheless, we claim GraphKit evaluators can describe the essential space behaviour of a wide variety of implementation techniques.

### 14.1.2 Defining and classifying space leaks

**Leakiness is relative (Section 7.1)** A space leak — an unexpected worsening in space complexity — is a relative property. This observation was formalised by defining a leak as a relation between graph evaluators: the implementation model and the expectation model. Leaks, or more generally, space-fault criteria, are defined as ternary relations on two evaluators and a transformation relation. To be leaky the implementation model needs to be worse than the expectation model for some program.

**Leakiness depends on the garbage collectors (Section 7.1)** Garbage is central to space usage so our ternary space-leak relations are really 5-ary relations — an implementation model evaluator and collector are leakier or not leakier than an expectation model evaluator and collector, with respect to some translation.

**Instances of leakiness (Section 7.1)** This general notion of worsening can be used in many ways. Our discussion centres on the interpretation $\mathbf{A}\leq\mathbf{B}$ wrt $\implies$: we compare different evaluation strategies relative to an initial graph translation. Some other useful questions are listed below.

- $\square\not\leq\square$ wrt $\implies$ — does the transformation $\implies$ make the encoding of any program less efficient?
• \( A \equiv \square \text{ wrt } \equiv_\text{id} \) — does \( A \) guarantee evaluation in bounded space?

• \( A \equiv B \text{ wrt } \{ (G, G) \} \) — is \( A \) worse than \( B \) for evaluating the graph \( G \)?

• \( A \equiv A \text{ wrt } \equiv \) — does the transformation \( \equiv \) worsen the run-time complexity of any program?

• \( A \equiv A ; gc' \text{ wrt } \equiv_\text{id} \) — does collector \( gc' \) improve the run-time complexity of any program?

Quantifying leakiness (Section 7.2) It is not so clear how much worsening in the GraphKit space-usage model constitutes the effect known as a space leak. Our definition is guaranteed to correspond to worse space complexity for some program — assuming that the space measure is sufficiently accurate. The abstraction from size to node counting is useful for showing that indirection-based evaluators like \([\text{ind}]\) have the same complexity as value-copying evaluators like \([\text{lazy}]\), so basing our leak definition of the space-measure was the right choice for our examples. The stronger space fault criteria such as \(\text{unsafe} \) clearly correspond to real worsening, and many of our example leaks belonged to this class. A further point of controversy is that our definitions include — and our methods rely on — the inclusion of non-terminating programs.

Causes of leakiness (Chapter 11) Our analysis separates the idea of a leak caused by inefficient encoding of some structure (a translation leak) from a leak caused by inefficiency in the evaluation strategy (an evaluation leak). Another useful distinction was discovered between active leaks caused by certain evaluation sequences and passive leaks which only occur in certain contexts.

14.1.3 Proving no-leak by simulation

Space relations (Chapter 8) A translation that is space-efficient in one direction is the basis of our method for showing no-leak. The Proportional Space Relation (PSR) framework supports the definition and checking of these space relations. Space relations are also useful as a way of specifying what a garbage collector does. GraphKit includes a simple extension to allow the definition of non-standard collectors as space relations defined by configuration rewrite systems.

s-lockstep simulation proofs (Section 9.1) To show no-leak requires a proof for all programs. We developed a specialisation of the simulation proof technique for this purpose: show that a space relation is an \textit{s-lockstep simulation}. In principle this is complete, but the proofs can be long and difficult so we developed a simulation proof framework based on PSRs and \textit{lockstep simulation} with which to compare our graph evaluators.

Lockstep simulation (Chapter 9) By restricting simulation to be lockstep we check and find no-leak proofs automatically. This amounts to showing that a PSR translation is preserved by every possible evaluation step. But there are problems: the complete proof must be represented by a small number of cases for each evaluation rule and we need to account for any garbage that may be generated after the evaluation
step. The properties of PSRs and some additional rules solve these problems in many cases.

**Local reasoning (Chapter 9)** Translations defined in the PSR framework allow a simulation proof to be constructed by just considering the effect an evaluation step has locally at the root of a graph. The PSR framework also solves the problem of accounting for any garbage generated by an evaluation step: if \( G \) translates to \( H \) then the garbage collection of \( G \) translates to the garbage collection of \( H \). However, the current scheme only works for the standard reachability-based garbage collector.

**Simulation completion (Section 10.1)** The simulation checker is extended to become a *proof editor* which infers new translation rules to complete part of a proof, asking the user to select one before continuing. Using the proof editor we can just specify a translation on initial states. The correspondence between the different control structures the evaluators use is discovered automatically.

**Global reasoning (Section 10.2)** The PSR translation framework encodes certain global properties but we sometimes found that it does not naturally represent other properties needed to complete a simulation proof. The PSR rule notation can be enriched to allow the translation of variable-sized sub-graphs. Extending the PSR *start* rule is another way to define a global translation; this is useful for translating structures like stacks, when the way in which the elements are connected is significant and local translation rules would lose some inter-dependency information needed for translation.

**Breaking lockstep (Section 10.3)** Most of the no-leak evaluator comparisons we tried turned out not to have lockstep simulation proofs. This problem can often be overcome by generalising from exact lockstep to *s-lockstep* or by showing *non-lockstep termination* in a few cases. More difficult comparisons require us to match a variable number of steps to a single step.

### 14.1.4 Leak-witness checking and search

**The Infin leak class (Section 11.3)** A single witness graph (and its translation) is enough to expose an *Infin* leak. The witness graph grows without limit on the leaky evaluator. It is often possible to give a regular expression which describes the evaluation trace of the witness.

**Active Xsfl leaks (Section 11.4)** Leak witnesses with an evaluation trace that regularly repeats the same sequence of rules belong to the *Xsfl* class. If the leak is active then the *super-rule* and *self-feeding loop* construction procedures can be used to build a candidate leak witness automatically. These procedures use a higher-order graph-pattern unifier to merge and specialise graph-evaluator rules.

**Witness checking (Chapter 12)** To check a candidate leak witness it is translated and its behaviour on the supposedly-safer evaluator is tested. By showing that the translation runs in constant space the leak witness is confirmed. This analysis uses the
super-rule and self-feeding loop constructors to test for non-termination in constant space.

**Passive leaks (Section 12.5)** Suspected passive-leak witness graphs can be confirmed using the active-witness checker tool. We have to supply extra information in the form of a graph context to make the self-feeding loop of its evaluation trace cause expansion.

**Searching for witnesses (Chapter 13)** By testing all possible rule sequences we can search for a smallest active leak witness. This proved successful on our simple small evaluators, but it is very slow because the search space generated by considering all rule sequences is huge and the super-rule constructor can generate multiple solutions for each sequence.

**Proof planning (Chapter 13)** We introduced some tactics which cut the search space sufficiently to find active Xsfl leak witnesses automatically for all of the evaluator comparisons where we believe such a witness to exist.

## 14.2 Further work

The graph model and proof methods have many limitations. Some are fundamental to the approach but many could be lifted to extend the applicability of the methods.

### 14.2.1 Improving GraphKit

**GraphKit the tool** Practically, GraphKit would benefit from a full implementation. The experimental implementation stores graphs as trees which makes their manipulation slow. The presentation could also be improved by including function-symbol precedence information. The output could be improved by providing a pictorial representation of graphs and evaluators.

**GraphKit the formalism** The term-graph model is inadequate for some evaluator descriptions. Enriched terms or some meta-notation would be useful for describing the variadic let function or large finite categories such as bounded integers. This leads to an enriched notation for evaluator rules and the possibility of encoding ideas like environments directly in GraphKit evaluators.

Including a type system would let is ignore the space usage of ill-typed programs. If general enough this might be used to describe type-based garbage collectors [MH97] or sized-type languages [Par01].

A system for specifying initial and terminal states would allow program states to be distinguished from other states and would define valid states as those reachable from initial states. We used translation relations to define initial states in many proofs, but that is a rather indirect solution to the problem.

**Space accuracy** The accuracy check could be generalised to permit more convenient definitions of some evaluators. The undecidability of accuracy should be established
formally. Our test for a garbage-generating rule is also incomplete. This could certainly be improved; it is not clear whether it is decidable.

**Analysis for explicit memory management** A problem of graphs is that they are sometimes much more general than the structures they are modelling. In the spirit of the stack-category analysis, other graph shapes could be handled as special cases with their own safe explicit deallocation analysis.

**Modelling Haskell and more** To use an operational semantics as a basis for discussing the space usage of Haskell programs requires that the Core language is formally defined. Transformations which the compiler may apply (both in translating Haskell to Core and in further optimisations) must also be specified. Then models of core features that we did not consider, such as dictionaries, strict function application and state monads, should be included. Designing GraphKit models of other evaluation strategies would enable a more direct comparison of our methods with those used by Clinger for comparing call-by-value strategies [Cl98]. It would be useful to investigate how graph rewriting could model implementation techniques like *regions* [TT94].

**Making more space faults detectable** The GraphKit space-usage model is not accurate enough for arguments about constant-factor space improvements. Such reasoning could be supported by taking a user-specified *size* measure, or by assuming more about how the model is implemented.

**Specifying garbage collectors** Using space relations to define garbage collectors is an under-specification because these garbage collectors are relations not functions. Their implementation has an unreasonable time complexity and there is no guarantee of correctness relative to an evaluator. Therefore it would be appropriate to find a more limited and realistic answer to the question of what a garbage collector is, and incorporate that into the GraphKit framework.

### 14.2.2 Better no-leak proofs

**Local or global translation** By defining translations locally we are able to give simple simulation proofs, but the translations need to have many properties. The question is whether specifying a translation relation locally or globally leads to a *better* no-leak simulation proof.

**Improving local translation** Locally-defined PSR translations are unrooted configuration-rewrite systems. Finding a way to allow the use of rooted translation rules would improve the time complexity of translation and the simulation proofs. The other main drawback of the PSR translation framework is its inadequate treatment of bound variables, as discussed in Section 8.5.

**Disadvantages of localisation** It seems that local translation rules can never include all the properties we might need, creating a need for special rules like *(Share)* in the proof system, or special treatment for stack-category nodes, for example. Sometimes translating a structure requires more information about how nodes are interconnected and it seems that a global translation scheme may be the best way to do
this. It would be interesting to see whether adding information to nodes about which
nodes they can reach, or which nodes they are reachable from, can be used to sim-
pify translation and avoid the need for a global scheme in some cases. A mix of global
translation for some structures in the graph with local translation elsewhere is another
possibility that we have used already (for example, the \texttt{proj} \texttt{prsh} proof of Example
10.12); the PSR framework is readily extensible in this way though the presentation
of a proof as a set of squares becomes less clear.

**Global translation opportunities** Ideas like translating sub-graphs which have
been duplicated any number of times, describing complex node inter-dependencies
and translating special structures which only occur at certain points in valid graphs
can all be expressed easily with a global translation scheme. The difficulty is how to
define such a translation without resorting to unwieldy layers of subscripted variables
making simulation proof squares too complicated to read and difficult to construct.

**Reasoning about possible states** Sometimes simulation proofs need to exclude
graphs which cannot be reached by evaluation from valid initial states. A global
translation scheme can avoid the need for such an exclusion argument in some cases,
but it is not clear whether this is always possible.

**Different garbage collectors** The use of non-standard garbage collectors to im-
prove space usage is a vital part of some implementation strategies. The simulation
proof method should be extended to include them. By keeping the principle that
translation is preserved after garbage collection, and assuming maximal collection, a
simulation proof with non-standard collectors should work in the same way as before.
Then it should be possible to verify the conjectures in Figure 10.1.

**Non-lockstep proofs** In general, the no-leak proof system needs to handle non-
lockstep simulations. It should provide a way for the user to indicate what kind of
non-lockstep technique needs to be applied. For proof-checking this might mean the
proof editor asking for the number of evaluation steps that should be taken when a
square fails. Specifying a maximum number of steps and letting the tools search for
a proof might be acceptable. But in the cases where one step corresponds to many
another scheme is needed.

**Changing the question** Rather than developing incomplete sound proofs to show
an evaluator is not leakier than another, it would be useful to develop approximations
to suggest whether an evaluator comparison is most likely to be a one-way leak, a
two-way leak or a mutual no-leak. We have not discussed this problem of how to con-
jecture whether a comparison is a leak or a no-leak — mainly because we have been
modelling evaluators with known faults. The tell-tale signs in our leakier evaluators,
such as the presence of more free variables in the right pattern of an evaluator rule,
could be formalised to help make conjectures if proofs cannot be found. Other inter-
pretations of the no-leak question are also open for investigation. Our translations
tend to include all programs but it would be interesting to ask instead what programs,
or what translations, an evaluator comparison is safe for. This could include ideas like
a space-safe strictness analysis.
14.2.3 Finding more leaks

Improving the search tools The present leak checker is adequate for our simple examples but almost all aspects of its operation could be improved. The super-rule construction is complete but produces multiple solutions and it would be worth investigating when more general solutions can be used. The self-feeding loop detection algorithm is incomplete, and could certainly be extended to give more, if not all, solutions; it also gives multiple-solutions, which may not always be necessary. The non-termination and termination analyses are necessarily incomplete but the current solutions are very basic. The search speed could be improved by sharing the partial super-rules from which larger ones are built. Further opportunities for sequence filtering include type-based filtering and eliminating sequences whose super-rule cannot be translated. The search space would be cut more effectively by working out how tactics can be combined to form efficient ready-filtered sequence generators.

Expressions for active-leak witnesses All active-leak witnesses can be expressed by defining their evaluation trace. For the $Xsfl$ leaks this is just the regular repetition of some rule sequence. Other active witnesses could be described similarly by giving some expression which summarises their evaluation trace — we could think of this the imperative definition of a declarative graph program. Checking and search functions could also be devised for these classes. It remains to be seen whether any active-leak witness can be defined and checked in this way.

Finding a passive-leak witness Given a self-feeding loop we could search for a passive-leak witness. If the loop expands before garbage collection then all possible contexts would be searched until one is found which makes the loop expand after garbage collection. With more analysis it may be possible to construct such a context, leading to the possibility of a complete search for $Xsfl$ passive-leak witnesses.

Passive leaks We have not considered how to show that a leak is passive — that it has no active witness. A method, or just a heuristic, for this problem would be useful as a first step in deciding how to go about searching for a leak rather than relying on conjecture or intuition.

Detecting inefficient translation Showing inefficient translation could be automated in a number of ways. One approach is to look for features of translation rules. Another is to treat the problem as looking for a non-terminating translation trace in which the translated graph grows ever larger; a kind of self-feeding translation loop (where the translation is defined by a configuration rewrite system).

Finding inefficient leak witnesses All the call-by-need evaluator comparisons had an Infin leak witness: an unbounded-space program. It may be possible to widen the class of detectable leaks to $\leadsto$ by extending the witness-checking method to construct a family of terminating programs from an expression which describes their evaluation trace. Another approach here is to give rules to turn an Infin witness into such a family of programs.
14.3 Conclusion

Are graphs, leaks, leak classifications and proof frameworks the right way to reason about space usage?

