Data Model Refinement, Generic Profiling, and Functional Programming

Nathan Richard Charles

Thesis submitted for the degree of DPhil in Computer Science

University of York
Department of Computer Science

10th May 2001
Abstract

Profiling has become an important part of writing functional programs. Most profiling tools are able to gather a lot of information about a computation, but users are often unable to efficiently extract the information they require. The raw data gathered by the tools is too complicated for direct use by the programmer, and the graphical views provided by the tools often display too little information. Yet, for large computations, they are also too crowded.

This thesis shows how to use data modelling techniques to create better profiling tools. We describe a tool integrating a query evaluator that allows efficient and controlled extraction of profiling data. We show how to realize refinement in data modelling and use this to construct a view compiler. The compiler’s input is an expression describing a chain of data models, and its output is a set of definitions for retrieving data for any of the data models in the chain, from a database based on a concrete data model.

Previous profiling tools generally presented a fixed level of abstraction to the user. This can be too low-level to effectively answer some queries, and at the same time too abstract to answer more complicated queries. We show how output from the view compiler is used to create a profiling tool that allows the user to view and query a computation using different abstraction levels.
Contents

Abstract i
Acknowledgements xii
Declaration xiii

1 Introduction 1
  1.1 Aim ........................................... 1
  1.2 Functional languages and profiling ......................... 1
  1.3 Profiling and data modelling ................................ 2
  1.3.1 Data model refinement ................................ 3
  1.4 Historical background ..................................... 3
  1.5 Thesis contributions ....................................... 4
  1.6 Thesis structure ......................................... 5

I Motivating application: Improving parallel functional programs 7

2 Parallel functional programming 9
  2.1 Introduction ......................................... 9
  2.1.1 The par and seq annotations ............................. 9
  2.1.2 Safety ............................................ 10
  2.2 Identifying possible parallelism ............................ 10
  2.2.1 Forward analysis .................................... 11
  2.2.2 Backward Analysis .................................... 14
  2.2.3 Other approaches .................................... 16
  2.3 Granularity of parallelism .................................. 16
  2.4 Parallel profiling tools .................................. 17
  2.4.1 Using profiling tools ................................ 18
  2.4.2 Problems with current parallel profiling tools .......... 24
  2.5 Summary ............................................. 25
3 Data modelling parallel graph reduction

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Data Modelling Parallelism</td>
<td>27</td>
</tr>
<tr>
<td>3.2.1 A Simple Parallel Graph Reduction Model</td>
<td>28</td>
</tr>
<tr>
<td>3.2.2 Introducing Time</td>
<td>29</td>
</tr>
<tr>
<td>3.2.3 Introducing Processors</td>
<td>30</td>
</tr>
<tr>
<td>3.2.4 Introducing Nodes</td>
<td>30</td>
</tr>
<tr>
<td>3.2.5 Introducing spark/thread distinction, thread migration, fetching</td>
<td>32</td>
</tr>
<tr>
<td>3.3 GranSim: data models, log-files and graphical charts</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1 Modelling the log-file</td>
<td>33</td>
</tr>
<tr>
<td>3.3.2 Modelling the graphs</td>
<td>34</td>
</tr>
<tr>
<td>3.4 Conversion into relational database form</td>
<td>34</td>
</tr>
<tr>
<td>3.4.1 Expressing queries using SQL</td>
<td>37</td>
</tr>
<tr>
<td>3.5 Summary</td>
<td>37</td>
</tr>
</tbody>
</table>

4 Interactive parallel profiling

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>39</td>
</tr>
<tr>
<td>4.2 Tool Overview</td>
<td>39</td>
</tr>
<tr>
<td>4.2.1 Using queries to extract information</td>
<td>41</td>
</tr>
<tr>
<td>4.2.2 Using graphs to extract information</td>
<td>41</td>
</tr>
<tr>
<td>4.2.3 Using graphs to compose queries</td>
<td>41</td>
</tr>
<tr>
<td>4.2.4 Using queries to generate graphs</td>
<td>42</td>
</tr>
<tr>
<td>4.3 An Example</td>
<td>43</td>
</tr>
<tr>
<td>4.3.1 GranSim graphs</td>
<td>43</td>
</tr>
<tr>
<td>4.3.2 Using queries to extract information</td>
<td>45</td>
</tr>
<tr>
<td>4.3.3 Using graphs to extract information</td>
<td>46</td>
</tr>
<tr>
<td>4.3.4 Using graphs to compose queries</td>
<td>47</td>
</tr>
<tr>
<td>4.3.5 Using queries to generate graphs</td>
<td>49</td>
</tr>
<tr>
<td>4.4 Implementation</td>
<td>50</td>
</tr>
<tr>
<td>4.5 Evaluation</td>
<td>53</td>
</tr>
<tr>
<td>4.6 Related work</td>
<td>55</td>
</tr>
<tr>
<td>4.7 Summary</td>
<td>56</td>
</tr>
</tbody>
</table>

II Technical development: Semantic and functional implementation of data model refinement

5 Refinement: Design

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Introduction</td>
<td>59</td>
</tr>
<tr>
<td>5.1.1 E-R notation</td>
<td>59</td>
</tr>
</tbody>
</table>
# III Experimental evaluation: A generic profiling system

8 **Refinement: Evaluation**

8.1 Introduction ........................................ 123
8.2 Evaluation – The parallel model .................. 123
  8.2.1 Expressing the data model chain in combinatorial form 123
  8.2.2 The primitives .................................. 124
  8.2.3 The views and fall-through information .......... 128
  8.2.4 Integrating views into the tool kit ............... 128
  8.2.5 Sample queries .................................. 129
  8.2.6 Experimental results ......................... 130
8.3 Evaluation – The heap model ..................... 137
  8.3.1 Data-modelling ................................. 137
  8.3.2 Expressing the data model chain in combinatorial form 142
  8.3.3 The primitives .................................. 142
  8.3.4 The views ....................................... 142
  8.3.5 Sample query .................................... 142
  8.3.6 Construction and retainer profiling .......... 142
  8.3.7 Discussion ...................................... 143
8.4 Summary ............................................ 143

9 **Generic profiling tools**

9.1 Introduction ........................................ 145
9.2 Making the tools generic ......................... 145
  9.2.1 View generation and Building the databases .... 146
  9.2.2 Stage 1: View compilation .................... 148
  9.2.3 Stage 2: Building the databases ............... 153
  9.2.4 Stage 3: The Front-end ....................... 154
9.3 Application: parallel profiling ................. 161
  9.3.1 Stage 1: Data modelling ...................... 161
  9.3.2 Stage 2: Building the databases .............. 162
  9.3.3 Stage 3: Using the browser .................. 162
  9.3.4 Experimental results .......................... 164
9.4 Application: heap profiling .................... 166
  9.4.1 Stage 1: Data modelling ...................... 166
  9.4.2 Stage 2: Building the databases .............. 166
9.4.3 Stage 3: Using the browser ................................................. 167
9.5 Implementation ................................................................. 169
   9.5.1 The Builder ................................................................. 169
   9.5.2 The front-end ............................................................... 173
   9.5.3 Implementation experience .............................................. 175
9.6 Related work .......................................................................... 176
9.7 Summary ................................................................................ 176

10 Conclusions ........................................................................... 179
   10.1 Data model refinement ........................................................ 179
   10.2 Profiling and data modelling .................................................. 180
   10.3 Criticism .............................................................................. 181
   10.4 Future Work ......................................................................... 182
   10.5 Concluding Thought ............................................................ 183

A Parallel profiling example data .................................................... 185
   A.1 Data model chain: primitive encoding ...................................... 185
   A.2 Views .................................................................................. 186
   A.3 DTDs .................................................................................. 188
      A.3.1 The minimal DTD ............................................................ 188
      A.3.2 The final DTD ................................................................. 190

B Heap profiling example data ....................................................... 193
   B.1 Data model chain: primitive encoding ...................................... 193
   B.2 Views .................................................................................. 193
   B.3 DTDs .................................................................................. 194
      B.3.1 The minimal DTD ............................................................ 194
      B.3.2 The final DTD ................................................................. 195
# List of Figures

1.1 Tail strict function .......................... 2

2.1 The \texttt{sumEuler} program (Version 0) .................. 19
2.2 Activity graph for \texttt{SumEuler} 9 .................. 20
2.3 Per-Thread graph for \texttt{SumEuler} 9 .................. 22
2.4 Per-Thread spark and block graph for \texttt{SumEuler} 9 .................. 22

3.1 Relational tables for a database using data generated by GranSim 35

4.1 The graphical interface of the interactive tool .................. 40
4.2 Mini per-thread graph example .................. 41
4.3 Query template for the per-thread graph .................. 42
4.4 The \texttt{sumEuler} program (Version 1) .................. 44
4.5 Activity graph for Version 1 of the program .................. 44
4.6 Per-thread graph for Version 1 of the program .................. 45
4.7 Average run-time for each \texttt{par-site} .................. 46
4.8 Per-thread graph for Version 2 of the program .................. 47
4.9 Mini per-thread graph .................. 47
4.10 Activity graph for Version 3 of the program .................. 48
4.11 How many sparks lie on each processor, where the user clicks? .................. 49
4.12 Filtered Per-thread graph for Version 1 of the program .................. 50
4.13 Structure of the interactive tool .................. 50

5.1 The \texttt{Schema} data-type .................. 60
5.2 The primitive refinement data type .................. 62
5.3 Auxiliary functions .................. 64
5.4 \texttt{B} compilation scheme: \texttt{EGeneration}, \texttt{RGeneration}, \texttt{HGeneration} .................. 64
5.5 Example \texttt{AGeneration}, with a new primary key .................. 66
5.6 \texttt{B} compilation scheme: \texttt{AGeneration}, \texttt{HGeneration} .................. 67
5.7 An example of application of \texttt{HGeneration} .................. 68
5.8 \texttt{T} compilation scheme for the \texttt{RSplit} primitive .................. 69
5.9 An example of entity decomposition .................. 70
5.10 \texttt{T} compilation scheme for the \texttt{E Decomposition} primitive .................. 71
5.11 $\mathcal{T}$ compilation scheme for the RDecomposition primitive. .... 73
5.12 $\mathcal{T}$ compilation scheme for the HDecomposition primitive. .... 74

6.1 Compilation equations for CT. ........................................ 81
6.2 More auxiliary functions ............................................. 83
6.3 The default compilation scheme, $\mathcal{D}$. .......................... 84
6.4 The $\mathcal{V}$ compilation scheme for the bottom-up primitives. .... 85
6.5 The $\mathcal{V}$ compilation scheme for RSplit ........................... 89
6.6 The $\mathcal{V}$ compilation scheme for EDecomposition .................. 91
6.7 An example application of EDecomposition ......................... 92
6.8 The $\mathcal{V}$ compilation scheme for RDecomposition ............... 94
6.9 The $\mathcal{V}$ compilation scheme for HDecomposition ............... 95
6.10 Rules defining when view definitions generate a valid tables. .... 97

7.1 Composition combinators. .............................................. 102
7.2 Bottom-up primitives combinators. ................................... 105
7.3 Attribute and label combinators. ..................................... 109
7.4 Combinators for the RSplit and HDecomposition primitives ....... 110
7.5 Decomposition combinators. ......................................... 112
7.6 Combinators for defining multiple refinements to the same concept. 114
7.7 A refinement built from two refinements of the same entity. ....... 114
7.8 Integrating view compilation into the combinators. ............... 116
7.9 Refinement chain with three abstraction levels. ................... 117
7.10 A section of the ViewCompiler module. ............................. 118

8.1 A chain of data models to describe parallel profiling (Part I). .... 125
8.2 A chain of data models to describe parallel profiling (Part II). .... 126
8.3 Parallel models described using the combinators. ................... 127
8.4 A theoretical chain of data models for heap profiling. ............. 138
8.5 The four biographical phases of a heap cell. ........................ 139
8.6 Data model chain for biographic heap profiling by producer. ..... 140
8.7 The two biographical phases of a heap cell in Figure 8.6, Model (c). 141

9.1 Structure of the specialized tool kit .................................. 146
9.2 Structure of stages one and two in the generic tool kit. .......... 147
9.3 Compilation equations for $\mathcal{X})$. ................................ 150
9.4 Template SQL queries to create activity and per-thread graphs. .. 156
9.5 Rules for deriving the query variables from a E-R schema. ........ 156
9.6 The query template to produce a thread activity graph. ......... 157
9.7 Example state activity and per-thread graphs. ....................... 158
9.8 Conditions under which graphs can be generated. .................. 159
9.9 Example showing locking. ............................................. 161
9.10 Sample output produced by the GranSim to XML log-file convertor. 162
9.11 Generic browser displaying Thread Activity graph. ............. 163
9.12 View selecting window. ........................................... 164
9.13 Sample output produced by hp2xml. ............................. 167
9.14 Generic browser displaying heap waste by producer .......... 168
9.15 Reading the XML file. ............................................. 172
List of E-R Models

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A naïve view.</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>Introducing blocking.</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>Introducing time.</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>Limiting the number of concurrent running threads.</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>Introducing processors.</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>Introducing graph nodes.</td>
<td>31</td>
</tr>
<tr>
<td>7</td>
<td>Introducing spark/thread distinction, thread migration, fetching.</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>The data stored in GranSim log-files.</td>
<td>34</td>
</tr>
<tr>
<td>9</td>
<td>The data represented in GranSim graphs.</td>
<td>35</td>
</tr>
<tr>
<td>10</td>
<td>Model suitable for constructing a database.</td>
<td>36</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Time taken to display graphs in specialized tool. ............... 54

5.1 Bottom-up refinement primitives (attribute details omitted) ... 63
5.2 Top-down refinement primitives (attribute details omitted) . . . 69

7.1 Precedence and fixity of the combinators. ................. 115

8.1 Comparison of database build times. .................. 133
8.2 Comparison of database sizes. ........................... 133
8.3 Comparison of query lengths. ............................. 134
8.4 Comparison of query times. ............................... 135
8.5 Comparison of query times. ............................... 135
8.6 Comparison of total times. .............................. 136

9.1 The time taken to build databases in the specialized and generic tools. .................................................. 165
9.2 The time taken to display graphs in the specialized and generic tools. 166
9.3 Time taken to display graphs in generic tool. ............... 173
9.4 Breakdown of the times taken to display graphs (Haskell version). 174
9.5 Breakdown of times taken to display graphs (C version). ....... 174
Acknowledgements

This work was supported by a Research Studentship from the British Engineering and Physical Sciences Research Council. Thanks are due to many people. First and foremost I would like to thank Colin Runciman, who made me feel his door was always open, and whose supervision guided and inspired me.

I am grateful to Jeremy Jacob and John Hughes, my viva examiners, for their constructive comments that lead to improvements in my thesis.

I thank Thomas Hallgren for his hospitality and help relating to the Fudget library during my stay in Göteborg.

Thanks to all the members of the programming and language and systems research group in York, whose input helped shaped the direction of my work. In particular I would like to thank Graeme Moss who was always there to chat to whenever I needed a break, and Phil Hassall whose friendship has been greatly appreciated.

Thanks to my parents the support and encouragement they have given me over the years.

Finally, I thank my wife Emma both for her support (financial and otherwise), patience and understanding, particularly during my writing-up period, and for being there when I wanted to forget about my thesis.
Declaration

A cut-down version of Chapter 3 has been published in the book *Research Directions in Parallel Functional Programming* [17]. An early version of Chapter 4 was presented at IFL ’98 and subsequently published in the workshop proceedings [16].

Unless otherwise stated in the text, the ideas presented in this thesis are solely the work of the author.
Chapter 1

Introduction

Profiling has become an important part of writing functional programs, yet, there has been little explicit data modelling for profiling tools. Making the data models explicit is useful for both the implementor and programmer: Implementors can identify inconsistencies between the implicit data model that their tools use, the data recorded in the log-files, and the information displayed by visualization tools; and it is clear to programmers exactly what data model is used.

1.1 Aim

The aim of this thesis is:

- to make functional languages more usable by improving the quality of functional language profiling using data modelling techniques.

This thesis argues that explicit data modelling helps to identify problems with current profiling tools, and consequently leads to better tools.

Whilst pursuing this aim this thesis also demonstrates the use of Haskell, a lazy functional programming language with strong typing, as a language of choice for developing profiling tools. This is in contrast to most profiling tools for functional languages that are implemented in other languages.

Section 1.2 gives a short introduction to functional language profiling and Section 1.3 summarizes the advantages of applying data modelling to profiling tools. Section 1.4 describes the motivation that lead to the directions taken in the thesis. Section 1.5 summarizes the contributions made by the thesis, whilst Section 1.6 outlines the thesis structure.

1.2 Functional languages and profiling

It has long been argued that functional languages are clearer and more succinct than imperative languages. The side-effect free environment is one reason why
\[
f :: \text{String} \rightarrow \text{Bool}
\]
\[
f = \text{let } f' r (y::xs@(x:xs)) = f' ((x/=y) && r) \text{ xxs}
\]
\[
f' r (x:_ \text{ }) = r
\]
\[
f' r \text{ } [] = []
\]
in \text{f'} \text{ True}

Figure 1.1: The function \(f\) is a Haskell function that takes a string and returns true if no two adjacent characters are the same. Otherwise it returns false. It is tail-strict: it demands to see all its input before giving any output. Given a large enough input, this function will exhaust all heap space.

the \textit{extensional} properties of a functional program (what it computes as its result) are argued to be easier to understand than those of imperative languages. However, the \textit{intensional} properties of a functional program (how it computes its result) are often harder to understand than those of an imperative one. Where lazy evaluation and higher-order functions are employed, this weakness is compounded.

Without this information, writing \textit{efficient} functional programs can be hard \cite{66}. For example, the function in Figure 1.1 is theoretically correct but in practice may consume all resources on a machine and fail to complete.

Several authors \cite{96, 107, 10} have developed theoretical frameworks that allow the user to reason about the intensional properties of a program, using similar algebraic methods to those used to reason about extensional properties. Although progress has been made in this area, it has yet to be widely adopted and used in practical situations.

In contrast, practical profiling tools that gather information during a program execution have become important tools in writing functional programs. Heap profilers have been applied to a wide range of programs, including large applications, such as compilers \cite{91, 85}, and are now a standard part of all major Haskell systems. The use of parallel profilers has become an established part of writing parallel functional programs because of the natural demand to understand what happens during a program’s execution.

1.3 Profiling and data modelling

The aim of profiling tools is to provide a bridge between the programmer’s high-level view of the computation, and the low-level details of exactly what happens during a program’s execution. Most profiling tools are able to gather a lot of in-
formation about a computation, but often the user is unable to efficiently extract enough information to answer some questions about the computation. The raw data gathered by the tools is too long for direct use by the programmer, and the limited graphical views that can be generated by the profiling tools often display too little information. Yet for large computations the graphical displays are also too crowded.

This thesis shows how data modelling techniques are used to create tools that allow efficient extraction of information. The data model used by profiling tools is typically left implicit, usually lying on some ad hoc point between a naïve programmer’s view and the technical details of the computation. By making the data model explicit we are able to construct a tool that provides efficient extraction of all the data recorded by the profiling tool, using a query interface.

1.3.1 Data model refinement

The majority of profiling tools use a fixed data model. It may be too abstract to answer some questions, and at the same time too low-level to effectively answer simple questions. The core of this thesis shows how to implement data model refinement, so that a profiling tool can present a range of data models to the programmer.

1.4 Historical background

This thesis is divided into three parts, exemplifying the changes in direction made during the thesis’s progress. My original goal was to improve parallel functional programming by automatic parallelization of lazy functional programs. The work began by developing a simple tool that uses strictness analysis to place parallel annotations within a lazy functional program. Two different parallel profilers were used to assess the quality of the generated output (the level of parallelism achieved). They work in a similar way to most profilers: during a program run they gather information into a log-file that can be processed later to generate graphs. Both reported the level of parallelism in the program run, and could produce numerous graphical summarizers of the computation. However, neither were sufficient to answer the questions required. The main problems were:

1. There was a difference between the data recorded in the log-file and the data displayed in the graphs;

2. The log-files were too large to extract meaningful information by hand;

3. The graphical charts were too static – it was easy to find the positions where there were problems, but no way to find out more about those areas.
At this point it became clear that the best way of improving parallel functional programming was to improve the profiling tools, so our goals were changed to reflect this. We started by establishing exactly what was recorded in the log-files so that we could construct data models to describe the model of concurrent evaluation used by the tools. To compare and assess the data models, we constructed a chain of data models representing different levels of abstraction [17]. We then developed an interactive profiling tool [16] based on one of the data models. This tool provides easy access to the profiling data, and does not suffer the problems listed above. However, the development inspired two questions:

- **Why present only a fixed level of abstraction to the user?** — during the data modelling process we constructed a chain of data models, each representing different levels of abstraction, but only one model was selected for use in the interactive tool.

- **Can the techniques be extended to other forms of profiling?**

The second question was answered by making the interactive tool more generic. But how can the interactive tool be extended to handle queries over multiple abstraction levels? An extensive search of the literature showed that there was no previous work on the semantics of E-R data model refinement that would provide the aforementioned facility. This lead to the work that provides the core of this thesis: data model refinement.

### 1.5 Thesis contributions

This section lists the research contributions made in this thesis:

1. **Realizing data model refinement in the Entity-Relationship approach.** Although the Entity-Relationship approach is the de facto standard for designing databases, there has been little research into refinement of Entity-Relationship models. This thesis shows how to realize data model refinement. Given a chain of data models described using primitives, we show how to generate definitions to retrieve data relating to an abstract model, from a database based on a concrete model.

2. **Refinement primitives.** This thesis describes a set of primitives for defining E-R model refinements. The exact meaning of each primitive is defined and informal proofs show that the primitives generate refinements which are valid when certain pre-conditions are satisfied.

3. **Refinement combinators.** This thesis describes a combinator library for describing data model refinements in a concise and readable way.
4. *Generic profiling tool*. This thesis describes a generic profiling tool that does not suffer from problems found in most previous tools. The tool provides simple extraction of data through the use of queries, and allows the user to select the level of abstraction they are interested in.

5. *Data models for parallel profiling, heap profiling*. This thesis shows how to apply data modelling for two different types of profiling.

### 1.6 Thesis structure

This thesis is arranged into three parts, each representing a separate strand of research. Part I provides the motivating application: improving parallel functional programs. Part II is the core of the thesis and describes the technical development for data model refinement. Part III describes applications of data model refinement.

**I (Motivating application): Improving parallel functional programs**

**Chapter 2** provides the background for the motivating application: improving parallel functional programs. It begins by reviewing static methods for identifying and controlling possible parallelism, and follows with a discussion of current parallel profiling and the importance of profiling tools. It finishes with an example that highlights some of the weaknesses of current profiling tools, providing motivation for the subsequent chapters.

**Chapter 3** applies formal data modelling techniques to parallel graph reduction, suitable for use by parallel profiling tools. It discusses the merits of explicit data modelling, and derives the relational tables used in the interactive profiling tool described in Chapter 4.

**Chapter 4** describes an interactive parallel profiling tool that provides a query interface and dynamic graphs to provide easy extraction of information about a computation. This tool is extended in Chapter 9 to handle more generic data.

**II (Technical development): Semantic and functional implementation of data model refinement**

**Chapter 5** presents a framework to describe the refinement of data models. A set of top-down and bottom-up primitives are suggested, and their exact meaning is defined.
Chapter 6 presents a methodology for realizing refinement in E-R modelling. Refinements are described in terms of the primitives described in Chapter 5, and a rule set determines the mappings to retrieve an abstract value from a concrete value.

Chapter 7 describes the implementation of a view compiler that generates definitions, enabling queries to be submitted using any data model from a chain of data models. The compiler’s input is combinator expression and its output is determined by the rules described in Chapter 6.

III (Experimental evaluation): A generic profiling system

Chapter 8 evaluates the view compiler by applying it to both a form of parallel profiling and a form of heap profiling. The output of the compiler is integrated into the interactive profiling tool (described in Chapter 4) for the parallel example, and the extra functionality is evaluated.

Chapter 9 describes a generic profiling tool. It integrates the interactive tool described in Chapter 4 and the view compiler in Chapter 7 with the aim to produce a single tool for many different forms of profiling.

Chapter 10 draws conclusions from the earlier chapters, and discusses directions for further work.

Appendix A shows the different output generated by the view compiler for the parallel data model chain.

Appendix B shows the different output generated by the view compiler for the heap profiling by producer data model chain.

All source-code segments within this thesis are written in the standard lazy functional language, Haskell [75], unless otherwise stated. The quotes around strings are omitted in some examples for presentation purposes.
Part I
Motivating application

Improving parallel functional programs
Overview

This part of the thesis provides the motivation for the other two parts. Chapter 2 reviews some of the current methods for introducing parallelism into functional programs. After reviewing compile-time techniques for introducing parallelism, it introduces current profiling techniques, and finishes with a summary of the weaknesses of current parallel profiling tools. Chapter 3 shows how to apply data modelling techniques to parallel profiling tools. Chapter 4 describes a profiling tool based on a data model from Chapter 3, that does not exhibit the weaknesses of current profiling tools.
Chapter 2

Parallel functional programming

2.1 Introduction

This chapter explains the background behind the thesis’s motivating application: improving parallel functional programs. It reviews methods for introducing and improving parallelism in a lazy functional language.

2.1.1 The par and seq annotations

A number of different annotations have been suggested to identify parallelism in functional languages [84, 37, 79, 3]. We use Roe’s par and seq combinators [84], as used in Glasgow’s Parallel Haskell compiler and GUM system [103]. The par combinator takes two arguments. The first identifies a sub-expression for potential evaluation in parallel with the second argument, and it returns the second argument as its result. The seq combinator does the same as par but in sequence.

Example 1 One possible parallelization of \((3+4)*(9-2)\) is:

\[
\text{let } l = 3+4 \\
\quad r = 9-2 \\
\text{in par } l \ (\text{seq } r \ (l*r))
\]

The seq combinator ensures both arguments of \(*\) are evaluated before the function body. Without this the evaluation of the function application could immediately block demanding the value of \(l\). Only after \(l\) has been evaluated could \(r\) be evaluated; all parallelism is lost.

One advantage of using the par and seq combinator model is that the mapping of tasks onto a parallel machine, and the organization of task communication is implicit, and is handled either by the compiler or by the run-time system. Not
only does this ease the burden from the programmer but it means that programs are not necessarily tied to a single architecture.

2.1.2 Safety

One possible strategy for the introduction of parallelism is to evaluate all arguments in parallel in every function application. However, using a lazy evaluation strategy, arguments are only evaluated if and when required. For example, when lazily evaluating \texttt{const (3\*4) (9-2)} , the expression \( (9-2) \) is not evaluated\(^1\). As a function may not evaluate all its arguments, it is not always \textit{safe} to evaluate them all in parallel. We define our notion of safety:

An evaluation strategy is \textit{safe} if it never commits all resources to \( \bot \)-valued computations unless the semantics of the original expression is \( \bot \).

Two schemes exist for satisfying this safety condition. The first is \textit{conservative} evaluation, where arguments are only evaluated in parallel if they are definitely required. The second is \textit{speculative} evaluation [18, 13], where any computation can be executed in parallel. The safety condition is then satisfied by assigning lower priority to speculative tasks (tasks not necessarily required) than \textit{mandatory} tasks (tasks definitely required). If and when it is established that a speculative task is not required, its priority is set to zero. This method has the advantage that idle processors can be utilized to evaluate tasks that may be required. This has applications in searching where, we may want to traverse the whole search space in parallel, although only one answer is required. Unfortunately safe speculative computation has proved to be costly and complex, with non-trivial priority changing required at run-time [18]. For this reason this chapter only deals with conservative parallelism.

Section 2.2 reviews compile-time techniques for identifying possible parallelism. Section 2.3 discusses the problems of task granularity. Section 2.4 highlights problems experienced using parallel profiling tools by analyzing a simple example.

2.2 Identifying possible parallelism

A function is \textit{strict} if and only if it always needs the value of an argument. This is formalized; a function \( f \) with \( n \) arguments is strict in its \( i \)th argument if:

\[
\quad f \ x_1 \ x_2 \ \ldots \ x_{i-1} \ \bot \ x_{i+1} \ \ldots \ x_n = \bot
\]

\(^1\)The \texttt{const} function is part of the standard prelude and defined by: \texttt{const x y = x}.
where \( \perp \) represents a non-terminating expression. That is, \( f \) is strict in its \( i \)th argument if its result is undefined when its \( i \)th argument is undefined.

If a function is strict in an argument then it follows that the argument is definitely required. Therefore it is safe to *evaluate the argument in parallel with the function body.*

**Example 2** Given the following definitions:

\[
\text{times } x \ y = x * y \\
\text{const } x \ y = x
\]

*times* is strict in \( x \) and \( y \), and *const* is only strict in \( x \). So the following expression is safe:

\[
\text{let } x = 2 + 3 \\
\quad y = 5 + 4 \\
\quad z = z -- \text{ evaluates to } \perp
\]

in \( \text{par } x \ \text{par } y \ \text{times} (\text{const } x \ z) \ y \)

It is impossible to determine whether a function is strict in an argument in the general case since this is analogous to the halting problem for Turing machines [104]. The remainder of this section reviews *strictness analysis* techniques that use approximations to determine the strictness of a function.

## 2.2.1 Forward analysis

**First Order Programs**

Mycroft presented the seminal idea on strictness analysis by *abstract interpretation* [69, 70]. He set up an abstract domain, with two elements, \( \top \) and \( \perp \), such that \( \perp \subseteq \top \). This is called the 2-point domain. \( \top \) represents expressions that may or may not terminate and \( \perp \) represents expressions that definitely do not terminate. This is represented diagrammatically as follows:

\[
\begin{array}{c}
\top & \text{may or may not terminate} \\
\mid \\
\perp & \text{definitely does not terminate}
\end{array}
\]

The function \( \# \) takes an expression from the concrete domain to the abstract one. For example, we define the \( \# \) function for *if . . . then . . . else* as:

\[
\text{if}\# \ x\# \ \text{then} \ y\# \ \text{else} \ z\# = x\# \ \sqcap (y\# \ \sqcup z\#)
\]

\(^2\text{Actually, Mycroft did not use } \top \text{ and } \perp \text{ but instead used the notation } 1 \text{ and } 0.\)
where \( \sqcup \) and \( \sqcap \) return the least upper bound and greatest lower bound, respectively. The \( \# \) function is calculated for user-defined function using the following equivalence:

\[
(f \ x)^\# = f^\# \ x^\#
\]

**Example 3** What is the strictness of \( g \)?:

\[
g \ x \ y \ z = \text{if} \ (x = 0) \ \text{then} \ (y + z) \ \text{else} \ (x + z)
\]

Using the equivalence above we propagate the \( \# \) inwards to obtain:

\[
g^\# \ x^\# \ y^\# \ z^\# = \text{if} \ (x^\# =^\# 0^\#) \ \text{then} \ (y^\# +^\# z^\#) \ \text{else} \ (x^\# +^\# z^\#)
\]

The individual components are then reduced:

\[
= (x^\# \sqcap 0^\#) \sqcap ((y^\# \sqcap z^\#) \sqcup (x^\# \sqcap z^\#))
\]

\[
= x^\# \sqcap z^\#
\]

To determine whether \( g \) is strict in \( x \) we compute the following:

\[
g^\# \ \bot \ \top \ = \ \bot \ \sqcap \top
\]

\[
= \bot
\]

If \( x \) is undefined and \( y \) and \( z \) may terminate, then the result of \( g \) will also be undefined. Since \( g \) is monotonic it follows that \( g \) is undefined whenever \( x \) is undefined. We can similarly compute whether \( g \) is strict in \( y \) and \( z \):

\[
g^\# \ \top \ \bot \ = \ \top
\]

\[
g^\# \ \top \ \bot \ = \ \bot
\]

So, \( g \) is strict in \( x \) and \( z \) but not in \( y \); it is safe to evaluate the arguments \( x \) and \( z \) in parallel with the body of \( g \) but not \( y \).