The graph and proof frameworks: are they over-restrictive and too complicated? Our formulation of term-graph rewriting is complex but precise. Higher-order terms with substitution help to create concise and safe graph manipulations rules. Useful ideas from operational semantics such as contexts fit into the theory naturally. The use of disequality constraints may be excessive for describing reduction, but explicating this aspect of the meaning of a pattern proved invaluable in our various other uses for graph patterns. Similarly, graph contexts may be an excessive notation if the only operation of interest is how to evaluate a graph but they are a powerful way to describe graph-pattern specialisation and generalisation, simplifying many of our descriptions.

The GraphKit framework restricts general graph rewriting to make it suitable for describing space usage. Rooted rewriting means that evaluators must include control structures and evaluation is not separable from evaluation strategy; meeting the conditions for accuracy can make formulating an evaluator more difficult than it might be and determinism can make evaluator specifications larger than they might be. But the restrictions simplify many things. Rooted rewriting eliminates the troublesome problems associated with the confluence of graph rewriting; accurate evaluators make graph rewriting a model of resource usage; determinism makes evaluators more realistic models.

Both proof frameworks are semi-decision procedures: a no-leak simulation may complete; a leak search may find a witness. Both are also incomplete and their restrictions often preclude a proof. Simulation completion only works when the evaluators under comparison are exact lockstep and when a locally-defined translation between their states is adequate. We have seen how to overcome the restrictions but the limits of the method are determined by a complex combination of the properties of the evaluation and translation rules. The search for leak candidates is in principle complete for a well-defined class, the active Xsfl leaks, but the implemented method for checking and confirming that a candidate is a witness damages this completeness somewhat.

Modelling space behaviour seems to expose improvement or worsening; are the proofs merely confirmation? Our example graph evaluators are designed to show the relationship between the space usage of different lazy evaluation techniques abstractly. The capabilities of the graph and proof frameworks are designed for those same examples. The general approach is finding a method that works for simple cases (whether the method be modelling or proof) and generalising until it becomes capable of handling enough interesting examples. In that sense all is fabrication.

Working out how to model an implementation technique does say a lot about its space usage, particularly where we have other models to compare it with. The formulation of an evaluator is often determined by the intention to compare it with certain other evaluators. Subtle changes in this formulation to ease a comparison may raise questions about whether the models really do say anything about real implementations. For example, the choice to black-hole both projections when one is looked up in \texttt{prsh} simplifies the comparison with \texttt{proj}.
So the way an implementation technique is modelled is informed by the expectation of how it compares with other techniques and ideas about how that comparison can be proved. But the hierarchy of evaluators (figures 10.1 and 13.1) was not drawn up then proved; it evolved as the subtle differences between the strategies and their effect on space usage came to light.

**The mechanised proofs are simple, but do they need to be so long and detailed?** Maybe the proofs should be more abstract in some way. Maybe there are better methods which summarise the essence of the space usage without needing to know everything about evaluators, garbage collectors and translations. Maybe.

Simulation proofs are necessarily large; it is essential to consider all possible evaluation rules, the effect of translation and to account for garbage collection. Combining all these elements through the PSR framework for translation allowed us to state a proof as just a translation relation and a collection of proof squares. The proofs are not lengthy when everything fits the framework.

When a simulation proof fails, the failure usually indicates what should be done to complete it manually. So this approach may not scale very well: the more complicated an evaluator becomes, the more difficult its comparison with other evaluators and the more likely we are to need extensions to the basic methods.

The leak proof by finding or searching for an example is very simple. It is slowed but not complicated by making evaluators larger. It is complicated by making evaluators behave very similarly in the main, with differences in space usage exposed only by a few combinations of rules with certain translations in certain contexts. The ease with which a leak witness can be found perhaps indicates the likelihood of a leak occurring in practice.

**Does this approach solve the problem of deciding whether there is a relative space leak?** Defining space leaks as ternary relations and classifying them further according to witness program behaviour turned out to be a fruitful way to describe and analyse leakiness. The generality of higher-order term graphs makes modelling easy and the example call-by-need evaluators are simple. Proofs can become complicated in such a general model where higher-order equation solving causes multiple solutions and the spaghetti structure of graphs allows the unintended. The proof frameworks are necessarily incomplete, but it was always possible to find a suitable extension; none of our example evaluator comparisons turned out to be proof-resistant. Hopefully the hours of sitting spent probing, quantifying and classifying leaks provide means which will aid the detection and repair of any cracks in new evaluators.

> "There’s a crack in the roof where the rain pours through.  
> That’s the place you always decide to sit.  
> Yeah I know I’m there for hours, the water running down my face.  
> Do you really think you keep it all that well hid? " [Tin95]
Appendix A

Using GraphKit

The software tool GraphKit and its extensions described through this dissertation are available for experimentation from http://www-users.cs.york.ac.uk/~ajb/kit.

GraphKit has the five main functions described here in addition to generators for many of the examples in the preceding chapters. GraphKit is a Haskell program. It uses a parser combinator library [SA99] which requires a language extension available in current Haskell implementations such as Hugs 98.

Graph evaluation

The function kit reads a file which defines a graph grammar, evaluator and graph as described in Chapter 4 and optionally a space-relation garbage collector as outlined in Section 8.2. Its output for a file sestudm defining the grammar of Example 4.1, a simple graph, the evaluator Scott and the garbage collector #gc is shown below. The components are checked and displayed and the graph is evaluated. The evaluation trace would be two steps longer without #gc which eliminates the unnecessary update markers introduced by the first and third (ScLookup) steps.

Main> kit "sestudm"

$$ROOT ::= \langle x, s \rangle$$

$$X ::= \text{LAM } x.X \quad \lambda x.X$$
$$\mid \text{APP } X \, x \quad X \, x$$
$$\mid \text{VAR } x \quad x$$
$$\mid \text{LET } x.X \, X' \quad \text{let } x = X \text{ in } X'$$
$$\mid \text{BOT} \quad \bot$$

$$S ::= \text{PSH } x \, s \quad x : s$$
$$\mid \text{UDM } x \, s \quad \#x \, s$$

$$\{ a \mapsto \text{let } y = E \text{ in } X \} a, s \longrightarrow \{ a \mapsto X[b/y], b \mapsto E[b/y] \} a, s \quad \text{(Let)}$$
$$\{ a \mapsto \lambda x.E, y \mapsto \bot, s \mapsto \#y \, t \} a, s \longrightarrow \{ a \mapsto \lambda x.E, y \mapsto \lambda x.E \} a, t \quad \text{(ScUpdate)}$$
$$\{ a \mapsto x, x \mapsto E \} a, s \longrightarrow \{ a \mapsto E, x \mapsto \bot, t \mapsto \#x \, s \} a, t \quad \text{(ScLookup)}$$
$$\{ a \mapsto \lambda y.E, s \mapsto x : t \} a, s \longrightarrow \{ a \mapsto E[x/y], t \} a, t \quad \text{(Reduce)}$$
$$\{ a \mapsto F \, x \} a, s \longrightarrow \{ a \mapsto F, t \mapsto x : s \} a, t \quad \text{(Push)}$$
\[(\{\}, \{a \mapsto \lambda x.E\}, \{b \mapsto \lambda y.F\})\]  
\(\xrightarrow{} (\{c.d\}, \{a.b, w \cdot z\}, \{e \mapsto E[w/x]\}, \{d \mapsto F[z/y]\})\)  
(Id.\lambda)

\[(\{\}, \{a \mapsto E \, x\}, \{b \mapsto F \, y\})\]  
\(\xrightarrow{} (\{c.d, x.y\}, \{a.b\}, \{c \mapsto E\}, \{d \mapsto F\})\)  
(Id.Apply)

\[(\{\}, \{a \mapsto x \mapsto y\}, \{b \mapsto y\})\]  
\(\xrightarrow{} (\{x.y\}, \{a.b\}, \{\}, \{\})\)  
(Id.Var)

\[(\{\}, \{a \mapsto \_\} \{b \mapsto \_\}\{\}, \{\})\]  
\(\xrightarrow{} (\{\}, \{a \mapsto \⊥\}, \{b \mapsto \⊥\})\)  
(Id.⊥)

\[(\{\}, \{s \mapsto \#u \mapsto v, t \mapsto \#y \mapsto v\}\)  
\(\xrightarrow{} (\{s.v\}, \{s.t\}, \{\}, \{\})\)  
(DeadUdm)

\{a \mapsto \_\}
\(\xrightarrow{} \{ba \mapsto \lambda f.f, g \mapsto h = ba \in h \text{ ba}g, \epsilon\)  
(Let)

\{e \mapsto \lambda h.h, a \mapsto e, e \mapsto e\}a, \epsilon  
(Let)

\{f \mapsto \lambda g.g, a \mapsto i, i \mapsto f, d \mapsto f : e\}a, d  
(Push)

\{e \mapsto \lambda h.h, a \mapsto e, e \mapsto e : e\}a, c  
(ScLookup)

\{f \mapsto \lambda g.g, d \mapsto \#j o, o \mapsto j : e, j \mapsto \_\}f, d  
(ScLookup)

\{e \mapsto \lambda h.h, a \mapsto \lambda k.k, c \mapsto e : e\}a, c  
(ScUpdate)

\{f \mapsto \lambda g.g, a \mapsto f\}a, \epsilon  
(Reduce)

\{e \mapsto \lambda h.h\}e, \epsilon  
(ScLookup)

space = 4, time = 8

No-leak proof by simulation with completion

The function sim implements the PSR simulation proof checker with completion described in chapters 9 and 10. Its inputs are two files defining evaluators as before and one defining a PSR translation. All three are checked and displayed; a PSR proof of the graphs is displayed if one can be found; then the simulation proof begins. Inferred PSR rules are presented to the user before they are added to the translation. The following example shows \(\text{\textit{test}} \xrightarrow{} \text{\textit{test}}\) wrt \(\equivX\). The output shows the translation \(\equivX\); the PSR proof trace showing that the example graph (the graph evaluated above) translates to itself; then the process of simulation square production and the inference of additional identity rules to translate the stack structures.
\[
\begin{aligned}
\{\{\}, \}, \{a \mapsto \lambda x. E, \{b \mapsto \lambda y. F\}\} & \quad (Id.\lambda) \\
\rightarrow \langle \{c \cdot d, \{a \cdot b, w \cdot z\}, \{c \mapsto E[w/x]\}, \{d \mapsto F[z/y]\}\rangle \\
\{\{\}, \}, \{a \mapsto E x\}, \{b \mapsto F y\}\rangle & \quad (Id.\text{Apply}) \\
\rightarrow \langle \{c \cdot d, x \cdot y\}, \{a \cdot b\}, \{c \mapsto E\}, \{d \mapsto F\}\rangle \\
\{\{\}, \}, \{a \mapsto x\}, \{b \mapsto y\}\rangle & \quad (Id.\text{Var}) \\
\rightarrow \langle \{x \cdot y\}, \{a \cdot b\}, \{\}, \{\}\rangle \\
\{\{\}, \}, \{a \mapsto \text{let } x = E \in B, \{b \mapsto \text{let } y = F \in D\}\} & \quad (Id.\text{let}) \\
\rightarrow \langle \{c \cdot d, e \cdot f, \{a \cdot b, w \cdot z\}, \{c \mapsto E[w/x], e \mapsto B[w/x]\}, \{d \mapsto F[z/y], f \mapsto D[z/y]\}\rangle
\end{aligned}
\]

\[
\begin{aligned}
\langle \{a \cdot a\}, \{\} \rangle, \\
\langle \{a \mapsto \text{let } id = \lambda x. x \text{ in } \text{let } s = id \text{ in } s \text{ id}\}, \{a \mapsto \text{let } id = \lambda x. x \text{ in } \text{let } s = id \text{ in } s \text{ id}\} \rangle \\
\langle \{b \cdot e, d \cdot g\}, \{c \cdot f\} \rangle, \\
\rightarrow \langle \{b \mapsto \lambda x. x, d \mapsto \text{let } s = c \text{ in } s \text{ c}\}, \{c \mapsto \lambda x. x, g \mapsto \text{let } s = f \text{ in } s \text{ f}\} \rangle & \quad (Id.\text{let}) \\
\langle \{d \cdot g, a \cdot i\}, \{c \cdot f, h \cdot j\} \rangle, \\
\rightarrow \langle \{d \mapsto \text{let } s = c \text{ in } s \text{ c, a } \mapsto h\}, \{g \mapsto \text{let } s = f \text{ in } s \text{ f, i } \mapsto j\} \rangle & \quad (Id.\lambda) \\
\langle \{d \cdot g\}, \{c \cdot f\} \rangle, \\
\langle \{d \mapsto \text{let } s = c \text{ in } s \text{ c}\}, \{g \mapsto \text{let } s = f \text{ in } s \text{ f}\} \rangle & \quad (Id.\text{Var}) \\
\langle \{a \cdot h, e \cdot j\}, \{c \cdot f, b \cdot i\} \rangle, \\
\rightarrow \langle \{a \mapsto c, e \mapsto b \text{ c}\}, \{h \mapsto f, j \mapsto i \text{ f}\} \rangle & \quad (Id.\text{let}) \\
\langle \{a \cdot h, d \cdot g\}, \{c \cdot f, b \cdot i\} \rangle, \\
\rightarrow \langle \{a \mapsto c, d \mapsto b\}, \{h \mapsto f, g \mapsto i\} \rangle & \quad (Id.\text{Apply}) \\
\langle \{d \cdot g\}, \{b \cdot i\} \rangle, \\
\rightarrow \langle \{d \mapsto b\}, \{\}, \{\} \rangle & \quad (Id.\text{Var}) \\
\langle \{g \mapsto i\} \rangle & \quad (Id.\text{Var}) \\
\langle \{\}, \{\} \rangle
\end{aligned}
\]

\[
\begin{aligned}
\langle \{g \cdot a, b \cdot s\}, \{\} \rangle, \\
\langle \{g \mapsto \text{let } h = A \text{ in } C\}, \{a \mapsto \text{let } y = E \text{ in } X\} \rangle & \quad (Id.\text{let}) \\
\rightarrow \langle \{b \cdot s, k \cdot n, m \cdot p\}, \{g \cdot a, l \cdot o\} \rangle \rightarrow \langle \{k \mapsto A[l/h], m \mapsto C[l/h]\}, \{n \mapsto E[o/y], p \mapsto X[o/y]\} \rangle
\end{aligned}
\]