**Recursive Equations**

Recursive functions are handled using *ascending Kleene chains (AKC)*. Mycroft identified that the least upper bound of an AKC of \( f^\# \) (an ascending sequence of approximations) gives a correct terminating expression for \( f^\# \). If:

\[
f^\# \ x^\#_1 \ x^\#_2 \ \ldots \ x^\#_n = F(f^\#, x^\#_1, \ldots, x^\#_n)
\]

then the AKC is defined by:

\[
f^{0^\#} x^\#_1 \ x^\#_2 \ \ldots \ x^\#_n = \bot
\]

\[
f^{k+1^\#} x^\#_1 \ x^\#_2 \ \ldots \ x^\#_n = F(f^{k^\#}, x^\#_1, \ldots, x^\#_n)
\]

where \( f^{i^\#} \) is the \( i \)th approximation of \( f \). The least upper bound of these approximations is called the fixed point (or fixpoint) and is defined as the point where:

\[
f^{n^\#} x^\#_1 \ x^\#_2 \ \ldots \ x^\#_n = f^{n+1^\#} x^\#_1 \ x^\#_2 \ \ldots \ x^\#_n \quad \text{(for any} \ x^\#_1, x^\#_2, \ldots, x^\#_n)\]
Higher Order Functions

Bunt, Hankin and Abramsky [34, 35, 11] handle higher-order functions by extending the abstract domain with interpretations of functions. For example, the abstract domain for the function space \( \alpha \rightarrow \beta \) is defined as \( (\lambda \, x. \bot) \subseteq (\lambda \, x. x) \subseteq (\lambda \, x. \top) \).

**Example 4** Is apply strict in its first argument, where apply is defined by?

apply :: (a \rightarrow b) \rightarrow a \rightarrow b
apply f x = f x

We substitute the first argument of apply\# with the bottom element of the abstract domain of the first argument:

\[
\text{apply}\# (\lambda \, x. \bot) \top = (\lambda \, x. \bot) \top = \bot
\]

so apply is strict in its first argument.

Non-flat domains

Wadler [106] shows how to handle non-flat domains such as lists, by extending the abstract domain. For example, the non-flat domain of lists of integers, is represented in the abstract domain by \( \top \in \), \( \bot \in \), \( \infty \) and \( \bot \), which are defined by:

\[ \begin{align*}
\top & \quad \text{— any finite list} \\
\mid & \\
\bot & \quad \text{— any finite list, some member of which is } \bot \\
\mid & \\
\infty & \quad \text{— any infinite list} \\
\mid & \\
\bot & \quad \text{— definitely does not terminate, } \bot
\end{align*} \]

where \( \bot \subseteq \infty \subseteq \bot \subseteq \top \).

Using the 4-point domain it is possible to capture two extra levels of strictness. If the following equation holds:

\[
f^\# \ x_1^\# \ldots \ x_{i-1}^\# \infty\ x_{i+1}^\# \ldots \ x_n^\# = \bot
\]

then the \( i \)th argument is \( \infty \)-strict (or tail-strict), and it is safe to evaluate the argument in parallel with its spine. If this equation holds:

\[
f^\# \ x_1^\# \ldots \ x_{i-1}^\# \bot \in \ x_{i+1}^\# \ldots \ x_n^\# = \bot
\]

then the \( i \)th argument is strict in both the head and the tail (\( \bot \)-strict) and so it is safe to evaluate the spine and all elements of the argument in parallel.
2.2.2 Backward Analysis

In forward analysis the process of determining the strictness of a function can be summarized as: construct the syntax tree for the function body; working from the leaves of the tree calculate whether an argument is strict; repeat for each argument. Backward analysis, on the other hand, propagates contextual information from the root of the tree. The advantage of this method is that the tree is traversed only once.

Projection based analysis

Johnsson [49] made the first attempt at backward analysis at about the same time that Mycroft presented the original work on forward analysis. However, it was less powerful. Later, Wadler and Hughes [108] presented a seminal paper on a projection-based analysis using the idea that context could be modelled by projections [98]. A continuous function $\alpha$ is a projection if:

$$\alpha \subseteq ID$$
$$\alpha \circ \alpha = \alpha$$

where $ID$ is the identity function (defined by $ID \ u = u$ for all $u$). In other words, projections only remove information from an object, and all this information is removed at once. The $ID$ projection specifies a context where its argument may or may not be required. To handle ordinary strictness, as defined in Section 2.2.1, they extend the 2-point domain with a new element $\bot$ (pronounced “abort”). This element is defined such that $\bot \subseteq \bot$ and all functions are strict in $\bot$. A projection $\alpha$ is then said to be strict if it requires a value more defined than $\bot$ (i.e. when $\alpha \bot = \bot$). Conversely, a projection $\alpha$ is non-strict when $\alpha \bot = \bot$. The strict projection $STR$ is then defined by:

$$STR \bot = \bot$$
$$STR \bot = \bot$$
$$STR \ u = u, \text{ if } \bot \subseteq u$$

We now define safety in this context: a function $f$ of $n$ arguments applied in an $\alpha$ context is $\beta$-safe in its $i$th argument if:

$$\alpha(f\ u_1\ u_2 \ldots u_n) \sqsupseteq f\ u_1 \ldots (\beta\ u_i) \ldots u_n$$

(2.3)

This is abbreviated to $f^i : \alpha \Rightarrow \beta$. So $f^1 : STR \Rightarrow STR$ if and only if $f$ is strict in its first argument. For example, the function const has the following properties:
So, if $\text{const}$ is called in a lazy context, then we do not know whether either its first or second arguments will be required, and therefore it is not safe to evaluate them in parallel. Alternatively, if $\text{const}$ is called in a strict context we know its first argument is definitely required (we still do not know about the second argument) and so we can evaluate its first argument in parallel with its function.

Wadler and Hughes show how to define projection transformers that take a projection for a function and return projections that can be safely applied to the arguments. This information can be used to determine whether an argument can be evaluated in parallel with its function body.

Launchbury and Baraki [57] have developed a projection-based backward analysis, which is able to deal with simple strictness analysis but which does not use $\forall$ (which they call the lifted domain). They use partial projections. An example of a partial projection for $\text{STR}$ is:

$$\text{STR } x = \langle \text{UNDEF} \rangle \text{ if } x = \perp$$

$$= x \text{ otherwise}$$

The advantage of using this representation is that the theory is more elegant, making it easier to analyse what is going on.

### Comparing backward and forward analysis

Since backward analysis takes contextual information into account it often achieves better results than forward analysis. For example, a forward analysis of the application sum (append x y) would conclude that x is $\perp$-strictly needed, and therefore it would only be safe to evaluate x to WHNF (weak head normal form) in parallel with sum (append x y). Conversely, a backward analysis would show a stronger property equivalent to x being $\perp_\varepsilon$-strictly needed\(^3\), allowing x to be evaluated to a greater degree. There are other advantages as well: the time to compute the results is significantly less using backward analysis; a result suggested by Hughes [42] and confirmed by Seward [99] where a forward analysis took 200 hundred times longer for some examples. This is mainly because in

\(^3\text{This is projection } ALL \text{ defined by:}\)

\[
\begin{align*}
\text{ALL} & = \text{NIL} \sqcup \text{CONS STR ALL} \\
\text{CONS } \alpha \beta \ u & = \text{STR } (u \sqcap \text{con} \ (\alpha \ (\text{head } u)) \ (\beta \ (\text{tail } u)))
\end{align*}
\]
a forward analysis all the points of a domain must be computed, whereas in a
backward analysis only those which are required are evaluated.

However, backward analysis cannot cope with higher-order functions. Hughes'
[41] solution is to mix the backward interpretation with a forward interpretation.
This seems to work reasonably well, although using the method described Section
2.2.1 it is still not able to capture the same higher-order strictness analysis
as is possible in forward analysis.

2.2.3 Other approaches

Burn’s thesis [12] develops evaluation transformers (inspired by wp-transformers
[21]) by combining a number of ideas from both abstract interpretation and an
early version of projection based analysis [40]. The evaluation transformer system
is able to cope with higher order functions by using the method in Section 2.2.1,
and uses the method in [1] to analyse polymorphic functions efficiently. However,
Wadler observed [106] that it is unable to capture the notion of head strictness
that is available by using a purely projection-based analysis (Hughes and Launch-
bury [43] show that forward analysis also cannot handle head strictness).

2.3 Granularity of parallelism

The previous section summarized different strictness analyses for identifying ex-
pressions that can be safely evaluated in parallel. However, when tasks are exe-
cuted on a parallel machine they have an associated communication cost. If the
task is too small in terms of computation time, then the cost may outweigh the
gain in parallelism.

Example 5 Consider the following naïve parallel \texttt{nfib} function:

\begin{verbatim}
nfib 0 = 1
nfib 1 = 1
nfib n = let n1 = n-1; n2 = n-2
          nf1 = par n1 (nfib n1); nf2 = par n2 (nfib n2)
          fstPlus = par nf1 (1 + nf1)
          in par fstPlus (fstPlus + nf2)
\end{verbatim}

If this program is run on a parallel machine where all \texttt{par} expressions are turned
into tasks, then \texttt{nfib 20} would result in over 40 000 tasks being created, of which
over half are simple arithmetic expressions. The costs of this number of parallel
tasks would easily out-weigh the gain in parallelism on any current architecture.
If the tasks are too large then there may be too few to maximize the level of parallelism. There has been extensive research in both the imperative and functional programming communities to improve task granularity. Loidl [61] reviews techniques for improving the granularity of lazy parallel function programs. A brief summary follows.

**Run-time methods:** The *evaluate-and-die* method of evaluation [94] generates asymmetric parallelism. Parent threads do not synchronize with child threads. If a child has not started to evaluate its sub-expression when the parent thread requires the result, the parent thread will evaluate the expression itself, and the child thread is never evaluated. Where there are many runnable threads this method can increase their granularity. It can also reduce the number of threads created when compared to the *fork-and-join* method of evaluation where the parent waits for the child to finish. Other methods include load based inlining [88, 81, 52] that uses the machine's load to determine whether a potential thread of computation is evaluated in parallel and lazy thread creation [68] (and variants [93, 24]) where tasks are created retroactively as processing resources become available.

**Compile-time methods:** Information gained from a complexity analysis can be used to completely remove any par-sites which are estimated to generate very small tasks or to add cost information to a parallel annotation. Complexity analyses for lazy functional languages are often a mixture of a strictness analysis and a complexity analysis for a strict functional language. Wegbreit [111] described the seminal work on automatic complexity analysis of strict *first-order* programs. Wadler [107] shows how strictness analysis can help analyse the time behaviour of lazy *first-order* programs. Sands [95, 96] shows how to handle lazy *higher-order* programs.

### 2.4 Parallel profiling tools

The strictness analyses summarized in Section 2.2 can help identify safe places to place parallel annotations within lazy functional programs. The granularity methods summarized in Section 2.3 may help to improve the granularity of tasks. However, what if the parallel program achieves little parallelism, or worse still, takes longer to run than the sequential version of the program? The programmer could add and remove different par-sites in the hope of improving the program’s performance, but this is a time consuming exercise and unlikely to produce significantly better results. It is this problem that inspired the development of parallel
profiling tools whose purpose is to summarize the events that take place during the execution of a parallel program.

### 2.4.1 Using profiling tools

This section illustrates the use of parallel profiling tools by summarising the analysis of a short parallel program using GranSim [31, 60, 62], the most advanced parallel profiling tool for Haskell [74]. GranSim’s use is similar to most parallel profiling tools for functional languages. A program is compiled for GranSim, to generate an executable. When this executable is run, a record of various events related to each parallel thread’s life-time, such as the start and end time of a thread, is recorded in a log-file. This log-file can then be processed to produce graphs using the various visualization tools that accompany GranSim.

Figure 2.1 shows a parallel Haskell program based on an example by Augustsson and Johnsson [3], that calculates the sum of the results of applying a naively defined Euler totient function to the numbers from one to nine. The placement of the par-sites has been determined by the strictness analysis technique in Section 2.2.1. Parallelism is introduced in four places:

1. within `sumEuler` there is a parallel annotation to evaluate `numbers`, the list from 1 to \( n \);

2. within `euler` there is also a parallel annotation to evaluate `numbers`;

3. the `sumEuler` function applies `parList`, a function which evaluates all the members of a list in parallel, to the list of `euler` applications from `euler 1` to `euler n`;

4. the `euler` function evaluates the spine of the list of relative primes in parallel with the calculation of the result.

GranSim is highly configurable and it is possible to test the performance of a program on imaginary machines with different characteristics. For the purpose of illustration, GranSim is configured to simulate a three-processor machine with no latency and no communication costs.

Figure 2.2 shows the activity profile for the program, using one of the graphs that can be generated by visualization tools. The bars show the numbers of running (dark grey), runnable (light grey) and blocked (black) threads during the execution of the program. There are seven regions where at least one processor is idle. This is manually indicated on the graph by the addition of our own annotations.
main = (print \(\circ\) show \(\circ\) sumEuler) 9

sumEuler :: Int \(\rightarrow\) Int
sumEuler n
= let eulerList = let numbers = [1..n]
    in par numbers (map euler numbers)
    in seq (parList eulerList) (sum eulerList)

euler :: Int \(\rightarrow\) Int
euler n
= let relPrimes = let numbers = [1..(n-1)]
    in par numbers (filter (relprime n) numbers)
    in par (spine relPrimes) (length relPrimes)

parList :: [Int] \(\rightarrow\) ()
parList = foldr par ()

spine :: [Int] \(\rightarrow\) ()
spine [] = ()
spine (_:xs) = spine xs

hcf :: Int \(\rightarrow\) Int \(\rightarrow\) Int
hcf x 0 = x
hcf x y = hcf y (rem x y)

relprime :: Int \(\rightarrow\) Int \(\rightarrow\) Bool
relprime x y = (hcf x y==1)

Figure 2.1: The \texttt{sumEuler} program (Version 0).
Figure 2.2: Activity graph for SumEuler, with annotations for regions where at least one processor is idle (added by hand).

Analyzing the whole example in detail is a lengthy process, so we limit the analysis to reasoning why there are idle processors within the first three regions.

Region (I)

Every parallel computation has an initial start-up period as it takes time to execute the code that creates the parallelism. The lack of parallelism in region (I) could be because:

- no par application has been evaluated yet;
- threads have not been distributed onto other processors;
- the expressions to be evaluated by the sparks created within this region are already evaluated.

It is not clear from the activity graph which of these explanations is correct. It is necessary to examine the beginning of the log-file:

<table>
<thead>
<tr>
<th>PE</th>
<th>Time</th>
<th>Event</th>
<th>Thread Id</th>
<th>Node/Spark Id</th>
<th>Optionals</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>START</td>
<td>0</td>
<td>0xbe420</td>
<td>[SiteName: Main]</td>
</tr>
<tr>
<td>0</td>
<td>4994</td>
<td>SPARK</td>
<td>0</td>
<td>0x309368</td>
<td>[sparks 0]</td>
</tr>
<tr>
<td>Line</td>
<td>Description</td>
<td>Address</td>
<td>Spark Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>---------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>PRUNED</td>
<td>0x309368</td>
<td>sparks 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SPARK</td>
<td>0x309398</td>
<td>sparks 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SPARK</td>
<td>0x3093e8</td>
<td>sparks 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>EXPORTED</td>
<td>0x309398</td>
<td>sparks 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ACQUIRED</td>
<td>0x309398</td>
<td>sparks 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>EXPORTED</td>
<td>0x3093e8</td>
<td>sparks 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>USED</td>
<td>0x309398</td>
<td>sparks 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>START</td>
<td>0x309398</td>
<td>[SiteName: parList]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To understand this portion of the log-file it is necessary to understand the distinction that GranSim makes between a spark and a thread. A spark is a task created by an evaluation of a par application, and may either be turned into a thread and begin running, or else may be pruned and discarded.

The log-file shows that three sparks are created before threads begin to run. Two are turned into threads (i.e. start to run) soon after being sparked, but one of the sparks is pruned. The log-file does not record the par-site that created the pruned spark. However, from a simple analysis of the evaluation order it is possible to see that the pruned spark is created by the par-site inside euler. This par-site causes the evaluation of numbers, taking less than 1/140th of total run-time to evaluate, so it is likely that, before the spark had chance to turn into a thread, the node had already been evaluated to normal form. This would cause the spark to be pruned. After this pruning two further sparks are created, they are exported onto different processors, and finally the parallelism begins. This confirms that the lack of parallelism in region (I) is due to the time taken to execute the code that produces parallel threads.

**Region (II)**

Within this region only two threads are running concurrently. From Figure 2.2 it is not clear why one processor is idle. A fuller insight can be gained by looking at the per-thread graph (see Figure 2.3), which shows the activity of each thread, rather than the number of running threads. Again, this graph is generated by the visualization tools. The thick grey lines show when the thread is running. The medium-width black lines show when the thread is runnable and the breaks in the bars show when a thread is blocked. The x-scale and extra annotations match the ones given in the activity profile.

This graph shows that the lull starts immediately after a thread has finished. No other thread begins running at the point at which the thread finishes, suggesting that there are no unused sparks or runnable threads. The current graphs produced by GranSim do not convey enough information to confirm this. To help our understanding of this example we produced an annotated version of the per-thread profile which we call the per-thread spark and block profile (Figure 2.4).
Figure 2.3: Per-thread graph for *SumEuler* 9, with annotations for regions where at least one processor is idle (added by hand).

Figure 2.4: Per-thread spark and block graph for *SumEuler* 9 with annotations identical to those in Figure 2.3.
This profile contains a lot more information, most of which can be extracted from current log-files\textsuperscript{4}. In this graph the following extra annotations have been introduced:

- **thread labels** – each thread is labelled on the y-axis with the par-site that created the thread;

- **processor numbers** – the number of the processor on which the thread is running is recorded at the beginning of the first bar for the thread (there is no thread migration so there are no numbers on subsequent bars);

- **spark lines** – for each spark that is turned into a thread, the spark’s creation time and the thread that created the spark are shown. These are indicated by the straight lines on the graph. The vertical lines point to the thread that creates the spark, and the horizontal lines point to the thread that the spark turns into;

- **pruned spark lines** – for each spark that is pruned, a horizontal line is placed on the thread that created the spark to indicate its creation and pruning time. A spark is pruned when its closure is evaluated by another thread;

- **block lines** – where a thread blocks, an arc connects the blocked thread (blockee) to the thread evaluating the closure the blockee is awaiting the value of (blocker). The line is curved to remind us that the blocker is not necessarily evaluating the node at the current time.

From this graph we can extract more information about the computation. For example, it is clear that the euler spark site produces considerably shorter computations than the parList spark site.

Within region (II) we see that, although there are suitable sparks, they are not residing on processor one, which is idle. A spark is soon exported from processor zero. When this arrives on processor one (immediately as there are no communication costs) the idle processor has some more work to do and so the level of parallelism picks up. The reason processor one ran out of sparks is self-evident. Following the work trail of processor one, we see that all the tasks exported to it are rather small and so the processor used up all its sparks before any other sparks are exported to it. In the optimal situation each processor has approximately the same amount of work as the other processors throughout execution.

\textsuperscript{4}Current log-files do not contain enough information to establish a link between the blockee and corresponding blocker. Also, the construction of the spark lines is only possible for some examples using the current log-files.
Region (III)

This region differs from region (II) because, although there are no runnable threads, there is a blocked thread waiting to be evaluated. Again it is only the per-thread spark and block profile that reveal what is happening. This graph clearly shows that the blocked thread is being blocked by the thread that created it. The computation attempted by the blocked thread has already begun. An examination of the source code shows that this is because the function euler (evaluated by the blockee) and function parList (evaluated by the blocker) will attempt the evaluation of the same computation at one point, namely the spine of relPrimes for a given euler function application. The per-thread spark and block profile confirms that the processor is idle during this period as it has no sparks or runnable threads. As soon as a spark is imported from another processor the parallelism starts up again.

Conclusions from the Example

Although this is a very small example, it highlights the problems often experienced when using the parallel profiling tools. A lot of manual analysis of the log-file and source-code is necessary in order to answer only a few questions. The activity graphs are able to give a very general view of the computation, but do not contain enough information to answer some questions. The per-thread spark and block profiles convey a lot more information than can be produced with previous tools, and it would not be too hard to develop a tool to produce these profiling graphs automatically. However, most examples have far more threads with far more activities, so the bars on a per-thread spark and block profile would be much too dense and unreadable — a scalability problem. The same is true for the normal per-thread graphs. The profiles could be split on to multiple pages, but this is unlikely to make them any more understandable and the compactness of a single-page profile is lost. The crux of the problem is that large quantities of information can be obtained from a parallel computation, whether it is held in a log-file or a graph.

2.4.2 Problems with current parallel profiling tools

These problems are not restricted to GranSim and are found in most parallel profiling tools for functional languages [84, 91, 32, 54]. We summarize these problems in the following list:

1. There is an inconsistency between the data recorded in log-files and the data displayed in graphs;

2. The log-files are too long for direct use by programmers;
3. The graphical display is often static and displays too little information.
   Yet, for large computations the graphs are too crowded. If the information
   needed to confirm some hypothesis is not displayed on any of the graphs,
   the programmer may have to resort to tedious analysis of the log-file.

The standard solution to this problem, used by a number of tools for imperative
parallel profiling [36, 83, 30], is to provide a wider range of graphical outputs
that can be generated. Although the different views of the data may help to
understand the data more, there is still no guarantee that user has convenient
access to all of the information recorded in the log-files.

2.5 Summary

This chapter has discussed a possible strategy for writing a parallel functional
program in a lazy language. It is hard to identify expressions that can be safely
evaluated in parallel in a lazy functional language, so we use strictness analyses.
It is also difficult to identify expressions that are worth evaluating in parallel
so methods to improve granularity are used. Finally, profiling tools help the
programmer to understand the behaviour of their program when it is evaluated.
Although profiling tools are good at gathering data relating to a computation,
efficient access to this data is missing in most tools.

The remainder of this thesis describes techniques for improving profiling tools.
We begin by examining data models for parallel graph reduction in Chapter 3.
Chapter 3

Data modelling parallel graph reduction

3.1 Introduction

Section 2.4.2 highlighted some inadequacies of current parallel profiling tools. There is a significant difference between what users need, what log-files record and what graphs show. One reason for this disparity is because no formal data modelling was done when these were developed. The model of concurrent evaluation is left implicit. It usually lies on some intermediate point between the naïve programmers view and the technical details of the computation. This chapter explores various data models for parallel graph reduction and explicates the models used by different parallel profiling tools. An explicit data model makes clear to the developer of the profiling tool the information that needs to be recorded, and to the user of the tool what the underlying model is.

Section 3.2 develops a set of data models for describing a parallel graph reduction system at different levels of abstraction. Section 3.3 compares the final data model in Section 3.2 to the data model implied by GranSim’s log-files and graph producing tools. Section 3.4 shows how a data model for GranSim can be converted into a form suitable for a relational database.

3.2 Data Modelling Parallelism

This section shows how to apply data modelling techniques to parallel profiling. For purposes of illustration we build up a model based on the features of GranSim. This is not because GranSim’s model is the right model. Indeed, one reason to make the model explicit is to assess its merits (see Section 3.3). Although we comment on the differences between different tools we do not attempt to synthesize an ideal model.
There are a number of ways to carry out conceptual data modelling. We use the most common: the Entity-Relationship (E-R) approach [112, 22, 56]. It does not provide the mathematical rigour of specification languages, such as Z [115], but it does provide a simple diagrammatic way of specifying a data model, which is convenient for both users and developers.

Starting with a simple model we introduce extra entities and relationships to reflect different parts of GranSim’s data model. Attribute details are omitted from the models in this section.

### 3.2.1 A Simple Parallel Graph Reduction Model

One simple view of parallel graph reduction has one main thread which creates further threads to do work in parallel. This model abstracts away from details of resources such as processors and memory, and is perhaps the one held by a naïve programmer who places par annotations anywhere in their program, hoping for parallelism. This model is naturally represented as an E-R diagram (see Model 1). In this diagram there is one entity which represents the threads, and a sparks relationship representing thread creation. A thread optionally sparks other threads; as indicted by the dashed arc connecting Thread. The relationship is one-to-many as a thread may spark many threads but a single thread can only be sparked by one thread.\(^1\)

![Model 1: A naïve view.](image)

![Model 2: Introducing blocking.](image)

When a thread tries to evaluate a shared data structure that is already being evaluated, the thread blocks until another thread has finished evaluating the expression. Using Model 1 it is not possible to ask any questions related to blocking or even to establish whether a thread blocks at all. A slight refinement of the original model is required to include this information, see Model 2. The blocksOn relationship represents a thread trying to evaluate an expression that is already being evaluated by another thread. One side of the relationship represents the thread that blocks (blockee). The other side represents the thread that evaluates the expression that causes the blockee to block (blocker).

It is possible to ask questions such as Does par-site p always create threads that block on thread t? This type of information can be used to identify bottlenecks.

---

\(^1\)The main thread is not really created by another thread so we take Main to be the unique fixed point of the function sparked by (inverse of sparks relation).
in the code, Runciman and Wakeling [91, 92] required information similar to this when profiling the soda program. So even a relatively simple data model can answer some questions that can be used to improve a program.

### 3.2.2 Introducing Time

Current profiling tools place a lot of emphasis on the order and duration of events. Log-files often record a time-stamp for each event recorded, and time is usually one of the dimensions in the graphs. There is already an implicit record of time in Model 2: a thread must be sparked before it can be blocked, and a thread can only spark a new thread when it has been sparked itself. Although this may be useful for determining the order in which threads are created, its abstraction level is too high to establish the duration of events. Knowledge of event duration is useful. For example, it helps to identify par-sites that create threads doing very little work. These par applications can in turn be removed in the hope of increasing the granularity of threads.

Model 3 incorporates a notion of time. In this model a thread can have many activities, where each activity is either running or blocked on the thread that is evaluating a shared expression. The horizontal bar under the Activity entity specifies that Run and Block are sub-types of Activity: they have all the attributes of Activity, plus some extra ones.

Model 3 is used both by HBC-PP [91], with an unbounded number of processors, and by GranSim-light². However, they each use different units of time. GranSim-light records time as an internal clock and HBC-PP records it as the number of reduction counts. These two methods of recording times can lead to

²A simplified version of GranSim with no processor limit and no communication costs.
very different results for exactly the same computation. So, even though HBC-PP and GranSim-light both fit the above data model, we cannot expect the same results when the same program is run on both systems. Even if they did use the same measure of time they still may give different results because the data model is only abstract and the concrete implementation details (e.g. scheduling policies) may be very different.

By making underlying models explicit for two different tools, it is possible to compare the differences between what data is implicit in the model, what is stored in the log-files, and what is currently accessible through use of the visualization tools. HBC-PP records all the information in the Model 3. GranSim-light does not record the data for the `blocks0n` relationship. None of the visualization tools that accompany GranSim or HBC-PP are able to show the `sparks` relationship or the `blocks0n` relationship. This highlights the gap between the data model used and the information that can be extracted using the high-level graphs.

### 3.2.3 Introducing Processors

Model 3 does not model a limit on the number of concurrent threads that can be evaluated at one time. A third activity, `queue`, is incorporated to model this behaviour. A thread queues when its not blocked but cannot run because the limit on the number of running threads has been reached. This queue is often called the queue of runnable threads. A slight refinement is made to incorporate this new state (see Model 4). In this model a thread may run, block or queue during a single activity period.

This is the model used by HBC-PP, where the number of processors are limited. Again, all the information required to answer any of the queries is recorded in the log-file, but the visualization tools do not present the `sparks` or `blocks0n` relationship in any way.

A limit on the number of concurrent running threads is one simple way to model a limited number of `processors`. This is particularly suited to a thread-oriented view of the computation. It is also possible to view the computation in a processor-oriented way. Model 5 extends the previous model to include an explicit representation of processors. The advantage of doing this is that it is now possible to ask questions about processor load and how well the sparks and threads are distributed across the processors.

### 3.2.4 Introducing Nodes

The goal of each thread is to evaluate some expression. When one thread blocks on another, the shared computation under evaluation by the blocker is not necessarily its `entire` expression but may be a sub-expression. Theoretically it is possible to
Model 5: Introducing processors. The thick lines represent $m : 1$ relationships between the Time and another entity. These reduce the size of the diagram.

Model 6: Introducing graph nodes.
record this distinction with the addition of an extra entity, `Node` (see Model 6), that records the sub-expression that the threads are trying to evaluate when they block. By doing this we enter the realm of tracing [100], which in practice would be too costly to integrate into a parallel profiling system. One compromise is to only record the cost-centre [32] of an expression.

### 3.2.5 Introducing spark/thread distinction, thread migration, fetching

The final model in this section attempts to capture the functionality of GranSim not exhibited in the previous models. GranSim is highly configurable. Its implicit data model depends on the options selected. In Model 7 we include:

- spark/thread distinction;
- spark migration;
- thread migration;
- fetching nodes;

in addition to the properties modelled previously.

![Diagram](image)

Model 7: Introducing spark/thread distinction, thread migration, fetching.

GranSim uses an *evaluate-and-die* mechanism to help increase the granularity of threads. *Sparks* and *threads* are different objects. A spark is created by a
thread and joins the spark pool on the processor evaluating the thread. This spark may then be pruned if its associated closure has already been evaluated. Alternatively the spark is turned into a thread which evaluates its associated expression. A task may be exported to other processors. This is essential: if it did not happen there would be no parallelism! The period between the time a spark is created and the time the spark is turned into a thread could be considered as an activity of a thread, and modelled in a similar way to the running, runnable and blocked states. However, the sparking period always happens at the start of a task's lifetime. To reflect this we treat sparking differently, as shown in the model.

Sometimes it is necessary for threads, as well as sparks, to migrate between processors [33]. GranSim explicitly fetches graph nodes from different processors when the current processor does not have access to the required node. Both of these actions are represented in Model 7 as new states of activity.

### 3.3 GranSim: data models, log-files and graphical charts

The previous section developed data models to describe the features implemented in GranSim. This section constructs data models describing the exact data recorded in both the log-files and graphs. One reason for doing this is so that we can assess the quality of the implicit data models.

#### 3.3.1 Modelling the log-file

Model 7 (in Section 3.2.5) captures features provided by GranSim, but Model 8 describes the exact data stored in the log-files.

Why are these models different? Firstly, we know what data items are stored in the log-file, so Model 8 explicitly identifies the attributes of the entities. Secondly, there is not enough information in the log-file to construct the blocksOn relationship or the becomes relationship.

The blocksOn relationship was introduced in the second data model, Model 2. Even using this simple data model it may be possible to extract information to improve programs. It can answer the question *Does par-site p always create threads that block on thread t?*, which generates similar information to that required by Runciman and Wakeling [91, 92] when profiling the sodah program. As this relationship cannot be extracted from the log-file we cannot answer this question. It is only possible to determine that a thread blocks, and not what causes the thread to block.
Model 8: The data stored in GranSim log-files.

The becomes relationship also cannot be extracted from the log-file, which means it is not possible to relate a thread to its associating spark. This severely restricts the class of queries that can be answered. It is not possible to determine what thread created a particular thread, and we cannot establish the par-sites that generate the sparks.

GranSim does not include these two relationships because they were not needed for their purposes. By explicit data-modelling we are able to show what queries can and cannot be answered. This information can be used to improve profiling tools so that they are more usable to a greater number of users.

3.3.2 Modelling the graphs

Model 9 is a data model describing all the data that can be extracted from both the activity and per-thread graphs. This is a much simpler model containing no information about sparks or explicit processors.

3.4 Conversion into relational database form

One purpose of E-R modelling is for database design. This section defines suitable relational database tables for constructing a database from GranSim log-files. For this we use Model 10. This is different from the log-file model (see Model 8) in the following ways:
Model 9: The data represented in GranSim graphs.

\[
\text{THREAD(} \text{ThreadId, ThStart, ThEnd, becomes*)} \\
\text{SPARK(} \text{SparkId, ParSite, CreatedAt, sparks*)} \\
\text{PROCESSOR(ProcessorNo)} \\
\text{ACTIVITY(} \text{ActivityNo, AcStart, AcEnd, Selector, has*, hosts*)} \\
\text{LIEON(} \text{LieOnId, LieFrom, LieTo, LieOnP*, LieOnS*)} \\
\text{RUN(} \text{ActivityNo*); QUEUE(} \text{ActivityNo*);} \\
\text{BLOCK(} \text{ActivityNo*); MIGRATE(} \text{ActivityNo*);} \\
\text{FETCH(} \text{ActivityNo*);}
\]

Figure 3.1: Relational tables for a database using data generated by GranSim.

- The becomes relationship is included in the model, and the Spark entity is attributed with the par-site rather than the Thread entity. The previous section highlighted the arguments for recording this information, so we include this even though modifications are required to GranSim.

- The logical entities representing time and connecting relationships are replaced by attributes. We do this as the final database will be more efficient.

- The primary keys of the entities are identified where appropriate.

The format of the final tables is shown in Figure 3.1. The fields emphasized in bold are the primary keys, and the * represent foreign keys. We now describe the methodology for generating these table formats. We use a standard approach found in the literature [7].
Model 10: Model suitable for constructing a database.

1. Identify primary keys and create intentional description.

**Example 6**  
\[ \text{PROCESSOR}(\text{ProcessorNo}); \]  
\[ \text{THREAD}(\text{ThreadId}, \text{ThStart}, \text{ThEnd}) \]

2. Translate hierarchies (copy primary key of parent to child entity, add selector attribute to the parent)

**Example 7**  
\[ \text{RUN}(\text{ActivityNo}^*); \text{QUEUE}(\text{ActivityNo}^*); \]  
\[ \text{ACTIVITY}(\text{ActivityNo}, \text{AcStart}, \text{AcEnd}, \text{Selector}) \]

3. Map the relationships:

(a) Many-to-one relationships, mandatory in the many direction (add the primary key of the master entity to the detail entity)

**Example 8**  
\[ \text{has: ACTIVITY}(\text{ActivityNo}, \text{AcStart}, \text{AcEnd}, \text{Selector}, \text{has}^*) \]  
\[ \text{lieOnP: LIEON}(\text{LieOnId}, \text{LieFrom}, \text{LieTo}, \text{LieOnP}^*) \]  
\[ \text{lieOnS: LIEON}(\text{LieOnId}, \text{LieFrom}, \text{LieTo}, \text{LieOnP}^*, \text{LieOnS}^*) \]

(b) Contingent One-to-one (add the primary key of the optionally related entity to the other entity)
Example 9

becomes: \textit{THREAD(ThreadId, ThStart, ThEnd, becomes*)}

The relational tables in Figure 3.1 are in third normal form (3NF).

3.4.1 Expressing queries using SQL

SQL [47, 72] is the \textit{de-facto} standard database query language (see [67] for an introduction). As the database tables are constructed from Model 10, this model is suitable for constructing the paths of queries.

Example 10 To ask Which thread \textit{t1} created thread \textit{t2}? we use the following path through the E-R model:

\[
\begin{align*}
\text{t2} & = \text{THREAD(ThreadId)} \quad \text{becomes} \quad \text{SPARK(SparkId)} \quad \text{sparks} \\
\text{RUN(ActivityNo)} & \quad \text{activity} \quad \text{ACTIVITY(ActivityNo)} \quad \text{has} \\
\text{THREAD(ThreadId)} & = \text{t1}
\end{align*}
\]

which corresponds the following SQL query:

\[
\begin{align*}
\text{select} & \quad \text{distinct} \quad t1.\text{ThreadId as creator, t2.ThreadId as created} \\
\text{from} & \quad \text{Thread t1, Thread t2, Spark, Run, Activity} \\
\text{where} & \quad t2.\text{becomes = Spark.SparkId} \\
& \quad \text{and Spark.sparks = Run.ActivityNo} \\
& \quad \text{and Run.ActivityNo = Activity.ActivityNo} \\
& \quad \text{and Activity.has = t1.ThreadId}
\end{align*}
\]

3.5 Summary

This chapter has presented a number of data models describing parallel graph reduction. Simple data models are sufficient to answer to some queries, but more complicated questions require more complex models. By explicating the data models we have been able to evaluate and compare different profiling tools. Finally, we have shown how to derive database tables for storing the data that is recorded in GranSim log-files. Chapter 4 describes a tool which integrates a query-interface based on these tables.

Part II develops the framework for describing and implementing data model refinement, and Chapter 8 shows how this is applied to the chain of data models described in this chapter.
Chapter 4

Interactive parallel profiling

4.1 Introduction

Section 2.4.2 highlighted some of the inadequacies of previous parallel profiling tools:

1. There is an inconsistency between the data recorded in log-files and the data displayed in graphs;

2. The log-file is too long for direct use by programmers;

3. The graphical display is often static and displays too little information.

Yet, for large computations the graphs are too crowded. If the information needed to confirm some hypothesis is not displayed on any of the graphs, the programmer may have to resort to tedious analysis of the log-file.

Chapter 3 developed explicit data models for parallel profiling and confirmed that the first point is true for the tools accompanying GranSim. This chapter describes an interactive tool for extracting information about a parallel computation in a structured way. A query interface is provided so that a user can extract the full or selected details of a computation. All results can be displayed in tabular form and some query results can be displayed as a graphical chart. Parameters of subsequent queries can be specified by selecting points within a previous chart.

Section 4.2 summarizes the tool’s functionality. Section 4.3 shows the tool being applied to an example. Section 4.4 summarizes the implementation. Section 4.5 discusses the tool’s performance and limitations. Section 4.6 summarizes related work.

4.2 Tool Overview

This section summarizes the functionality provided by an interactive tool for analyzing profiling data generated by GranSim. The tool’s purpose is to provide
Figure 4.1: The graphical interface of the interactive tool. The two main components are the graph viewer and query editor. The query editor is showing the template for the activity graph.

the functionality needed to extract any information recorded in a log-file without the problems highlighted above. As such the design is influenced by the following aims:

- It is clear to the programmer which data model is being used;
- It is possible to extract any data present in the log-file;
- The user can control exactly what data is extracted;
- Graphical summaries of the computation can be produced.

Figure 4.1 shows the main interface to the interactive tool. It presents a query editor and graph viewer. A summary of the tool’s features follows.
4.2.1 Using queries to extract information

Information about a parallel computation can be extracted using SQL [47, 67, 72] queries, with results displayed in a tabular form. GranSim’s model of parallel computation is described by Model 10 in Section 3.4, and Figure 3.1 defines the corresponding format of the database tables for use in the queries. The user can submit any valid SQL query: no restrictions are made to the queries other than those present in the database system used (see Section 4.4).

4.2.2 Using graphs to extract information

Since the activity and per-thread graphs, produced by the tools accompanying GranSim, have been useful for improving some parallel programs [63, 51], the interactive tool can display similar graphs within a graphical window.

4.2.3 Using graphs to compose queries

Regions of a per-thread graph can be clicked on to help complete parts of a query. The smallest component in a per-thread graph is a bar representing a period of activity (e.g., a period of running). Each of these bars corresponds to a single record in the Activity table. As a record is uniquely identified by its primary key, clicking on one of the bars returns its associating ActivityNo (the primary key of Activity).

Example 11 Given the Activity table with the following entries:

<table>
<thead>
<tr>
<th>ActivityNo</th>
<th>AcStart</th>
<th>AcEnd</th>
<th>Selector</th>
<th>has(ThreadId)</th>
<th>hosts(ProcessorNo)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>Run</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>Block</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>8</td>
<td>Run</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>7</td>
<td>Run</td>
<td>b</td>
<td>1</td>
</tr>
</tbody>
</table>

we can produce the mini per-thread graph in Figure 4.2. Each of the four bars represents one of the records in the table. If the light bar is clicked the number ‘2’
Per-Thread graph template:

```
SELECT a1.AcStart, a1.AcEnd, a1.Selector, a1.has, a1.ActivityNo
FROM Activity a1
WHERE
GROUP BY
HAVING
ORDER BY a1.has, a1.AcStart
```

Figure 4.3: Query template for generating a per-thread graph. The shaded fields are locked and cannot be modified. To remove duplicate rows, SELECT is translated into `SELECT DISTINCT` before submitting to the database engine.

...is sent to the query editor. The user then completes the query. So, the following query returns information about sparks created during the period that the bar represents:

```
select distinct Spark.* from Spark, Activity
  where Activity.ActivityNo = 2
  and Activity.AcStart <= Spark.CreatedAt
  and Activity.AcEnd >= Spark.CreatedAt
```

### 4.2.4 Using queries to generate graphs

Figures 4.1 and 4.3 show the SQL queries that generate the two types of graph. By modifying templates the user can choose exactly what to display. Some of these modifications generate data in an incompatible format for building the graphs, so some of the fields are locked. For example, the per-thread graph template only allows the user to add to the `FROM` and `WHERE` clauses. The rest of the fields are locked. Even with these restrictions, it is possible to create a highly complex filter.

**Example 12** We can use the following query to display a per-thread graph for threads created on processor 0 by `parList`, that last longer than the average thread and never block:
4.3 An Example

The sumEuler program in Figure 4.4 is used to illustrate the functionality of the tool. This program is derived from the sumEuler program in Figure 2.1. Changes are made to reflect the analysis in Section 2.4.2, and the problem size is increased to calculate the Euler totient function from one to one hundred. Parallelism is now introduced in two places within the program:

1. the sumEuler function applies parList, a function that evaluates all the members of a list in parallel, to the list [euler 1, ..., euler n];

2. the euler function evaluates the spine of the list of relative primes in parallel with the calculation of the result.

4.3.1 GranSim graphs

When the sumEuler example is run on a simulated 32 processor distributed memory machine with fairly low latency, GranSim generates a log-file and reports a run-time of 281954 cycles. This figure alone is fairly meaningless. Traditionally the log-file is processed using the tools that accompany GranSim to produce Postscript™ graphs. The interactive tool displays similar graphs but in an X-Window as illustrated in Figures 4.5 and 4.6.
main = (print ∘ show ∘ sumEuler) 100

sumEuler :: Int → Int
sumEuler n
  = let eulerList = map euler [1..n]
    in seq (parList eulerList) (sum eulerList)

euler :: Int → Int
euler n
  = let relPrimes = filter (relprime n) [1..(n-1)]
    in par (spine relPrimes) (length relPrimes)

parList :: [a] → ()
parList = foldr par ()

spine :: [a] → ()
spine [] = ()
spine (_:xs) = spine xs

hcf :: Int → Int → Int
hcf x 0 = x
hcf x y = hcf y (rem x y)

relprime :: Int → Int → Bool
relprime x y = (hcf x y==1)

Figure 4.4: The sumEuler program (Version 1).

Figure 4.5: Activity graph for Version 1 of the program. The bands from bottom to top represent running, runnable, fetching, and blocked tasks.
Figure 4.6: Per-thread graph for Version 1 of the program. The lifetime of each thread is represented by a horizontal line, shaded grey when the thread is running, light-grey when it is runnable and black when it is blocked.

### 4.3.2 Using queries to extract information

The activity graph for sumEuler shows that not all the processors are used during the whole computation. To find out why, we begin our investigation by asking how much parallelism each par-site generated. This simple question is hard to answer using current tools: the programmer has to search through the log-file, which in this case is 2876 lines long! This question can be asked in the interactive tool using an SQL query. To ask one interpretation of the question *How much parallelism did each par-site generate?* the programmer enters the following query:

```sql
SELECT DISTINCT Spark.ParSite, AVG(Activity.AcEnd - Activity.AcStart) 
FROM Activity, Spark, Thread, Run 
WHERE Spark.SparkId = Thread.becomes 
AND Thread.ThreadId = Activity.has 
AND Activity.ActivityNo = Run.ActivityNo 
GROUP BY Spark.ParSite
```

In plain English: *What is the average length of time of all the running periods, for threads created by each par-site?* The answer is shown in Figure 4.7.

The par-site inside the euler function generates smaller threads than the other par-site and main thread. The cost of creating a thread and controlling communication can be high. Where a thread does not perform much evaluation, the costs can outweigh the benefit of parallelism. To test whether this is the case in this example we remove the par-site within the euler function.
Figure 4.7: Average run-time for each par-site. The left column contains the par-site names: 0 is the main thread; euler refers to the par-site inside the euler function, and similarly for parList. The right column shows average length of running periods for each par-site.

\[
euler :: \text{Int} \rightarrow \text{Int} \\
euler n = \text{length} \ (\text{filter} \ (\text{relprime} \ n) \ [1..(n-1)])
\]

Call this Version 2 of the program. Using GranSim with the same set up as the previous run, the run-time drops by more than 30% to 182791 cycles; so the par inside the euler function was wasteful.

### 4.3.3 Using graphs to extract information

Figure 4.8 shows the per-thread graph for Version 2 of the program. The dominant solid grey shading reassures us that the threads spend most of their time running. Life-times of the threads increase over time. This means at the beginning of the execution all the processors were hungry for tasks, most of which resulted in little computation. Is it better if some of the larger tasks were created at the beginning of the execution, so that the processors’ idleness is distributed across the computation? One way to do this is to reverse the order of the list of euler function applications. In Version 3 of the program sumEuler is defined accordingly:

\[
\text{sumEuler} :: \text{Int} \rightarrow \text{Int} \\
\text{sumEuler} \ n = \text{let eulerList} = \text{map euler} \ [n,n-1,..1] \\
\quad \text{in seq} \ (\text{parList eulerList}) \ (\text{sum eulerList})
\]

The reader may wonder why creating larger tasks at the beginning and small tasks at the end could be an improvement. The simplest way to understand this is to look at a mini per-thread graph. If we assume that a program creates four tasks, lasting 1, 1.5, 2, and 2.5 ticks respectively, and the main thread takes 0.5 ticks to evaluate the expressions that create the parallelism, then we would get per-thread graphs (omitting the main thread) resembling those in Figure 4.9.

Running Version 3 of the program results in a run-time of 158497: 15% faster than Version 2, and 45% faster than Version 1. This part of the analysis has not
used any novel feature of the interactive tool, but it serves to illustrate the usefulness of supporting the traditional graphs in connection with the query interface.

### 4.3.4 Using graphs to compose queries

Figure 4.10 shows the activity graph for Version 3 of the program. During the first quarter of the computation the level of parallelism rises steadily. However, in the period that follows, a drop in parallelism is followed by an increase, and then another peak and trough.

So, why do we get the two drops in the level of parallelism? The graph in Figure 4.11 is the per-thread graph for the same run. To work out why we get the tidal effect we use an important feature of the tool: the ability to **click on regions of the graph to help complete a query**.

Figure 4.11 shows us that there are no *runnable threads* during the drop in parallelism, but we do not know whether there are any *unevaluated sparks*\(^1\) waiting to be evaluated.\(^1\)

---

\(^1\)When a `par` application is evaluated a `spark` is created to evaluate an expression in parallel.
Figure 4.10: Activity graph for Version 3 of the program.

ing to be evaluated, and if so, on what processors these sparks lie. To answer this question we click on the end of one of the threads on the graph at a point where the level of parallelism is low. A reference to this point is sent to the query editor (shown in Figure 4.11 as the highlighted 72). We fill in the rest of the query to ask how many sparks lie on the different processors at the point we clicked. Submitting this query returns the table:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
</tbody>
</table>

There are 25 sparks but they all lie on processor 0. Clicking on other areas with low parallelism results in much the same table being produced. This information holds the key to understanding what is happening: one reason for the drop in parallelism is because all the threads terminate at approximately the same time. This would cause no problems if each of the idle processors had some sparks waiting to be evaluated, but because all the parallelism is sparked by the main thread, all the sparks lie on the processor which is evaluating the main thread. It takes time for each of the processors to obtain more work. As soon as each of the processors imports a spark the level of parallelism picks up again.

Had there been no limit on the number of processors, the peak and trough effect would not have emerged. The activity graph for an unlimited processor machine would be approximately the same as the activity graph for a 32 processor machine with the peaks stacked on top of each other. This would make the per-

When this evaluation begins, the spark is turned into a thread.
thread graph look similar to the right-hand mini per-thread graph in Figure 4.9. The lack of parallelism in some regions of the graph is due to the ineffective distribution of sparks across the processors. The task of working out how to distribute the sparks more effectively is left as an exercise for the reader.

### 4.3.5 Using queries to generate graphs

One feature of the tool not explored so far is the ability to use queries to select what information is displayed in a graph. Selective display is useful because a display of the full graph is too crowded.

The very first question we asked in Section 4.3.2 was, *How much parallelism does each par-site generate?*. This was answered using a query, whose result was displayed in a table. An alternative way to answer this question is to filter the information so that only the data for selected par-sites is displayed. Figure 4.12 shows the per-thread graph for only threads created by par-site parList. A simple query (shown in the query editor) filtered the information displayed. A similar query is used to display only the threads created by par-site euler, for which the threads are considerably shorter.
Figure 4.12: Per-thread graph for Version 1 of the program, filtered to display only threads created by \texttt{par-site parList}.

Figure 4.13: Structure of the interactive tool.

4.4 Implementation

Figure 4.13 shows the five main components of the interactive tool.

GranSim

GranSim compiles and runs the parallel programs to produce a log-file.

Builder

The Builder takes a GranSim-generated log-file and processes it to produce relational tables. There are four phases to building the relational tables. All but the first are implemented in Haskell.
1. Remove redundant data and sort – GranSim log-files are quite verbose and contain a lot of redundant data. We use the UNIX utilities awk and sort to provide quick removal of redundant data and to sort the file.

2. Parse the log-file.

3. Create relational tables from the parse tree.

4. Submit tables to the database engine using the GreenCard pre-processor [77] to generate the routines that communicate with the database server.

DBMS

To save on production time a popular database server is used to process queries. This has the advantage that file storage and implementation of clever algorithms to retrieve the data efficiently, are left to the database server.

The builder and query evaluator (both Haskell applications) communicate with the database server using a C API and GreenCard 2 [77] – a foreign language interface pre-processor. PostgreSQL [114], a freely distributable relational database system supporting a fairly large subset of SQL, is used as the database server. Originally the Mini-SQL database engine was used but bugs effectively prevented any use of indexes. One surprising thing was the ease of converting from one database server to another. Although they were designed around the C API that accompanies Mini-SQL, it took no more than a couple of hours to make the changes required to use PostgreSQL. One reason for this was due to the similarities in the C API’s between both database servers. However, the main reason was because the GreenCard functions integrate easily into Haskell. Changes to the C API meant little or no changes to the Haskell code, and only moderate changes to the GreenCard sources.

Query Evaluator

The query evaluator exchanges messages between the GUI and the database server. It communicates with the GUI through socket streams, implemented using GreenCard [77], and with the database server using PostgreSQL’s C API and GreenCard.

GreenCard was found to be very suitable for interfacing with the database server. It was useful to get Haskell to garbage collect C data structures automatically, and convenient that side-effect free C functions could be treated as pure functions in Haskell. For example, consider the function getValue:

2If HBC (the compiler used for the GUI) had a GreenCard pre-processor, then this part of the system would not have been required as the GUI could have communicated directly with the C API.
\%fun getValue :: PG_Result_Ptr \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{String} \\
\%call (pg\_Result\_Ptr x) (\text{int} \ col) (\text{int} \ row) \\
\%code res = PG\_getValue(x,\text{col},\text{row}); \\
\%result (\text{string} \ res)

This function takes a pointer to a database result table, a column and row number, and returns the value at that position by calling the C function \text{PG\_getValue}. To return the whole table we can use the function \text{getTable}:

\text{getTable} :: PG\_Result\_Ptr \rightarrow [[\text{String}]]
\text{getTable} \text{ptr} = \text{map} \ \text{fetchRow} [1..(\text{numRows} \ \text{ptr})]

where

\text{fetchRow} \text{n} = \text{map} (\text{getValue} \ \text{ptr} \text{n}) [1..(\text{numFields} \ \text{ptr})]

As \text{getValue} is a pure function (because \text{PG\_getValue} is side-effect free) the result of \text{getValue} is only translated into Haskell form on demand. So given an expression such as \text{(getTable \text{ptr} !! 3)} that returns the fourth row of the table, only the entries in the fourth row are translated into Haskell. As some tables returned from SQL queries are very large this saves a lot of time and space.

One problem experienced with GreenCard is that there is no standard way to access Haskell file handles within C across different Haskell implementations. So there is no portable way to read/write a file opened in C using the standard Haskell \text{readFile} and \text{writeFile} routines. This also seems to be the case for other foreign language interfaces to Haskell [23].

\textbf{GUI}

\textbf{Implementation overview:} With the aim to implement as much in Haskell as possible the GUI is also a Haskell application. The Fudget Library [28] was chosen to provide the graphical functions because it is both a purely functional solution, and had previously been used to write substantial applications including a web browser [14].

Fudgets are stream processors with a high-level and low-level stream. The low-level stream is connected to the I/O system and is usually transparent to the programmer. Fudgets communicate by sending and receiving messages along high-level streams connected by the programmer.

The GUI is implemented as a parallel composition of fudgets, wrapped in a loop (so that the fudgets can communicate with each other):

\text{mainF} = \text{loopThroughRightF \ handler \$}

\hspace{1cm} \text{graphOutputF} \gg\gg \quad \text{-- Display graphs}

\hspace{1cm} \text{graphInputF} \gg\gg \quad \text{-- Handle input from query evaluator}

\hspace{1cm} \text{sqlEditorF} \gg\gg \quad \text{-- SQL editing box}
When the user clicks on a button to evaluate a query `buttonsF` handles the request, passing an appropriate message to `graphInputF` that sends a request to the query evaluator. When the query evaluator returns the result it is passed to `graphOutputF` for display as a graph or table.

**Communication between fudgets:** In a parallel composition of two fudgets (e.g. `fud1F` <<< `fud2F`) an incoming message is sent to the appropriate Fudget by tagging the message with `Left` or `Right`. As anyone who has written a program using Fudgets will know, when there are several fudgets in a parallel composition the tags become long and can be confusing to program. For example, to send a message to `sqlEditorF` above, the tag `Right ◦ Right ◦ Left` is used. As our program increased in size, arranging communication between fudgets became tricky. After this problem was discussed with Thomas Hallgren, he wrote a new fudget to arrange communication between \( n \) fudgets using \( n \) one-level tags.

Fudgets are composed in parallel with a new combinator `>>>`. This combinator extends the standard parallel combinator `>>>` in addition to the composite fudget itself it returns a dictionary of one-level tags that can be used to communicate with the component fudgets directly. For example:

```
TagF combF insideH
   (toGraphOutputF :&: toGraphInputF :&: toSqlEditorF :&: ...) =
   tagGraphOutputF >>> tagGraphInputF >>> tagSqlEditorF >>> ...
```

In this definition the `TagF` structure has three components: the composed fudget `combF`; `insideH`, which is required but whose role is not of concern to us; and a dictionary of message tags (toGraphOutputF :&: toGraphInputF :&: ...) whose structure matches the parallel composition on the right-hand side. Now to send a message to `sqlEditorF` the one-level tag `toSqlEditorF` is used. Use of this new fudget increased the speed of development and made the source-code much more understandable.

### 4.5 Evaluation

**Speed:** One of the first questions potential users may ask is *How fast is it?*, especially as the tool is written in Haskell. One of the aims of the tool is to speed up the development of parallel programs. If the tool was considerably slower than previous tools then this aim might not be achieved. Table 4.1 gives the time in
<table>
<thead>
<tr>
<th>Method</th>
<th>Compile</th>
<th>Run</th>
<th>Process data</th>
<th>Run browser</th>
<th>Display graphs</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current</td>
<td>21.8</td>
<td>1.0</td>
<td>12.3</td>
<td>1.0</td>
<td>6.0</td>
<td>41.8</td>
</tr>
<tr>
<td>New Tool</td>
<td>21.8</td>
<td>1.0</td>
<td>31.0</td>
<td>7.0</td>
<td>7.1</td>
<td>67.9</td>
</tr>
</tbody>
</table>

Table 4.1: Time (in seconds) taken to compile, run, and display both the activity graph and per-thread graph using GranSim’s tools and our tool.

seconds taken to compile, run, process, and display both the activity graph and the per-thread graph for the **sumEuler** example using:

1. The tools that accompany GranSim

2. The interactive tool described in this chapter.

Each time is a minimum over three runs. Each experiment was performed on a 270 MHz Sun Ultra 5 Server, 192MB of RAM, GUI running with 25MB heap, Query evaluator with 500KB heap.

The table shows that the interactive tool takes longer to complete the measured tasks than GranSim’s tools. Of all the tasks, building the database takes the most time. A closer examination of the builder reveals that index creation is the main reason for this. Indexes are created for every primary key in every table. Some of these tables are rarely used so could be removed. However, even with the indexes as they stand, the time taken to process the data is not necessarily longer in the interactive tool. Table 4.1 only shows the times taken to process the data for two graphs. The interactive tool processes the data only once. Other tools need to process the data every time a new graph or view of a graph is required. So, a more realistic estimate of the time taken to process the data for this example is \((q \times 12.3)/2\) seconds, where \(q\) is the number of queries. The time taken to process the data using our tool is constant, 31.0 seconds. If more than five queries are required then for this example our tool will be faster.

---

David King has written faster versions of these tools in C rather than Perl and Bash as part of the APSET project [53], but our measurements are against the standard tools.

It was necessary to measure the *real* times rather than the *user* and *system* times because otherwise any work performed by the database server would not have been accounted.
Space usage: One problem we have found with the GUI part of the tool is that it is sometimes necessary to allocate a very large heap. The main reason for this is that some of our code is too strict.

Composing queries: Writing SQL queries can be laborious. Sometimes this is because of the poor language facilities provided by SQL and a more concise functional query language may help [71]. However, a lot of the time it is because the data model is more complex than required. Chapter 9 describes a generic version of the interactive tool that uses the data model refinement work described in Part II to provide the user with a selection of data models of which queries can be composed.

4.6 Related work

The most similar work is that of Lei et al. [59]. They have implemented a performance tuning tool for message-passing parallel programs. They also use a database query system but with a lower-level data model, designed mainly to extract communication and synchronization costs. The results of a query can be displayed in either tables or graphical visualizations. However, one of the most important features of the interactive tool, the ability to interact with the graphs to help fill in the queries, is not implemented in their tool. One interesting feature of their tool is the incorporation of a 3-D spreadsheet to analyse data between program runs. For example, it is possible use the 3-D spreadsheet package to compare the run-times for different program runs and display them in a bar chart.

Halstead’s Vista system [29] (based on an earlier system called Pablo [82]) provides a generic browser that can be used to analyse different types of performance data. In particular, Halstead has used Vista to analyse parallel symbolic programs [30]. Vista presents the programmer with different views of the parallel execution which resemble the activity and per-thread graphs generated by the interactive tool. The programmer can interact with the graphs, to choose what information to display and how to display it. Vista provides a wide range of options to filter the information displayed, but does not allow the complex filtering of data that our query interface provides.

The data model used by the interactive tool uses the par-site as a reference to the source-code. Two independent projects are working on extending GranSim to record alternative source-code references. Hammond et al. [32] have extended GranSim with the cost-centre mechanism. King et al. [54] have developed a profiler for programs that introduce parallelism using evaluation strategies [102].
4.7 Summary

This chapter has described a tool for profiling parallel programs. Its advantage over other tools that it is possible *both* to view high-level graphical views of a computation *and* to pose queries over lower-level information in a structured way. In most profiling tools the only link between the log-file and the graphs is a hidden translation. The interactive tool allows the programmer to click on the graphs to fill in parts of query, and to use queries to control and filter the information that is displayed.

The tool as described in this chapter only allows queries over a single data model. It sometimes frustrating to compose queries over a reasonably complex data model when a query over one of the earlier data models in Chapter 3 would have been sufficient. Part II realizes refinement in E-R modelling, and Part III shows how this can integrated into the interactive tool to provide multiple data models to the user. Chapter 9 shows how the interactive tool is made more generic, to handle other forms of profiling.
Part II

Technical development

Semantic and functional implementation of data model refinement
Overview

In conceptual database design[7] the implementor often starts with a simple data model and applies successive refinements to this until their required data model is produced. The result of this development process is a chain of data models, similar to those produced for GranSim in Chapter 3. The final data model is generally used as a basis to create a database structure, as shown in Chapter 4, and the intermediate data models are discarded. This is unfortunate because, as we saw in Chapter 3, queries over the intermediate data models can answer some questions without having the complexity of the final data model. This part of this thesis describes the implementation of a compiler that produces output such that the user can submit queries based on any of the data models in a given chain. The design of the compiler is influenced by the following aims:

1. The output should have a theoretical justification;

2. It should be possible to define all types of refinement users require;

3. There should be a direct correspondence between the input to the compiler and the refinement chain it describes;

4. The compiler’s input should be typed checked, in common with most modern compilers.

Chapter 5 and Chapter 6 outline the theoretical framework for generating a database that can answer queries using two levels of abstraction. Chapter 7 builds on this framework to provide a compiler that takes as input a combinator expression describing a refinement chain, and generates definitions that enable a user to submit queries on any of the data models from a single database.

The effectiveness of the compiler and its output are analysed in Part III. Chapter 8 applies the compiler both to the parallel data models in Chapter 3, and a data model chain for heap profiling. Chapter 9 integrates the compiler into a generic profiling system.
Chapter 5

Refinement: Design

5.1 Introduction

Chapter 3 developed a chain of data models for describing the GranSim data model. This chapter describes a framework for defining data model refinements. Chapter 6 builds on this framework, describing a theoretically sound method of constructing a database that can answer queries on both a source and refined schema.

Section 5.2 defines a non-diagrammatic syntax for describing schemas. Section 5.3 defines a set of primitives for describing refinements. Section 5.4 summarizes related work.