\[
\begin{aligned}
\langle \{g \cdot a, b \cdot s\}, \{\} \rangle, \\
\langle \{g \mapsto C[a/h], a \mapsto A[a/h]\}, \{a \mapsto X[b/y], b \mapsto E[b/y]\} \rangle & \quad (\text{Ren, Ren, } H\text{Ren, } H\text{Ren}) \\
\rightarrow \langle \{k \cdot n, m \cdot p, b \cdot s\}, \{l \cdot o, g \cdot a\} \rangle \rightarrow \langle \{k \mapsto A[l/h], m \mapsto C[l/h]\}, \{n \mapsto E[o/y], p \mapsto X[o/y]\} \rangle
\end{aligned}
\]
\[
\begin{align*}
\langle \{c \cdot a, b \cdot s\}, \{\} \rangle, \\
\langle \{d \mapsto A, c \mapsto d\}, \\
\{a \mapsto x, x \mapsto E\} \rangle & \quad \text{(Id.Var)} \\
\mapsto & \quad \langle \{b \cdot s, d \cdot x\}, \{c \cdot a\}, \\
\{d \mapsto A\}, \\
\{x \mapsto E\} \rangle
\end{align*}
\]
Inferred PSR:

\[
\langle \{ \}, \{ \}, \{ a \mapsto f : b \}, \{ b \mapsto x : s \} \rangle \\
\to \langle \{ b \cdot s, f \cdot x \}, \{ a \cdot b \}, \{ \} \rangle \\
\quad \text{(New)}
\]

\[
\begin{aligned}
\langle \{ f \cdot a, n \cdot s \}, \{ \}, \{ n \mapsto d : e, f \mapsto \lambda g. A \}, \{ a \mapsto \lambda g. E, s \mapsto x : t \} \rangle & \quad \text{(New, Id.\lambda)} \\
\to & \quad \langle \{ e \cdot t, d \cdot x, j \cdot l \}, \{ f \cdot a, k \cdot m \}, \{ j \mapsto A[k/g], \{ l \mapsto E[m/y] \} \rangle \\
\to & \quad \langle \{ d \cdot x, j \cdot l, e \cdot t \}, \{ k \cdot m, f \cdot a \}, \{ j \mapsto A[k/g], \{ l \mapsto E[m/y] \} \rangle \\
\end{aligned}
\]

\[
\begin{aligned}
\langle \{ f \cdot a, e \cdot t \}, \{ \}, \{ f \mapsto A[d/g], \{ a \mapsto E[x/y] \} \rangle & \quad \text{(Ren, HRen)} \\
\to & \quad \langle \{ d \cdot x, j \cdot l, e \cdot t \}, \{ k \cdot m, f \cdot a \}, \{ j \mapsto A[k/g], \{ l \mapsto E[m/y] \} \rangle \\
\end{aligned}
\]

\[
\begin{aligned}
\langle \{ e \cdot a, b \cdot s \}, \{ \}, \{ e \mapsto A \cdot f \}, \{ a \mapsto F \cdot x \} \rangle & \quad \text{(Id.\ Apply)} \\
\to & \quad \langle \{ b \cdot s, i \cdot j, f \cdot x \}, \{ e \cdot a \}, \{ i \mapsto A \}, \{ j \mapsto F \} \rangle \\
\end{aligned}
\]

\[
\begin{aligned}
\langle \{ e \cdot a, a \cdot b \}, \{ \}, \{ e \mapsto A, a \mapsto f : b \}, \{ a \mapsto F, b \mapsto x : s \} \rangle & \quad \text{(Ren, New)} \\
\to & \quad \langle \{ i \cdot j, b \cdot s, f \cdot x \}, \{ e \cdot a \}, \{ i \mapsto A \}, \{ j \mapsto F \} \rangle \\
\end{aligned}
\]

### Checking leak witnesses

The function `witness` implements the leak-witness checker `witness`. Its arguments are two files containing evaluator definitions (with optional space-relation garbage collectors) and another file defining a PSR translation. The example below involves a six-rule sequence which shows that `sest` with the standard collector `gc` is leakier than the same evaluator with the `#gc` collector.

```
Main> witness "sest" "sestudm" "psridx"
  [
    "Push","ScLookup","ScUpdate","Reduce","Let","ScLookup"]
```

\[
\begin{aligned}
\{ a \mapsto g \cdot e, g \mapsto \lambda h. \text{let } i = g \cdot e \text{ in } i \} & a, e \\
\to^6 \text{sest} & \{ a \mapsto g \cdot e, d \mapsto \bot, e \mapsto \#d \cdot e, g \mapsto \lambda h. \text{let } i = g \cdot e \text{ in } i \} a, e \\
\quad \text{(Push, ScLookup, ScUpdate, Reduce, Let, ScLookup)} \\
\end{aligned}
\]

\[
\begin{aligned}
\{ a \mapsto g \cdot e, g \mapsto \lambda h. \text{let } i = g \cdot e \text{ in } i \} & a, e \\
\to^6 \text{sestudm} & \{ f \mapsto \lambda j. \text{let } l = f \cdot e \text{ in } l, a \mapsto f \cdot e \} a, e \cdots \\
\quad \text{(Push, ScLookup, ScUpdate, Reduce, Let, ScLookup)} \\
\end{aligned}
\]
Searching for leak candidates

The function candidates implements the leak-candidate search function candidates. The example below shows the result of a search of sequences of length five in \texttt{sest} using the \texttt{apxSeqs} tactic.

Main> candidates apxSeqs "sest" 5

$$\{a \mapsto g \in \epsilon, g \mapsto \lambda i.g \in \epsilon\}a, \epsilon$$

$$\longrightarrow^5_{\text{sest}} \{a \mapsto g \in \epsilon, g \mapsto \lambda i.g \in \epsilon, d \mapsto \epsilon : \epsilon\}a, d$$

(Push, Push, ScLookup, ScUpdate, Reduce)

<table>
<thead>
<tr>
<th>nrtSeqs</th>
<th>pIfSeqs</th>
<th>expSeqs</th>
<th>apxSeqs</th>
<th>Srs</th>
<th>Sfs</th>
<th>Xsfs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025</td>
<td>209</td>
<td>153</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Searching for leak witnesses

The function witnesses implements witnesses. The example below shows the first witness found by a search of sequences to length six using the \texttt{apxSeqs} tactic. The output shows witnesses as they are found and a final summary of how many sequences were considered as in the candidates search.

Main> witnesses apxSeqs "sest" "sestudm" "psridx" 6

$$\{a \mapsto \text{let } c = h \in \epsilon \text{ in } c, h \mapsto \lambda j.\text{let } c = h \in \epsilon \text{ in } c\}a, \epsilon$$

$$\longrightarrow^6_{\text{sest}} \{a \mapsto \text{let } c = h \in \epsilon \text{ in } c, h \mapsto \lambda j.\text{let } c = h \in \epsilon \text{ in } c, d \mapsto \bot, e \mapsto \#d \epsilon\}a, \epsilon$$

(Let, ScLookup, Push, ScLookup, ScUpdate, Reduce)

$$\implies^i$$

$$\{a \mapsto \text{let } c = h \in \epsilon \text{ in } c, h \mapsto \lambda j.\text{let } c = h \in \epsilon \text{ in } c\}a, \epsilon$$

$$\longrightarrow^6_{\text{sestudm}} \{u \mapsto \lambda f.\text{let } m = u \in m, a \mapsto \text{let } i = u \in i\}a, \epsilon \cdots$$

(Let, ScLookup, Push, ScLookup, ScUpdate, Reduce)

<table>
<thead>
<tr>
<th>nrtSeqs</th>
<th>pIfSeqs</th>
<th>expSeqs</th>
<th>apxSeqs</th>
<th>Srs</th>
<th>Sfs</th>
<th>Xsfs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1864</td>
<td>657</td>
<td>466</td>
<td>7</td>
<td>49</td>
<td>16</td>
<td>5</td>
</tr>
</tbody>
</table>
Appendix B

Relatively-No-Leak Proofs

This appendix gives the full details of the no-leak proofs for the call-by-need evaluator comparisons referred to in Chapter 9 and Chapter 10. All the proofs compare two evaluators with respect to the identity translation on expressions, $\equiv_{id}$. The completed translation needed for each proof is given along with the simulation proof squares and any propositions necessary to support the proof. Many of the squares are shared between several proofs, so they are all numbered and collected together in Section B.5, the proofs just list the squares needed with the safer-evaluator rule covered by each.

B.1 Fully automated proof checks

The following simulation proofs are checked automatically. The full translation is specified for each. Any previously-defined rules are abbreviated to their names.

**Proposition B.1 (No trimming does not repair black hole trimming)**

\[
\text{not} \models \text{bhtr} \quad \text{wrt} \quad \equiv_{id} \quad \Rightarrow
\]

\[
\equiv_{id} = (\text{start}, \equiv_{id} \cup \{(Id.\bot), (Id.:), (Id.\#), (Env\text{Ret})\})
\]

\[
\langle \{\}, \{\}, \{s \mapsto Env x u, \{t \mapsto \text{Ret} v\} \rangle \rightarrow \langle \{u \cdot v\}, \{s \cdot t\}, \{\}, \{\} \rangle \quad \text{Env\text{Ret}}
\]

**Proof**

Squares: RetReturn(B.1), Update(B.2), Lookup(B.3), RetReduce(B.4), Push(B.5), Let(9.10).

**Proposition B.2 (No (Push) black holing is no repair)**

\[
\text{noah} \models \text{lazy} \quad \text{wrt} \quad \equiv_{id} \quad \Rightarrow
\]

\[
\equiv_{id} = (\text{start}, \equiv_{id} \cup \{(Id.\bot), (Id.:), (Apply ::)\})
\]

\[
\langle \{\}, \{\}, \{s \mapsto a : u, a \mapsto F x, \{t \mapsto y : v\} \rangle \rightarrow \langle \{x \cdot y, u \cdot v\}, \{s \cdot t\}, \{\}, \{\} \rangle \quad \text{Apply ::}
\]

**Proof**

Squares: Update(B.2), Lookup(B.3), Let(9.10), Reduce(B.6), Push(B.7).

**Proposition B.3 (No stack is no repair)**

\[
\text{nost} \models \text{lazy} \quad \text{wrt} \quad \equiv_{id} \quad \Rightarrow
\]

\[
\equiv_{id} = (\text{start}, \equiv_{id} \cup \{(Id.\bot), (\# \# \bot), (\# \bot)\})
\]

\[
\langle \{\}, \{\}, \{a \mapsto \# b\}, \{s \mapsto \# y t, y \mapsto \bot\} \rangle \rightarrow \langle \{b \cdot t\}, \{a \cdot s, a \cdot y\}, \{\}, \{\} \rangle \quad (\# \# \bot)
\]

\[
\langle \{\}, \{\}, \{a \mapsto \# b\}, \{y \mapsto \bot\} \rangle \rightarrow \langle \{\}, \{a \cdot y\}, \{\}, \{\} \rangle \quad (\# \bot)
\]

**Proof**

Squares: Let(9.10), Reduce(9.9), Push(9.6), Update(B.8), Lookup(B.9).
B.2 Fully automated proof inference

The following simulation proofs are completed automatically. All comparisons begin with the translation \( \vdash_{\text{delay}} \); inferred translation rules and the final set of simulation squares are listed in each case.

**Proposition B.4 (No trimming does not repair black hole trimming)**

\[
\text{not} \iff \text{htr} \quad \text{wrt} \quad \vdash_{\text{delay}}
\]

**Proof**

Inferred translation: \( (\bot \# \bot \#) \) below, \( (\text{Id.} \bot) \), \( (\text{Id.} :) \) and \( (\text{EnvRet}) \) of Proposition B.1.

A less general stack translation than \( \vdash_{\text{tr}} \) in Proposition B.1.

\[
\langle \{\}, \{d \mapsto \bot, d \mapsto \#c : e \}, \{a \mapsto \bot, b \mapsto \#a : s\} \rangle
\rightarrow \langle \{e : s\}, \{c \cdot a, d \cdot b\}, \{\}, \{\}\rangle \\
(\bot \# \bot \#)
\]

Squares: \text{RetReturn}(B.1), \text{Update}(B.10), \text{Lookup}(B.11), \text{RetReduce}(B.4), \text{Push}(B.5), \text{Let}(9.10).

\[\square\]

**Proposition B.5 (No (Push) black holing is no repair)**

\[
\text{noah} \iff \text{lazy} \quad \text{wrt} \quad \vdash_{\text{delay}}
\]

**Proof**

Inferred translation: \( (\bot \# \bot \#) \) of Proposition B.4, \( (\text{Id.} \bot) \), \( (\text{Apply} :) \) of Proposition B.2. A less general stack translation than \( \vdash_{\text{noah}} \) in Proposition B.2.

Squares: \text{Update}(B.10), \text{Lookup}(B.11), \text{Let}(9.10), \text{Reduce}(B.6), \text{Push}(B.7).

\[\square\]

**Proposition B.6 (No stack is no repair)**

\[
\text{host} \iff \text{lazy} \quad \text{wrt} \quad \vdash_{\text{delay}}
\]

**Proof**

Inferred translation: \( (\text{Id.} :) \), \( (\#\bot) \) and \( (\# \bot) \) of Proposition B.3. The same translation as \( \vdash_{\text{host}} \) in Proposition B.3.

Squares: \text{Let}(9.10), \text{Reduce}(9.9), \text{Push}(9.6), \text{Update}(B.8), \text{Lookup}(B.9).

\[\square\]

B.3 Proofs using global reasoning

The translations used by these evaluator comparisons break the PSR or simulation proof framework in some way. So we supply extra information to complete them (see also Section 10.2).

**Proposition B.7 (No stack is no leak)**

\[
\text{host} \not\iff \text{lazy} \quad \text{wrt} \quad \vdash_{\text{delay}}
\]

**Proof**

This simulation cannot be checked automatically because the translation \( \vdash_{\text{host}} \) used to show \( \text{host} \iff \text{lazy} \) is not a PSR when it is inverted — the inversion of rule \( (\# \bot) \) fails the checks that guarantee a space bound. But actually, the situation covered by \( (\# \bot) \) never arises. Proposition B.8 argues that \text{GcTrace} still holds without this rule, assuming that all initial programs are in \( \vdash_{\text{delay}} \). Therefore the proof squares used for \( \text{host} \iff \text{lazy} \) in Proposition B.6 are sufficient here too.

\[\square\]
Proposition B.8 (A black hole has an update marker)

\( G \xrightarrow{idX} H \land H \rightarrow_{\text{nost}} H' \land \{a \mapsto \bot\} \subseteq \text{gc } H' \Rightarrow \exists s, t \cdot \{t \mapsto \# a \ s\} \subseteq \text{gc } H' \)

If an initial graph \( H \) evaluates to \( H' \) which contains a black-holed variable then there must be an update marker for that black-holed variable. Therefore the PSR rule \( (\# \bot) \) in \( \rightarrow_{\text{nost}} \) is never needed.