5.1.1 E-R notation

This chapter uses a slightly different E-R notation to that used in Chapter 3 as the analysis requires different information. This happens regularly in the literature. This is partly because of the lack of a standard E-R notation but also because of the different abstraction levels required for different purposes. The main changes to the notation are:

- All the data models contain full information about the attributes of the entities and relationships. This is required, as data models are used for implementation purposes. Compare this with the data models in Chapter 3, where it was necessary to add attribute information to the final data model (Model 10, Section 3.4) in order to construct a database.

- The E-R models in this chapter do not include any information about the degree (how many instances of an entity may take part in a relationship) and optionality of a relationship.
```haskell
type Schema = Set Concept

data Concept = E Entity
  | R Relationship
  | H Hierarchy
  | A Attribute

type Entity = (Name, AttributeName)
type Name = String

type Attribute = (Name, [AttributeName])
type AttributeName = Name

type Relationship = (Name, [(Name, Maybe Name)])
type Hierarchy = (Name, [Name])
```

Figure 5.1: The Schema data-type.

### 5.2 Schema syntax

Although E-R diagrams are helpful to the user, a non-diagrammatic notation is beneficial for schema analysis and manipulation. Figure 5.1 gives the data-type that defines our schema syntax, loosely based on the syntax used by Santucci [97], without the complexity of role information (see Section 5.4). A brief introduction to the syntax follows.

A schema is a set of concepts. Each of these concepts is either an entity, a relationship, a generalization hierarchy, or attributes of an entity. An entity is identified by a distinct name and a primary key. A relationship is identified by a name and has two or more connections to some entities. Additionally, an entity has one or more attributes (including the primary key), and a relationship has zero or more attributes.

**Example 13** The following schema contains two entities connected by a binary relationship:

![Diagram](https://via.placeholder.com/150)

and is represented in our syntax as:

```plaintext
{E (Thread, ThreadId), E (Processor, Number),
 A (Thread, [ThreadId]), A (Processor, [Number]),
 R (RunsOn, [(Thread, Nothing), (Processor, Nothing)])}
```
If a relationship has two or more connections with the same entity then labels are required to distinguish them.\(^1\)

Example 14 The following schema contains a single entity whose instances can be interrelated:

\[
\begin{align*}
\text{THREAD} & \quad 
\begin{array}{c}
\text{creator} \\
\text{created}
\end{array} \\
\text{ThStart} & \\
\text{ThEnd} & \\
\text{ThreadId} & \\
\text{Sparks}
\end{align*}
\]

and its syntactic representation is:

\[
\{ E (\text{Thread}, \text{ThreadId}), A (\text{Thread}, [\text{ThreadId}, \text{ThStart}, \text{ThEnd}]), \\
R (\text{Sparks}, [(\text{Thread}, \text{Just creator}), \\
(\text{Thread}, \text{Just created})]) \}
\]

A generalisation hierarchy (also called a sub-type relation) is defined by a parent entity, and one or more child entities. The primary key of the child entities is the same as that of the parent entity.

Example 15 The hierarchy in the schema:

\[
\begin{align*}
\text{ACTIVITY} & \quad 
\begin{array}{c}
\text{AcEnd} \\
\text{AcStart} & \\
\text{ActivityNo}
\end{array} \\
\text{RUN} & \\
\text{BLOCK} & \\
\text{ActivityNo}
\end{align*}
\]

is represented as:

\[
\{ E (\text{Activity}, \text{ActivityNo}), \\
A (\text{Activity}, [\text{ActivityNo}, \text{AcStart}, \text{AcEnd}]), \\
E (\text{Run}, \text{ActivityNo}), A (\text{Run}, [\text{ActivityNo}]), \\
E (\text{Block}, \text{ActivityNo}), A (\text{Block}, [\text{ActivityNo}]), \\
H (\text{Activity}, [\text{Run}, \text{Block}]) \}
\]

Although the above examples contain only binary relationships, it is possible for relationships to have two or more connections.

\(^1\)More precisely, labels are required where the primary keys of the connection are the same.
data PrimRefinement
  -- Bottom-up primitives
  = AGeneration EnName (Maybe AttributeName) [AttributeName]
  | EGeneration EnName AttributeName
  | RGeneration ReName [ArcLabel]
  | HGeneration EnName [EnName]
  | HEGeneration Name Name
  -- Top-down primitives
  | RSplit ReName [ReName]
  | EDecomposition EnName ReName [EAttribute] [(ArcLabel,Name)]
  | RDecomposition ReName EnName [ReName]
  | HDecomposition Name Name [ArcLabel]

newtype EAttribute
  = EAttribute (Name,AttributeName,[AttributeName])
type ArcLabel = (Name,Maybe Name)

Figure 5.2: The primitive refinement data type.

5.3 Refinement primitives

We now have a non-diagrammatic representation of an E-R data model, but need a way to describe the refinement between data models. To do this we introduce a set of primitives. These should be both simple and a near minimal set of operations required to describe standard transformations, making any compilation easier to define. Rather than create a new set of primitives we base ours on those given by Batini et al. [6]. They define a small set of simple primitives, which they claim are complete in the sense that, given a schema containing a single entity, it is possible to generate any other schema. Batini et al. also sketch a proof to justify their claim. However, they neglect to formally define the primitives and so their full interpretation is left open. This section formally defines the behaviour of each primitive using a denotational semantics that describes the affect of a primitive refinement on a given schema.

Each primitive refinement is represented by a single constructor of the type PrimRefinement, given in Figure 5.2. Batini et al. divide their primitives into top-down and bottom-up primitives. The latter primitives generate new concepts, the former primitives split or decompose already existing concepts. This division is convenient for our purposes.

The reader not interested in the intricate details of the primitives can refer to Tables 5.1 and 5.2 and pass over the remainder of this chapter without loss of continuity.
### 5.3.1 Bottom-up primitives

Bottom-up primitives share the property that they introduce new concepts into a schema. Table 5.1 introduces five bottom-up primitives, together with example E-R transformations. The full meaning of the primitives is defined by the $B$ compilation scheme in Figures 5.4 and 5.6. A rule in the compilation scheme:

$$B_s[p] = t$$

is interpreted as: Applying the bottom-up primitive $p$ to source schema $s$ results in the target schema $t$. The auxiliary functions in Figure 5.3 are assumed in the compilation rules, and explicit constructors are omitted, making the definitions easier to read. An informal description of the primitives and their meaning follows. Pre-conditions on the use of the primitives are defined in important cases but standard conditions, such as it is not possible to introduce an entity with the same name as an already existing entity, are omitted.

**EGeneration EnName AttributeName**

The EGeneration constructor introduces a new entity into a schema. Its first argument is the schema name, and its second is the compulsory primary key attribute. Figure 5.4 defines the meaning of the EGeneration primitive. Its effect is to add a schema definition for both the entity, and the attribute.
findConcept\((n,s)\) = find concept \(n\) in schema \(s\)

findPrim\((n,s)\) = find the primary key for entity \(n\) in schema \(s\)

findAttrs\((n,s)\) = find field names for entity \(n\) in schema \(s\)
(includes selector)

findAttrs'\((n,s)\) = find attribute names for entity \(n\) in schema \(s\)

sortConcepts\((s)\) = sort concepts of schema \(s\) into entities,
relationships, hierarchies, and attributes

Figure 5.3: Auxiliary functions

\[B_s[EGeneration \ e \ a] = newE \cup newA \cup s\]
where \(newE = \{(e,a)\}\), \(newA = \{(e,\{a\})\}\)

\[B_s[RGeneration \ r \ links] = newR \cup s\]
where \(newR = \{(r,\{links\})\}\)

\[B_s[HGeneration \ e \ es] = newH \cup s\]
where \(newH = \{(e,\{es\})\}\)

Figure 5.4: The \(B\) compilation scheme for the \(E\)Generation, \(R\)Generation, and \(H\)Generation primitives.

**Example 16** We introduce two entities, first \textit{Thread}, then \textit{Processor}:

\[r_1 = B_{\{\}}[EGeneration \ Thread \ ThreadId] \]
\[= \{E (Thread,ThreadId), A (Thread,[ThreadId])\}\]

\[r_2 = B_{\{\}}[EGeneration \ Processor \ No] \]
\[= \{E (Processor,No), E (Thread,ThreadId),
A (Processor,[No]), A (Thread,[ThreadId])\}\]

\[\text{RGeneration ReName [ArcLabel]}\]

The \textit{RGeneration} constructor specifies a new relationship, named in the first argument and defined by the connections in the second argument. Labels are used where there is more than one connection to the same entity.
Example 17  We introduce a new relationship between the two entities:

\[ r_3 = B_{r_3}[\text{RGeneration hosts } [(\text{Processor,Nothing}),(\text{Thread,Nothing})]] \]

\[ \{ \text{E (Processor,No), E (Thread,ThreadId),} \]
\[ \text{R (hosts,[(Processor,Nothing),(Thread,Nothing)])} \]
\[ \text{A (Processor,[No]), A (Thread,[ThreadId])} \} \]

followed by a relationship, \emph{sparks}, for recording the creator of each thread:

\[ r_4 = B_{r_4}[\text{RGeneration sparks } [(\text{Thread,Just creator}),] \]
\[ (\text{Thread,Just created})]] \]

\[ \{ \text{E (Processor,No), E (Thread,ThreadId),} \]
\[ \text{R (hosts,i [(Processor,Nothing),(Thread,Nothing)]]} \]
\[ \text{R (sparks, [(Thread,Just creator),(Thread,Just created)])} \]
\[ \text{A (Processor,[No]), A (Thread,[ThreadId])} \} \]

HGeneration EnName [EnName]  

Where a number of entities share exactly the same primary key attribute, it is possible to label these under a single hierarchy. Any relationships with these entities still relate to the same entity.

AGeneration EnName (Maybe AttributeName) [AttributeName]  

The AGeneration primitive adds the attributes listed in the third argument to the entity or relationship named in the first argument. The second argument is optional, and if used specifies an attribute to use as the new primary key. This feature is used when the current primary key no longer uniquely identifies all the records.

Example 18  In Figure 5.5 the \emph{name} attribute is a suitable primary key for identifying an account when the only information available is the person's name and their account's value. When account numbers are added into the model, making multiple accounts per person possible, the account number is a better primary key.

So far the compilation rules have used only simple set operations. The rules for AGeneration (see Figure 5.6) use extra notation. The pattern \langle \ldots \rangle represents a sequence. A sequence can have an optional subscript to define the domain of the variables, and also a superscript that defines the operator needed to glue the elements of the sequence together. The meaning of the subscript and superscript
Figure 5.5: An example of AGeneration, where a new primary key is identified.

is defined by:

\[
\langle \text{exp}(i) \rangle_{i \in \{1,2,\ldots,n\}} \equiv \langle \text{exp}(1), \text{exp}(2), \ldots, \text{exp}(n) \rangle
\]

\[
\langle a_1, a_2, \ldots, a_n \rangle \equiv a_1 \otimes a_2 \otimes \ldots \otimes a_n
\]

\[
\langle \text{exp}(i) \rangle_{i \in \{1,2,\ldots,n\}} \equiv \text{exp}(1) \otimes \text{exp}(2) \otimes \ldots \otimes \text{exp}(n)
\]

So \(\langle e_i \rangle_{i \in \{1,2\}} \equiv \langle e_1, e_2 \rangle\) and \(\langle e = e_i \rangle_{i \in \{1,2,3\}} \equiv e = e_1\ or\ e = e_2\ or\ e = e_3\). Convenient concatenation of sequences is provided by the + operator, defined by:

\[
\langle a_1, \ldots, a_n \rangle + \langle b_1, \ldots, b_m \rangle \equiv \langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle
\]

**Example 19** We add an attribute to the hosts relationship to record the duration of the hosting:

\[r_5 = B_n[A\text{Generation} \ \text{hosts} \ \text{Nothing} \ [\text{Duration}]]\]

\[= \{E \ (\text{Processor}, \text{No}), \ E \ (\text{Thread}, \text{ThreadId}), \]
\[R \ (\text{hosts}, [(\text{Processor}, \text{Nothing}), (\text{Thread}, \text{Nothing})])\]
\[A \ (\text{Processor}, [\text{No}]), \ A \ (\text{Thread}, [\text{ThreadId}]),\]
\[A \ (\text{hosts}, [\text{Duration}])\}\]

then add a spark reference attribute to Thread, setting this as the new primary key:

\[r_6 = B_{n+1}[A\text{Generation} \ \text{Thread} \ (\text{Just SparkId}) \ [\text{SparkId}]]\]

\[= \{E \ (\text{Processor}, \text{No}), \ E \ (\text{Thread}, \text{SparkId}), \]
\[R \ (\text{hosts}, [(\text{Processor}, \text{Nothing}), (\text{Thread}, \text{Nothing})])\]
\[A \ (\text{Processor}, [\text{No}]), \ A \ (\text{Thread}, [\text{SparkId}, \text{ThreadId}]),\]
\[A \ (\text{hosts}, [\text{Duration}])\}\]

**HEGeneration Name Name**

The HEGeneration primitive introduces a new child entity into an pre-existing hierarchy. This primitive does not decompose the parent entity into a further
\[ B_s[AGeneration \ e \ \llangle \text{Nothing} \rrangle \ as] = E \cup R \cup A' \cup H \]
where
\[ A' = A \setminus \{ (e, ) \} \cup \{ (e, as' + as) \mid as' \in as \} \]
\[ (E', R', A', H') = \text{sortConcepts}(s) \]

\[ B_s[AGeneration \ e \ a \ as] = E' \cup R \cup A' \cup H \]
where
\[ E' = E \setminus \{ (e, a) \} \]
\[ A' = A \setminus \{ (e, a) + as' + as' \mid as' \in as \} \]
\[ (E', R', A', H') = \text{sortConcepts}(s) \]

\[ B_s[HEGeneration \ e \ e'] = E' \cup R \cup A' \cup H' \]
where
\[ E' = E \cup (e', \text{findPrim}(e, s)) \]
\[ A' = A \cup (e', \{ \text{findPrim}(e, s) \}) \]
\[ H' = \{ (n, e') \mid (n, s) \in H, n = e \} \cup \{ (n, s) \mid (n, s) \in H, n \neq e \} \]
\[ (E', R', A', H') = \text{sortConcepts}(s) \]

Figure 5.6: The \( B \) compilation scheme for the \text{AGeneration} and \text{HEGeneration} primitives.
child entity, as is possible with the top-down primitive HDecomposition, but adds a new entity that represents new information. For example, in Figure 5.7 the COUNTRY entity in the source schema would contain no information about the population of African countries. The meaning of this primitive is defined in Figure 5.6.

5.3.2 Top-down primitives

The main difference between bottom-up and top-down transformations is that the former add new concepts into a schema whilst the latter manipulate pre-existing concepts. Batini et al. [6] define three conditions of a top-down transformation:

1. they have a simple structure (the starting pattern is a single concept);

2. each logical link should be inherited by a single concept of the target schema;

3. names should preferably be refined into new names, describing the original concept at a lower abstraction level.

The first two conditions are suitable aims for our top-down primitives. However, the third condition is too restrictive. For example, if this is enforced then it is not possible to use the name Thread in more than one of the data models in Chapter 3. Table 5.2 shows an example of each of our top-down primitives. The meaning of these primitives are defined in Figures 5.8, 5.10, 5.11 and 5.12 by the T compilation scheme, the top-down equivalent of the bottom-up B scheme. All these primitives conform to condition 1 and 2. A description of each primitive follows.
### Table 5.2: Top-down refinement primitives (attribute details omitted)

<table>
<thead>
<tr>
<th>Refinement primitive</th>
<th>Source sub-schema</th>
<th>Target sub-schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSplit</td>
<td><img src="#" alt="RSplit Diagram" /></td>
<td><img src="#" alt="RSplit Diagram" /></td>
</tr>
<tr>
<td>EDecomposition</td>
<td><img src="#" alt="EDecomposition Diagram" /></td>
<td><img src="#" alt="EDecomposition Diagram" /></td>
</tr>
<tr>
<td>RDecomposition</td>
<td><img src="#" alt="RDecomposition Diagram" /></td>
<td><img src="#" alt="RDecomposition Diagram" /></td>
</tr>
<tr>
<td>HDecomposition</td>
<td><img src="#" alt="HDecomposition Diagram" /></td>
<td><img src="#" alt="HDecomposition Diagram" /></td>
</tr>
</tbody>
</table>

\[
\mathcal{T}_s[\text{RSplit } r \text{ rs}] = \\
E \cup R' \cup A' \cup H \\
\text{where} \\
R' = R \setminus \text{oldr} \cup \{(r', \text{links}) \mid r' \in rs\} \\
A' = \{(r', as) \mid (n, as) \in A, n = r, r' \in rs\} \cup \{(n, as) \mid (n, as) \in A, n \neq r\} \\
\text{oldr} \equiv \left(\left\langle \_\text{links} \right\rangle \right) = \text{findConcept}(r, s) \\
(E, R, A, H) = \text{sortConcepts}(s)
\]

Figure 5.8: The \( \mathcal{T} \) compilation scheme for the RSplit primitive.

### RSplit ReName [ReName]

The RSplit primitive refines a single relationship into multiple relationships. For example, a relatedTo relationship could be broken down into two relationships: childOf, and siblingOf. The structure of the new relationships is inherited from the original relationships.

The \( \mathcal{T} \) compilation equations for this primitive, given in Figure 5.8, are fairly straightforward, bearing in mind that if the relationship being split has attributes then so should the new relationships.
EDecomposition \textit{Thread} has
\[\left[\left(\text{Thread, ThreadId, [ThStart, ThEnd]}\right), \left(\text{Activity, ActivityNo, []}\right)\right]\]
\[\left[\left(\text{sparks, Just creator, Thread}\right)\right]\]

Figure 5.9: An example of entity decomposition.

\begin{itemize}
\item \textbf{EDecomposition EName RName [EAttribute]}
\item \textbf{[(ArcLabel, Name)]}
\end{itemize}

The \textbf{EDecomposition} primitive decomposes a single entity into multiple entities, connected by a new relationship. The purpose of this primitive is to decompose a relationship between some of the attributes into an explicit relationship. For example, the transformation in Figure 5.9 makes explicit the relationship between the primary key ActivityNo, and the ThreadId attribute.

The attributes of the decomposed entity must be distributed across all of them, with none of the attributes in more than one entity, and one of the new entities must have the same primary key as the original.

The first argument specifies the entity to split, the second the name of the new relationship, and the third defines the new entities. The fourth argument determines what to do with relationships connected to the original entity. This begs the question, \textit{what should happen to relationships with the entity that is decomposed?}. It is not necessary to consider these type of questions for the bottom-up primitives, since no concepts from the original schema are decomposed. However, in a top-down transformation a concept is removed from the original model (to be replaced by more refined concepts) so the question is valid. Clearly, it is not satisfactory to delete or somehow reduce the number of connections in the relationship, so the connection should be inherited by the resulting entities. To conform to the second condition of a top-down primitive \textit{each logical link should be inherited by a single concept of the target schema}, so we need to decide which
\[ T_s[E\text{decomposition } e \ \epsilon \ \mathcal{E}] = \]

- \[ \langle \left\{ \langle a_i \cup as_i \rangle \cap \left( \{a_j \cup as_j \} \neq \emptyset \right)_{i \neq j} \right\}, \exists i \in I, \forall a \in \text{findAttrs}(e, s) \rangle \]

\[ \land \left( \text{findPrim}(e, s) \in \{a_i \ | \ i \in I\} \right) \] \[
\land \left( \text{findPrim}(e, s) \in \{a_i \ | \ i \in I\} \right) \]

\[ = \{(r, \langle \langle e_i, \langle \text{Nothing} \rangle \rangle \rangle)_{i \in I}\} \cup E' \cup R' \cup A' \cup H' \]

where

\[ E' = E \setminus (e, \text{findPrim}(e, s)) \cup \{(e_i, a_i) \ | \ i \in I\} \]

\[ A' = A \setminus (e, \_ \cup \{(e_i, \{a_i \cup as_i \}) \ | \ i \in I\} \]

\[ R' = \{(r', \mathcal{L}_{E'}[\langle \text{if } e_j' = e \text{ then remaplt}(l_j) \text{ else } e_j', l_j' \rangle]_{j \in J}) \] \[
| (r', \langle (e_j', l_j') \rangle_{j \in J}) \in \mathcal{R} \}
\]

\[ \text{remaplt}(l) = \text{if } (r, l, \_ \in \mathcal{R} \text{ then } e', \text{ where } (r, l, e') \in \mathcal{R} \] \[
\text{else } e_i', \text{ where } a_i = \text{findPrim}(e, s), i \in I \]

\[ H' = \text{updtChildren}(\text{updtParents}(H)) \]

where

\[ \text{updtParents}(H) = \{(n, ns) \ | \ (n, ns) \in H, n \neq e\} \cup \]

\[ \{(e_i, ns) \ | \ (n, ns) \in H, n = e, i \in I, a_i = \text{findPrim}(e, s)\} \]

\[ \text{updtChildren}(H) = \{(n, \{e_i \ | \ e' \in ns, e' = e, i \in I, a_i = \text{findPrim}(e, s)\} \cup \]

\[ \{e' \ | \ e' \in ns, e' \neq e\}) \ | \ (n, ns) \in H\} \]

\[ (E, R, A, H) = \text{sortConcepts}(s) \]

\[ \mathcal{L}_s[\langle e_i, l_i \rangle_{i \in I}] = \]

\[ \left( \langle e_i, \text{ if findPrim}(e_i, s) \in \{\text{findPrim}(e_j, s) \ | \ j \in I, j \neq i\} \right) \right)_i \]

\[ \text{then } l_i \text{ else } \langle \text{Nothing} \rangle \] \]

\[ \right)_{i \in I} \]

Figure 5.10: The \( T \) compilation scheme for the EDecomposition primitive.

of the new entities the relationship should connect to. This is the purpose of the fourth argument, a list of pairs of connections and new entities. This is used in Figure 5.9 to specify that the creator arc of sparks relationship should connect to the Thread entity.

It can be tedious to specify every connection, so by default, if the connection is not mentioned in the third argument, the relationship connects to the new entity with same primary key attribute as the decomposed entity.

Figure 5.10 defines the \( T \) compilation rule for the EDecomposition primitive. This is considerably more complex than any of the \( B \) rules. This to be expected though since top-down primitives manipulate current concepts in the schema which can have side effects on neighbouring concepts.
As the EDecomposition rule is complex we step through the definition.

- \( \langle \{a_i\} \cup as_i \rangle \cap (\{a_j\} \cup as_j) = \emptyset \rangle \), \( i, j \in I, i \neq j \)
  - \( \text{findAttr}(e, s) = \{a_i \mid i \in I\} \cup \{a \mid i \in I, a \in as_i\} \)
  - \( \text{findPrim}(e, s) = \{a_i \mid i \in I\} \)

These pre-conditions determine that all attributes of the old entity must be distributed across the new entities, that the primary key attribute must be inherited as a primary key of one of the new entities, and each attribute can only be inherited by one entity. This disallows meaningless transformations, and transformations which do not reflect the data model properly as attributes are lost.

\[
E' = E \setminus (e, \text{findPrim}(e, s)) \cup \{(e_i, a_i) \mid i \in I\}
\]

\[
A' = A \setminus (e, \_ \_ ) \cup \{(e_i, \{a_i\} \cup as_i) \mid i \in I\}
\]

The target entity set is obtained by removing the original one from the source set, and adding definitions for the new entities. A similar process is used to derive the target attribute set.

\[
R' = \{(r', \mathcal{L}_E\[ \langle (\text{if } e_j' = e \text{ then remapLt}(l_j) \text{ else } e_j', l_j') \rangle \rangle_{j \in J} \}) \mid (r', \langle (e_j', l_j') \rangle \rangle_{j \in J} \in R\}
\]

The target relationship is derived by analysing the connections of each relationship. If a connection is with the entity that is being decomposed then the connection is changed according to the remapLt function, otherwise the connection is unchanged. The \( \mathcal{L} \) compilation scheme removes any labels that are no longer required. For example, in Figure 5.9 the labels creator and created are no longer required because the relationship is between the two new entities, rather than a single entity.

\[
\text{remapLt}(l) = \text{if } (r, l, \_ ) \in rels \text{ then } e', \text{ where } (r, l, e') \in rels \text{ else } e_i, \text{ where } a_i = \text{findPrim}(e, s), i \in I
\]

The remapLt function updates relationship connections. If the new connection is described in the fourth argument, rels, this is used, otherwise the connection defaults to the new entity with the same primary key as the original one.

\[
H' = \text{updtChildren(updtParents}(H))
\]

The new hierarchy set is obtained by updating any parents that split, followed by any children.

\[
\text{updtParents}(H) = \{(n, ns) \mid (n, ns) \in H, n \neq e\} \cup \{(e_i, ns) \mid (n, ns) \in H, n = e, i \in I, a_i = \text{findPrim}(e, s)\}
\]
\[ \mathcal{T}_s[R\text{Decomposition} \ r \ e \ \langle r_i \rangle_{i \in I}] = \\
E' \cup R' \cup A' \cup H \]

where
\[ E' = E \cup \{(e, a)\} \]
\[ R' = R \setminus \text{oldr} \cup \{(r_i, (e_i, \langle \text{Nothing} \rangle), (e, \langle \text{Nothing} \rangle)) \mid i \in I\} \]
\[ A' = \{(e, \text{attrs})\} \cup \{(n, as) \mid (n, as) \in A, n \neq r\} \]
\[ \text{oldr} \equiv \langle (e_i, l_i) \rangle_{i \in I} = \text{findConcept}(r, s) \]
\[ \text{attrs} \equiv \langle a, \ldots \rangle = as \in \{as \mid (n, as) \in A, n = r\} \]
\[ (E, R, A, H) = \text{sortConcepts}(s) \]

Figure 5.11: The \( \mathcal{T} \) compilation scheme for the RDecomposition primitive.

If the parent of the hierarchy is the entity being decomposed then the new parent is identified as the new entity with the same primary key as the original.

\[ \text{updtChildren}(H) = \{(n, \{e_i \mid e' \in ns, e' = e, i \in I, a_i = \text{findPrim}(e, s)\} \cup \\
\{e' \mid e' \in ns, e' \neq e\}) \mid (n, ns) \in H\} \]

If a child (sub-type) of a hierarchy is decomposed, the new entities with the same primary key replace the sub-type in the hierarchy.

RDecomposition ReName EnName [ReName]

RDecomposition decomposes a relationship into an entity connected by multiple relationships. The first argument defines the relationship to decompose. It is important that the relationship has at least one attribute, as the first one is used as the primary key for the new entity. The number of new relationships must equal the number of entities that the source relation is related to, because each relationship connection is replaced by one these relationships.

The \( \mathcal{T} \) compilation rules in Figure 5.11 show the effect of the relationship decomposition primitive. The new relationships require no labels as they are all binary relationships between the new entity and a different one.

HDecomposition EnName EnName [ArcLabel]

HDecomposition is the final top-down primitive. It refines an entity by creating a sub-type entity. If a hierarchy does not exist then a new one is created, otherwise the new entity is added to the existing hierarchy. The first argument refers to the (perhaps new) entity to use as a super-type entity, and the second the new entity.
\[ \mathcal{T}_s [\text{HDecomposition } e \to e' rs] = \]
\[ E' \cup R' \cup A' \cup H' \]
where
\[ E' = E \cup \{(e', \text{findPrim}(e, s))\} \]
\[ R' = \left\{ (r, \left( a_i, l_i, \begin{cases} (e = e) \land (e_i, l_i) \in rs & \text{if } (e_i, l_i) \in R \\ (e', l_i) & \text{else} \end{cases} \right) \}_{i \in I} \right\} \]
\[ A' = A \cup \{(e', \text{findPrim}(e, s))\} \]
\[ H' = \{(e, \langle e', n_1, \ldots, n_i \rangle) \mid \langle n, n_1, \ldots, n_i \rangle \in H, n = e \} \cup \{(n, ns) \mid \langle n, ns \rangle \in H, n \neq e \} \]
\[ (E, R, A, H) = \text{sortConcepts}(s) \]

Figure 5.12: The \( \mathcal{T} \) compilation scheme for the \textbf{HDecomposition} primitive.

The third argument lists the relationship connections that previously connected to the original parent entity, but should now connect to the new child entity.

Figure 5.12 defines the meaning of the \textbf{HDecomposition} primitive.

### 5.4 Related Work

Each primitive described in this chapter corresponds to one of the primitives given by Batini \textit{et al.} [6], but there are some differences between the two sets:

- We do not have a primitive that corresponds to their bottom-up primitive, \textbf{Attribute Collection}, that creates an entity seen as an aggregation of previously defined attributes. Use of this primitive implies that it is possible to define attributes that have no connection to any entity or relationship. This is not possible in the E-R model, so we have no need for the primitive.

- We do not include a primitive that corresponds to their top-down primitive, \textbf{Entity-Unrelated Entity}, that splits an entity into two or more unrelated entities. The reasoning behind this is that this primitive does not define refinements: it is only possible to retrieve the abstract data from the concrete data in a limited number of cases as there is no relationship between the resulting entities. A better way of simulating a splitting of an entity is by using the \textbf{HDecomposition} primitive where an entity is sub-divided into further different entities but still related by a parent entity.
Example 20 Their example use of Entity-Unrelated Entity is where the entity Geographic reference is split into two unrelated entities Territorial geographic reference and Administrative geographical reference. An alternative way to express this is as an HDecomposition with Geographic reference as the parent with the other two entities as children.

- They define a top-down primitive Attribute Development that has the same functionality as our bottom-up primitive AGeneration. There are valid arguments for categorizing this primitive as either top-down or bottom-up. We treat the primitive as bottom-up as it is more consistent with our naming scheme.

- They imply that in the case of the Generalisation Generation primitive, if all the entities joining the hierarchy are connected to relationships with the same name, then all of these relationships can be combined. The connection is passed on to the parent of the hierarchy. Our bottom-up primitive HGeneration does not provide this functionality. Furthermore, it is not even possible to have two different relationships with the same name.

- We define the formal meaning of the primitives.

McBrien and Poulosilis [65] define primitives for adding and removing entities, relationships and attributes. Whilst their primitives describe a lower level of detail than ours, they are sufficient to define the primitives in this chapter. However, our primitives are more suitable for defining rules for retrieving data from a refined data model. In their system a sequence of primitives is required to describe a refinement whereas with our system a single primitive describes a single refinement.

5.5 Summary

We now have a framework for defining refinement transformations using a set of formally defined primitives. Chapter 6 defines retrieve functions for each of these primitives, and shows that they produce valid refinements when certain pre-conditions are met.
Chapter 6

Refinement: Semantics

6.1 Introduction

The aim of this chapter is to describe a mechanism to allow an implementor to design a data model using refinement, and subsequently allow the user to query a database using any of the intermediate data models. Chapter 5 has defined the syntax and primitives needed to describe a data model (schema) and refinement transformations. This chapter shows how, when given a database based on the final data model, to use this refinement information to generate a valid automatic method for extracting the data for an intermediate data model.

Section 6.2 gives a short introduction to views. Section 6.3 considers what is meant by a valid refinement. Section 6.4 defines the table format to represent schemas. Section 6.5 defines the compilation rules. Section 6.6 examines related work.