Proof

Initially there are no black-holed variables by the definition of \( \xrightarrow{idX} \). The rules \( (\text{Let}), (\text{Push}) \) and \( (\text{Reduce}) \) do not involve black-hole nodes and do not affect the reachability via the stack nodes of any existing ones. \( (\text{NosUpdate}) \) destroys a black hole and does not affect any others. \( (\text{NosLookup}) \) introduces a black hole reachable from a new update marker, leaving existing black holes unaffected. \( \square \)

Proposition B.9 (No black holing does not repair \( (\text{Lookup}) \) black holing)

\( \text{noabh} \not\equiv \text{nabh} \text{ wrt } \xrightarrow{nabh} \)

Proof

Squares: \( \text{Let}(9.10), \text{Reduce}(B.6), \text{Push}(B.7), \text{UpdateNbh}(B.18), \text{LookupNbh}(B.19). \)

Square \( (B.18) \) needs to be supported by Proposition B.10 because the usual definition of \( \Rightarrow \) does not cover this case. \( \square \)

Proposition B.10 (Square \( (B.18) \) is complete)

\[
\begin{pmatrix}
\{i \cdot k, d \cdot y, g \cdot t\}, \\
\{e \cdot a, j \cdot l\}, \\
\{d \mapsto B, i \mapsto A[j / f]\}, \\
\{y \mapsto X, k \mapsto E[l / x]\}
\end{pmatrix} \Rightarrow \begin{pmatrix}
\{g \cdot t, i \cdot k\}, \\
\{j \cdot l, e \cdot a, d \cdot y\}, \\
\{i \mapsto A[j / f]\}, \\
\{k \mapsto E[l / x]\}
\end{pmatrix}
\]

Proof

Extending the definition of node \( y \) in the left configuration using any of the six possibilities \( (\text{TransE}), (\text{Id.} \bot), (\text{Id. Var}), (\text{Id.} \lambda), (\text{Id. Apply}) \) or \( (\text{Id. Let}) \) yields a configuration which implies the right configuration by the usual definition. \( \square \)

### B.4 Proofs using non-lockstep reasoning

These evaluator comparisons require some non-lockstep reasoning (see Section 10.3). As before, we define the translation used for each proof and then give the squares and other supporting evidence required for the simulation proof.

Proposition B.11 (No black holing does not repair \( (\text{Push}) \) black holing)

\( \text{noabh} \not\equiv \text{nabh} \text{ wrt } \xrightarrow{nabh} \)

Proof

This proof is very similar to \( \text{nabh} \not\equiv \text{lazy} \) in Example 10.9 — another case of non-lockstep termination — but different squares are needed for the \( (\text{ReduceNah}) \) and \( (\text{PushNbh}) \) redexes.

Squares: \( \text{Let}(9.10), \text{ReduceNah}(B.16), \text{PushNah}(B.17), \text{Update}(B.12, B.13, B.14, B.15), \text{Lookup}(10.14, B.20, B.21, B.22). \) \( \square \)
Proposition B.12 (Shortcutting projections is no leak)

\[ \text{\textbf{proj}} \vdash \text{\textbf{prsh}} \text{ wrt } \vdash_{\text{id}X_{\pi}} \]

where

\[ \vdash_{\text{id}X_{\pi}} = \left( \begin{array}{c}
\text{start}_\pi, \{(\text{Id. }:)\}, (\text{Id.}\{\cdot\}), (\text{Id.}\lambda), (\text{Id.}\text{App}), (\text{Id.}\text{let}), \\
(\text{Id.}\text{case}), (\text{Id.}\text{if}), (\text{Id.}\text{LIST}), (\text{Id.}\text{[]}), (\text{Id.}:), (\text{Id.}\text{ALN}), (\text{Id.}\text{ALC}), \\
(\text{Id.}\text{case}), (\text{Id.}\text{if}), (\text{Id.}\text{BOOL}), (\text{Id.}\text{True}), (\text{Id.}\text{False}), \\
(\pi \text{hth}), (\pi \text{hht}), (\pi \text{ght}), (\pi \text{t}: t), (\pi \text{h}: h), (\pi \text{h}: h) \end{array} \right) \]

The translation \( \vdash_{\text{id}X_{\pi}} \) is an identity on most function symbols (; is overloaded, being used for cons and pushed arguments in these grammars), except those involving projections which are handled by the new rules below and the pushed projections and update markers which are handled by the \( \text{start}_\pi \) rule below. The expressions included in \( \vdash_{\text{id}X_{\pi}} \) by identity rules are the \( \lambda \)-calculus with let and let-cons pattern bindings plus boolean constructors, list constructors, if-expressions, list case-expressions (and their alternatives). Stack terms for pushed arguments and pushed if and case alternatives are also translated by identity rules.

\[
\begin{align*}
\langle \{ \}, \{ \} \rangle, \{ a \mapsto \pi_h c, b \mapsto \pi_t c \}, \{ a' \mapsto \pi_h c' @ (: b'), b' \mapsto \pi_t c' @ (a' :) \} & \rightarrow \langle c' c, \{ a \cdot a', b \cdot b' \}, \{ \}, \{ \} \rangle & \text{(hht)} \\
\langle \{ \}, \{ \} \rangle, \{ a \mapsto \pi_h c \}, \{ a' \mapsto \pi_h c' @ (: b'), b' \mapsto \pi_t c' @ (a' :) \} & \rightarrow \langle c' c, \{ a \cdot a', \}, \{ \} \rangle & \text{(hht)} \\
\langle \{ \}, \{ \} \rangle, \{ b \mapsto \pi_t c \}, \{ a' \mapsto \pi_h c' @ (: b'), b' \mapsto \pi_t c' @ (a' :) \} & \rightarrow \langle c' c, \{ b \cdot b', \}, \{ \} \rangle & \text{(ht)} \\
\langle \{ \}, \{ \} \rangle, \{ a \mapsto \pi_h c, c \mapsto h : t \}, \{ a' \mapsto h', c' \mapsto h' : t' \} & \rightarrow \langle h \cdot h', t \cdot t', \{ a \cdot a', c \cdot c' \}, \{ \}, \{ \} \rangle & \text{(h : h)} \\
\langle \{ \}, \{ \} \rangle, \{ a \mapsto \pi_h c, c \mapsto h : t \}, \{ a' \mapsto h' \} & \rightarrow \langle h \cdot h', \{ a \cdot a' \}, \{ \}, \{ \} \rangle & \text{(h : h)} \\
\langle \{ \}, \{} \rangle, \{ b \mapsto \pi_t c, c \mapsto h : t \}, \{ b' \mapsto t', c' \mapsto h' : t' \} & \rightarrow \langle h \cdot h', t \cdot t', \{ b \cdot b', c \cdot c' \}, \{ \}, \{ \} \rangle & \text{(t : t)}
\end{align*}
\]

\[
\begin{align*}
\text{start}_\pi & \left( \begin{array}{c}
s_i \mapsto \# a_i s_{i-1}, a_i \mapsto \bot | i \in U \\
s_i \mapsto \# \pi_h a_i s_{i-1}, a_i \mapsto \bot, b_i \mapsto \pi_t a_p | i \in H \\
s_i \mapsto \# \pi_h a_i s_{i-1}, a_i \mapsto \bot | i \in H' \\
s_i \mapsto \# \pi_t a_i s_{i-1}, b_i \mapsto \pi_h a_p, a_i \mapsto \bot | i \in T \\
s_i \mapsto \# \pi_t a_i s_{i-1}, a_i \mapsto \bot | i \in T'
\end{array} \right) \\
G_{a_m, s_n} & \left( \begin{array}{c}
s'_i \mapsto \# a'_i s'_{i-1}, a'_i \mapsto \bot | i \in U \\
s'_i \mapsto \# \pi_h a'_i (a'_i : b'_i) s_{i-1}, a_i \mapsto \bot, b_i \mapsto \bot | i \in H H' \\
s'_i \mapsto \# \pi_h a'_i (a'_i : b'_i) s_{i-1}, b_i \mapsto \bot, a_i \mapsto \bot | i \in T T' \\
H_{a'_m, s'_n} & \right)
\end{align*}
\]

\[
= (\{ a_m, a'_m \} \{ s_i, s'_i | 1 \leq i \leq m \}, \{ a_i, a'_i, s_i, s'_i | i \in U H H' T T' \} \{ b_i, b'_i | i \in H T \}, G, H)
\]

where

\[ n \in \mathbb{N}, U H H' T T' \subseteq \{1, \ldots, n\}, m = n + 1, \]

\[ s_0 = e, p_i = \min(\{m\} U H H' T T' - \{1, \ldots, i\}) \]

\( \text{start}_\pi \) maps the stack update markers for variables and projections in an \textbf{proj} graph
to the equivalent \(\text{prsh}\) graph. The simulation proof needs to use the \(\text{(Share)}\) axiom in the squares for \(\text{(Update)}, \text{(LUpdate)}\) and \(\text{(UpdateCtr)}\) redexes. The translation rules involving projections are not sharable so we need Proposition B.13 to validate these steps.

**Proof**

See Example 10.12. \(\square\)

**Proposition B.13 \((\text{id}_{\text{proj}}X)\) is sharable**

\[\text{noInternal}_\pi(G) \land (G \to_{\text{proj}} G' \lor G \to_{\text{prsh}} G') \Rightarrow \text{noInternal}_\pi(G')\]

where \(\text{noInternal}_\pi(G) = \{\text{symbols } \Sigma | (a \mapsto F \pi.N) \in G \} \downarrow \{\pi_h, \pi_t, \pi_h@(): \pi_t@():\}\)

Both \(\text{proj}\) and \(\text{prsh}\) preserve the invariant that projection function symbols only occur outermost in graph nodes. Therefore the \(\text{(Share)}\) axiom may be applied to sub-terms during a PSR proof because the \(\text{id}_{\text{proj}}X\) rules which do not involve projections are sharable.

**Proof**

Follows from the definitions of \(\text{proj}\) and \(\text{prsh}\). \(\square\)

### B.5 Simulation proof squares

The squares for the proofs discussed in chapters 9 and 10 and this appendix are listed below. They are not grouped by evaluator comparison as many occur in several proofs.

Squares (B.1 – B.22) are all generated automatically by the GraphKit `sim` function. Squares (B.23 – B.26) use the enriched PSR rule notation (see also Example 10.6), so they are produced by hand. Similarly, squares (B.27 – B.44) are generated by hand because they involve translations which combine PSR rules with an enriched `start` rule.

\[
\begin{align*}
\left\{e\cdot a,b\cdot s\right\} & \downarrow, & \left\{e \mapsto \lambda f.A, b \mapsto \text{Env } d : g\right\} & \to & \left\{i\cdot k,g\cdot t\right\} & \downarrow, & \left\{e\cdot a,j\cdot l\right\}, \\
\left\{a \mapsto \lambda x.E, s \mapsto \text{Ret } t\right\} & \to & \left\{i \mapsto A[j/f]\right\} & \to & \left\{k \mapsto E[l/x]\right\}
\end{align*}
\]

\[
\text{(EnvReturn)} \downarrow \text{ (RetReturn)}
\]

\[
\begin{align*}
\left\{e\cdot a,g\cdot t\right\} & \downarrow, & \left\{e \mapsto \lambda f.A\right\} & \to & \left\{g\cdot t,i\cdot k\right\} & \downarrow, & \left\{j\cdot l,c\cdot a\right\}, \\
\left\{a \mapsto \lambda x.E\right\} & \to & \left\{i \mapsto A[j/f]\right\} & \to & \left\{k \mapsto E[l/x]\right\}
\end{align*}
\]

\[
\text{(Id},\lambda, \text{EnvRet} \right) \downarrow \text{ (Id},\lambda, \text{Ren} \right)
\]

\[
\begin{align*}
\left\{e\cdot a,b\cdot s\right\} & \downarrow, & \left\{e \mapsto \lambda f.A,b \mapsto \#d g, d \mapsto \bot\right\} & \to & \left\{i\cdot k,g\cdot t\right\} & \downarrow, & \left\{e\cdot a,j\cdot l,d\cdot y\right\}, \\
\left\{a \mapsto \lambda x.E, y \mapsto \bot, s \mapsto \#y t\right\} & \to & \left\{i \mapsto A[j/f]\right\} & \to & \left\{k \mapsto E[l/x]\right\}
\end{align*}
\]

\[
\text{(Update)} \downarrow \text{ (Update)}
\]

\[
\begin{align*}
\left\{d\cdot y,g\cdot t\right\} & \downarrow, & \left\{d \mapsto \lambda f.A,e \mapsto \lambda f.A\right\} & \to & \left\{g\cdot t,i\cdot k\right\} & \downarrow, & \left\{j\cdot l,e\cdot a,d\cdot y\right\}, \\
\left\{y \mapsto \lambda x.E,a \mapsto \lambda x.E\right\} & \to & \left\{i \mapsto A[j/f]\right\} & \to & \left\{k \mapsto E[l/x]\right\}
\end{align*}
\]

\[
\text{(Share},\text{Id},\lambda, \text{Ren}, \text{HRen}) \downarrow
\]

\[
\begin{align*}
\left\{d\cdot y,g\cdot t\right\} & \downarrow, & \left\{d \mapsto \lambda f.A,e \mapsto \lambda f.A\right\} & \to & \left\{g\cdot t,i\cdot k\right\} & \downarrow, & \left\{j\cdot l,e\cdot a,d\cdot y\right\}, \\
\left\{y \mapsto \lambda x.E,a \mapsto \lambda x.E\right\} & \to & \left\{i \mapsto A[j/f]\right\} & \to & \left\{k \mapsto E[l/x]\right\}
\end{align*}
\]

\[
\text{(Share},\text{Id},\lambda,\text{Id},\#), \text{Id},\bot) \downarrow \text{ (Share},\text{Id},\lambda,\text{Ren},\text{HRen}) \downarrow
\]

\[
\begin{align*}
\left\{d\cdot y,g\cdot t\right\} & \downarrow, & \left\{d \mapsto \lambda f.A,e \mapsto \lambda f.A\right\} & \to & \left\{g\cdot t,i\cdot k\right\} & \downarrow, & \left\{j\cdot l,e\cdot a,d\cdot y\right\}, \\
\left\{y \mapsto \lambda x.E,a \mapsto \lambda x.E\right\} & \to & \left\{i \mapsto A[j/f]\right\} & \to & \left\{k \mapsto E[l/x]\right\}
\end{align*}
\]

\[
\text{(Id},\#), \text{Id},\bot) \downarrow \text{ (Share},\text{Id},\lambda,\text{Ren},\text{HRen}) \downarrow
\]

\[
\begin{align*}
\left\{d\cdot y,g\cdot t\right\} & \downarrow, & \left\{d \mapsto \lambda f.A,e \mapsto \lambda f.A\right\} & \to & \left\{g\cdot t,i\cdot k\right\} & \downarrow, & \left\{j\cdot l,e\cdot a,d\cdot y\right\}, \\
\left\{y \mapsto \lambda x.E,a \mapsto \lambda x.E\right\} & \to & \left\{i \mapsto A[j/f]\right\} & \to & \left\{k \mapsto E[l/x]\right\}
\end{align*}
\]
\[
\begin{align*}
\langle \{c \cdot a, b \cdot s \}, \{\} \rangle \\
\langle \{c \mapsto d\}, \\
\{a \mapsto x\} \rangle \\
\stackrel{(\text{Id. Var})}{\Rightarrow} \\
\langle \{b \cdot s, d \cdot x\}, \{c \cdot a\} \rangle \\
\langle \{\} \rangle \\
\langle \{\} \rangle
\end{align*}
\]

(\text{Lookup}) \downarrow (\text{Lookup})
\[
\begin{align*}
\langle \{d \cdot x, a \cdot b\}, \{\} \rangle \\
\langle \{c \mapsto \perp, a \mapsto \#c b\} \rangle \\
\langle \{a \mapsto \perp, b \mapsto \#a s\} \rangle \\
\stackrel{(\text{Id. \perp, \text{Id. \#})}}{\Rightarrow} \\
\langle \{d \cdot x, b \cdot s\}, \{c \cdot a\} \rangle \\
\langle \{\} \rangle \\
\langle \{\} \rangle
\end{align*}
\]

(\text{EnvReduce}) \downarrow (\text{RetReduce})
\[
\begin{align*}
\langle \{e \cdot a, b \cdot s\}, \{\} \rangle \\
\langle \{e \mapsto \lambda f. A, b \mapsto d : g\} \rangle \\
\langle \{a \mapsto \lambda x. E, s \mapsto y : t\} \rangle \\
\stackrel{(\text{Id. \lambda, \text{Id. :})}}{\Rightarrow} \\
\langle \{i \cdot k, d \cdot y, g \cdot t, \{e \cdot a, j \cdot l\}\} \rangle \\
\langle \{i \mapsto A[j/f]\} \rangle \\
\langle \{k \mapsto E[l/x]\} \rangle
\end{align*}
\]

(\text{EnwRet})
\[
\begin{align*}
\langle \{\} \rangle \\
\langle \{\} \rangle
\end{align*}
\]

\[
\begin{align*}
\langle \{e \cdot a, b \cdot s\}, \{\} \rangle \\
\langle \{e \mapsto A[f/x]\} \rangle \\
\langle \{a \mapsto E[y/x], s \mapsto \text{Ret : } t\} \rangle \\
\stackrel{(\text{Ren, HRen})}{\Rightarrow} \\
\langle \{d \cdot y, i \cdot k, g \cdot t, \{j \cdot l, e \cdot a\}\} \rangle \\
\langle \{i \mapsto A[j/f]\} \rangle \\
\langle \{k \mapsto E[l/x]\} \rangle
\end{align*}
\]

(\text{Push}) \downarrow (\text{Push})
\[
\begin{align*}
\langle \{e \cdot a, b \cdot s\}, \{\} \rangle \\
\langle \{e \mapsto A[f]\} \rangle \\
\langle \{a \mapsto F[x]\} \rangle \\
\stackrel{(\text{Ren, \text{Id. :})}}{\Rightarrow} \\
\langle \{i \cdot j, f \cdot x, b \cdot s\}, \{e \cdot a\} \rangle \\
\langle \{i \mapsto A\} \rangle \\
\langle \{j \mapsto F\} \rangle
\end{align*}
\]

\[
\begin{align*}
\langle \{g \cdot a, o \cdot s\}, \{\} \rangle \\
\langle \{o \mapsto d : e, d \mapsto A[f], g \mapsto \lambda x. B\} \rangle \\
\langle \{a \mapsto \lambda y. E, s \mapsto x : t\} \rangle \\
\stackrel{(\text{Apply ::, \text{Id. \lambda})}}{\Rightarrow} \\
\langle \{f \cdot x, e \cdot t, k \cdot m, \{g \cdot a, l \cdot n\}\} \rangle \\
\langle \{k \mapsto B[l/h]\} \rangle \\
\langle \{m \mapsto E[n/y]\} \rangle
\end{align*}
\]

(\text{ReduceNah}) \downarrow (\text{Reduce})
\[
\begin{align*}
\langle \{g \cdot a, e \cdot t\}, \{\} \rangle \\
\langle \{g \mapsto B[f/h], d \mapsto A[f]\} \rangle \\
\langle \{a \mapsto E[x/y]\} \rangle \\
\stackrel{(\text{Ren, \text{HRen})}}{\Rightarrow} \\
\langle \{f \cdot x, k \cdot m, e \cdot t, \{l \cdot n, g \cdot a\}\} \rangle \\
\langle \{k \mapsto B[l/h]\} \rangle \\
\langle \{m \mapsto E[n/y]\} \rangle
\end{align*}
\]

(\text{B.6})
\[
\begin{align*}
\langle \{e \cdot a, b \cdot s\}, \{\} \rangle & \quad \text{(Id.Apply)} & \langle \{b \cdot s, i \cdot j, f \cdot x\}, \{e \cdot a\} \rangle \\
\langle e \mapsto A f \rangle, & \quad \langle \{i \mapsto A\}, \{j \mapsto F\} \rangle \\
\langle a \mapsto F x \rangle & \quad \text{\rightarrow} \\
\end{align*}
\]

\[
\text{(PushNah)} \downarrow \text{(Push)} \\
\begin{align*}
\langle e \cdot a, c \cdot b\}, \{\} \rangle, & \quad \text{(Ren,} \quad \langle \{i \mapsto A \}, \{j \mapsto F\} \rangle \\
\langle e \mapsto A, a \mapsto A f, c \mapsto a \mapsto b\rangle, & \quad \text{Apply :)} \\
\langle a \mapsto F, b \mapsto x \mapsto s \rangle & \quad \text{\rightarrow} \\
\end{align*}
\]

\[
\begin{align*}
\langle \{e \cdot a, b \cdot s\}, \{\} \rangle, & \quad \text{(Id.\lambda)} & \langle \{i \cdot k, d \cdot t\}, \{e \cdot a, j \cdot l, b \cdot y\} \rangle \\
\langle e \mapsto \lambda f.A, b \mapsto \#d\rangle, & \quad \#\#\# & \langle \{i \mapsto A[j/f]\}, \{k \mapsto E[l/x]\} \rangle \\
\langle a \mapsto \lambda x.E, y \mapsto \bot, s \mapsto \#y t\rangle & \quad \text{\rightarrow} \\
\end{align*}
\]

\[
\text{(NosUpdate)} \downarrow \text{(Update)} \\
\begin{align*}
\langle b \cdot y, d \cdot t\rangle, \{\} \rangle, & \quad \text{(Share, Id.\lambda,} \quad \langle \{d \cdot t, i \cdot k\}, \{j \cdot l, e \cdot a, b \cdot y\} \rangle \\
\langle b \mapsto \lambda f.A, e \mapsto \lambda f.A\rangle, & \quad \text{Ren, HRen,} \\
\langle y \mapsto \lambda x.E, a \mapsto \lambda x.E\rangle & \quad \text{\rightarrow} \\
\end{align*}
\]

\[
\begin{align*}
\langle \{c \cdot a, b \cdot s\}, \{\} \rangle, & \quad \text{(Id.\Var)} & \langle \{b \cdot s, d \cdot x\}, \{c \cdot a\} \rangle \\
\langle c \mapsto d\rangle, & \quad \langle \{\}, \{\} \rangle \\
\langle a \mapsto x\rangle & \quad \text{\rightarrow} \\
\end{align*}
\]

\[
\text{(NosLookup)} \downarrow \text{(Lookup)} \\
\begin{align*}
\langle \{d \cdot x, c \cdot b\}, \{\} \rangle, & \quad \text{(\#\#\#)} \\
\langle c \mapsto \#b\rangle, & \quad \langle \{d \cdot x, b \cdot s\}, \{c \cdot a\} \rangle \\
\langle a \mapsto \bot, b \mapsto \#a s\rangle & \quad \text{\rightarrow} \\
\end{align*}
\]

\[
\text{(Update)} \downarrow \text{(Update)} \\
\begin{align*}
\langle \{d \mapsto \bot, e \mapsto \#d f, g \mapsto \lambda h.A\}, \{\} \rangle, & \quad \text{(\bot\#\#\#,} \quad \langle \{f \cdot t, k \cdot m\}, \{d \cdot y, g \cdot a, l \cdot n\} \rangle \\
\langle a \mapsto \lambda x.E, y \mapsto \bot, s \mapsto \#y t\rangle & \quad \text{Id.\lambda,} \\
\end{align*}
\]

\[
\begin{align*}
\langle \{d \mapsto \bot, e \mapsto \#d f, g \mapsto \lambda h.A\}, \{\} \rangle & \quad \langle \{k \mapsto A[l/h]\}, \{m \mapsto E[n/x]\} \rangle \\
\langle d \cdot y, f \cdot t\}, \{\} \rangle, & \quad \text{(Share, Id.\lambda,} \\
\langle \{d \mapsto \lambda h.A, g \mapsto \lambda h.A\}, \{\} \rangle & \quad \text{Ren, HRen,} \\
\langle y \mapsto \lambda x.E, a \mapsto \lambda x.E\rangle & \quad \text{\rightarrow} \\
\end{align*}
\]

\[
\begin{align*}
\langle \{d \mapsto \bot, e \mapsto \#d f, g \mapsto \lambda h.A\}, \{\} \rangle & \quad \langle \{f \cdot t, k \cdot m\}, \{d \cdot y, g \cdot a, l \cdot n\} \rangle \\
\langle d \cdot y, f \cdot t\}, \{\} \rangle, & \quad \text{(Share, Id.\lambda,} \\
\langle \{d \mapsto \lambda h.A, g \mapsto \lambda h.A\}, \{\} \rangle & \quad \text{Ren, HRen,} \\
\langle y \mapsto \lambda x.E, a \mapsto \lambda x.E\rangle & \quad \text{\rightarrow} \\
\end{align*}
\]
\[
\begin{align*}
\langle \{\cdot a, b \cdot s\}, \{\cdot\}, \rangle & \quad \text{\{Id.Var\}} & \langle \{b \cdot s, d \cdot x\}, \{\cdot a\}, \rangle \\
\langle \{c \mapsto d\}, \{\cdot\}, \rangle & \quad \to & \langle \{\cdot\}, \{\cdot\} \rangle \\
\langle \{d \cdot x, a \mapsto b\}, \{\cdot\} \rangle & \quad \to & \langle \{d \cdot x, b \cdot s\}, \{\cdot a\}, \rangle \\
\text{(Lookup) \downarrow (Lookup)} & & \text{(B.11)}
\end{align*}
\]

\[
\begin{align*}
\langle \{\cdot e \cdot a, b \cdot s\}, \{\cdot\}, \rangle & \quad (\lambda \perp, \text{Id.} \lambda, \text{Id.} \#) & \langle \{\cdot i \cdot k, h \cdot t\}, \{\cdot m \cdot y, e \cdot a, j \cdot l\}, \rangle \\
\langle \{m \mapsto \lambda d \cdot A\}, \{\cdot\}, \rangle & \quad \text{Id.} \# & \langle \{i \mapsto B[j/f]\}, \rangle \\
\langle e \mapsto \lambda f \cdot B, b \mapsto \#m h\rangle & \quad \to & \langle k \mapsto E[l/x]\rangle \\
\langle a \mapsto \lambda x \cdot E, y \mapsto \perp, s \mapsto \#y t\rangle & \quad \to & \langle \{\cdot\}, \{\cdot\} \rangle \\
\text{(UpdateNb) \downarrow (Update)} & & \text{(B.12)}
\end{align*}
\]

\[
\begin{align*}
\langle \{e \cdot a, b \cdot s\}, \{\cdot\}, \rangle & \quad (\text{Apply} \perp, \text{Id.} \lambda, \text{Id.} \#) & \langle \{i \cdot k, h \cdot t\}, \{\cdot m \cdot y, e \cdot a, j \cdot l\}, \rangle \\
\langle \{m \mapsto A d\}, \{\cdot\}, \rangle & \quad \text{Id.} \# & \langle \{i \mapsto B[j/f]\}, \rangle \\
\langle e \mapsto \lambda f \cdot B, b \mapsto \#m h\rangle & \quad \to & \langle k \mapsto E[l/x]\rangle \\
\langle a \mapsto \lambda x \cdot E, y \mapsto \perp, s \mapsto \#y t\rangle & \quad \to & \langle \{\cdot\}, \{\cdot\} \rangle \\
\text{(UpdateNb) \downarrow (Update)} & & \text{(B.13)}
\end{align*}
\]

\[
\begin{align*}
\langle \{e \cdot a, b \cdot s\}, \{\cdot\}, \rangle & \quad (\text{let} \perp, \text{Id.} \lambda, \text{Id.} \#) & \langle \{i \cdot k, h \cdot t\}, \{\cdot m \cdot y, e \cdot a, j \cdot l\}, \rangle \\
\langle \{m \mapsto \text{let } d = \text{Ain} \cdot C\}, \{\cdot\}, \rangle & \quad \text{Id.} \# & \langle \{i \mapsto B[j/f]\}, \rangle \\
\langle e \mapsto \lambda f \cdot B, b \mapsto \#m h\rangle & \quad \to & \langle k \mapsto E[l/x]\rangle \\
\langle a \mapsto \lambda x \cdot E, y \mapsto \perp, s \mapsto \#y t\rangle & \quad \to & \langle \{\cdot\}, \{\cdot\} \rangle \\
\text{(UpdateNb) \downarrow (Update)} & & \text{(B.14)}
\end{align*}
\]
\[
\begin{align*}
\{b_j, b_j', s \cdot s', \}, \quad & \{e, e', t \cdot t', \} \\
\{b \mapsto I a, a \mapsto \lambda x. E, \} \quad & \{b \mapsto I a, a \mapsto \lambda x. E, \} \\
\{y \mapsto \perp, s \mapsto \#y \, t, \} \quad & \{y \mapsto \perp, s \mapsto \#y \, t, \} \\
\{c \mapsto \lambda y. E, s \mapsto x : t, \} \quad & \{c \mapsto \lambda y. E, s \mapsto x : t, \} \\
\{c' \mapsto \lambda y'. E', \} \quad & \{c' \mapsto \lambda y'. E', \} \\
\{c' \mapsto \lambda y'. E', \} \quad & \{c' \mapsto \lambda y'. E', \} \\
\{c' \mapsto \lambda y'. E', \} \quad & \{c' \mapsto \lambda y'. E', \} \\
\{c' \mapsto \lambda y'. E', \} \quad & \{c' \mapsto \lambda y'. E', \} \\
\{c' \mapsto \lambda y'. E', \} \quad & \{c' \mapsto \lambda y'. E', \} \\
\{c' \mapsto \lambda y'. E', \} \quad & \{c' \mapsto \lambda y'. E', \} \\
\{c' \mapsto \lambda y'. E', \} \quad & \{c' \mapsto \lambda y'. E', \}
\end{align*}
\]
\[
\langle \{a' \cdot b', \overline{a} \cdot b' \}, \{ a_i' \cdot a_i' \}, \\
\{ a \mapsto \text{let } x = E \text{ in } B \}, \\
\{ a' \mapsto \text{let } x' = E' \text{ in } B' \} \rangle \\
\text{(Let)} \downarrow \text{(Let)} \\
\langle \{ a' \cdot b', \overline{a} \cdot b' \}, \{ a_i' \cdot a_i' \}, \\
\{ a \mapsto B[y/x], y \mapsto E[y/x] \}, \\
\{ a' \mapsto B'[y'/x'], y' \mapsto E'[y'/x'] \} \rangle \\
\text{(HRen,} \text{ HRen,} \\
\text{Ren, Ren)} \\
\langle e \mapsto E[z/x], b \mapsto B[z/x], \\
e' \mapsto E'[z'/x'], b' \mapsto B'[z'/x'] \rangle
\]