6.2 Refinement using views

We start by assuming that there is only one abstract model, which can be refined using a single primitive refinement to produce a concrete data model. Our aim is to allow the retrieval of queries based on the abstract model, from a database based on the concrete data model. In Chapter 7 we consider the case of multiple refinements. There are a number of ways of achieving our aim:

1. Take the data from the concrete database and construct an entirely new database based on the abstract model. Queries over the abstract and concrete data models are then processed using the appropriate database.

2. Intercept the user queries over the abstract model and translate into appropriate queries over the concrete model. The result of the query is also
intercepted, and translated into an appropriate result in the abstract domain.

To answer any query \( q_A \) over the abstract database to get the result \( r_A \) in the mapping:

\[
\begin{array}{c}
S_A \xrightarrow{q_A} r_A \\
t \downarrow \\
S_C \xrightarrow{f(q_A)} r_C \\
g(r_C) \\
\end{array}
\]

where \( t \) is the refinement from abstract schema \( S_A \) to concrete \( S_C \), we would need to implement the mapping functions \( f \) and \( g \).

3. Define SQL \textit{views} to construct the abstract tables from the concrete tables, then use the database’s built-in handling of \textit{views} to retrieve the results.

Given a refinement \( t \) and source schema \( S_A \), the view function \( v(S_A, t) \) adds view definitions to the concrete database to describe the tables in the abstract database. The resulting database \( S_{C,A} \) can handle queries from both the abstract data model \( S_A \) and the concrete data model \( S_C \).

\[
\begin{array}{c}
S_A \xrightarrow{q_A} r_A \\
t \downarrow \\
S_C \xrightarrow{q_A} r_A \\
v(S_A, t) \\
S_{C,A} \\
\end{array}
\]

The advantage of the first solution is that only one transformation of the data is required, but the entire abstract database does need to be constructed. Although the second solution requires only one database, it requires the definition of two separate transformations. We adopt the third alternative as it combines the advantages of the other two solutions, without the associating disadvantages: Only one database is constructed, and only one set of transformation equations need to be defined.

### 6.2.1 A short introduction to views

Since the conception of database systems there has always been a demand for viewing a single database in multiple ways. For example, an implementor requires a different view of the database to that required by the end-user. These different views are called \textit{user views}. The SQL standard helps the implementor maintain these views by providing a primitive for specifying different views of a single data set. An SQL view is defined by an SQL query.
Example 21 To define a view of all the males from table of people, the following SQL query\(^1\) might used:

```sql
create view Male as
    select distinct Person.Age, Person.Name from Person
    where Person.sex = 'male'
```

The following query would then return the names of all the adult males:

```sql
select distinct Male.Name from Male
    where Male.Age > 18
```

An important property of SQL views is that they are generally stored as translation equations, and whenever the view is used the translation equations are substituted into the query. This may lengthen the time the query takes to evaluate but it means the view is always up-to-date, and there is no replication of data items. Views are indistinguishable to the end user from normal tables. It follows that a view of a view behaves in the same way as a view of a table with the same data.

### 6.3 What is a valid refinement?

Before we can specify views that define the mapping from a concrete schema to an abstract schema, it is important to understand what is meant by a valid refinement, and furthermore whether the refinement primitives produce valid refinements. The two main definitions that define the correctness of transformations are soundness and completeness. For binary set transformations the definitions are straightforward.

**Definition 1 (Soundness of binary set transformations)**

If \( x \in X \) and \( t : X \rightarrow X \) then the transformation \( t \) is **sound** if \( x \subseteq t(x) \).

**Definition 2 (Completeness of binary set transformations)**

If \( x \in X \) and \( t : X \rightarrow X \) then the transformation \( t \) is **complete** if \( x \supseteq t(x) \).

A transformation is sound if no data is lost when the transformation is applied, and the transformation is complete if no data is gained during transformation.

Much of the literature on schema transformations is concerned with transformations that preserve the original data model for the purposes of schema optimization or normalization \([25, 55, 65, 2, 64, 86, 87, 27]\). In this case it is necessary for the transformations to be sound and complete. However, for refinement transformations it is only necessary for the transformation to be sound.

\(^1\)An SQL query can be a command as well as a selection.
Adding the condition that the transformation must be complete would not allow the models to be extended.

**Definition 3 (Schema refinement)** A schema refinement is a schema transformation that is sound.

To define what is meant by a sound transformation we introduce some preliminary definitions. In standard data refinement theory [50, 80, 113], retrieve functions describe the relationship between the concrete data type and the abstract data type. In schema refinement we also define retrieve functions that define the relationship between the concrete and abstract schemas. We use \( \text{repr}(\text{retr}, S) \) to mean a representation using the concrete schema \( S \), and the retrieve function \( \text{retr} \) to return results based on the abstract schema.

**Definition 4 (Adequacy of schema transformations)** Given two schemas \( S_A \) and \( S_C \), and the function \( \text{inst}(S) \) that returns the set of all databases modelled by scheme \( S \), \( \text{repr}(\text{retr}, S_C) \) is an adequate representation of \( S_A \) if, for each \( i \in \text{inst}(S_A) \), there exists an \( i' \in \text{inst}(S_C) \) which is mapped by \( \text{retr} \) (the retrieve function) onto \( i \).

**Definition 5 (Soundness of schema transformations)** Given two schemas \( S_A \) and \( S_C \), where \( t(S_A) = S_C \). The transformation \( t \) is sound if there exists a retrieve function \( \text{retr} \) such that \( \text{repr}(\text{retr}, S_C) \) is an adequate representation of \( S_A \).

It follows from these definitions that a set of views defining a mapping from the concrete database to the abstract database is a retrieve function. Therefore, defining a set of compilation rules \( \mathcal{V} \) that translates primitive refinement into SQL views, is sufficient to show that the primitives are valid refinements.

### 6.4 The table format

In Chapter 3 standard normalisation techniques were used to convert the data model into database tables. This required explicit knowledge of the degree and optionality of the relationships. For example, if a relationship is mandatory \( (1 : m) \) then the primary key of the master entity is added to the relation of the detail entity. We have purposely not recorded this information to simplify the view compilation process.

Assuming all data is of type integer, the CT compilation scheme in Figure 6.1 is a simpler way of translating E-R schemas into relational tables. The entities are translated into tables with a field representing each of their attributes. If an
\[ CT_s[E \ (e, \_)] = \]
\[
\text{create table } e \ (((a \int4 \text{ not null})_{a \in \{a(e', as) \in A', e=e', a \in as \} \cup \text{isHierarchy}(s, e)}) \]
\[
\text{where}
\]
\[
isHierarchy(s, e) = \{ \text{selector char(255) not null} \mid (e', \_ \in H', e = e') \}
\]
\[
(E', R', A', H') = \text{sortConcepts}(s)
\]
\[
CT_s[R \ (r, \_)] = \]
\[
\text{create table } r \ (((shLab(\text{findPrim}(s, e), lab) \int4 \text{ not null})_{(e, lab) \in c})
\]

Figure 6.1: Compilation equations for CT. \( CT_s[c] \) defines the table format for concept \( c \) in schema \( s \).

---

entity is the parent of a hierarchy then it has an extra field, \text{selector}, used as a key to identify the appropriate sub-type.

**Example 22** Applying CT to the second schema in Figure 5.9 results in the following table definitions:

\[
\text{create table } \text{Thread as (ThreadId int4 not null, ThStart int4 not null, ThEnd int4 not null)}
\]
\[
\text{create table } \text{Activity as (ActivityNo int4 not null)}
\]
\[
\text{create table } \text{sparks as (ThreadId int4 not null, ActivityNo int4 not null)}
\]
\[
\text{create table } \text{has as (ThreadId int4 not null, ActivityNo int4 not null)}
\]

This method of translation has the following advantages over standard techniques:

- The optionality of the relationships does not have to be specified.

- There is a very close relationship between the data model and resulting tables.

- Constructing automatic views is easier, as we do not have to deal with the different optimizations.

However, the possible disadvantages are:

- Queries may be longer if the user does not apply mental optimization techniques.

**Example 23** Suppose we want to determine which threads created which from a database based on the second data model in Figure 5.9. If the data
model is converted into database form using the traditional method then the following query is used:

```sql
SELECT DISTINCT t1.ThreadId, t2.ThreadId
FROM Thread t1, Thread t2, Activity
WHERE t1.has = Activity.ActivityNo
    AND Activity.sparks = t2.ThreadId
```

If the data model is converted using the CT rules then the following query is used:

```sql
SELECT DISTINCT t1.ThreadId, t2.ThreadId
FROM Thread t1, Thread t2, Activity, has, sparks
WHERE t1.ThreadId = has.ThreadId
    AND has.ActivityNo = Activity.ActivityNo
    AND Activity.ActivityNo = sparks.ActivityNo
    AND sparks.ThreadId = t2.ThreadId
```

which is clearly, this query is much longer. However, by applying two optimizations to remove redundant joins this query can be reduced to:

```sql
SELECT DISTINCT has.ThreadId, sparks.ThreadId FROM has, sparks
WHERE has.ActivityNo = sparks.ActivityNo
```

which is shorter than the first query, a query that cannot be shortened.

- Query evaluation is not as efficient as more joins may be required. This is related to the previous point, and can be minimized by automatic optimizations techniques, as performed in Example 23.

- It does not achieve an important goal in database design: keeping redundancy to a minimum.

The CT method results in similar tables than would be derived using the standard transformational approach where all the relationships are \( m : n \). This is not surprising as to assume the most general level of optionality is the safest option.

### 6.4.1 Valid table data

The CT rules define the structure of the database tables but do not define what is valid table data. We define the following conditions for a table whose structure is created using CT to be a valid table:

1. The field representing the primary key in an entity table must have no repeated data items. This ensures that the primary key uniquely identify the rows.
\[
\begin{align*}
\text{shLab}(a, \langle \text{Nothing} \rangle) &= a \\
\text{shLab}(a, \text{lab}) &= \text{lab} \\
\text{mkLab}(a, \langle \text{Nothing} \rangle) &= a \\
\text{mkLab}(a, \text{lab}) &= \text{a as lab}
\end{align*}
\]

Figure 6.2: More auxiliary functions

2. There must be no repeated rows. The first point disallows this for entities, but is also necessary for relationships.

3. The table data cannot contain null values. The CT rules do not allow this anyway.

4. The fields representing connections in a relationship table should contain only values from the primary key field of the entity that is connected.

6.5 Translating the primitives

In this section we define the retrieve function \( \mathcal{V} \) for each of the primitives. The \( \mathcal{V}_s \) requires the state \( s \) representing the abstract schema. As the concrete and abstract schemas may have entities and relationships with the same names, the concrete table names are accented with a circumflex (indicating a higher level of refinement) to avoid confusion.

Example 24 The abstract version of table \( e \) is \( \hat{e} \) and the concrete version is \( \acute{e} \).

The view compilation rules, \( \mathcal{V}_s \), make use of the auxiliary functions defined earlier in Figure 5.3, as well as the extra ones in Figure 6.2. Also to make the definitions more concise, we define the function view:

\[
\text{view } n \; s = \text{create view } \hat{n} \text{ as } s
\]

All the compilation rules assume the concrete tables contain valid table data, as defined above. We begin by defining the view compilation rules for the bottom-up primitives, and then move on to look at the top-down ones.

A single primitive refinement only constructs and deconstructs a few of the concepts in a schema. The rest of the concepts, and hence tables, remain the same in both the concrete and abstract schemas. For these cases we define default compilation rules in Figure 6.3. It produces views for every entity and relationship in the abstract schema, which is not in an opt-out list, that essentially defines a copy of the concrete tables. Views are not created for hierarchies and attributes as they are represented as fields within an entity or relationship table.
\[ \mathcal{D}(s, cs) = \{ \text{view } c \ (\text{select } \dot c \text{. } \ast \text{ from } \dot c) \mid c \in E \cup R, \text{noneNamed}(c, cs) \} \]

where

\[
\text{noneNamed}(c, cs) = \text{true, if no concept named } c, \text{ false otherwise} \\
(E, R, A, H) = \text{sortConcepts}(s)
\]

Figure 6.3: The default compilation scheme, \( \mathcal{D} \).

### 6.5.1 Bottom-up primitives

The purpose of a bottom-up primitive is to introduce new concepts not present in the abstract schema. Therefore the abstract database should be the same as the concrete database, removing any tables and fields created because of the new concepts. Figure 6.4 defines the view compilation rules for the bottom-up primitives.

**EGeneration**

The effect of the EGeneration primitive is to create a new entity, which consequently creates a new table in the concrete database. Therefore the abstract database should contain all the data contained in the concrete database excluding the table for the new entity. So the view compilation rule for this primitive uses only the default rule.

**Example 25** When the schema \{E (E₁, a), A (E₁, [a, b])\} is refined with:

EGeneration E₂ c

the following schema is produced:

\{E (E₁, a), E (E₂, c), A (E₁, [a, b]), A (E₂, [c])\}

The associating (concrete) database has tables E₁ and E₂. The abstract database is then constructed with the following view:

create view E₁ as

select distinct E₁.* from E₁

which is the view generated by \( \mathcal{V} \).

In Section 6.3 we said that \( \mathcal{V}_s[t] \) is a valid retrieve function for schemas transformed using \( t \), if for every abstract database \( X \), based on schema \( s \), there exists a concrete database instance such that \( X = \mathcal{V}_s[t] \). Therefore, to show that \( \mathcal{V}_s[[\text{E Decomposition...}]] \) is a valid retrieve function it is sufficient to show that,
\[
\begin{align*}
\forall_s\text{EGeneration } & = D[(s, \{\})]
\end{align*}
\]

\[
\begin{align*}
\forall_s\text{RGeneration } & = D[(s, \{\})]
\end{align*}
\]

\[
\begin{align*}
\forall_s\text{HGeneration } & = D[(s, \{\})]
\end{align*}
\]

\[
\begin{align*}
\forall_s\text{AGeneration } & \equiv \langle \text{Nothing} \rangle = \langle vE \rangle + + D[(s, \{e\})]
\end{align*}
\]

where

\[
\begin{align*}
vE & = \text{view } e \left( \begin{array}{c}
\text{select distinct } \langle \hat{e}.a \rangle_{a \in \text{findAttrs}(e, s)} \\
\langle \hat{e}.\text{shLab}(\text{findPrim}(e, s), l') \rangle_{(e', l') \in \text{el}}
\end{array} \right) \\
\text{from } \hat{e}
\end{align*}
\]

\[
\begin{align*}
\forall_s\text{AGeneration } & \equiv \langle vE, vR \rangle + + D[(s, \{e\} \cup \{r \mid (r_e) \in R\})]
\end{align*}
\]

where

\[
\begin{align*}
vE & = \text{view } e \left( \begin{array}{c}
\text{select distinct } \langle \hat{e}.a \rangle_{a \in \text{findAttrs}(e, s)} \\
\text{from } \hat{e}
\end{array} \right) \\
vR & = \left( \begin{array}{c}
\text{view } r \\
\text{from } \hat{r}, \langle \hat{e} t_i \rangle_{i \in I, e_i = e} \\
\text{where } \langle t_i.a = \hat{r}.\text{shLab}(a, l_i) \rangle_{i \in I, e_i = e}
\end{array} \right)
\end{align*}
\]

\[
\begin{align*}
\forall_s\text{HEGeneration } & \equiv \langle e'^{\prime} \text{ rels} \rangle = vH + + D[(s, \{e\})]
\end{align*}
\]

\[
\begin{align*}
vH & = \text{view } e \langle \text{remove} \rangle \\
\text{select distinct } \langle \hat{e} \text{.Selector} \rangle + + \langle \hat{e}.a \rangle_{a \in \text{findAttrs}(e, s)} \\
\text{remove} & = \text{from } \hat{e} \\
\text{where } \hat{e} \text{.Selector} & \equiv \langle e' \rangle
\end{align*}
\]

\[
(E, R, A, H) = \text{sortConcepts}(s)
\]

Figure 6.4: The $\forall$ compilation scheme for the bottom-up primitives. In the definition of $vR$, \text{sel dis} is an abbreviation for \text{select distinct}.
given any abstract database instance, there is a way of constructing a concrete
database such that the \( \mathcal{V} \) produces views that generate the abstract database.

If the concrete database is generated by copying the abstract database and
adding an empty table for the new entity, then clearly the view compilation rule
retrieves the original abstract database. Therefore \( \mathcal{V}_s[\text{EGeneration...}] \) is a valid
retrieve function, and so the refinements defined in terms of the primitive are
sound.

**RGeneration and HGeneration**

The view compilation rules for the \textit{RGeneration} and \textit{HGeneration} primitives are
defined in the same way as the \textit{EGeneration} primitive, and so the same argument
is used to show that they also produce sound refinements.

**AGeneration**

The \textit{AGeneration} rule cannot be defined in the same way as the other bottom-up
primitives because its effect is to introduce new fields into a table, rather than
introducing a completely new table.

We first consider the case where the primary key is left unchanged. Here the
only table that changes between the abstract and concrete databases is that which
represents the entity or relationship where new attributes are added. The abstract
version of this table is constructed by selecting, from the concrete version of the
table, only the fields which represent the attributes that exist before refinement.
Where the new attributes are added to a relationship, it is also necessary to select
fields representing the relationship connections from the concrete table. All other
tables are re-constructed using the default rules.

**Example 26 If the primitive:**

\textit{AGeneration Thread Nothing [Start,End]}

is applied to the schema:

\{E (Thread, a), A (E, [ThreadID, Location])\}

then the following view definition constructs the abstract database from the con-
tcrete database:

\begin{verbatim}
create view Thread as
select distinct \_Thread.ThreadID, \_Thread.Location from \_Thread
\end{verbatim}

If an entity's primary key is changed, a view definition is required for any
relationships that connect to the entity. This is because changing a primary
key has the side-effect of changing the attribute that is used as the key in any
connecting relationships. The abstract relationship tables are re-constructed by copying their associating concrete relationship tables, and replacing the values which represents the connection. These values are replaced by the values of the old primary key, in the rows of the concrete entity table where the value of new primary key is equal to the value of the current connection.

**Example 27** If the refinement:

AGeneration Thread (Just SparkId) []

is applied to the schema:

\{E (Thread,ThreadId), A (Thread,[ThreadId,Start,End]), E (Activity,ActivityNo), A (Activity,[ActivityNo]), R (has,[(Thread,Nothing),(Activity,Nothing)])\}

then the following views re-construct the entity table Thread, and the relationship table has.

```sql
create view Thread as
select distinct Thread.ThreadId, Thread.Start, Thread.End
from Thread
```

```sql
create view has as
select distinct t1.ThreadId, has.ActivityNo
from has, Thread t1
where t1.SparkId = has.SparkId
```

*The Activity table is re-constructed using the default rule. So, given the concrete tables:*

<table>
<thead>
<tr>
<th>Thread</th>
<th>has</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>SparkId</td>
<td>ThreadId</td>
<td>Start</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

*the view definitions extract the following abstract tables:*

<table>
<thead>
<tr>
<th>Thread</th>
<th>has</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThreadId</td>
<td>Start</td>
<td>End</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>
The important thing to notice is that the values of ThreadId in has are obtained from the ThreadId entries in Thread, where the values of SparkId in the same row are equal to the values of SparkId in has.

We now look at the definition of the view compilation in Figure 6.4. Where a relationship has more than one connection to the same entity the mkLab function create labels required. The shLab function refers to the connections using labels instead of primary key names, as required. Different instances of the entity table \( t_i \) are used to re-construct each connection with the entity. If we did not do this and instead used the same entity for each connection, then it would not be possible to obtain different values of the primary key for different connections. If a relationship has no connection to the entity then the resulting view is similar to one produced by the default rule.

We show that the AGeneration primitive is sound in the same way that we demonstrated the soundness of other bottom-up primitives. Let \( D \) be the tables in the abstract database not related to the entity that the attributes are added to. Let \( R \) be the abstract tables for the relationships that connect to the entity, and let \( E \) be the table for the entity. Then if the concrete database is constructed from:

- All tables in \( D \).
- The \( R \) tables, replacing any field names equal to the name of the old primary key with the name of the new primary key.
- The table for \( E \), but with new fields for the new attributes, generated by copying the field for the old entity primary key.

\( \mathcal{V} \) will return the abstract database, and so the AGeneration primitive is also sound.

**HEGeneration**

The \( \mathcal{V} \) in Figure 6.4 re-constructs the parent of the hierarchy in the abstract database by removing all the entries in the concrete table that are associated with the new child. All other tables are handled by the default rule.

**Example 28** The following view definition re-constructs the Country table in Figure 5.7:

```sql
create view Country as
    select distinct \( \hat{C}.Selector, \hat{C}.GDP, \hat{C}.Population, \hat{C}.Name \)
    from Country \( \hat{C} \)
    where \( \hat{C}.Selector <> 'Africa' \)
```

The Europe and Asia tables are constructed using the default rule.
\[ V_s[RSplit \ r \ \tilde{r}] = \langle \text{view } \ r \ \langle \text{select } \tilde{r}' \ast \text{ from } \tilde{r}'_{r' \in r} \text{union} \rangle + +D[(s, \{ r \})] \]

Figure 6.5: The \( V \) compilation scheme for RSplit

If the concrete tables are set as identical to the abstract tables with an extra empty table for the new entity, then \( V \) returns the abstract database. It follows that the HGeneration primitive is sound.

### 6.5.2 Top-down primitives

We now define the view compilation rules for the top-down primitives. Unlike, the bottom-up primitives, it is sometimes necessary to specify pre-conditions that define when the views are valid retrieve functions. We use the notation \( \rho(e, i, a) \) to mean the \( i \)-th row of the field named \( a \) in the table for entity \( e \). \( \pi(e) \) is the number of rows for the table for entity \( e \). So, the whole table for entity \( e \) is described by:

\[
\left\{ \langle \rho(e, j, a) \rangle_{a \in \text{find Attrs}(e, s)} \right\}_{j \in \pi(e)}
\]

**RSplit**

The view compilation rule for the RSplit primitive in Figure 6.5 re-constructs the abstract relationship table by merging the rows of the concrete relationship tables.

**Example 29** The following primitive refines the meaning of a relationship between two people such that it is possible to distinguish between children and brothers:

**RSplit** isRelatedTo [isChildOf, isBrotherOf]

The original isRelatedTo is reconstructed by merging together the isChildOf and isBrotherOf relationships. The result of \( V \) in this case is:

create view isRelatedTo as

```sql
select isChildOf. * from isChildOf union
select isBrotherOf. * from isBrotherOf
```

So given the tables:
**isChildOf**

<table>
<thead>
<tr>
<th>person1</th>
<th>person2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jack</td>
<td>John</td>
</tr>
<tr>
<td>Jack</td>
<td>Jane</td>
</tr>
<tr>
<td>Bill</td>
<td>Tim</td>
</tr>
<tr>
<td>Bill</td>
<td>Jane</td>
</tr>
</tbody>
</table>

**isBrotherOf**

<table>
<thead>
<tr>
<th>person1</th>
<th>person2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jack</td>
<td>Bill</td>
</tr>
<tr>
<td>Bill</td>
<td>Jack</td>
</tr>
<tr>
<td>Tim</td>
<td>John</td>
</tr>
</tbody>
</table>

The query:

```sql
SELECT DISTINCT isRelatedTo.person2 FROM isRelatedTo
WHERE isRelatedTo.person1 = 'Jack'
```

would return a table with the entries Bill, John, and Jane.

We can show that V is a valid retrieve function for RSplit by constructing a concrete database where one of the new relationship tables is an exact copy of the abstract relationship table, and the other new relationship tables are empty. V would then return the desired table. It follows that RSplit is sound.

**EDecomposition**

The full rules for the EDecomposition primitive are given in Figure 6.6. Although definitions are quite long and appear complicated, this is mainly because of the need to create re-mapped relationships. The view to reconstruct the decomposed entity is reasonably simple.

The EDecomposition primitive decomposes a single entity into multiple entities, so the natural way of reconstructing the abstract entity table is by use of a composition of the concrete entity tables (see vE in Figure 6.6).

**Example 30** Where a task is decomposed into a spark which becomes a thread:

**EDecomposition Task becomes**

```
[(Thread,[ThreadId,Start,End]),(Spark,[SparkId,ParSite])] []
```

The following view re-constructs the Task table by using a composition between the Thread table and the Spark table, using the becomes relationship table as an intermediary composition:

```sql
CREATE VIEW Task AS
SELECT DISTINCT Spark.* Thread.*
FROM Spark becomes Thread
WHERE Spark.SparkId = becomes.SparkId
AND becomes.ThreadId = Thread.ThreadId
```

As an entity decomposition can have side-effects on other relationships and hierarchies, specialized views are also required for other concepts.
\[ \forall \varepsilon [E \text{ Decomposition} \cdot e \cdot r \cdot \langle e_i, a_i, as_i \rangle_{i \in I \cdot \text{rels}}]

\]

\[ \cdot \langle \rho(e, j, a_t) = \rho(e, k, a_t) \Rightarrow \rho(e, j, a) = \rho(e, k, a) \rangle_{i \in I \cdot j, k \in \pi(e), a \in \text{as}_i} \]

\[ \land \langle \rho(e, j, a_t) = \rho(e, k, a_t) \Rightarrow j = k \rangle_{j, k \in \pi(e), i \in \{i \mid e = e', e = e' \in \text{rels} \}} \]

\[ = \langle vE \rangle \triangleleft vH \triangleleft vR \]

\[ \triangleleft vD([s, \{e, r \mid (rn, \_ \_ ) \in R \mid rn \in \bar{r} \} \cup \{n \mid (n, ns) \in H, e \in ns \}]) \]

where

\[ vE = \text{view } e \left( \text{select distinct } \langle e_i \rangle_{i \in I} \text{ from } \langle e_i \rangle_{i \in I}, r \right) \]

\[ \text{and } \langle e_i, a_i = \hat{r}.a_i \rangle_{i \in I} \]

\[ vR = \langle \text{view } \hat{r} \rangle \left( \text{select distinct } \langle \text{sel}(j) \rangle_{j \in J} \triangleleft \langle \hat{r}.a \rangle_{a \in \text{findAttrs}(r, s)} \right) \]

\[ \text{from } \langle \hat{r}.a \rangle_{a \in \text{findAttrs}(r, s)} \]

\[ \text{where } \langle \text{findPrim}(x_j, s), l_j \rangle_{j \in J} \]

\[ \hat{r}.\text{shLab}(x_j, l_j) \]

\[ \{ \langle \text{findPrim}(x_j, s), l_j \rangle_{j \in J} \}

\[ \hat{r}.\text{shLab}(\text{findPrim}(x_j, s), l_j) \]

\[ \text{if } e' = x_j \land l_j = l' \]

\[ \hat{r}.\text{shLab}(\text{findPrim}(x_j, s), l_j) \]

\[ \text{else if } x_j \neq e \land l_j = l' \]

\[ \hat{r}.\text{shLab}(\text{findPrim}(x_j, s), l_j) \]

\[ \text{else } e' = x_j \lor x_j \neq e \]

\[ \hat{r}.\text{shLab}(\text{findPrim}(e, s), l_j) \]

\[ \text{otherwise} \]

\[ \text{sel}(j) = \]

\[ \bar{R} = \{ (rn, \langle (e_j, l_j) \rangle_{j \in J}) \mid (rn, \langle (e_j, l_j) \rangle_{j \in J}) \in R, rn \in \bar{r} \}

\[ \text{where } (rn, \langle (e_j, l_j) \rangle_{j \in J}) \in \text{rels, } \bar{r} = \{rn \mid (rn, \_ \_ \_ ) \in R, (rn, \_ \_ \_ ) \in \text{rels, } \bar{r} = \{rn \mid (rn, \_ \_ \_ ) \in R, (rn, \_ \_ \_ ) \in \text{rels, } \bar{r} = \{rn \mid (rn, \_ \_ \_ ) \in \}

\[ e' = e_i, \text{where } \text{findPrim}(e, s) \in \text{as}_i \}

\[ E, R, A, H = \text{sortConcepts}(s) \]

\[ vH = \langle \text{view } n \rangle \langle \text{remove}(n), \text{keep}(n) \rangle_{e \in \text{ns}, (n, ns) \in H}

\[ \text{select distinct } \langle \hat{n}.\text{Selector} \rangle \triangleleft \langle \hat{n}.a \rangle_{a \in \text{findAttrs}(n, s)} \]

\[ \text{from } \hat{n} \]

\[ \text{where } \hat{n}.\text{Selector} \leftrightarrow 'e' \]

\[ \text{select distinct } \langle 'e' \rangle \triangleleft \langle \hat{n}.a \rangle_{a \in \text{findAttrs}(n, s)} \]

\[ \text{keep}(n) = \]

\[ \text{from } \hat{n} \]

\[ \text{where } \hat{n}.\text{Selector} = 'e' \]

Figure 6.6: The \( \mathcal{V} \) compilation scheme for E Decomposition.
The hierarchies are re-constructed as follows: where the entity being decomposed is the child of a hierarchy, the references to the new entity in the Selector field of the parent entity table are replaced by references to the decomposed entity. All hierarchies are reconstructed using the default rule.

Re-constructing relationships connecting the decomposed entity requires more thought. If none of the relationship’s connections are listed in the third argument then any connections to the abstract entity connect to the entity that inherits the primary key in the concrete model. In this case the default rule is sufficient in re-constructing the abstract relationship because the value of the connections do not change.

However, if some of the relationship’s connections are listed in the third argument, then the connections are re-constructed by using the new relationship as the translation rule from one primary key to another.

Example 31 Entity $E_1$ is decomposed in Figure 6.7. By default any relationships with $E_1$ connect with $E_{1,1}$ after the refinement, but relationship $R_1$ connects with $E_{1,2}$ instead, as this is listed defined in the third argument. The following view re-constructs the abstract relationship $R_1$, by translating the field representing the connection to $E_{1,2}$, with primary key $b$, into appropriate values of primary key $a$ determined by $R_2$:

create view $R_1$ as

select distinct $\hat{R}_2.a, \hat{R}_1.c$ from $\hat{R}_1, \hat{E}_{1,2}, \hat{R}_2$

where $\hat{R}_2.b = \hat{E}_{1,2}.b$

and $\hat{E}_{1,2}.b = \hat{R}_1.b$

The $vE$ rule in $\mathcal{V}$ generates an optimized version that removes the redundant
intermediate reference to \( \hat{E}_{1,2} \):

create view \( R_1 \) as

select distinct \( \hat{R}_2.a, \hat{R}_1.c \) from \( \hat{R}_1, \hat{R}_2 \)
where \( \hat{R}_2.b = \hat{R}_1.b \)

The function \( \text{sel} \), used to define \( vR \) in \( \mathcal{V} \), handles the different methods required to generate the correct field names in different circumstances.

Example 32 The following is a possible select line in a relationship view:

select distinct \( R.a, R.b \) as left, \( R.right, t_1.c \) from \( R_2 t_1, \ldots \)

It contains four types of reference:

- standard primary key;

- primary key, that requires a label in the abstract model, not present in the concrete model;

- primary key referenced by a label, in both the abstract and concrete models;

- and a key extracted from a copy of the \( R_2 \) table.

The pre-conditions define the instances of the schema where \( \text{edecomposition} \) primitive is sound and \( \mathcal{V} \) is a suitable retrieve function. The first pre-condition eliminates any instances of the decomposed entity that would produce entity tables where the primary keys do not uniquely identify the rows. The second pre-condition ensures that re-mapped relationships can be reconstructed.

Is it sound? If the concrete table database is constructed by:

- creating table for the entities by copying the relevant fields (the ones for the attributes in the new entity) in the table for the entity being decomposed, removing duplicates. The pre-conditions ensure that the primary key of each of the new tables uniquely identifies rows.

- creating a table for the new relationship by copying all the fields for the attributes used as primary keys in the new entities, from the table for the entity being decomposed.

then the views retrieve the decomposed entity in the abstract database. Further rules are required to create tables for the re-mapped entities, and hierarchies.
\[ \forall [\text{RDecomposition}] \quad \forall i, j \in \pi(r) \quad \rho(r, i, \text{attr}) = \rho(r, i, \text{attr}) \Rightarrow i = j \]

\[ = \left\langle vR \right\rangle \text{++} D \left[ (s, \{r\}) \right] \]

where

\[
\begin{align*}
\text{vR} &= \text{view} \ r \\
&= \left( \begin{array}{c}
\text{select distinct} \quad \left\langle \hat{r}_i.\text{mkLab}((e_i, s), l_i) \right\rangle_{i \in I} \\
\text{from} \quad \hat{e} \quad \left\langle \hat{r}_i \right\rangle_{i \in I} \\
\text{where} \quad \left\langle \hat{r}_i.\text{attr} = \hat{e}.\text{attr} \right\rangle_{i \in I}
\end{array} \right)
\end{align*}
\]

\[\left\langle (e_i, l_i) \right\rangle_{i \in I} = \text{findConcept}(r, s)\]

\[\left\langle a_i \right\rangle_{i \in I} = \text{findAttrs}(s, r)\]

attr = a_1

---

**RDecomposition**

Although the EDecomposition and RDecomposition primitives are similar, the \( \forall \) compilation rules for the RDecomposition primitive in Figure 6.8 are less complex than those for EDecomposition because the primitive does not affect any concepts apart from the relationship being decomposed. The same method is used to reconstruct the decomposed concept though.

**Example 33** To re-construct the relationship table `worksWith` from the refinement:

**RDecomposition**

`worksWith` `Manager [worksWith, ofDepartment]`

`where` the `worksIn` relationship has a single attribute, Name, the fields of worksWith and `ofDepartment` are composed together:

**create view** `worksIn` **as**

**select distinct** `worksWith`.Person `ofDepartment`.Department

**from** `worksWith` `Manager` `ofDepartment`

**where** `worksWith.Name` = `Manager.Name`

**and** `Manager.Name` = `ofDepartment.Name`

The pre-condition ensures that the primary key of the new concrete entity table is unique.
\[ \mathcal{V} \left[ \text{HDecomposition } e, e' \ rels \right] = vH \leftarrow D \left[ (s, \{ e \}) \right] \]

\[ vH = \text{view } e \ (\text{remove, keep}) \]

\[
\begin{align*}
\text{select distinct} & \quad \text{if } (e, \_ ) \in H \text{ then } \langle \hat{e}.\text{Selector} \rangle \text{ else } \langle \rangle \\
\text{remove} & = \quad \langle \hat{e}.a \rangle_{a \in \text{findAttrs}'(e, s)} \\
\text{from} & \quad \hat{e} \\
\text{where} & \quad \hat{e}.\text{Selector} \not< 'e' \\
\text{select distinct} & \quad \text{if } (e, \_ ) \in H \text{ then } \langle 'e' \rangle \text{ else } \langle \rangle \quad \langle \hat{e}.a \rangle_{a \in \text{findAttrs}'(e, s)} \\
\text{keep} & = \quad \langle \hat{e} \rangle \\
\text{from} & \quad \hat{e} \\
\text{where} & \quad \hat{e}.\text{Selector} = 'e' \\
\end{align*}
\]

\[(E, R, A, H) = \text{sortConcepts}(s)\]

Figure 6.9: The \( \mathcal{V} \) compilation scheme for HDecomposition

If a concrete database is constructed by:

- copying all the tables from the abstract database apart from the table for the relationship being decomposed;

- adding a table for the new entity, with the primary key field containing all the numbers from one up to the number of rows in the table for the relationship being decomposed;

- adding a new table for each of the new relationship, where each table is constructed by: copying its relevant field within the table for the relationship being decomposed and setting field values for the connection to the new entity equal to the row number.

Then the views retrieve the abstract database, and so the primitive is sound.

**HDecomposition**

Figure 6.9 defines the view compilation rule for the HDecomposition primitive. It has similarities to the \( \mathcal{V} \) rule for HGeneration (see Figure 6.4) as they have similar functionality. The important differences are:

- The HDecomposition primitive can create, as well as extend, hierarchies, so the Selector field is removed when creating a view for the entity before creating the hierarchy.

- When a child is added to a hierarchy using HDecomposition we are identifying values of the parent of the hierarchy that are to be attributed to a
new child. So the abstract parent table still contains the data relating to the child, although the **Selector** field is left missing for those rows (as they are no longer part of the hierarchy).

No special treatment is required for the relationships whose connections change from the parent to the new child, as the values of the primary keys, and hence the relationship values, do not change.

If the concrete database is constructed by copying the abstract database with an extra empty table for the new child, it is clear that the views retrieve the abstract database. It follows that the **HDecomposition** primitive is sound.

### 6.5.3 Valid concrete tables

We have shown how to construct a concrete database where \( V \) acts as a retrieve function. Clearly, in these cases, the resulting views generate valid table data. But we cannot guarantee that \( V \) will generate valid table data for *any* concrete database, even if it is valid itself.

**Example 34** *Given the following concrete tables for the transformation in Example 30:*

<table>
<thead>
<tr>
<th>Thread</th>
<th>becomes</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThreadId</td>
<td>Start</td>
<td>End</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

*The views in Example 30 construct the following abstract table for **Task**:*

<table>
<thead>
<tr>
<th>SparkId</th>
<th>ParSite</th>
<th>ThreadId</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

*which is not a valid table as none of the attributes are a satisfactory primary key.*

Figure 6.10 defines the concrete tables where the views generate valid table data. These mainly ensure that primary keys are observed in the abstract tables.
app(EGeneration _ _) = true
app(RGeneration _ _) = true
app(HGeneration _ _) = true
app(AGeneration e a _) = \( \forall i, j \in \pi(\hat{e}) \cdot \rho(\hat{e}, \text{findPrim}(s, e), i) = \rho(\hat{e}, \text{findPrim}(s, e), j) \Rightarrow i = j \)
app(RSplit _ _) = true
app(EDecomposition e r \langle e_i, a_i, a_s_i \rangle_{i \in I} \text{rels}) = \( \forall i, j \in I \cdot \rho(\hat{r}, \text{findPrim}(s, e), i) = \rho(\hat{r}, \text{findPrim}(s, e), j) \Rightarrow i = j \)
app(RDecomposition _ _) = true
app(HDecomposition _ _) = true

Figure 6.10: The app function takes a refinement that defines the concrete data model, and returns true if its corresponding view definitions generate valid tables for the current database.

6.6 Related Work

6.6.1 Schema refinement

Most of the literature on schema transformations is concerned with the semantic equivalence of transformed schemas [25, 55, 65, 2, 64, 86, 87, 27]. This has applications for normalisation and methods for optimizing user schemas for implementation use. However, these transformations are too restrictive for our purposes, as refinements between the data models, by definition, add new concepts to the schemas.

Although there is some previous work on database schema refinement [6, 5, 97, 15] it is mainly concerned with methods to define and validate data model refinements. The first two papers advocate using the intermediate data models as abstraction levels for documentation purposes, but do not suggest any way to use these abstraction levels in the final database.

Santucci [97] claims to define the semantics of schema refinements. However, his approach seems flawed\(^2\). Firstly, the meaning of the concrete entities and relationships are encoded within the meaning of the abstract one. So, the meaning of the abstract schema changes as more schema refinements are made! Secondly, al-

\(^2\)It is not possible to say whether or not the semantics described by Santucci [97] is correct, because the paper does not communicate enough information. Attempts were made to contact Santucci but no replies were received
though Santucci does consider soundness and completeness, his definitions appear to categorize the majority of valid refinements incorrectly.

Castelli and Pisani [15] describe a schema refinement language for defining correct refinements but do not show how to retrieve the data for the abstract models.

### 6.6.2 View integration

Different views of the same database are often required. We have shown how to construct a database with different views defined by refinement. There exists a body of work called view (or schema) integration [8] that attempts to integrate multiple independent schemas into a single schema. One possible alternative to the view compilation rules defined in this chapter would be to integrate all the data models using the view integration methods. However, this would be a much more complicated process and the problem of whether or not views can be integrated is undecidable [19].

### 6.7 Summary

We now have a method for generating a database that can process queries, using two levels of abstraction determined by a refinement process. Chapter 5 defined the meaning of the primitive set for describing a refinement from abstract to concrete data models. This chapter defined a compilation scheme that generates views, used as retrieve functions, for extracting an abstract instance of a database from a concrete instance. When defining these rules we have also shown the primitives are sound and hence produce valid refinements.

Chapter 7 uses compilation rules to produce a practical tool, a view compiler, which is then integrated into a generic profiling system in Chapter 9. We shall test the effectiveness of the view compiler and the views generated in Chapter 8.
Chapter 7

Refinement: Realization and Implementation

7.1 Introduction

Chapter 6 developed a low-level method for describing schema refinements, and corresponding compilation rules for generating valid retrieve functions. This chapter describes a view compiler that integrates this theory into a usable tool. The compiler’s input is an expression describing a refinement chain. Its output is a set of view definitions for retrieving data for abstract data models in the chain from a database based on a concrete data model. The following tasks suggest one way to implement the compiler using the compilation rules as a starting point.

1. Select a suitable input language;

2. define mappings from the input language to the primitive refinements;

3. and extend the compilation rules to handle a chain of refinements.

The refinement primitives are sufficient to describe most schema transformations but they are not a suitable input language for the user: even simple refinements may require multiple primitive refinements, and constructing the refinements using complex data constructors can be error prone. A high-level input language is required. We could design a specialized language and implement the compiler using language pre-processors, but instead we define higher-order Haskell combinators for describing refinements. The refinements are defined directly using the combinators, and the evaluation of a combinator expression produces the view definitions. The advantage of this method, over using a specialized language, is that parsing and type-checking are provided for free by the Haskell compiler.
7.1.1 Design rationale

At the beginning of this part of the thesis we stated the following aims for the compiler:

1. The output should have a theoretical justification;

2. It should be possible to define all types of refinement that users require;

3. There should be a direct correspondence between the input to the compiler and the refinement chain it describes;

4. The compiler’s input should be type checked, in common with most modern compilers.

The first aim is addressed by using the $\mathcal{V}$ rules in Chapter 6 in an appropriate way. The combinators should be able to define all refinements definable using the primitives; the amount the second aim is achieved is then only limited by the primitives. The third aim is addressed by not requiring explicit usage of type annotations and explicit constructors. The fourth aim is satisfied by implementing the combinators in Haskell.

The design of the combinators is also influenced by the following more subjective aims:

1. Combinator expressions should be clear and look self explanatory.

2. The refinements should be concisely definable.

It is not possible to assign an objective test to the first of these aims. However, we can compare the relative lengths of a combinator expression and its corresponding direct primitive encoding, as a simple metric of conciseness. We could measure the length of an expression by counting the number of characters used, but this can be inaccurate as it is easy to change the number of characters required by a combinator. So instead the length is measured in terms of the number of lexical tokens used.

**Example 35** The expression, the: [cat, sat, on, the, mat], uses 13 lexical tokens. Invisible quotes required around the strings are not counted.

We consider a combinator expression to be concise if it is 10% shorter (uses 10% fewer tokens) than an equivalent primitive refinement expression. This is an arbitrary figure, but provides a suitable target to claim for.
It is not only important to state what our aims are but also to state what our aims are not. Specifically we make no aim that the combinators should be space or time efficient.

Some of the symbols used as combinators in this chapter are not legal Haskell, and are used for presentation purposes only. The is equivalent combinators in the library are defined using the available symbols.

Section 7.2 defines a set of combinators for describing data-model refinement and Section 7.3 shows how these are modified to produce appropriate view definitions. Section 7.4 describes the whole compiler. Related work is discussed in Section 7.5.

### 7.2 Refinement combinators

This section defines a combinator library for describing schema transformations, where the result of a combinator expression is a chain of data models, defined by a source schema, and a list of primitive refinements. In Section 7.3 we extend these combinators to generate view definitions.

#### 7.2.1 The refinement data type

We begin with a type to describe refinement:

```haskell
type Refinement = (Schema → (Schema, [[PrimRefinement]]))
```

Refinement is a function that takes a schema as its argument and returns both the source schema before the refinement and a sequence of lists of primitive refinements. Each of these lists represents one level of abstraction. So the length of the sequence returned by a refinement is equal to the number of levels of abstraction. The target schema is obtained by applying all the primitive refinements in the second argument in turn, to the source schema in the first argument.

The `apply` function returns the result of the refinement, when applied to an empty schema, and the function `prims` retrieves all the primitives.

```haskell
apply :: Refinement → (Schema, [[PrimRefinement]])
apply = flip ($) nullSchema

prims = snd ∘ apply
```

#### 7.2.2 Composition combinators

One of the main advantages of using combinators over explicit data constructors is the ease at which they can be composed. Figure 7.1 defines three composition
\[ (\implies) :: \text{Refinement} \rightarrow \text{Refinement} \rightarrow \text{Refinement} \]
\[ f \implies g = g' \circ f \]

where
\[ g' (s, \text{refs}) = (s, \text{refs} \uplus (\text{snd} \circ g \circ \text{getTarget} \ s) \ \text{refs}) \]
\[ \text{getTarget} \ s \ \text{refs} = \text{foldl} \ \text{refine} \ s \ (\text{concat} \ \text{refs}) \]

\[ \text{refine} :: \text{Schema} \rightarrow \text{PrimRefinement} \rightarrow \text{Schema} \]
\[ \text{refine} \ s \ \text{prim} = T_s[\text{prim}] \cup B_s[\text{prim}] \]

\[ (\iff) :: \text{Refinement} \rightarrow \text{Refinement} \rightarrow \text{Refinement} \]
\[ f \iff g = g \implies f \iff g \]

\[ (\overline{++}) :: \text{Refinement} \rightarrow \text{Refinement} \rightarrow \text{Refinement} \rightarrow \text{Refinement} \]
\[ f \overline{++} g = \text{flatten} (f \implies g) \]

\[ \text{flatten} :: \text{Refinement} \rightarrow \text{Refinement} \]
\[ \text{flatten} \ f = b \circ f \]

where
\[ b (s, l_s) = (s, \text{[concat} \ l_s]) \]

\[ \text{refineSteps} :: \text{[Refinement]} \rightarrow \text{Refinement} \]
\[ \text{refineSteps} = \text{foldr1} (\implies) \circ \text{map} \ \text{flatten} \]

Figure 7.1: Composition combinators.
combinators. The \( \Longrightarrow \) combinator applies the first refinements followed by the second ones, and similarly, the \( \Longleftarrow \) combinator applies the refinements in reverse order. These two combinators treat each refinement as different abstraction levels. On the other hand, the +++ combinator combines two refinements into a single abstraction level. We examine in closer detail the definition of \( \Longrightarrow \).

- \( f \Longrightarrow g = g' \circ f \)
  
The refinement \( f \Longrightarrow g \) is obtained by applying \( g' \) to the result obtained from \( f \).

- \( g' \ (s, \text{refs}) = (s, \text{refs} \oplus (\text{snd} \circ g \circ \text{getTarget} \ s) \ \text{refs}) \)
  
- \( g' \) takes the output from \( f \) and adds the primitives produced by \( g \) onto the primitives produced by \( f \).

- \( \text{getTarget} \ s \ \text{refs} = \text{foldl} \ \text{refine} \ s \ (\text{concat} \ \text{refs}) \)
  
The \text{getTarget} function retrieves the state of the schema after applying all the refinements from \( f \) to the source schema. This function uses \text{refine} which requires Haskell implementations of the \( T \) and \( B \) compilation schemes in Section 5.3.

The \text{flatten} function converts a refinement using multiple abstraction levels into a refinement using a single abstraction level. The \text{refineSteps} function combines a list of refinements, creating a single abstraction level for each refinement.

### 7.2.3 Bottom-up primitive combinators

The composition combinators combine multiple refinements but further combinators are required to describe the actual refinements. To achieve our aim that all refinements definable using the primitives are definable using the combinators we define combinators that replicate the exact functionality of each primitive.

We start by defining combinators for the bottom-up primitives. Apart from the \text{AGeneration} primitive, the bottom-up primitives have only two arguments. So given the function \text{generate}:

\[
\text{generate} \ \text{prims} = \backslash \text{schema} \rightarrow \\
\text{(schema,} \) \text{[prims]}\) \text{ -- The source schema} \text{ -- The primitives}
\]

that creates a refinement from a list of primitives, the primitive refinement:

\[
\text{Prim} \ x \ y :: \text{PrimRefinement}
\]
is represented by the combinator expression \( x \oplus y \), where \( \oplus \) is defined by:

\[
(\oplus) :: a \rightarrow b \rightarrow \text{Refinement} \\
x \oplus y = \text{generate} \ [\text{Prim} \ x \ y]
\]

Figure 7.2 defines primitive combinators for the bottom-up primitive refinements using this method. The \texttt{AGeneration} primitive is translated in a slightly different way to the other primitives. The two different uses of \texttt{AGeneration} (changing or not changing primary key) are handled by two different combinators, and where the primary key is changed the second and third arguments are integrated. Also some of the combinators have additional functionality to their associating primitive, allowing them to handle common cases. For example, the \( \triangleright \) combinator allows specification of an initial attribute set when the new entity is introduced into the model.

**Example 36** The following data-structure describes a refinement that creates two entities in the first abstraction level, followed by two relationships in the next abstraction level, by direct encoding using the primitive refinements. The outer-list represents the abstraction levels, and the inner-lists the primitives defining each refinement:

\[
[[\text{EGeneration} \ \text{Thread} \ \text{ThreadId}, \ \text{EGeneration} \ \text{Processor} \ \text{No}]], \\\n[\text{RGeneration} \ \text{hosts} \ \[((\text{Processor}, \text{Nothing}), \ (\text{Thread}, \text{Nothing}))], \\\n\text{RGeneration} \ \text{sparks} \ \[((\text{Thread}, \text{Just creator}), \ (\text{Thread}, \text{Just created}))]]
\]

This equivalent to the combinator expression:

\begin{verbatim}
prims $ \\
Thread\triangleright\triangleleft\text{PAttribute} \ (\text{ThreadId}, []) \\
     \oplus\oplus \text{Processor}\triangleright\triangleleft\text{PAttribute} \ (\text{No}, []) \\
\Rightarrow \text{hosts} \rightharpoonup [((\text{Processor}, \text{Nothing}), (\text{Thread}, \text{Nothing})) \oplus\oplus \\
     \text{sparks} \rightharpoonup [((\text{Thread}, \text{Just creator}), (\text{Thread}, \text{Just created}))]
\end{verbatim}

### 7.2.4 Mixed-fixed combinators and optional arguments

One way to construct combinators for a type with more than two arguments is to use mixed-fixed combinators; multiple combinators chained together.

**Example 37** Given the data-type:

\begin{verbatim}
data XYZ = XYZ X Y Z
\end{verbatim}

we define two combinators:
-- \textit{EGeneration}
\[
(\triangleright) :: \text{Name} \rightarrow \text{PAttribute} \rightarrow \text{Refinement} \\
\text{name} \triangleright (\text{PAttribute (a,[])} \) = \text{generate [EGeneration name a]}
\]
\[
\text{name} \triangleright (\text{PAttribute (a,as)}) \\
\quad = \text{generate [EGeneration name a,}
\quad \quad \text{AGeneration name Nothing as]}
\]

-- \textit{RGeneration}
\[
(\triangleright\triangleright) :: \text{Name} \rightarrow [\text{ArcLabel}] \rightarrow \text{Refinement} \\
\text{name} \triangleright\triangleright \text{as} = \text{generate [RGeneration name as]}
\]

-- \textit{AGeneration}
\[
(\triangleright) :: \text{Name} \rightarrow \text{PAttribute} \rightarrow \text{Refinement} \\
\text{name} \triangleright (\text{PAttribute (a,as)}) \\
\quad = \text{generate [AGeneration name (Just a) as]}
\]
\[
(\triangleright\uparrow) :: \text{Name} \rightarrow [\text{AttributeName}] \rightarrow \text{Refinement} \\
\text{name} \triangleright\uparrow \text{as} = \text{generate [AGeneration name Nothing as]}
\]

-- \textit{HGeneration}
\[
(\downarrow) :: \text{Name} \rightarrow [\text{Name}] \rightarrow \text{Refinement} \\
\text{name} \downarrow \text{names} = \text{generate [HGeneration name names]}
\]

-- \textit{HEGeneration}
\[
(\downarrow\uparrow) :: \text{Name} \rightarrow [\text{Name}] \rightarrow \text{Refinement} \\
\text{name} \downarrow\uparrow \text{names} \\
\quad = \text{generate ([HEGeneration name name' | name' <- names]})
\]
\[
\text{newtype PAttribute = PAttribute (AttributeName,[AttributeName])}
\]

Figure 7.2: Bottom-up primitives combinators.
(⊕) :: X → Y → (Z → XYZ)
x ⊕ y = XYZ x y

(⊗) :: (Z → XYZ) → Z → XYZ
xy ⊗ z = xy z

to construct objects of type XYZ:

x ⊕ y ⊗ z :: XYZ

We now consider how to construct combinators for data-types with three arguments, one of which is optional.

Optional third argument

In the data-type XYZ:

data XYZ = XYZ X Y (Maybe Z)

the third argument is optional, so we might hope to be able to write the following combinator expressions:

x ⊕ y ⊗ z :: XYZ
x ⊕ y :: XYZ

when the argument is supplied and not supplied, respectively. There are two ways to define combinators of this form.

1. (⊕) :: X → Y → XYZ
   x ⊕ y = XYZ x y Nothing

   (⊗) :: XYZ → Z → XYZ
   (XYZ x y _) ⊗ z = XYZ x y (Just z)

This is the simplest implementation. The ⊕ combinator returns an object of type XYZ with the third argument equal to Nothing. If there is an outer application of ⊗, then the third argument is updated with the new value. One disadvantage of this approach is that it is possible to construct combinator expression such as, w ⊕ x ⊗ y ⊗ z. Also this technique is not as safe if XYZ has more than one constructor, as shown in Example 38.

Example 38 If type XYZ is re-defined as:

data XYZ = XYZ X Y (Maybe Z) | ABC A B C

then (ABC a b c) ⊗ z is type correct but causes a run-time error.
2. infix 4 ⊕ ; infix 5 ⊗

(⊕) :: YZc a ⇒ X → a → XYZ
x ⊕ yz = let YZ (y,z) = unYZ yz in XYZ x y z

(⊗) :: Y → Z → YZ
y ⊗ z = YZ (y,Just z)

newtype YZ = YZ (Y,Maybe Z)
class YZc a where unYZ :: a → YZ
instance YZc YZ where unYZ yz = yz
instance YZc Y where unYZ y = YZ (y,Nothing)

This implementation defines a class (YZc). The instances define valid second arguments of ⊕. The main advantage of this solution is that it does not matter if XYZ has more than one constructor. Its disadvantage is that it uses the class system, that can complicate types and may force explicit type signatures for some expressions.

Optional second argument

In the data-type RST:

data RST = RST R (Maybe S) T

the second argument is optional. Rather than using the type system, we use a visual trick. We create two combinators. The first for when the second argument is supplied:

r ⊕ s ⊗ t :: RST

and the second, where the argument is not supplied:

r ⊗⊗ t :: RST    -- ⊗⊗ is only one combinator

The implementation of the combinators is straight-forward:

(⊕):: R → S → (T → RST)
r ⊕ s = RST r (Just s)

(⊗):: (T → RST) → T → RST
rs ⊗ t = rs t

(⊗⊗):: R → T → RST
r ⊗⊗ t = RST r Nothing t
7.2.5 Attribute and label combinators

The combinator expression in Example 36 uses explicit constructors and is longer than the primitive refinement encoding\(^1\), against the original aims. Figure 7.3 defines combinators for constructing the common types within the PrimRefinement data-type, removing the need for explicit constructors. The combinators are implemented using the methods for handling optional arguments in Section 7.2.4.

These are some of expressions that can be formed with these combinators:

1. \( y \uparrow z :: \text{PAttribute} - \text{primary key } y, \text{with other attributes } z. \)
2. \( x ! y \uparrow z :: \text{EPAttribute} - \text{entity } x, \text{primary key } y, \text{and attributes } z. \)
3. \( x ! y :: \text{EPAttribute} - \text{entity } x, \text{with primary key } y, \text{and no attributes.} \)
4. \( x ! z :: \text{EAttribute} - \text{entity } x, \text{with attributes } z, \text{and no primary key.} \)
5. \( x \mapsto y :: \text{ArcLabel} - \text{a connection with an entity or relationship } x, \text{with label } y. \)
6. \( (x#) :: \text{ArcLabel} - \text{a connection with an entity or relationship } x, \text{where no label is required.} \)

To be consistent the definitions of \(\triangleright\) and \(\triangleright\triangleright\) are re-defined in Figure 7.3 to handle the optional empty list of attributes, example uses include:

7. \( e \triangleright a - \text{new entity } e, \text{with primary key } a; \)
8. \( e \triangleright a \uparrow [b] - \text{new entity } e, \text{primary key } a, \text{and an extra attribute } b; \)
9. \( e \triangleright\triangleright a - \text{add the new primary key } a \text{ to } e; \)
10. \( e \triangleright\triangleright a \uparrow b - \text{add the new primary key } a \text{ to } e, \text{and attribute } b; \)
11. \( e \triangleright\triangleright\uparrow b - \text{add attribute } b \text{ to } e. \)

**Example 39** The refinements in Example 36 are written using the attribute and label combinators as:

\[
\text{prims } \$ \text{Thread } \triangleright \text{ThreadId } \quad \text{+++ } \text{Processor } \triangleright \text{No} \\
\implies \text{hosts } \mid \text{--} \mid [(\text{Processor#}), (\text{Thread#})] \\
\text{+++ } \text{sparks } \mid \text{--} \mid [\text{Thread} \mapsto \text{creator}, \text{Thread} \mapsto \text{created}] \\
\]

This combinator expression uses twelve (25\%) fewer tokens than the primitive refinement encoding, and does not require explicit constructors.

\(^1\)The increase is because the \(\triangleright\) combinator has more functionality and so its types are more complicated.
\[
\begin{align*}
(\uparrow) & : \text{AttributeName} \to [\text{AttributeName}] \to \text{PAttribute} \\
& \quad a \uparrow \text{as} = \text{PAttribute} (a, \text{as}) \\
(\downarrow) & : \text{PA} a \Rightarrow \text{Name} \to a \to \text{EAttribute} \\
& \quad n \downarrow a = \text{let} (\text{PAttribute} (p, \text{as})) = \text{unPA in} = \text{EAttribute} (n, p, \text{as}) \\
(\uparrow\downarrow) & : \text{Name} \to [\text{AttributeName}] \to \text{EAttribute} \\
& \quad n \uparrow\downarrow p = \text{EAttribute} (n, p) \\
(\rightarrow) & : \text{Name} \to \text{Name} \to \text{ArcLabel} \\
& \quad n \rightarrow l = \text{ArcLabel} (n, \text{Just} l) \\
(#) & : \text{String} \to \text{ArcLabel} \\
& \quad n = \text{ArcLabel} (n, \text{Nothing})
\end{align*}
\]

--- \textit{EGeneration} ---

\[
\begin{align*}
(\leftarrow) & : \text{PA} a \Rightarrow \text{Name} \to a \to \text{Refinement} \\
& \quad \text{name }\leftarrow \text{pa} = \text{let} (\text{PAttribute} (a, \text{as})) = \text{unPA pa} \\
& \quad \quad \text{atts }| \text{as}=[] = [] \\
& \quad \quad \text{otherwise }= [\text{AGeneration name Nothing as}] \\
& \quad \quad \text{in generate } [\text{EGeneration name a]}++\text{atts}
\end{align*}
\]

--- \textit{AGeneration} ---

\[
\begin{align*}
(\Rightarrow) & : \text{PA} a \Rightarrow \text{Name} \to a \to \text{Refinement} \\
& \quad \text{name }\Rightarrow \text{pa} = \text{let} (\text{PAttribute} (a, \text{as})) = \text{unPA} \\
& \quad \quad \text{in generate } [\text{AGeneration name (Just a) as}]
\end{align*}
\]

class PA a where unPA : a \to PAttribute

instance PA String\footnote{It is not possible to give an instance of a class for the type synonym \text{String}, without creating a \text{newtype}. This would mean the user would have to provide constructors, which is against our objectives so we use a work-around to obtain an instance for \text{String}.} where unPA s = PAttribute (s, [])

instance PA PAttribute where unPA x = x

instance PA Char where unPA c = PAttribute ([c], [])

instance PA a \Rightarrow PA [a] where

\[
\begin{align*}
\text{unPA} [] &= \text{PAttribute} ([], []) \\
\text{unPA} (x:xss) &= \text{let} \text{PAttribute} ([c], \_ ) = \text{unPA x} \\
& \quad \text{PAttribute} (s, \_ ) = \text{unPA} xss \\
& \quad \text{in} \text{PAttribute} (c:s, [])
\end{align*}
\]

---

Figure 7.3: Attribute and label combinators.
-- RSsplit

(RSplit) :: Name → [Name] → Refinement
name ⊿→ names = generate [RSplit name names]

-- HDecomposition

(HDecomposition) :: Name → [(Name,[ArcLabel])] → Refinement
name | maps
     = generate ([HDecomposition name e rels | (e,rels) ← maps])

none ::: [ArcLabel]
none = []

newtype SInfo = SInfo ([EAttribute],[[ArcLabel,Name]])

(→) :: ArcLabel → Name → (ArcLabel,Name)
x → y = (x,y)

Figure 7.4: Combinators for the RSplit and HDecomposition primitives.

7.2.6 Top-down primitive combinators

The combinators for RSplit and HDecomposition in Figure 7.4 are defined in the same way as the bottom-up primitives as they also have two main arguments\(^2\). However, the remaining primitives, EDecomposition and RDecomposition are translated into mixed-fixed combinators.