\[
\langle \{ a' \cdot b', \overline{a} \cdot b' \}, \{ a_i' \cdot a_i' \}, \\
\{ a \mapsto E \cdot x \}, \\
\{ a' \mapsto E' \cdot x' \} \rangle \\
\text{(Id. Apply)} \downarrow \text{(Push) \downarrow (Push)} \\
\langle \{ a' \cdot b', \overline{a} \cdot b' \}, \{ a_i' \cdot a_i' \}, \\
\{ a \mapsto E \cdot x \cdot x', \overline{a} \cdot a_i' \}, \\
\{ a' \mapsto E' \cdot x' \} \rangle \\
\text{(Ren)} \\
\langle e \mapsto E \cdot x \cdot x', \overline{a} \cdot b' \rangle \\
\langle \{ a' \cdot a_i', \overline{a} \cdot a_i' \}, \\
\{ e \mapsto E \cdot x \cdot x', \overline{a} \cdot b' \} \rangle \\
\text{(HRen, Ren)} \\
\langle e \mapsto E[z/y], e' \mapsto E'[z'/y'] \rangle \\
\text{(Id.} \lambda) \\
\langle \{ a' \cdot b', \overline{a} \cdot b' \}, \{ a_i' \cdot a_i' \}, \\
\{ a \mapsto \lambda y \cdot E \}, \\
\{ a' \mapsto \lambda y' \cdot E' \} \rangle \\
\text{(Reduce) \downarrow (Reduce)} \\
\langle e \mapsto E[x/y], e' \mapsto E'[x'/y'] \rangle \\
\text{(Share) \downarrow (Update)} \\
\langle \{ a' \cdot b', \overline{a} \cdot b' \}, \{ a_i' \cdot a_i' \}, \\
\{ a \mapsto \lambda x \cdot E, y \mapsto \lambda x \cdot E \}, \\
\{ a' \mapsto \lambda x' \cdot E', y' \mapsto \lambda x' \cdot E' \} \rangle \\
\text{(Share)} \\
\langle a \mapsto \lambda x \cdot E \}, \\
\{ a' \mapsto \lambda x' \cdot E' \} \rangle \\
\text{(Share) \downarrow (Update)} \\
\langle \{ a' \cdot b', \overline{a} \cdot b' \}, \{ a_i' \cdot a_i' \}, \\
\{ a \mapsto \lambda x \cdot E \}, \\
\{ a' \mapsto \lambda x' \cdot E' \} \rangle
\[
\begin{align*}
&\langle a \cdot a', b_i \cdot b'_i, \{y \cdot y', a_i \cdot a'_i\}, \{a \mapsto \lambda x. E, \{a' \mapsto \lambda x'. E'\}\} \rangle \quad (Id, \lambda) \quad \Rightarrow \quad \langle \{c \cdot c', b_i \cdot b'_i\}, \{a \cdot a', z \cdot z', y \cdot y', a_i \cdot a'_i\}, \{e \mapsto E[z/x], \{c' \mapsto E'[z'/x']\}\} \rangle \\
&(Ret \text{Return}^*, \text{Update}) \downarrow \text{(Update)} \\
&\langle y \cdot y', b_i \cdot b'_i, \{a_i \cdot a'_i\}, \{a \mapsto \lambda x. E, y \mapsto \lambda x. E\}, \{a' \mapsto \lambda x'. E', y' \mapsto \lambda x'. E'\}\rangle \quad (Share, \text{Id}, \text{Ren}) \quad \Rightarrow \quad \langle \{c \cdot c', b_i \cdot b'_i\}, \{a \cdot a', z \cdot z', y \cdot y', a_i \cdot a'_i\}, \{e \mapsto E[z/x], \{c' \mapsto E'[z'/x']\}\} \rangle
\end{align*}
\]

\[
\begin{align*}
&\langle \{a \mapsto x\}, \{a' \mapsto x'\}\rangle \quad (Id, \text{Var}) \quad \Rightarrow \quad \langle x \cdot x', b_i \cdot b'_i, \{a \cdot a', a_i \cdot a'_i\}\rangle \\
&(\text{Lookup}) \downarrow \text{(Lookup)} \\
&\langle x \cdot x', b_i \cdot b'_i, \{a \cdot a', a_i \cdot a'_i\}\rangle \quad (\text{Id}) \quad \Rightarrow \quad \langle \{x \cdot x', b_i \cdot b'_i, \{a \cdot a', a_i \cdot a'_i\}\}\rangle
\end{align*}
\]

\[
\begin{align*}
&\langle a \cdot a', s_i \cdot s'_i, \{a_i \cdot a'_i\}, \{a \mapsto \text{let } (x : y) = E \text{ in } B\}, \{a' \mapsto \text{let } (x' : y') = E' \text{ in } B'\}\rangle \quad (\text{Id, let : }) \quad \Rightarrow \quad \langle \{c \cdot c', b \cdot b', s_i \cdot s'_i\}, \{a \cdot a', w \cdot w', z \cdot z', a_i \cdot a'_i\}, \{e \mapsto E[w/x, z/y]\}, \{b' \mapsto B'[w'/x', z'/y']\}\rangle \\
&(\text{LetCons}) \downarrow \text{(LetCons')} \\
&\langle a \cdot a', s \cdot s', \{a_i \cdot a'_i\}, \{a \mapsto B[b/x, c/y]\}, \{b \mapsto \pi_h d, d \mapsto E[b/x, c/y], c \mapsto \pi_t d\}, \{a' \mapsto B'[b' / x', c'/y']\}, \{b' \mapsto \pi_h d' @ (c'), c' \mapsto \pi_t d' @ (b' :), d' \mapsto E'[b'/x', c'/y']\}\rangle
\end{align*}
\]

\[
\begin{align*}
&\langle a \cdot a', s_i \cdot s'_i, \{a_i \cdot a'_i\}, \{a \mapsto \pi_h c\}, \{a' \mapsto \pi_h c' @ (: b'), b' \mapsto \pi_t c' @ (a' :)\}\rangle \quad (\pi hht) \quad \Rightarrow \quad \langle \{c \cdot c', s_i \cdot s'_i\}, \{a \cdot a', a_i \cdot a'_i\}\rangle \\
&(\text{PihLook}) \downarrow \text{(PihPush)} \\
&\langle \{c \cdot c', s_i \cdot s'_i\}, \{a \cdot a', a_i \cdot a'_i\}, \{\{\}, \{\}\}\rangle \quad (\text{PihPush}) \quad \Rightarrow \quad \langle \{c \cdot c', s_i \cdot s'_i\}, \{a \cdot a', a_i \cdot a'_i\}\rangle
\end{align*}
\]
\[ \{a \cdot a', s_i \cdot s'_i\}, \{a_i \cdot a'_i\}, \{a \mapsto \pi_h c, b \mapsto \pi_t c\}, \{a' \mapsto \pi_h c'\@ (b' \cdot), b' \mapsto \pi_h c'\@ (a')\} \]

\[ \{c \cdot c', s_i \cdot s'_i\}, \{a \cdot a', b \cdot b', a_i \cdot a'_i\}\]

\[ (\Pi h h t) \quad \rightarrow \quad \{c \cdot c', s_i \cdot s'_i\}, \{a \cdot a', b \cdot b', a_i \cdot a'_i\}\]

\[ (\Pi h h t) \quad \rightarrow \quad \{c \cdot c', s_i \cdot s'_i\}, \{a \cdot a', b \cdot b', a_i \cdot a'_i\}\]

\[ \{a \cdot a', s_i \cdot s'_i\}, \{a_i \cdot a'_i\}, \{a \mapsto \pi_t c\}, \{a' \mapsto \pi_t c'\@ (b' \cdot), b' \mapsto \pi_h c'\@ (a')\} \]

\[ \{c \cdot c', s_i \cdot s'_i\}, \{a \cdot a', a_i \cdot a'_i\}\]

\[ (\Pi h t) \quad \rightarrow \quad \{c \cdot c', s_i \cdot s'_i\}, \{a \cdot a', a_i \cdot a'_i\}\]

\[ \{a \cdot a', s_i \cdot s'_i\}, \{a_i \cdot a'_i\}, \{a \mapsto \pi_h c, b \mapsto \pi_h c\}, \{a' \mapsto \pi_h c'\@ (b' \cdot), b' \mapsto \pi_h c'\@ (a')\} \]

\[ \{c \cdot c', s_i \cdot s'_i\}, \{a \cdot a', b \cdot b', a_i \cdot a'_i\}\]

\[ (\Pi h h t) \quad \rightarrow \quad \{c \cdot c', s_i \cdot s'_i\}, \{a \cdot a', b \cdot b', a_i \cdot a'_i\}\]

\[ \{a \cdot a', s_i \cdot s'_i\}, \{a_i \cdot a'_i\}, \{a \mapsto \pi_t c, b \mapsto \pi_h c\}, \{a' \mapsto \pi_h c'\@ (b' \cdot), b' \mapsto \pi_h c'\@ (a')\} \]

\[ \{c \cdot c', s_i \cdot s'_i\}, \{a \cdot a', b \cdot b', a_i \cdot a'_i\}\]

\[ (\Pi h h t) \quad \rightarrow \quad \{c \cdot c', s_i \cdot s'_i\}, \{a \cdot a', b \cdot b', a_i \cdot a'_i\}\]

\[ \{a \cdot a', s_i \cdot s'_i\}, \{a_i \cdot a'_i\}, \{a \mapsto d \cdot e\}, \{a' \mapsto d' \cdot e'\} \]

\[ \{c \cdot c', s_i \cdot s'_i\}, \{a \cdot a', b \cdot b', a_i \cdot a'_i\}\]

\[ (\text{Id. var, } \pi_t : t) \quad \rightarrow \quad \{d \cdot d', e \cdot e', s_i \cdot s'_i\}, \{a \cdot a', b \cdot b', c \cdot c', a_i \cdot a'_i\}\]

\[ \{b \cdot b', s_i \cdot s'_i\}, \{a_i \cdot a'_i\}, \{b \mapsto d, c \mapsto \pi_t a, a \mapsto d \cdot e\}, \{b' \mapsto d', c' \mapsto e', a' \mapsto d' \cdot e'\} \]

\[ \{d \cdot d', e \cdot e', s_i \cdot s'_i\}, \{a \cdot a', b \cdot b', c \cdot c', a_i \cdot a'_i\}\]
\[
\begin{align*}
\langle \{a \cdot a', s_i \cdot s_i'\}, \{b \cdot b', a_i \cdot a_i'\} \rangle & \quad \text{(Id.)} & \langle \{d \cdot d', c \cdot c', s_i \cdot s_i'\}, \{a \cdot a', b \cdot b', a_i \cdot a_i'\} \rangle \\
\{a \mapsto d : e\}, & \quad \rightarrow & \{\}, \{\}
\end{align*}
\]

\[
\begin{align*}
\langle \{a \cdot a', s_i \cdot s_i'\}, \{b \cdot b', c \cdot c', a_i \cdot a_i'\} \rangle & \quad \text{(Id.)} & \langle \{d \cdot d', c \cdot c', s_i \cdot s_i'\}, \{a \cdot a', b \cdot b', c \cdot c', a_i \cdot a_i'\} \rangle \\
\{a \mapsto d : e\}, & \quad \rightarrow & \{\}, \{\}
\end{align*}
\]

\[
\begin{align*}
\langle \{a \cdot a', s_i \cdot s_i'\}, \{c \cdot c', a_i \cdot a_i'\} \rangle & \quad \text{(Id.)} & \langle \{d \cdot d', c \cdot c', s_i \cdot s_i'\}, \{a \cdot a', c \cdot c', a_i \cdot a_i'\} \rangle \\
\{a \mapsto d : e\}, & \quad \rightarrow & \{\}, \{\}
\end{align*}
\]

\[
\begin{align*}
\langle \{a \cdot a', s_i \cdot s_i'\}, \{c \cdot c', s_i \cdot s_i'\}, \{a_i \cdot a_i'\} \rangle & \quad \text{(Id.)} & \langle \{d \cdot d', c \cdot c', s_i \cdot s_i'\}, \{a \cdot a', c \cdot c', a_i \cdot a_i'\} \rangle \\
\{a \mapsto d : e\}, & \quad \rightarrow & \{\}, \{\}
\end{align*}
\]

\[
\begin{align*}
\langle \{a \cdot a', s_i \cdot s_i'\}, \{c \cdot c', a_i \cdot a_i'\} \rangle & \quad \text{(Id.)} & \langle \{d \cdot d', c \cdot c', s_i \cdot s_i'\}, \{a \cdot a', c \cdot c', a_i \cdot a_i'\} \rangle \\
\{a \mapsto d : e\}, & \quad \rightarrow & \{\}, \{\}
\end{align*}
\]

\[
\begin{align*}
\langle \{a \cdot a', s_i \cdot s_i'\}, \{a_i \cdot a_i'\} \rangle & \quad \text{(Id.)} & \langle \{d \cdot d', c \cdot c', s_i \cdot s_i'\}, \{a \cdot a', c \cdot c', a_i \cdot a_i'\} \rangle \\
\{a \mapsto d : e\}, & \quad \rightarrow & \{\}, \{\}
\end{align*}
\]

\[
\begin{align*}
\langle \{h \cdot h', t \cdot t', s_i \cdot s_i'\}, \{a \cdot a' a_i \cdot a_i'\} \rangle & \quad \text{(Id.)} & \langle \{h \cdot h', t \cdot t', s_i \cdot s_i'\}, \{a \cdot a', c \cdot c', a_i \cdot a_i'\} \rangle \\
\{c \mapsto h : t\}, & \quad \rightarrow & \{\}, \{\}
\end{align*}
\]
$$\left\{ a' \cdot a', s_i \cdot s_i', \{ a_i \cdot a_i' \} \right\} \xrightarrow{(\pi t : t :)} \left\{ a' \cdot a', a' \cdot a_i', s_i' \cdot s_i' \right\}$$