Decomposition combinators

The RDecomposition and EDecomposition primitives are similar to each other in that they both decompose a single concept into multiple concepts connected by a new entity or relationship. So we define a shared decomposition combinator, \(\rightarrow\), whose first argument is the concept to be decomposed, and second argument is a function that performs the decomposition:

\((\rightarrow) :: Name → (Name → PrimRefinement) → Refinement\)

\(n \rightarrow f = \text{generate } [f n]\)

Entity decomposition combinators

The \(\rightarrow\) combinator is the outer-most combinator for an entity decomposition, but we still need combinators for constructing the actual decomposition. Using\(^2\) The primitive relating to the combinator \(| |\) (HDecomposition) actually has three arguments, but the second and third argument are paired to form one argument so that the combinator has a similar usage to the \(| |\) combinator that has very similar functionality.
the strategy used so far, we introduce a new combinator, say $\rightarrow$, that allows the following expression:

$$\text{becomes } \rightarrow \ [\text{Thread} \! \text{ThreadId} \uparrow \text{[ThStart,ThEnd]},
    \text{Spark} \! \text{SpkId} \uparrow \text{[PSite]}]$$

However, as the most common entity decomposition results in two new entities, the single combinator and list syntax are replaced by two combinators, and the above expression is written as follows:

$$\text{Thread} \! \text{ThreadId} \uparrow \text{[ThStart,ThEnd]} \leftarrow \text{becomes } \rightarrow \ \text{Spark} \! \text{SpkId} \uparrow \text{[PSite]}$$

This reads more easily because its structure is similar to the corresponding E-R diagram, plus it also requires fewer tokens. Decompositions into more than two entities are handled by further applications of $\rightarrow$:

$$\text{Thread} \! \text{ThreadId} \uparrow \text{[ThStart,ThEnd]} \leftarrow \text{becomes } \rightarrow \ \text{Spark} \! \text{SpkId} \uparrow \text{[PSite]}
    \rightarrow \ \text{AnotherE} \! \text{a}$$

The relative precedence of $\rightarrow$ and $\leftarrow$ has a considerable effect of their types and implementation.

1. If $\leftarrow$ has a higher precedence than $\rightarrow$ and $\rightarrow$ is left associative, suitable types for $\leftarrow$ and $\rightarrow$ are:

$$(\leftarrow) :: \text{EAttribute} \rightarrow \text{Name} \rightarrow (\text{Name} \rightarrow \text{PrimRefinement})$$

$$(\rightarrow) :: (\text{Name} \rightarrow \text{PrimRefinement}) \rightarrow \text{EAttribute} \rightarrow
    (\text{Name} \rightarrow \text{PrimRefinement})$$

This implementation does not enforce the condition that an entity must be decomposed into at least two entities, and expressions such as $(e \rightarrow e_1 \leftarrow)$ are type-correct, despite being meaningless.

2. If $\rightarrow$ has a lower precedence than $\leftarrow$, and again $\rightarrow$ is left associative, then in the expression $(r \rightarrow e_1) \rightarrow e_2$ the first argument of the inner-application needs to be of type Name, and the first argument of the outer-application needs to be of type:

$$(\text{Name}, [\text{EAttribute}], [(\text{ArcLabel}, \text{Name})])$$

Although this can be handled using a sum-type:

$$(\leftarrow) :: \text{EAttribute} \rightarrow \text{EInfo} \rightarrow (\text{Name} \rightarrow \text{PrimRefinement})$$

$$(\rightarrow) :: \text{EInfo} \rightarrow \text{EAttribute} \rightarrow \text{EInfo}$$

data EInfo = N Name | E (Name, [EAttribute], [(ArcLabel, Name)])
\[
(\rightarrow) :\ Name \rightarrow (Name \rightarrow \text{PrimRefinement}) \rightarrow \text{Refinement}
\]
\[
n \mapsto f = \text{generate } [f \ n]
\]

---

**R**\text{De}**composition**

\[
(\Rightarrow) :\ Name \rightarrow \text{RInfo} \rightarrow (Name \rightarrow \text{PrimRefinement})
\]
\[
(!\leftarrow) :\ \text{RI } a \Rightarrow a \rightarrow \text{Name } \rightarrow \text{RInfo}
\]

---

**E**\text{De}**composition**

\[
(!\leftarrow) :\ \text{EPAAttribute } \rightarrow \text{EInfo} \rightarrow (Name \rightarrow \text{PrimRefinement})
\]
\[
a \leftarrow (\text{EInfo } (n,\text{as},\text{rels})) = \text{e } \rightarrow \text{EInfo} \ e \ n \ (a:\text{as}) \text{rels}
\]
\[
(\Rightarrow) :\ \text{EI } a \Rightarrow a \rightarrow \text{EPAAttribute } \rightarrow \text{EInfo}
\]
\[
a \leftarrow (\text{EInfo } (s,\text{as++[b]},\text{remap})
\]
\[\text{where } \text{EInfo } (s,\text{as},\text{remap}) = \text{unEI } a
\]
\[
\text{but } :\ \text{EInfo } \rightarrow [(\text{ArcLabel,Name})] \rightarrow \text{EInfo}
\]
\[
a \text{‘but’ } b = \text{EInfo } (x,\text{y},b) \text{ where } \text{EInfo } (x,\text{y},\_ ) = a
\]

class EI a where unEI :: a \rightarrow EInfo

instance EI String where
\[\text{unEI } s = \text{EInfo } (s,\[],\[])
\]

instance EI EInfo where unEI x = x

Figure 7.5: Decomposition combinators.

explicit constructors would be required in combinator expressions. So instead we use Haskell’s class system again:

\[
(\leftarrow) :\ \text{EPAAttribute } \rightarrow \text{EInfo } \rightarrow (\text{Name } \rightarrow \text{PrimRefinement})
\]
\[
(\Rightarrow) :\ \text{EI } a \Rightarrow a \rightarrow \text{EPAAttribute } \rightarrow \text{EInfo}
\]
\[
\text{instance } \text{EI String }...
\]
\[
\text{instance } \text{EI EInfo }...
\]
\[
\text{newtype } \text{EInfo } = \text{EInfo } (\text{Name}, [\text{EPAAttribute}], [(\text{ArcLabel, Name})])
\]

where explicit constructors are not required.

Figure 7.5 defines \(\leftarrow\) and \(\Rightarrow\) using this last approach. The figure also defines the \textbf{but} combinator for specifying non-default relationship connections, in the same way the \textbf{with} combinator is used for the entity splitting. Its use is optional and is applied to the entity decomposition.

**Example 40** The entity decomposition in Figure 5.9 in Section 5.3.2 is represented in the following combinator form:
prims \( \rightarrow \) \( \text{Thread} \mid \text{ThreadId} \uparrow \left[ \text{ThStart}, \text{ThEnd} \right] \)

\( \left< \text{has} \right> \rightarrow \text{Activity} \mid \text{ActivityNo} \)

'but' \( \rightarrow \left[ \text{sparks} \rightarrow \text{creator} \sim \text{Thread} \right] \)

using fourteen (34%) less tokens than when defined directly.

**Relationship decomposition combinators**

The relationship decomposition combinators \( \triangleright \) and \( \triangleleft \) are similar to the entity decomposition combinators. The type signatures are given in Figure 7.5, with the implementation details omitted but easily derivable from the definitions of \( \triangleleft \) and \( \triangleright \).

**Example 41** The following relationship decomposition when applied in a single abstraction level:

RDecomposition \( \text{worksWith} \) Manager \( \left[ \text{worksWith}, \text{ofDepartment} \right] \)

is represented in the following combinator form:

prims \( \rightarrow \) \( \text{worksIn} \rightarrow \text{worksWith} \triangleright \text{Manager} \triangleleft \text{ofDepartment} \)

using three (25%) less tokens\(^4\).

**7.2.7 Other combinators**

More than one primitive refinement of the same concept may be required to describe some refinements. Figure 7.6 defines two combinators, \( \$\$ \) and \( \$+ \), that handle this concisely. The \( \$+ \) combinator concatenates partial function applications requiring a concept name to produce a primitive refinement. The \( \$\$ \) combinator then defines the concept to supply to all the partial applications, producing a refinement. Haskell contains syntactic sugar for sections that provide a convenient syntax for partial application of binary operators. The section syntax is used with the primitive combinators to define the partial function applications required by \( \$\$. \) So the expression:

\[
e \oplus \exp_1 \leftrightarrow e \otimes \exp_2 \leftrightarrow e \oplus \exp_3
\]

can be written as:

\[
e \$\$ \left( \oplus \exp_1 \right) \$+ \left( \otimes \exp_2 \right) \$+ \left( \oplus \exp_3 \right)
\]

Although more tokens are used when refinements are written in this form, it is clearer that the same concept is being refined.

\(^3\)The relationship decomposition combinators are distinguished from the entity decomposition combinators because the angled-half of the combinator is set in bold.

\(^4\)This figure is calculated by including the extra list brackets required around the primitive to encode the single abstraction level.
($) :: a → (a → Refinement) → Refinement
x $ y = y $ x

($+) :: (Name → Refinement) → (Name → Refinement) → (Name → Refinement)
  x $+ y = 

Figure 7.6: Combinators for defining multiple refinements to the same concept.

(a) AGeneration Thread (Just ActivityNo) []
(b) EDecomposition Thread has

[(Thread, ThreadId, [ThStart, ThEnd]), (Activity, ActivityNo, [])]
[]

Figure 7.7: A refinement built from two refinements of the same entity.

Example 42 Figure 7.7 introduces a new primary key, creating an implicit relationship between the new and old primary keys. This relationship is exposed by decomposing the entity. The transformation is expressed in the following combinator form:

prims $ Thread $ (⇒ ActivityNo)
  $+ (← Thread ! ThreadId↑ [ParSite, ThStart, ThEnd]
  ↩ has ✓ Activity ! ActivityNo)

using eleven (28%) less tokens.
### Table 7.1: Precedence and fixity of the combinators.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Left-associative</th>
<th>Non-associative</th>
<th>Right associative</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td></td>
<td>↑</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>→, !, !↑</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>＞, ＜</td>
<td>＞, ＜</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>‘but’, ‘with’</td>
<td>＞, ＜</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>＄+</td>
<td>＄, ＄＞, ＄＞＞, ＄＞＞＞, ＄&gt;, ＄&gt;↑, [;], [;]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>＋＋</td>
<td>＋＋</td>
<td>＝＝</td>
</tr>
<tr>
<td>2</td>
<td>＝⇒</td>
<td>＝⇒</td>
<td>＝＝</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>＝⇒</td>
<td>＝＝</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>＝⇒</td>
<td>＝＝</td>
</tr>
</tbody>
</table>

**7.2.8 Precedence**

Table 7.1 defines the precedence and associativity of all the combinators. Without these definitions most of the examples require appropriately placed brackets. The composition combinators bind the least tightly, followed by the primitive combinators, then the attribute combinators that bind the most tightly. Most of the primitive combinators are non-associative as multiple applications makes no sense. All ten of Haskell’s precedence levels are used. It is not possible to define a combinator that binds tighter than ↑. Fortunately, we do not require one, but it shows how the arbitrary selection of ten precedence levels in Haskell is perhaps too low.

**7.3 View compilation combinators**

**7.3.1 Updating the combinators**

The combinators in the previous section generate refinement descriptions in terms of chains of primitive refinements. We now modify the combinators to generate the full view definitions to satisfy our original aim. First the **Refinement** type is updated to reflect that each primitive refinement results has an associating list of views (to obtain a database for the model before the primitive is applied, from a database based on the model after the primitive is applied):

```haskell
type Refinement
    = Schema → (Schema, [(PrimRefinement, [View])])
```
prims = map (map fst) o snd o apply
views = map (map snd) o snd o apply

\( \Rightarrow \Rightarrow f = g' \circ f \)
where
\( g' (s, \text{refs}) = (s, \text{refs} \uplus (\text{snd} \circ g \circ \text{getTarget}) s) \)
getTarget s = foldl refine s (concatMap (map fst) refs)

flatten :: Refinement \( \rightarrow \) Refinement
flatten f = b \circ f
where
b (s, ls) = (s, [createViews (getPrims ls) s])
getPrims = map fst o concat

generatePrims = \schema \rightarrow
(scheme, [createViews prims schema])

createViews :: [PrimRefinement] \( \rightarrow \) Schema \( \rightarrow \) [(PrimRefinement, [View])]
createViews ps s = [(p, \mathcal{V}_s[p]) | (s, p) \leftarrow \text{zip ss ps}]
where
ss = scanl refine s ps

Figure 7.8: Integrating view compilation into the combinators. Additions and changes to the original definitions are shown in bold.

As a testimony to the adaptability of the combinatorial style, only a few changes need to be made to the combinators to make them generate the view definitions, as shown in Figure 7.8. The \texttt{createViews} function generates the views for a single level of abstraction by creating a set of views for every primitive refinement using the \( \mathcal{V} \) compilation rules defined in Chapter 6. Although this means that views are created for intermediate data models that are not required by the user, it is simpler than the alternative of creating views for transformations involving more than one primitive.

### 7.3.2 Level and fall-through information

The compilation equations described by \( \mathcal{V} \) in the last chapter rely on there being one abstract model, refined using one primitive refinement to create a single concrete model. The tables in the concrete model are accented to distinguish them from the tables of the same name in the abstract model. The refinements described using the combinators can have several abstraction levels, each defined by one or more primitive refinements. Rather than using multiple accents to
distinguish tables representing concepts with the same names in different refinement levels, the table names are appended with a pair of numbers representing the abstraction level used by the table.

Example 43 The combinator expression:

\[
Thread \triangleright ThreadId \upharpoonright ThreadId \upharpoonright \leftarrow [\text{ProcessornNo}]
\]

\[
\Rightarrow (Thread \leftrightarrow Thread \leftarrow ThreadId \leftarrow \text{RunsOn} \leftarrow \text{ProcessornNo} \leftrightarrow \text{Speed})
\]

defines the refinement chain in Figure 7.9, where (a), (b) and (d) are the different user abstraction levels, and (c) is an intermediate model. The user provides the concrete tables Thread_{2.0}, Processor_{2.0} and RunsOn_{2.0} based on second and final level of abstraction (d), and the compiler generates the following views:

1. \text{Thread}_{1.1}, \text{Processor}_{1.1} and \text{RunsOn}_{1.1}, defined in terms of \text{Thread}_{2.0}, \text{Processor}_{2.0} and \text{RunsOn}_{2.0}, are created for intermediate data model (c).

2. \text{Thread}_{1.0}, defined in terms of \text{Thread}_{1.1}, \text{Processor}_{1.1} and \text{RunsOn}_{1.1}, are created for data model (b).

No views are required for the start schema as it contains no concepts.

Using this methodology a view is created for every entity and relationship in every abstraction level and intermediate data model. In most cases the majority of concepts are unaffected by a transformation and are handled by the default rule, that simply copies the table. One way to remove this waste is to remove intermediate views and update the views for each abstraction level to incorporate the filtering and selection previously performed by intermediate views. However, this analysis is likely to be non-trivial. A far simpler way to reduce the waste is to only generate views for entities and relationships that change characteristics, and record fall-through information for unchanged entities.

Example 44 In the transformation in Example 43 the only views that need to be defined are: \text{Processor}_{1.1} and \text{Thread}_{1.0}. Using the fall-through informa-
module ViewCompiler(module Combinators, viewCompile)

import Combinators

viewCompile :: String -> Refinement viewCompile filename refs
    = writeFile filename $ showViews (views r)
        ++ showFallThrough (fallThrough r)
        ++ showTableFormat (tableFormat r)
    where r = apply refs

Figure 7.10: A section of the ViewCompiler module.

tion, queries involving RunsOn_{1,1} and Thread_{1,1} are replaced by RunsOn_{2,0} and Thread_{2,0} respectively.

Only modest changes need to be made tocreateViewes to incorporate this information.

7.4 The compiler

The refinements combinators are stored in a Haskell module Combinators. The implementor gains access to the combinators by importing the ViewCompiler module. Figure 7.10 outlines the structure of this module. It exports all the combinators and the top-level function viewCompile for generating the output file. The view compiler’s input is a refinement combinator expression and its output is a file containing the view definitions, fall-through information, and the format for the concrete tables. The output is generated in one of two ways:

- by compiling the program using a Haskell compiler, and running the compiled program;
- or, by running the program within a Haskell interpreter.

Example 45 The following module generates the view definitions for a chain and places them into the file outFile:

import ViewCompiler
main = viewCompile "outFile" chain
chain = ...

The outFile is generate by compiling and running this program.
7.4.1 Error handling

Although the view compiler does not enforce the pre-conditions defined in the \( V \) compilation scheme, it does provide minimal error handling. Clearly type errors are picked up by the Haskell compiler at compile-time. The combinators also integrate some error handling which is generated when the Haskell program is executed.

7.5 Related work

One application of combinators is as intermediate code in compilers [105, 39, 38, 78], where combinator expressions are akin to machine-code. This chapter uses an alternative application of combinators where they function as convenient building blocks for constructing expressions specific to an application (view compilation). These types of combinator libraries are prevalent in functional language programming. Indeed, the generic profiling tool, described in Chapter 9, makes use of combinator libraries to parse [46], pretty print [44], graphically display [14], and manipulate XML [110].

Most combinator libraries are used to construct combinator expressions that can be used to process input to produce output. For example, we might use a parser combinator library to construct a combinator expression that takes a string and returns a data structure. The refinement combinators are slightly different, as their structure defines the output and so the top-level combinator expression requires no real input. In this respect they are similar to the combinators used as intermediate code in compilers, which is not surprising as the refinement combinators also define a compilation process.

7.6 Summary

We now have a view compiler. Its input is a Haskell combinator expression defining a refinement chain, and its output is a list of view definitions for retrieving data for different abstraction levels from a single database.

Chapter 8 applies the view compiler to both the GranSim data-model chain in Chapter 3 and a heap profiling data model chain. Chapter 9 shows how the refinement work can be integrated into a generic profiling tool set.
Part III
Experimental evaluation

A generic profiling system
Overview

This final part of the thesis applies the techniques described in Part II.

Chapter 8 evaluates the output of the view compiler described in Chapter 7 when it is applied to two forms of profiling. Chapter 9 shows that the techniques described in this thesis are not restricted to parallel profiling. It describes a generic toolkit, that integrates the interactive tool described in Chapter 4 with the view compiler described in Chapter 7. To show that the tool is more generic that the specialized interactive tool it is applied to the original application, parallel profiling, and heap profiling.
Chapter 8

Refinement: Evaluation

8.1 Introduction

Chapter 6 introduced a theory of refinement and a view generation process for the E-R model. Chapter 7 described a view compiler based on this theory. This chapter shows how the compiler can be applied to two different forms of profiling.

Section 8.2 formulates the parallel profiling data model in Chapter 3 in terms of the view compilation combinators. It examines the resulting view definitions and assess the usefulness of the views. Section 8.3 then shows that the view compilation methodology is not restricted to only one form of profiling; we apply it to a form of heap profiling.

8.2 Evaluation – The parallel model

8.2.1 Expressing the data model chain in combinatorial form

The parallel E-R data model chain needs to be converted into combinatorial form to use the view compiler. However, before this is done the data model chain needs to be finalised because, although the data models in Chapter 3 suggest a possible data model chain, the final chain is not made explicit. It is also necessary to adapt the schemas to include extra information, such as the full attribute information described in Section 5.1.1.

The finalised chain

Figures 8.1 and 8.2 define the data model chain suitable for translation into combinators. Each model represents a different level of abstraction. The chain includes all of the models in Chapter 3 and some extra abstraction levels, which although they are not required, may be of interest to the user. The only differences
between the models in Chapter 3 and their associating models in the explicit chain are the extra attribute information and the removal of optionality information. No changes are made to the structure of the models.

**Converting into combinator form**

The main task in converting an E-R chain into combinatorial form is to establish which transformations describe the movement from one data model to the next. For example, the transformation from model (e) to (f)\(^1\) could be described by:

- **Activity** $\rightarrow^+ [\text{ProcessorNo}]$ $\rightarrow^+ [\text{ActivityNo}]$ $\rightarrow^+ [\text{AcStart, AcEnd}]$

  Decompose Activity into the Processor and Activity entities, related by hosts;

- **Processor** $\rightarrow^+ \text{ProcessorNo}$ $\rightarrow^+ [\text{Activity#}, \text{Processor#}]$

  Introduce new entity Processor, which is connected to entity Activity by a new relationship hosts.

Here the second case applies as the Activity entity is not being decomposed, rather a new concept (Processor) is being introduced to the model.

Figure 8.3 contains a combinator expression for describing the entire data model chain in Figures 8.1 and 8.2. Each element of the list represents an abstraction level.

### 8.2.2 The primitives

One of the original aims, as formulated in Section 7.1, is that combinator expressions should be shorter than the equivalent direct primitive refinement encodings. To ensure this is true for the GranSim data model chain, the chain should be at least 10% shorter when written in combinatorial form. The encoding in Appendix A.1 uses 294 lexical tokens, compared with the combinator expression in Figure 8.3 that uses 64 (22%) fewer tokens, so our goal is achieved. More importantly, the combinator expression is easier to understand and modify than the primitive encoding that resembles assembly language code. As both the combinator and primitive refinement encodings model the same data model chain, the result of prims main is exactly the list of primitive refinements in Appendix A.1.

\(^1\) Excluding the introduction of migrateTo and fetchFrom.
Figure 8.1: A chain of data models to describe parallel profiling (Part I).
Figure 8.2: A chain of data models to describe parallel profiling (Part II).
main = refineSteps $
[\text{Thread} \triangleright \text{ThreadId} \triangleright [\text{ParSite}, \text{ThStart}, \text{ThEnd}]
\implies \text{sparks} \leftarrow [\text{Thread} \rightarrow \text{creator}, \text{Thread} \rightarrow \text{created}]
, \text{Thread} \triangleright \text{ActivityNo}
\implies \text{Thread} \leftarrow \text{Thread} \triangleright \text{ThreadId} \triangleright [\text{ParSite}, \text{ThStart}, \text{ThEnd}]
\leftarrow \text{has} \rightarrow \text{Activity} ! \text{ActivityNo}
\text{but} \implies [\text{sparks} \rightarrow \text{created} \rightarrow \text{Thread}]
\implies \text{Activity} \triangleright \uparrow [\text{AcStart}, \text{AcEnd}]
, \text{Activity} \mid [\text{(Run}, [(\text{sparks} #)]), (\text{Block}, [])]
, \text{Activity} \mid [\text{Queue}]
, \text{Activity} \mid [\text{Migrate}, \text{Fetch}]
, \text{Processor} \triangleright \text{ProcessorNo}
\implies \text{hosts} \leftarrow [\text{(Processor#), (Activity#)}]
\rightarrow \text{migrateTo} \leftarrow [\text{(Processor#), (Migrate#)}]
\rightarrow \text{fetchFrom} \leftarrow [\text{(Processor#), (Fetch#)}]
, \text{Thread} \triangleright \uparrow [\text{SparkId}]
\implies \text{Thread} \leftarrow \text{Thread} \triangleright \text{ThreadId} \triangleright [\text{ThStart}, \text{ThEnd}]
\leftarrow \text{becomes} \rightarrow \text{Spark} \triangleright \text{SparkId} \triangleright [\text{ParSite}]
\text{but} \implies [\text{(sparks} #) \rightarrow \text{Spark}]
\implies \text{Spark} \triangleright \uparrow [\text{CreatedAt}]
\implies \text{liesOn} \leftarrow [\text{(Spark#), (Processor#)}]
, \text{liesOn} \triangleright \uparrow [\text{LieOnId}, \text{LieFrom}, \text{LieTo}]
\implies \text{liesOn} \leftarrow \text{lieOnS} \rightarrow \text{lieOn} \left< \text{lieOnP}]

Figure 8.3: Parallel models described using the combinators.
8.2.3 The views and fall-through information

The views for each abstraction level is given in Appendix A.2. There are a total of sixteen view definitions, averaging two per abstraction level. Compare this to the compilation scheme containing no fall-through information where over ten times more (170) views need to be created! It is possible to reduce the number of views to just eleven by removing intermediate views, as suggested in Section 7.3.2. However, only five fewer would be created, providing some evidence that developing an automatic analysis to remove intermediate views is not significantly beneficial, especially considering the added complexity that would need to be added to the view compiler.

8.2.4 Integrating views into the tool kit

The simplest way to integrate the views into the toolkit is to submit the view definitions to the DBMS when a database is created. Unfortunately, experimentation shows that PostgreSQL’s implementation of views is limited: it can only handle a limited depth of views defined in terms of other views; view of unions are not allowed; and the distinct keyword is not implemented for views. As a work-around the view definitions are replaced with tables that store the data represented by a view. So the view definition:

```
create view a as exp from exp2
```

is replaced by:

```
select exp into a from exp2
```

However, there are some disadvantages of using this method:

- building the database takes longer as sixteen new tables are built;
- extra storage space is required to store the tables, which is not required for views as they are recorded as translation equations;
- updates to the concrete tables do not automatically filter down to the abstract tables. This is not a problem since the profiling database does not change after build time.

None of these disadvantages change the appearance of the database to the user.

**Level selection**

To submit a query using a set level of abstraction the user:
1. selects the abstraction level by entering a number between one to eight that corresponds to one of the data models in Figures 8.1 and 8.2, one being the first model (a), and eight being the concrete model (h);

2. and then submits the query.

The query is intercepted and parsed, and the level and sub-levels tags are attached to the table names using the fall-through information, before being submitted to the database.

**Example 46** To find out which threads queue at some time, we select abstraction level four (which corresponds to model (d)) and submit the query:

```sql
select distinct  Thread.ThreadId
from Thread, Activity, has
where Thread.ThreadId = has.ThreadId
and has.ActivityNo = Activity.ActivityNo
and Activity.selector = ‘queue’
```

which is translated into the following query before being passed to the database:

```sql
select distinct  Thread6.0.ThreadId
from Thread6.0, Activity4.0, has8.0
where Thread6.0.ThreadId = has8.0.ThreadId
and has8.0.ActivityNo = Activity6.0.ActivityNo
and Activity6.0.selector = ‘queue’
```

**Graphical output**

Chapter 9 shows how the view selection, and query results are integrated in a generic setting.

### 8.2.5 Sample queries

The next three examples show how different queries are defined using different abstraction levels.

**Example 47** What is the average amount of time for each of the par-sites?

This query is only concerned with threads, and so can be written using the highest level of abstraction:

```sql
select distinct  Thread.ParSite,
            avg(Thread.ThEnd - Thread.ThStart)
from Thread
group by  Thread.ParSite
```

which is shorter than if the query was expressed using the lowest abstraction level:
select distinct Spark.ParSite, 
    avg(Thread.ThEnd - Thread.ThStart) 
from Spark, becomes, Thread 
where Spark.SparkId = becomes.SparkId 
    and becomes.ThreadId = Thread.ThreadId 
group by Spark.ParSite

Example 48 Which thread created which thread?

select distinct * from sparks

select distinct t1.ThreadId as Creator, t2.ThreadId as Created 
from Thread t1, Thread t2, Spark, sparks, becomes, 
    Activity, Run, has 
where t1.ThreadId = has.ThreadId 
    and has.ActivityNo = Activity.ActivityNo 
    and Activity.ActivityNo = Run.ActivityNo 
    and Run.ActivityNo = sparks.ActivityNo 
    and sparks.SparkId = Spark.SparkId 
    and Spark.SparkId = becomes.SparkId 
    and becomes.ThreadId = t2.ThreadId

Example 49 What processors did all the threads start on?

select distinct Thread.ThreadId, Processor.ProcessorNo 
from Thread, Activity, Processor, has, hosts 
where Thread.ThreadId = has.ThreadId 
    and has.ActivityNo = Activity.ActivityNo 
    and Activity.ActivityNo = hosts.ActivityNo 
    and hosts.ProcessorNo = Processor.ProcessorNo 
    and Thread.ThStart = Activity.AcStart 
    and Activity.selector = 'run'

8.2.6 Experimental results

We have shown that the combinators are sufficient to describe the parallel data 
model chain in Figures 8.1 and 8.2. But is a database that can handle queries 
over all the models any better or worse than a database that can only handle 
queries over a single data model?

As our aim to produce better tools for understanding and improving parallel 
functional programs, one way to define when one type of database is better than 
another is when using it results in better programs. However, the two types of
databases contain the same data (arranged in different tables) so this measurement is not helpful. Instead we compare the databases by asking the following questions:

1. Is the time taken to construct databases any different?

2. Is the time taken to answer queries any different?

3. Is it easier to compose queries in either database?

The first two questions are easily translated into objective tests. However, the third question is more subjective. It is also the most important test given our original claim that it is easier to compose queries over a more abstract model than a concrete model. The best way to test this hypothesis is to run a statistical experiment using a sample of programmers, each trying to understand and improve their parallel programs using the two different types of database. However, with limited resources it is necessary to use a simpler objective test.

**Definition 6** Given query A and query B. Query A is easier to compose than query B, if query A is shorter than query B.

It follows from this definition, that it is easier to compose queries in one type of database than another, if answering the same questions in one type of database leads to shorter queries than the other type of database. The length of a query is measured in terms of the number of lexical tokens used (cf. length of combinator expressions), as this is a fairer test than measuring the number of characters that can easily be manipulated.

Consequently, we use the following tests to determine whether one type of database is better than another:

1. Is the time taken to construct the database faster?

2. Is the time taken to answer queries shorter?

3. Is the length of queries shorter?

The remainder of this section summarises the results of these tests using three different types of database:

A. a database with all the data models in the chain available;

B. one with only the final (most concrete) data model in Figure 8.2 available;

C. and a database based on the original data model in Section 3.4.
Databases are constructed for two different examples, and one set of queries is used in the tests. The first example is the **BarnesHut** program from the **nofib** parallel benching suite [73], an implementation of the Barnes-Hut N-body particle simulation algorithm. It is modified slightly so that the location of each par-site is explicitly recorded. The second example is the first version of the **SumEuler** program described in Chapter 4.

The queries used in the tests are based on a log from an initial analysis of the **BarnesHut** program, where any duplicates and queries that differ only slightly have been removed. Three of the queries correspond to the those in Examples 47, 48 and 49.

All tests were performed on the following architecture: Linux 2.2, AMD™K6-200Mhz PC compatible, with 64MB memory². GranSim produced the log-files, and PostgreSQL was used as the database server. Each test was performed at least three times, with the minimum time/size recorded as the final result.

The databases are built with no indexes because choosing different indexes on different keys can dramatically change the database construction and query execution times. The time taken to built the indexes can often outweigh the speed benefit gained from using indexes. This is especially evident when multiple abstraction levels are included, as indexes need to be constructed for the tables in every refinement level³. It is also not clear on what fields indexes should be created. A natural choice is to create indexes for all the primary keys. However, observation has shown that some entities are rarely used and building indexes for attributes of the most commonly used entities is more worthwhile.

Henceforth we use views to mean the tables created by the view compiler.

**How much time is taken to construct the databases?**

Table 8.1 shows the times taken to build the databases. The database based on the data model in Section 3.4 (type C) is built more quickly than the other two databases for both programs. This is to be expected as the data model was designed using standard translation techniques that ensures little duplicated data. In contrast, the other two data models are translated using the method described in Section 6.4 that leads to more duplicated data, so it takes longer to submit the tables to the database server.

---

²This is a different architecture than the one used in Chapter 4 as I had no local or LAN access to the original machine when the experiments were performed. If the experiments were conducted over the Internet, network delays may have skewed the results. Preliminary tests suggest that similar results would be produced on the original machine.

³If we did not use the work-around (where copies of tables are used instead of views) it may not be necessary to create indexes for every refinement level as the views would automatically use the indexes of the concrete tables.
<table>
<thead>
<tr>
<th>Example</th>
<th>Time taken to build database (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>BarnesHut</td>
<td>19.31</td>
</tr>
<tr>
<td>SumEuler</td>
<td>13.23</td>
</tr>
</tbody>
</table>

Table 8.1: The time taken to build the three different types of database.