$$\langle a \mapsto \pi t c, c \mapsto h : t \rangle, \{ a' \mapsto t', c' \mapsto h' : t' \} \xrightarrow{} \langle h \cdot h', t \cdot t', s_i \cdot s_i' \rangle$$

$$\langle h \cdot h', t \cdot t', s_i \cdot s_i' \rangle \xrightarrow{} \langle a' \cdot a', c' \cdot c' \cdot a_i \cdot a_i' \rangle$$

$$\langle a' \cdot a', s_i \cdot s_i' \rangle, \{ a \cdot a' \cdot a \cdot a_i, a' \cdot a_i' \} \xrightarrow{(\pi t : t :)} \langle a' \cdot a', s_i \cdot s_i' \rangle$$

$$\langle a' \mapsto t', \{ c \mapsto h : t \}, \{ c' \mapsto h' : t' \} \rangle \xrightarrow{(\text{Id.} :)} \langle a' \cdot a', \{ a' \mapsto a', a_i' \cdot a_i \}, \{ a \cdot a' \cdot a_i \cdot a_i' \} \rangle$$

$$\langle a' \cdot a', s_i \cdot s_i' \rangle \xrightarrow{(\text{Id.} :)} \langle a' \cdot a', s_i \cdot s_i' \rangle$$

$$\langle \{ t \cdot t', s_i \cdot s_i' \}, \{ a \cdot a' \cdot a_i \cdot a_i' \} \rangle \xrightarrow{(\text{Pih} \cdot \text{Look}, \text{Pih} \cdot \text{Update}, \text{Lookup}) \downarrow \text{(Lookup)}} \langle \{ t \cdot t', s_i \cdot s_i' \}, \{ a \cdot a' \cdot a_i \cdot a_i' \} \rangle \xrightarrow{(\text{Pih} \cdot \text{Look}, \text{Pih} \cdot \text{Update}, \text{Lookup}) \downarrow \text{(Lookup)}} \langle \{ t \cdot t', s_i \cdot s_i' \}, \{ a \cdot a' \cdot a_i \cdot a_i' \} \rangle$$
Appendix C

Candidate Leak Witnesses

Table C.1 summarises the results of the candidate-witness trials of Chapter 13. The results of the search candidates apxSeqs lazy are represented as follows. Only one candidate is shown for each rule sequence. Permutation loops are eliminated by the search strategy. For lengths 5 and 6 all candidates are shown. At length 6, 3 of the 5 candidates are essentially the length 5 candidate plus a benign Let rule. For longer sequences there are many such candidates: at length 7 there are 13 and at length 8 there are 70; none of these are shown. The final column suggests a cure for the unbounded space behaviour of the candidates. The vast majority leak space by applying a function to too many arguments so they are rejected by a type system. The remainder leak space by creating useless update markers so a #gc-like collector could be used to run them in bounded space.

Table C.1: Candidate leak witnesses of lazy.

<table>
<thead>
<tr>
<th>Length</th>
<th>Self-feeding loop and candidate leak witness</th>
<th>Cure</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Push, Lookup, Update, Reduce, Push</td>
<td>ill-typed</td>
</tr>
<tr>
<td></td>
<td>{a \mapsto f \epsilon, f \mapsto \lambda x. f \epsilon \epsilon } a, \epsilon</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Push, Let, Lookup, Update, Reduce, Push</td>
<td>ill-typed</td>
</tr>
<tr>
<td></td>
<td>{a \mapsto (\text{let } y = \mathcal{E} \text{ in } f) \epsilon, f \mapsto \lambda x. (\text{let } y = \mathcal{E} \text{ in } f) \epsilon \epsilon } a, \epsilon</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Push, Lookup, Update, Reduce, Let, Lookup</td>
<td>#gc</td>
</tr>
<tr>
<td></td>
<td>{a \mapsto f \epsilon, f \mapsto \lambda x. \text{let } y = f \epsilon \text{ in } y } a, \epsilon</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Push, Lookup, Update, Reduce, Push, Let</td>
<td>ill-typed</td>
</tr>
<tr>
<td></td>
<td>{a \mapsto f \epsilon, f \mapsto \lambda x. (\text{let } y = \mathcal{E} \text{ in } f) \epsilon } a, \epsilon</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Push, Lookup, Update, Reduce, Push, Push</td>
<td>ill-typed</td>
</tr>
<tr>
<td></td>
<td>{a \mapsto f \epsilon, f \mapsto \lambda x. f \epsilon \epsilon } a, \epsilon</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Push, Lookup, Update, Reduce, Push, Let, Lookup</td>
<td>ill-typed</td>
</tr>
<tr>
<td>Line</td>
<td>Code</td>
<td>Cure</td>
</tr>
<tr>
<td>------</td>
<td>----------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>7</td>
<td>( \text{Push, Lookup, Update, Reduce, Let, Push, Lookup} ) ( a \rightarrow f \varepsilon, f \rightarrow \lambda x. \text{let } y = f \varepsilon \text{ in } y \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>7</td>
<td>( \text{Push, Lookup, Update, Reduce, Push, Reduce, Push} ) ( a \rightarrow f \varepsilon, f \rightarrow \lambda x. (\lambda y. f \varepsilon) \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>7</td>
<td>( \text{Push, Lookup, Update, Reduce, Push, Push, Reduce} ) ( a \rightarrow f \varepsilon, f \rightarrow \lambda x. (\lambda y. f \varepsilon) \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>7</td>
<td>( \text{Push, Reduce, Lookup, Update, Reduce, Push, Push, Push} ) ( a \rightarrow (\lambda y. f) \varepsilon, f \rightarrow (\lambda x. (\lambda y. f) x) \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>7</td>
<td>( \text{Push, Lookup, Update, Reduce, Push, Push, Push} ) ( a \rightarrow f \varepsilon, f \rightarrow \lambda x. f \varepsilon \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Lookup, Push, Lookup, Update, Reduce, Let, Let} ) ( a \rightarrow o, o \rightarrow i, i \rightarrow j \varepsilon, j \rightarrow \lambda l. \text{let } c = j \varepsilon \text{ in } let g = c \text{ in } g \varepsilon ) ( a, \varepsilon )</td>
<td>#gc</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Lookup, Let, Lookup, Push, Lookup, Update, Reduce, Let} ) ( a \rightarrow g, g \rightarrow h \varepsilon, h \rightarrow \lambda l. \text{let } c = h \varepsilon \text{ in } c \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Lookup, Push, Lookup, Update, Reduce, Push, Let} ) ( a \rightarrow g, g \rightarrow h \varepsilon, h \rightarrow \lambda l. (\lambda k. \text{let } c = h \varepsilon \text{ in } c \varepsilon) \varepsilon ) ( a, \varepsilon )</td>
<td>#gc</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Lookup, Push, Push, Lookup, Update, Reduce, Let} ) ( a \rightarrow g, g \rightarrow h \varepsilon, h \rightarrow \lambda l. (\lambda k. \text{let } c = h \varepsilon \text{ in } c \varepsilon) \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Lookup, Push, Push, Push, Lookup, Update, Reduce, Push, Let} ) ( a \rightarrow g, g \rightarrow h \varepsilon, h \rightarrow \lambda l. (\lambda k. \text{let } c = h \varepsilon \text{ in } c \varepsilon) \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Lookup, Push, Push, Push, Push, Update, Reduce, Push, Let} ) ( a \rightarrow g, g \rightarrow h \varepsilon, h \rightarrow \lambda l. (\lambda k. \text{let } c = h \varepsilon \text{ in } c \varepsilon) \varepsilon ) ( a, \varepsilon )</td>
<td>#gc</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Lookup, Push, Push, Push, Push, Push, Update, Reduce, Let} ) ( a \rightarrow g, g \rightarrow h \varepsilon, h \rightarrow \lambda l. (\lambda k. \text{let } c = h \varepsilon \text{ in } c \varepsilon) \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Lookup, Push, Push, Push, Push, Push, Push, Update, Reduce, Let} ) ( a \rightarrow g, g \rightarrow h \varepsilon, h \rightarrow \lambda l. (\lambda k. \text{let } c = h \varepsilon \text{ in } c \varepsilon) \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Lookup, Push, Push, Push, Push, Push, Push, Push, Update, Reduce, Let} ) ( a \rightarrow g, g \rightarrow h \varepsilon, h \rightarrow \lambda l. (\lambda k. \text{let } c = h \varepsilon \text{ in } c \varepsilon) \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Push, Lookup, Update, Reduce, Push, Let} ) ( a \rightarrow g \varepsilon, g \rightarrow i \varepsilon, i \rightarrow \lambda j. \text{let } c = i \varepsilon \text{ in } c \varepsilon \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Push, Push, Push, Push, Update, Reduce, Push, Let} ) ( a \rightarrow g \varepsilon, g \rightarrow i \varepsilon, i \rightarrow \lambda k. \text{let } c = i \varepsilon \text{ in } c \varepsilon \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Push, Push, Push, Push, Push, Update, Reduce, Let} ) ( a \rightarrow g \varepsilon, g \rightarrow i \varepsilon, i \rightarrow \lambda l. \text{let } c = i \varepsilon \text{ in } c \varepsilon \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Push, Push, Push, Push, Push, Push, Update, Reduce, Let} ) ( a \rightarrow g \varepsilon, g \rightarrow i \varepsilon, i \rightarrow \lambda l. \text{let } c = i \varepsilon \text{ in } c \varepsilon \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Push, Push, Push, Push, Push, Push, Push, Update, Reduce, Let} ) ( a \rightarrow (\lambda g. h) \varepsilon, h \rightarrow j \varepsilon, j \rightarrow \lambda l. \text{let } c = j \varepsilon \text{ in } (\lambda g. c) \varepsilon ) ( a, \varepsilon )</td>
<td>#gc</td>
</tr>
<tr>
<td>8</td>
<td>( \text{Push, Push, Push, Push, Push, Push, Push, Push, Update, Reduce, Let} ) ( a \rightarrow g \varepsilon, g \rightarrow j, j \rightarrow \lambda l. \text{let } c = j \varepsilon \text{ in } c \varepsilon \varepsilon ) ( a, \varepsilon )</td>
<td>ill-typed</td>
</tr>
<tr>
<td>Length</td>
<td>Self-feeding loop and candidate leak witness</td>
<td>Cure</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8</td>
<td><code>Push, Push, Lookup, Push, Lookup, Update, Reduce, Let</code>&lt;br&gt;{a \mapsto g \epsilon \epsilon, g \mapsto j \epsilon, j \mapsto \lambda l.\text{let } c = j \text{ in } c \epsilon \epsilon} a, \epsilon`&lt;br&gt;ill-typed</td>
<td></td>
</tr>
</tbody>
</table>
Glossary of Notation and Definitions

Mathematical notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{dom}$</td>
<td>domain</td>
<td>Def 4.2</td>
</tr>
<tr>
<td>$\text{rng}$</td>
<td>range</td>
<td>Def 4.2</td>
</tr>
<tr>
<td>$\text{function}$</td>
<td>is a function</td>
<td>Def 4.2</td>
</tr>
<tr>
<td>$\mathcal{I}$</td>
<td>disjoint sets</td>
<td>Def 4.5</td>
</tr>
<tr>
<td>$\text{injective}$</td>
<td>is an injective function</td>
<td>Def 4.18</td>
</tr>
<tr>
<td>$+$</td>
<td>sequence concatenation</td>
<td>Def 4.8</td>
</tr>
<tr>
<td>$#$</td>
<td>set cardinality</td>
<td>Def 5.2</td>
</tr>
<tr>
<td>$\div$</td>
<td>integer division</td>
<td>Prop 7.2</td>
</tr>
<tr>
<td>$a\cdot b$</td>
<td>element of relation: $(a, b)$</td>
<td>Ex 8.1</td>
</tr>
<tr>
<td>$\text{fix}_f(x)$</td>
<td>least fixpoint of $f$ starting from $x$</td>
<td>(3.3)</td>
</tr>
</tbody>
</table>

Term graphs and graph grammars

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{G}$</td>
<td>graph grammar</td>
<td>Def 4.1</td>
</tr>
<tr>
<td>$\text{ROOT}$</td>
<td>root category</td>
<td>Def 4.1</td>
</tr>
<tr>
<td>$\mathcal{FSymbols}$</td>
<td>set of all function symbols</td>
<td>Def 4.1</td>
</tr>
<tr>
<td>infinite category</td>
<td></td>
<td>Sect 4.1</td>
</tr>
<tr>
<td>$\text{Var}$</td>
<td>set of all variables</td>
<td>Def 4.2</td>
</tr>
<tr>
<td>$T : C$</td>
<td>term $T$ of category $C$</td>
<td>Def 4.2</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>null variable</td>
<td>Sect 4.2.1</td>
</tr>
<tr>
<td>$\mathcal{E}$</td>
<td>null term</td>
<td>Def 4.3</td>
</tr>
<tr>
<td>$\text{var}$</td>
<td>variables of term</td>
<td>Def 4.4</td>
</tr>
<tr>
<td>$\text{var}$</td>
<td>variables in graph</td>
<td>Def 4.6</td>
</tr>
<tr>
<td>$\text{fv}$</td>
<td>free variables of term</td>
<td>Def 4.4</td>
</tr>
<tr>
<td>$\text{fv}$</td>
<td>free variables in graph</td>
<td>Def 4.6</td>
</tr>
<tr>
<td>$\text{bv}$</td>
<td>bound variables of term</td>
<td>Def 4.4</td>
</tr>
<tr>
<td>$\text{bv}$</td>
<td>bound variables in graph</td>
<td>Def 4.6</td>
</tr>
<tr>
<td>$\theta()$</td>
<td>substitution into term</td>
<td>Def 4.4</td>
</tr>
<tr>
<td>$\equiv$</td>
<td>identical terms</td>
<td>Def 4.4</td>
</tr>
<tr>
<td>$\equiv$</td>
<td>identical graphs</td>
<td>Def 4.6</td>
</tr>
<tr>
<td>$=$</td>
<td>equal terms</td>
<td>Def 4.6</td>
</tr>
<tr>
<td>$=$</td>
<td>equal graphs</td>
<td>Def 4.6</td>
</tr>
<tr>
<td>$\text{Graph}$</td>
<td>rooted term graph</td>
<td>Def 4.5</td>
</tr>
<tr>
<td>$\text{wf}$</td>
<td>well-formed graph</td>
<td>Def 4.6</td>
</tr>
<tr>
<td>$\text{reach}$</td>
<td>variables reachable in graph</td>
<td>Def 4.7</td>
</tr>
<tr>
<td>$\text{gc}$</td>
<td>reachability-based garbage collector</td>
<td>Def 4.7</td>
</tr>
<tr>
<td>$\text{BaseCats}$</td>
<td>base categories of grammar</td>
<td>Def 4.19</td>
</tr>
<tr>
<td>$\text{size}$</td>
<td>term and graph size</td>
<td>Def 5.1</td>
</tr>
<tr>
<td>$\text{StackCat}$</td>
<td>stack category of grammar</td>
<td>Def 5.7</td>
</tr>
</tbody>
</table>
### Patterns, contexts and evaluators

**⊕** pattern-check environment combination \(\text{Def 4.8 \ 60}\)

**cv** capture variables in environment \(\text{Def 4.8 \ 60}\)

**Holes** set of all holes \(\text{Def 4.1 \ 61}\)

**Pat** term pattern \(\text{Def 4.9 \ 61}\)

**UnrootedPat** unrooted graph pattern \(\text{Def 4.10 \ 62}\)

**GraphPat** rooted graph pattern \(\text{Def 4.10 \ 62}\)

**LhsPat** left pattern \(\text{Def 4.11 \ 62}\)

**RhsPat** right pattern \(\text{Def 4.12 \ 62}\)

**Rule** graph-rewrite rule \(\text{Def 4.13 \ 63}\)

**Rewrite** rewrite rule with effect \(\text{Def 4.20 \ 67}\)

**Evaluator** graph evaluator \(\text{Def 4.14 \ 63}\)

**\(A\)** \(A\) is an evaluator \(\text{Def 4.14 \ 63}\)

**holes** holes in pattern \(\text{Def 4.15 \ 63}\)

**T** term context \(\text{Def 4.16 \ 64}\)

**\((M||D)\)** term context \(\text{Def 4.16 \ 64}\)

**G** graph context \(\text{Def 4.17 \ 64}\)

**\((G, T)\)** graph context \(\text{Def 4.17 \ 64}\)

**G** graph-pattern unifier \(\text{Def 5.9 \ 80}\)

**→** evaluation relation \(\text{Def 4.18 \ 65}\)

**→_{gc}\** evaluation with maximal garbage collection \(\text{Def 4.18 \ 65}\)

**→_{gc}** evaluation with maximal garbage collection \(\text{Def 5.2 \ 76}\)

**put \(Ch\)** evaluation with put effect \(\text{Def 4.21 \ 68}\)

**get \(Ch\)** evaluation with get effect \(\text{Def 4.21 \ 68}\)

**space** space/node usage of graph \(\text{Def 5.2 \ 72}\)

**space** asymptotic space usage of graph with input \(\text{Def 5.3 \ 73}\)

**size Usage** size usage of graph \(\text{Def 5.2 \ 72}\)

**time** time usage of graph \(\text{Def 5.2 \ 72}\)

**accurate** accurate evaluator \(\text{Def 5.4 \ 74}\)

**uniform Accurate** uniformly-accurate evaluator \(\text{Def 5.4 \ 74}\)

**extend** rule right-pattern extension \((5.1) \ 76\)

**ggen** garbage-generating rule \(\text{Def 5.5 \ 76}\)

**symbols** function symbols in graph \(\text{Def 5.6 \ 77}\)

**trim** trim evaluator for a graph \(\text{Def 5.6 \ 77}\)

**deterministic** deterministic evaluator \(\text{Def 5.8 \ 80}\)

**unify** term-pattern disunification \(\text{Def 5.10 \ 80}\)

**unify** graph-pattern disunification \(\text{Def 5.12 \ 82}\)

**unify** full disunification \(\text{Def 12.3 \ 188}\)

**unify** hole disunification \(\text{Def 12.4 \ 191}\)

**⊙** unifier composition \(\text{Def 5.11 \ 81}\)

**trace** evaluation trace of graph \(\text{Def 11.8 \ 181}\)

**apxUnify** approximate disunification \(\text{Def 13.7 \ 208}\)

**\(E\)** the null context \(\text{Def 12.5 \ 193}\)

### Evaluatoras

**∅** empty evaluator \(\text{Ex 7.1 \ 107}\)

**lazy** call-by-need (cbneed) \(\text{Ex 4.1 \ 53}\)

**if** cbneed with if-expressions \(\text{Ex 4.12 \ 66}\)

**ifio** cbneed with if and boolean IO \(\text{Ex 4.13 \ 68}\)
| stack     | cbneed with single-node stack | Ex 5.2 74 |
| call-by-value | | Ex 5.3 75 |
| vlazy     | non-deterministic cbneed | Ex 5.8 84 |
| list      | cbneed with list data type | Ex 6.2 89 |
| proj      | cbneed with lists and projections | Ex 6.3 90 |
| pint      | cbneed with an integer | Ex 6.4 91 |
| hio       | cbneed with character IO | Sect 6.1.4 94 |
| nost      | cbneed with pointer-reversal stack | Def 6.1 96 |
| sest      | cbneed with separated control expression | Def 6.2 97 |
| ind       | cbneed with value indirections | Def 6.3 97 |
| #ind      | cbneed with indirect update markers | Def 6.4 98 |
| nobh      | cbneed without (Lookup) black-holing | Def 6.5 99 |
| badbh     | cbneed without variable black-holing | Def 6.6 99 |
| noah      | cbneed without (Push) black-holing | Def 6.7 99 |
| noabkh    | cbneed without (Push), (Lookup) black-holing | Def 6.8 99 |
| prsh      | cbneed with lists and projection short cuts | Def 6.9 100 |
| nottr     | cbneed without environment trimming | Def 6.10 101 |
| bhtr      | cbneed with black-hole environment trimming | Def 6.11 101 |
| binary    | encodes numbers in binary | Ex 7.1 107 |
| unary     | encodes numbers in unary | Ex 7.1 107 |
| counter   | cbneed building chain of allocated nodes | Prop 7.3 109 |
| leaker    | leaky for terminating programs | Prop 7.3 109 |
| horder    | repeatable higher-order term reduction | Ex 12.10 189 |
| bad       | follows chains inserting new nodes | Ex 11.5 178 |
| good      | follows chains | Ex 11.5 178 |

**Leaks and leak criteria**

- leakier
- repairs
- not leakier
- does not repair
- mutually leakier
- mutually not leakier

**Space fault criterion**

- asymptleak asymptotic leak | Def 7.4 111 |
- inefficient inefficiency criterion | Def 7.6 112 |
- stronglyinefficient strong-inefficiency criterion | Def 7.6 112 |
- unsafe unsafe criterion | Def 7.6 112 |
- sizeleak size-usage inefficiency criterion | Def 7.7 113 |
- sizestrongleak size-usage strong-inefficiency criterion | Def 7.7 113 |
- sizeunsafe size-usage unsafe criterion | Def 7.7 113 |

**Space relations and simulation proofs**

- \( \equiv \) bisimilar graphs | Def 8.1 119 |
- \( \mathcal{G}_1 \equiv \mathcal{G}_2 \) configuration-rewrite system | Def 8.3 121 |
- \( (\theta, \phi, G, H) \) configuration | Def 8.3 121 |
- configuration-rewrite relation | Def 8.3 121 |
- identity configuration-rewrite relation | Ex 8.4 123 |
- bad id configuration-rewrite relation | Ex 8.6 128 |
- id expression configuration-rewrite relation | Def 9.2 137 |
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>configuration context</td>
<td>Def 8.3</td>
<td>121</td>
</tr>
<tr>
<td>start</td>
<td>initial configuration</td>
<td>Def 8.3</td>
<td>121</td>
</tr>
<tr>
<td>end</td>
<td>trace end predicate</td>
<td>Def 8.3</td>
<td>121</td>
</tr>
<tr>
<td>$Id.F$</td>
<td>identity configuration rewrite rule</td>
<td>Ex 8.4</td>
<td>123</td>
</tr>
<tr>
<td>gc</td>
<td>configuration garbage collector</td>
<td>Def 8.5</td>
<td>126</td>
</tr>
<tr>
<td>$gc$</td>
<td>pattern-configuration garbage collector</td>
<td>Def 9.8</td>
<td>146</td>
</tr>
<tr>
<td>SubTrace</td>
<td>configuration sub-trace property</td>
<td>Def 8.6</td>
<td>126</td>
</tr>
<tr>
<td>GcTrace</td>
<td>configuration garbage-collection property</td>
<td>Def 8.6</td>
<td>126</td>
</tr>
<tr>
<td>PSR</td>
<td>proportional space relation</td>
<td>Def 8.7</td>
<td>128</td>
</tr>
<tr>
<td>self-consistent</td>
<td>self-consistent PSR</td>
<td>Def 8.8</td>
<td>132</td>
</tr>
<tr>
<td>Trans$e$</td>
<td>null variable translation rule</td>
<td>Def 8.9</td>
<td>132</td>
</tr>
<tr>
<td>Trans$E$</td>
<td>null term translation rule</td>
<td>Def 8.9</td>
<td>132</td>
</tr>
<tr>
<td>PSR rule addition</td>
<td></td>
<td>Prop 8.8</td>
<td>135</td>
</tr>
<tr>
<td>simulation</td>
<td></td>
<td>Def 9.1</td>
<td>137</td>
</tr>
<tr>
<td>mutual simulation</td>
<td></td>
<td>Def 9.1</td>
<td>137</td>
</tr>
<tr>
<td>s-lockstep simulation</td>
<td></td>
<td>Def 9.3</td>
<td>138</td>
</tr>
<tr>
<td>lockstep simulation</td>
<td></td>
<td>Def 9.4</td>
<td>139</td>
</tr>
<tr>
<td>$\Rightarrow$</td>
<td>configuration implication</td>
<td>Def 9.5</td>
<td>141</td>
</tr>
<tr>
<td>$\Downarrow$</td>
<td>configuration implication</td>
<td>Def 9.5</td>
<td>141</td>
</tr>
<tr>
<td>translate</td>
<td>graph translation</td>
<td>Def 9.6</td>
<td>142</td>
</tr>
<tr>
<td>FCS</td>
<td>forward closures</td>
<td>Def 9.6</td>
<td>142</td>
</tr>
<tr>
<td>sqTops</td>
<td>graph-pattern translation</td>
<td>Def 9.7</td>
<td>143</td>
</tr>
<tr>
<td>FFCS</td>
<td>finite forward closures</td>
<td>Def 9.7</td>
<td>143</td>
</tr>
<tr>
<td>Ren</td>
<td>renaming rule</td>
<td>Def 9.9</td>
<td>148</td>
</tr>
<tr>
<td>HRen</td>
<td>hole-substitution renaming rule</td>
<td>Def 9.8</td>
<td>150</td>
</tr>
<tr>
<td>Share</td>
<td>duplication-elimination rule</td>
<td>Def 9.10</td>
<td>152</td>
</tr>
<tr>
<td>inferPSR</td>
<td>PSR rule inference</td>
<td>Def 10.1</td>
<td>157</td>
</tr>
<tr>
<td>choices</td>
<td>inferred rule selection</td>
<td>Def 10.2</td>
<td>159</td>
</tr>
<tr>
<td>indexed PSR rule notation</td>
<td></td>
<td>Ex 10.5</td>
<td>162</td>
</tr>
<tr>
<td>lockstep or bounded termination</td>
<td></td>
<td>Prop 10.4</td>
<td>165</td>
</tr>
</tbody>
</table>

**Space-relation garbage collectors**

- gc: standard collector
- Igc: indirection collector
- #/gc: update-marker collector
- $\pi$ gc: projection-shortcutting collector
- let gc: dead-binding collector

**Translation relations**

- $\Rightarrow$: all chains of term $E$
- $\Rightarrow^u$: unary to binary
- $\Rightarrow^u$: unary chain to unary term
- $\Rightarrow^b$: binary tree isomorphism
- $\Rightarrow^{sub}$: illustrative identity subset
- $\Rightarrow^{id\lambda}$: booleans to $\lambda$-calculus
- $\Rightarrow^{id}$: identity
- $\Rightarrow^{idX}$: id on expressions/program states
- $\Rightarrow^{idX#}$: id adding update marker to stack
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>(id)</td>
<td>projections to shortcut projections</td>
<td>Prop B.11 233</td>
</tr>
<tr>
<td>(\mapsto)</td>
<td>translates (\text{test}) graphs</td>
<td>Ex 10.10 166</td>
</tr>
<tr>
<td>(\mapsto)</td>
<td>translates (\text{host}) graphs</td>
<td>Prop B.3 231</td>
</tr>
<tr>
<td>(\mapsto)</td>
<td>translates (\text{incl}) graphs</td>
<td>Ex 10.5 162</td>
</tr>
<tr>
<td>(\mapsto)</td>
<td>translates (\text{nobl}) graphs</td>
<td>Ex 10.8 164</td>
</tr>
<tr>
<td>(\mapsto)</td>
<td>trimming to black-hole trimming</td>
<td>Prop B.1 231</td>
</tr>
<tr>
<td>(\mapsto)</td>
<td>translates (\text{nobl}) graphs</td>
<td>Prop B.2 231</td>
</tr>
<tr>
<td>(\mapsto)</td>
<td>translates (\text{badlh}) graphs</td>
<td>Prop 9.2 141</td>
</tr>
<tr>
<td>(\mapsto)</td>
<td>translates (\text{bhtr}) graphs</td>
<td>Ex 10.13 169</td>
</tr>
<tr>
<td>(\mapsto)</td>
<td>id with commutativity</td>
<td>Ex 11.9 181</td>
</tr>
</tbody>
</table>

**Leak classes, checking and search functions**

Inefficient translation

- \(\text{translation-leak}\) translation leak
- \(\text{evaluation-leak}\) evaluation leak
- \(\text{InfSpace}\) all unbounded-space programs
- \(\text{InfIn}\) InfIn leak class
- \(\text{sfloops}\) self-feeding loops
- \(\text{xsfloops}\) expanding self-feeding loops
- \(\text{Xsf}\) expanding self-feeding loop leaks
- active witnesses
- passive witnesses
- active leaks
- passive leaks
- \(\text{super}\) super-rule of two rules
- \(\text{superRule}\) super-rule of rule sequence
- \(\text{sflUnify}\) self-feeding loop test
- \(\circ\) matcher-unifier composition
- \(\text{candidate}\) candidate witness checker
- \(\text{witness}\) leak witness checker
- \(\text{candidates}\) candidate witness search function
- \(\text{witnesses}\) leak witness search function
- \(\text{allSeqs}\) all sequences search tactic
- \(\text{distinctSeqs}\) distinct-sequences search tactic
- \(\text{nrt}\) next-rule table
- \(\text{nrtSeqs}\) next-rule-table search tactic
- \(\text{plSeqs}\) permutation-loop-free search tactic
- \(\text{expSeqs}\) expanding-sequences search tactic
- \(\text{apxSeqs}\) approximate-sequences search tactic

Page numbers are shown in brackets next to the definitions.
Bibliography