<table>
<thead>
<tr>
<th>Example</th>
<th>Disk usage (MBs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>BarnesHut</td>
<td>4.50</td>
</tr>
<tr>
<td>SumEuler</td>
<td>3.14</td>
</tr>
</tbody>
</table>

Table 8.2: The total amount of disk space taken by each database.

The table shows that it takes approximately twice as long to create a database that can handle all data models (type A) than a database based on the original data model (type C). Only one third of this time is spent creating the view tables the rest is accounted for by the duplication of data in the tables. Considering that an extra seven data models can be handled by this database it is a small penalty, especially given that databases are only built once for each set of data.

The amount of disk space used by the databases (see Table 8.2) follows a similar trend to the time taken to build the databases. Given the current prices of disk space (less than one pence per MB), the relatively small amounts used by the databases are negligible.

**How long are the queries?**

Table 8.3 shows the lengths of queries required to produce the same tables using the different types of database. As it is possible to optimize queries to remove extraneous joins, the length of an unoptimized and optimized formation of each query is measured, for each database type. Where the optimized version is identical to the non-optimized version a dash is shown in the table.

**Example 50** The following query:

```sql
select distinct has.ThreadId, sparks.SparkId from has, Activity, sparks
where has.ActivityNo = Activity.ActivityNo
and Activity.ActivityNo = sparks.ActivityNo
```
Table 8.3: Number of tokens required to express eight different queries, using the lowest complexity level (i), abstraction level 8, and the original tables described in Section 3.4. For each of these levels two lengths are measured: the normal query, and one where the query is optimized to remove unnecessary joins. The dashes indicate the query cannot be optimized.

can be optimized to remove the intermediate join:

```sql
SELECT DISTINCT has.ThreadId, sparks.SparkId FROM has, sparks
WHERE has.ActivityNo = sparks.ActivityNo
```

As the type A databases can handle eight different data models the length of the query written using the simplest possible data model is used. $A(i)$ refers database $A$, data model $i$, where $i \in \{1 \ldots 8\}$, $A(1)$ is the simplest data model, and $A(8)$ is the concrete data model.

Queries 2, 7, and 8 are identical when written in any of the data models, so the lengths are the same.

The totals shows that the queries are shortest when written using a database containing all the data models in the chain (type A). They are almost a third shorter than a database containing only the concrete data model (type B). Furthermore, they are 14% shorter than queries written using the original data model (type C), and even if the original data model queries are optimized they only manage to match the length of the unoptimized queries written using the data model chain!

**How long do the queries take to complete?**

Tables 8.4 and 8.5 show the minimum times taken to complete the queries using the different databases for the two examples. Although the times differ, only one
<table>
<thead>
<tr>
<th>Query</th>
<th>Level i</th>
<th>Minimum completion time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A(i)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.423</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.101</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.065</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0.109</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0.900</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>1.176</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0.100</td>
</tr>
<tr>
<td>Sub-Total</td>
<td></td>
<td>2.874</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>140.1</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>143.0</td>
</tr>
</tbody>
</table>

Table 8.4: The minimum time taken to answer the eight queries when applied to the **BarnesHut** example. This table is in the same format as Table 8.3.

<table>
<thead>
<tr>
<th>Query</th>
<th>Level i</th>
<th>Minimum completion time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A(i)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.220</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.126</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.060</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0.047</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0.731</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0.952</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0.060</td>
</tr>
<tr>
<td>Sub-Total</td>
<td></td>
<td>2.195</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>43.48</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>45.68</td>
</tr>
</tbody>
</table>

Table 8.5: The minimum time taken to answer the eight queries when applied to the **SumEuler** example. This table is in the same format as Table 8.3.
<table>
<thead>
<tr>
<th>Example</th>
<th>Minimum total time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>Barnes Hut</td>
<td>156.2</td>
</tr>
<tr>
<td>SumEuler</td>
<td>58.9</td>
</tr>
<tr>
<td>Total</td>
<td>215.1</td>
</tr>
</tbody>
</table>

Table 8.6: The time taken to build databases and evaluate all the queries.

of the queries lasts long enough to be distinguishable to the user. The sub-totals that exclude this query show that the queries written using database type C are fastest, followed by type A, and finally type B. However, when we include the eighth query the results change. In the SumEuler example, the times for C and A differ slightly, whilst the B queries take about 10% longer. In the Barnes Hut example, A is the clear winner for the shortest times, followed by C, and then B.

It is not surprising that the queries written using only the concrete data model take the longest. Relationships are represented as tables instead of the standard way of translating E-R models where relationships are represented as attributes of entities. Consequently, more joins are required in B than C. However, the tables suggest that by allowing queries over all the data models in the chain, these time disadvantages are significantly reduced.

In the Barnes Hut tests the eighth query evaluates at different speeds in the different databases. But, as pointed out above, this query is identical in all the databases and they generate identical results, so why do the times differ? Although the queries are the same, the table data is not. The third level of refinement used in this query models only the running and blocking states. Consequently, only data relating to these states are present in the Activity table. In contrast, the concrete and original models also model queueing, blocking, migrating and fetching, and therefore the corresponding tables include this extra data.

Discussion

Although only a small sample of examples and queries have been tested, there are clear trends in the data. The length and speed of queries in a database which provides all the data models in the chain, is better than a database which only provides the concrete data model. It follows from our definition that a database that can handle queries over all the abstraction levels is easier to use than a database that can handle queries over only the concrete data model.

As expected, the more inefficient way of converting of E-R models into database
tables is costly. If the tables are converted using the standard method, where relationships are represented as attributes of any entity, fewer joins are required and the length and speed of queries is shorter and faster respectively. However, when the full range of data models is available, both the length and speed of queries are reduced. Indeed, Table 8.6 shows that if both of the examples are analysed from scratch using the queries then less time is spent using the multiple data model database than either of the other two databases.

Even if the sample and tests do not represent real world practice, experience has shown the views make writing queries a lot easier. Writing queries based on any data model in the chain is considerably easier than writing queries where the data model is fixed. For simple queries it is tedious to write extra joins required when using the complicated models, and it is easier to compose complex queries when you can choose a data model that does not contain the entity and relationships not required. I found it much easier to write Query 8 for the tests when I was able to select refinement level three, and ignore the complexity introduced in later models, and in this case the resulting query is identical in all the data models!

It is hard to argue that having all the parallel data models available in the refinement chain makes it easier for the user as the test is so subjective. However, the objective tests described in this section do give some evidence that providing all the models does make it easier for the user.

8.3 Evaluation – The heap model

8.3.1 Data-modelling

Figure 8.4 contains a first attempt at modelling the different features provided by the heap profiler in NhC98. We summarize the final model as:

- Each program component produces, constructs and retains a number of heap cells.

- Production and construction are static attributes.

- Cell retention is a dynamic attribute (the cells retained by a program component change during its evaluation), and more than one program component may retain the same cell at the same time.

- Each heap cell has many activities in its life-time. At any one time a cell may lag, drag, be void or be used (LDVU), where definitions of these four activities are those used by Röjemo and Runciman [85], see Figure 8.5.
Figure 8.4: A theoretical chain of data models for heap profiling.
This chain encapsulates an internal description of the information recorded in the heap cells, but cannot be used for a number of reasons. Firstly, although the heap profiler records information about each heap cell, due to the sheer amount of data that would result it does not record all this information in the log-file. Secondly, the current version of the heap profiler only allows a single producer, construction, or retainer profile to be generated per program run. Furthermore, it is not possible to merge producer, construction and retainer profiles from three different program runs, because even with identical input the samples are not produced at the same time and there is no way of determining how the producers, constructors, and retainers relate to each other. Nevertheless, it is possible to obtain LDVU information for producer and construction profiles in a single run, and using the two-pass technology [89] LDVU information can be obtained for retainer profiling.

The main consequences of this are that heap cells need to be grouped together, and a different chain of data models is required for each type of profiling (except biographical profiling as LDVU information can be integrated into the other chains).

**Producer profiling**

The heap profiling log-files record information about the frequency of heap cells with the same producer, constructor, or retainer set and with exactly the same biographical information. Using this as a basis Figure 8.6 contains a data model chain for producer profiling. The refinement process is summarized as:

(a) Each program component (e.g. top-level function) has a source code reference that we call **name**.

(b) Each component produces heap cells. These are grouped into blocks with the same creation time (**startTime**) and last use time (**endTime**).

---

*It is possible to perform limited restrictions, such as show producer information where the constructor is (:) though.*
Figure 8.6: A chain of data models for biographic heap profiling by producer. The final combinator expression defining the transformation is: `main = refineSteps [(a), (b), (c), (d), (e)]`. 
Figure 8.7: The two biographical phases of a heap cell in Figure 8.6, Model (c).

(c) A cell’s life-time can be divided into two phases: use, where the cell is in use or going to be in use in the future; and waste, where the cell still in the heap but is not going to be used again. The two phases are illustrated in Figure 8.7.

(d) Cell waste is divided into heap drag and heap void whose definitions correspond to that in Figure 8.5.

(e) Cell use is divided into lag and use. In this final data model the lag, drag, void and use phases are exactly equivalent to the LDVU phases in biographical profiling (see Figure 8.5).

In Model (c) the use phase is equivalent to the lag and use phases in standard biographic profiling [85]. Is it a meaningful abstraction to put these two phases together? In the closing discussion of their original biographic profiling article [85], Röjemo and Runciman say:

“... classifying lag cells as ‘useless’ is indeed rather harsh. They are not useless in the same sense that drag and void cells are. ... lag cells are not redundant: they will be needed eventually. ... So a more generous assessment of the useful part of the heap might include both use and lag cells ...”

Following this argument, treating lag as useful, or at least not as wasteful as void and drag, is a valid abstraction: the user is able to distinguish between the heap cells that could be garbage collected and those that cannot. Interestingly for purposes of biographic retainer profiling Runciman and Röjemo [89] sort the four phases into two groups using the question has a cell been used?. The effect is to group use and drag, and lag and void. Although this is useful for the purpose of the implementation, would such a division be a useful abstraction for the purpose of understanding heap usage in a program? Grouping together use and drag is counter-intuitive, it is hard to argue that drag cells are ever useful!
8.3.2 Expressing the data model chain in combinatorial form

Figure 8.6 gives the combinatorial representation of the data-model refinement under each successive data-model. Converting most of the refinements is a straightforward translation process.

8.3.3 The primitives

If the data model chain is written directly using the primitives (see Appendix B.1), approximately 13\% more tokens (12) are required than the equivalent combinator expression. We have achieved our aim that the combinator expressions should be concise for the heap profiling by producer.

8.3.4 The views

The full set of view definitions generated by the compiler are given in Appendix B.2. A total of eight views are generated, averaging just over one per refinement level.

8.3.5 Sample query

Example 51 What is the maximum amount of waste during the computation?

```
<table>
<thead>
<tr>
<th>Level 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>select distinct</strong></td>
</tr>
<tr>
<td><strong>from</strong></td>
</tr>
<tr>
<td><strong>where</strong></td>
</tr>
<tr>
<td><strong>and</strong></td>
</tr>
<tr>
<td><strong>and</strong></td>
</tr>
<tr>
<td><strong>order by</strong></td>
</tr>
</tbody>
</table>
```

In this query each line of the resultant table contains the amount of waste (calculated as \(\text{sum}(h2.population)\) where \(h2.selector = \text{‘Waste’}\)) at each possible time where waste may increase (as determined by \(h1.startTime\)). The results are then ordered so that the maximum waste is at the top of the table.

8.3.6 Construction and retainer profiling

The data models in Figure 8.6 only handle producer profiling, to apply the view compiler to construction and retainer profiling two similar chains are required.
8.3.7 Discussion

We now ask what extra functionality is gained using a database that imports the views compared to a database that does not. Since the final data model in the refinement chain is relatively simple, with only one relationship, it follows that the other refinement levels represent only small changes in abstraction. In fact any query over one of the abstract models is a valid query on the concrete model and the same result table will be generated using both models! This begs the question: *Is the refinement work described in Part II only of real use for parallel profiling, and other forms of profiling can only expect small benefits?*

The reason why the parallel profiling example achieves better results than the heap profiling example is because of the differences in the complexity of the concrete data models. The concrete data model in the parallel chain is relatively complex and therefore offers many possible abstractions. On the other hand, the concrete data model in heap profiling chain is relative simple and therefore does not offer many possible abstractions. So it is fairer to say that: *The refinement work described in Part II is most useful when the concrete data model is sufficiently complex to afford abstraction.* If it were possible to combine the different forms of profiling (constructor, retainer and producer) then there would be greater opportunities for abstraction.

8.4 Summary

In this chapter we applied the view compiler to two different forms of profiling. The combinator library described in Chapter 7 is sufficient to describe the two data model chains in a concise manner, achieving one of the aims of the combinator.

We have shown that, when the chain of data models is available for parallel profiling, queries can be more concise and evaluate faster. The heap profiling example does not achieve the same level of success. This can mainly be attributed to the simplicity of the concrete data model. The refinement process is more valuable where a model contains sufficient complexity to warrant higher levels of abstraction.

Chapter 9 shows how the view compilation process is fully integrated within a generic toolkit.
Chapter 9

Generic profiling tools

9.1 Introduction

Although this thesis was originally motivated by the need for better parallel profiling tools for functional languages, the methodologies used are not restricted to parallel profiling. They are more generic, and can be applied to other forms of profiling. This chapter develops generic versions of the tools presented in the previous chapters, and shows how they can be applied to heap profiling. Making the tools more generic has obvious advantages. The implementor no longer has to develop new browsing tools for different forms of profiling. The user continues to work in a familiar environment, and new forms of profiling immediately benefit from existing tools.

Section 9.2 describes changes to the current set of tools to make them more generic, and shows in what respect the tools demonstrate this property. Section 9.3 shows how these new, more generic tools can be applied to parallel profiling, and assesses changes in performance and functionality through the use of these tools compared to those of the specialized parallel tools described in previous chapters. Section 9.4 gives some evidence that the tools are more generic by applying them to a form of heap profiling. Section 9.5 describes the implementation of the new tools. Section 9.6 compares the generic profiling system with other related works.

9.2 Making the tools generic

One way to show that the techniques in previous chapters can be applied to other types of profiling is to produce a complementary set of tools for another form of profiling. A better way of doing this is to develop a set of generic tools that can handle any profiling data arranged in a standardized way. This section describes changes to make the tools more generic.
Figure 9.1: Structure of the tool kit. The dotted lines separate the 3 distinct stages. The solid lines show the direction of input/output, and the dashed lines show where the output from one component affected the implementation of another. The encapsulating box show the boundaries of the tool kit and the different user and implementor inputs and outputs.

We start by reviewing the current tool kit (see Figure 9.1). It has five main components: the View compiler, the Builder, the Database, the Query evaluator, and the GUI. We can divide the use of these components into three distinct stages:

1. GranSim's data model developed in Chapter 3 is submitted to the view compiler described in Chapter 7. This produces the format of the database tables and view definitions, used to create the components used in stages two and three.

2. The user submits a GranSim log-file to the Builder

3. The user extracts information by interacting with the GUI.

Our aim is to generalize each of the stages, removing any parallel profiling specialization. We begin with the first two stages, grouped together because the output of the first stage depends heavily on the data required by the second.

### 9.2.1 View generation and Building the databases

The view compiler is not specialized for parallel profiling even though it was developed with it in mind. Its input is a list of refinement combinators, and its output is a list of views, neither of which is necessarily related to parallel profiling. So the Builder is the main component to be generalized in the first two stages.

The Builder takes a GranSim log-file and uses this to build database tables. Clearly, a generic Builder has to remove the need for the input to be a GranSim

---

1 This is not strictly correct. As pointed out in Chapter 7 the query tool was implemented around a set of database tables as described in Chapter 3. The tool had to have minor modifications to handle the new database table format generated by the view compiler.
log-file. It would be unreasonable to expect a tool to be able to take any log-file in any format and interpret the data, so the log-file is standardized. A number of approaches have been suggested for achieving this. Both Halstead’s Vista system [29] and the Pablo system [83] use a form of self-describing log-files, specially designed for performance related data. The log-files contain a header which contains information describing how to interpret the body of the message. However, we use the standard log-file format proposed by Jarvis et al. [48]. Their motivation was that a common format for compiler-generated log-files would simplify the sharing of profile browsers for different types of profiling within the Haskell community.

Jarvis et al. ’s log-file format, XML-Data\textsuperscript{lite} (XML-DL), is a subset of XML-Data — a file format suitable for storing database related information — which in turn is a subset of XML. XML-Data [58], proposed by Microsoft\textsuperscript{TM}, and used in new versions of their software [20], has not yet been ratified by the standards agency, W3C. Since developing XML-DL alternative competing proposals (XML Schema [101, 9]) have got further in the standards process, and are expected to be ratified instead. However, we use this format because at least one functional language compiler (ghc [76]) is planning to base its profiling data format on XML-DL.

An XML-DL log-file is composed of two files: a DTD that describes the format of the data file\footnote{The authors’ choice of naming this file as a DTD can be confusing, as it is not a strict XML DTD as defined in the XML specification but a schema definition as defined in the XML-DL specification.} and the actual data file. Using these files as a basis to develop the generic versions of the tools, stages one and two are structured as shown in Figure 9.2.

![Diagram](image-url)

Figure 9.2: Structure of stages one and two in the generic tool kit..

In stage one the implementor submits a chain of data models defined in terms of refinement combinators. This produces two main outputs: the SQL views and table definitions, and a minimal DTD file. As the name suggests the minimal DTD contains a specification of the minimum amount of data that must appear in the XML file to build the database, based on the chain of data models submitted...
to the view compiler. The implementor then updates the minimal DTD (by hand) with extra information, and ensures that generated XML log-files conform to this new DTD. Section 9.2.3 gives more details about how and why the DTD can be modified.

In stage two the user submits the full DTD and a conforming XML log-file to the generic Builder. This then selects the appropriate information from the XML file and builds a database. A fuller description of these two stages follows.

9.2.2 Stage 1: View compilation

In the original view compiler there were three outputs; the SQL views, the final database table definitions, and the fall-through information. In the generic case the view compiler also generates a minimal DTD. For the moment we assume that this is the actual DTD given to the Builder.

We begin with a short introduction to XML-DL DTDs, before looking at the rules for creating a DTD. A DTD has an XML version header `<XML version='1.0'>` and a body enclosed within the schema tags. Within the schema definition is a list of nested tags.

1. The `elementType` tag has an `id` attribute used for naming purposes and one or more sub-elements. It is effectively a record construction tag.

2. The `element` tag declares a sub-element (cf. record field). It has two attributes: `key`, used to associate a name to the data, and `type`, that gives the type of the data.

   Example 52 A thread record with two fields, `id` of type `int`, and `parSite` of type `string`.

   ```xml
   <elementType id="thread">
     <element key="id" type="int"/>
     <element key="parSite" type="string"/>
   </elementType>
   ```

3. The `foreignKey` tag is a different type of sub-element, used to reference a key of an previously declared `elementType`. It has two attributes: `range`, that defines the `elementType` being referenced, and `key`, that defines the `element` referenced.

   Example 53 A sparks record has two references to the `id` field in the thread record and a time field.
<elementType id="sparks">
  <foreignKey range="#thread" key="#id"/> <!-- creator -->
  <foreignKey range="#thread" key="#id"/> <!-- created -->
  <element key="time" type="int"/>
</elementType>

4. The group tag declares a sequence of fields. The attribute occurs defines whether the sequence is optional, required, of length one or more, or length zero of more.

Example 54 A log-file has a filename and a list of thread records.

<elementType id="log-file">
  <element key="filename" type="string"/>
  <group occurs="ZEROORMORE">
    <foreignKey range="#thread" key="#id"/>
  </group>
</elementType>

Translating an E-R schema into a DTD

The implementor will use the DTD as a specification for the XML file format, and the Builder will use the DTD both to parse the XML file and to construct database tables. Therefore, the DTD needs to contain a description of all the tables, including the types of the fields.

The refinement combinators do not record the types of the attributes, so the view compiler assumes that all the fields are of type integer, except for the special Selector field which is of type string.

Figure 9.3 contains the rules for creating the DTD from an E-R schema. The resulting DTD has three parts: a description of the overall log-file, the translation of of the entities, and the translation of the relationships.

Translating the entities

Each entity is represented as a single element with sub-elements for each of its attributes. If the entity is a super-type then sub-element represents the Selector attribute. Where the entity is a sub-type its primary key is inherited from its associating super-type, and the foreignKey tag is used to reference this.

Example 55 Given the schema containing the two entities and a hierarchy:

E ("Activity", ["AcStart","AcEnd","ActivityNo"])  
E ("Run", ["ActivityNo"])  
H ("Activity", "ActivityNo", ["Run"])
Compilation of the entire schema

\( \mathcal{X}[s] = \)

```xml
<?xml version="1.0" ?>
<s:schema id="date">
  ⟨\( \mathcal{X}[e] \)\( e \in E \)⟩
  ⟨\( \mathcal{R}[r] \)\( r \in R \)⟩
  <elementType id="logfile">
    <group groupOrder="OR" occurs="ZEROORMORE">
      ⟨element type="#n"/>\( n \in \{r \mid (r, e) \in R \} \cup \{e \mid e \in s\} \)
    </group>
  </elementType>
</s:schema>
```

where

\( (E, R, A, H) = \text{sortConcepts}(s) \)

Compilation of the entities

\( \mathcal{X}[e (e, \_)] = \)

```xml
<elementType id="e">
  ⟨element key="a" type=int/⟩\( \forall a \in as \} \)\( \forall a \notin \{\text{findPrim}(p, s)\} \)\( (p, ns) \in H, c \in ns \)⟩
  ⟨element key="Selector" type=string/⟩ (p, ns) \in H, p = e ⟩
  ⟨foreignKey range="#p" key="#(\text{findPrim}(p, s))"/>\( p \in \{p \mid (p, ns) \in H, c \in ns \} \)
</elementType>
```

where

\( as = \text{findAttr}s(e, s) \)

\( (E, R, A, H) = \text{sortConcepts}(s) \)

Compilation of the relationships

\( \mathcal{R}[r (r, ros)] = \)

```xml
<elementType id="r">
  ⟨foreignKey range="#c" key="#(\text{findPrim}(e, s))"/>\( e, \text{Nothing} \)\( \in ros \)
  ⟨foreignKey range="#c" key="#(\text{findPrim}(e, s))" label="lab"/>⟩\( (e, \text{lab}) \in ros \)
</elementType>
```

Figure 9.3: Compilation equations for \( \mathcal{X} \).
the following definitions are derived:

```xml
<elementType id="Activity">
  <element key="AcStart" type="int"/>
  <element key="AcEnd" type="int"/>
  <element key="ActivityNo" type="int"/>
  <element key="Selector" type="string"/>
</elementType>

<elementType id="Run">
  <foreignKey range="#Activity" key="#ActivityNo"/>
</elementType>
```

An example XML file conforming to this partial DTD would be:

```xml
<Activity>
  <AcStart>0</AcStart><AcEnd>5</AcEnd><ActivityNo>0</ActivityNo>
  <Selector>Run</Selector>
</Activity>
<Run>0</Run>
<Activity>
  <AcStart>2</AcStart><AcEnd>3</AcEnd><ActivityNo>1</ActivityNo>
  <Selector>Run</Selector>
</Activity>
<Run>1</Run>
```

Translating the relationships

Given that relationships are between entities, each relationship is represented as a number of foreignKeys. A relationship between distinct entities translates into an element with a foreignKey for each participant in the relationship, with the range equal to the entity being related and the key equal to the attribute involved in the relation.

Example 56  The relationship

R (sparks, [(ActivityNo, Nothing), run], (SparkId, Nothing), Thread)

translates into the DTD form:

```xml
<elementType id="rel">
  <foreignKey range="#Run" key="#ActivityNo"/>
  <foreignKey range="#Thread" key="#SparkId"/>
</elementType>
```
Where a relationship involves the same entity more than once, each of the arcs is labelled to distinguish them (e.g. creator and created). There is no way to label the references in XML-DL, so if a relationship relates to the same entity more than once we must distinguish the different fields by their order of occurrence. Although this may be acceptable for most uses of XML-DL the labels need to be encoded as they are required by the Builder. Also the implementor will use the DTD to create the XML files, so they need to distinguish the different fields. We now look at four ways to do this, the first three are XML-DL compliant, and the last one requires an extension to XML-DL. The methods will be illustrated by representing the following relationship in DTD form:

\[
\text{rel} \leftarrow \rightarrow [e \rightarrow \text{left, } e \rightarrow \text{right}]
\]
\[
\simeq \text{R (rel, [(e,Just left), (e,Just right)])}
\]

where a is the primary key of e.

1. Rather than using the foreignKey tag to reference the objects, we copy the DTD definitions being referenced, replacing the key attribute with the labels.

```xml
<elementType id="rel">
  <element key="left" type="int"/>
  <element key="right" type="int"/>
</elementType>
```

Referencing is completely lost. So if the implementor changes the type of the attribute a in this example they must also remember to change the type of left and right.

2. The relationship is converted in the usual way, with the name of the label in a comment.

```xml
<elementType id="rel">
  <foreignKey range="#e" key="#a" <!-- label="left" -->/>
  <foreignKey range="#e" key="#a" <!-- label="right" -->/>
</elementType>
```

Although this is valid XML, placing important information inside comments is a hack and emphasizes the holes in XML-DL.

3. We create new records for the labels. This is similar to the correct way of doing this in XML Data, where additional key tags are required.
The problem with this method is that no two relationships can have the same labels.

4. We add an extra optional argument to a foreignKey tag to include a label.

This is the cleanest approach but is not valid XML-DL.

Although the fourth option is not XML-DL compliant it is XML compliant. So, we make this extension to XML-DL and use it to translate the relationships in X/R (see Figure 9.3).

**The log-file description**

The X/R and XE translation rules give the format for the different data items. The final tag is logfile which defines the top-level tag used to store the log-file information.

**Example 57** A DTD for the GranSim parallel data model schema in Section 8.2.1 is given in Appendix A.3.1.

### 9.2.3 Stage 2: Building the databases

As mentioned earlier, the view compiler will produce a minimal DTD which the implementor can modify and pass to the Builder, but the implementor cannot edit the DTD without restriction. They must not delete any definitions, or change any present definitions. The implementor is allowed to make the following modifications:
• Changing the types of the objects. Each of the types defaulted to be an integer in view compiler. Where the implementor wishes to use a different type in the XML file they change the appropriate definition in the DTD.

Example 58 The user may wish to change the type of parSite to string in the parallel dtd (Appendix A.3.1).

```xml
<elementType id="Spark">
   <element key="SparkId" type="int"/>
   <element key="ParSite" type="string"/>
   <element key="CreatedAt" type="int"/>
</elementType>
```

• Adding extra information to the DTD. One advantage of using XML files for storing profiling information is that the same XML file can be used as input to a number of different profile browsers. Different browsers require different information, so for effective use of the XML files it is not a good idea for the view compiler to dictate exactly what must appear in the DTD. Instead the implementor can add both extra fields and extra records to the DTD before passing it to the Builder.

Example 59 A different profiling tool may want to know which colour to use to represent each spark, so a new line is added within the spark definition.

```xml
<elementType id="Spark">
   <element key="SparkId" type="int"/>
   <element key="ParSite" type="string"/>
   <element key="CreatedAt" type="int"/>
   <element key="Colour" type="int"/>
</elementType>
```

The Builder takes the DTD and a XML file, builds the database tables from these, then submits the tables to the database. It also automatically passes on the view definitions to the database.

9.2.4 Stage 3: The Front-end

The specialized tool provides SQL query templates for creating activity and per-thread graphs, that mimic the graphs produced by the GranSim toolkit. If a query does not match one of these templates then it is not possible to produce
a graphical representation. A generic profiling tool could use standard data visualisation techniques [4] to display any query result. Instead, the generic tool implements a simpler and more limited graphical system based on the specialized tool. This section describes the initial research integrating the view information and interactive profiling within the generic setting. The result is not, and does not try to be as complete as a general data visualisation system.

View selector

In the specialized tool a user selects the view level by entering a number representing the level of refinement (see Section 8.2.4). As the chain of data models is fixed the user quickly learns which data model corresponds to which number. However, in the generic situation the chain of views is not fixed, so this is unlikely to happen. This situation is even worse if the user is not the implementor; they may have no idea what the final data model is.

This problem is solved by providing a browser showing the different data models which are available. The refinement combinators may be too low-level for the user to understand, so instead the view browser presents the equivalent E-R diagram for each view level. Not only does this help the user to choose the view level they require, but it also shows them all the table and fields names required to construct SQL queries.

Queries to generate graphs

The specialized tool uses two query templates to construct graphs. We identify seven query-variables in these templates:

1. \textit{thr} - an entity table representing a thread of computation;

2. \textit{thrPrim} - the primary key of \textit{thr};

3. \textit{act} - a super-type entity table representing the different activities relating to a thread. Its fields include:

4. \textit{actPrim} - the primary key of \textit{act};

5. \textit{start} - a start time attribute;

6. \textit{end} - an end time attribute;

   Its associating sub-types represent the different states of the thread of computation.

7. \textit{rel} - a table representing the relationship between the thread and its activities.
<table>
<thead>
<tr>
<th>select distinct</th>
<th>a1.start, a1.end, (a1.count)</th>
<th>a1.Selector</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>a1.act</td>
<td></td>
</tr>
<tr>
<td>order by</td>
<td>a1.start</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>select distinct</th>
<th>a1.start, a1.end, a1.Selector, tl.thrPrim, a1.actPrim</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>a1.act, tl.thr, rl.rel</td>
</tr>
<tr>
<td>where</td>
<td>a1.actPrim = rl.actPrim</td>
</tr>
<tr>
<td>and</td>
<td>tl.thrPrim = tl.thrPrim</td>
</tr>
<tr>
<td>order by</td>
<td>tl.thrPrim, a1.start</td>
</tr>
</tbody>
</table>

Figure 9.4: SQL queries used to create the activity and per-thread graphs, respectively. Substituting \( \text{start} = \text{AcStart}, \text{end} = \text{AcEnd}, \text{actPrim} = \text{ActivityNo}, \text{act} = \text{Activity}, \text{thr} = \text{Thread}, \text{rel} = \text{has}, \text{thrPrim} = \text{ThreadId} \) and omitting the \( \text{count} \) variable gives the queries needed for the parallel model.

\[
\mathcal{TF}[s] = 
\begin{cases} 
((\text{thr}, \text{findPrim}(s, \text{thr})), (\text{act}, (\text{findPrim}(s, \text{act}), \text{start}, \text{end}), \text{rel})) \\
H(\text{act}, \_ ) \leftarrow s, A(\text{act}, \_ ) \leftarrow s, \\
\text{start} \leftarrow \text{as}, \text{end} \leftarrow \text{as}, \text{start} \neq \text{end}, \\
\text{start} \neq \text{findPrim}(s, \text{act}), \text{end} \neq \text{findPrim}(s, \text{act}), \\
R(\text{rel}, \{(a, \_), (t, \_ )\}) \leftarrow s, a = \text{act}, t = \text{thr} 
\end{cases}
\]

Figure 9.5: Rules for deriving the query variables from a E-R schema.

Figure 9.4 gives the queries for the activity and per-thread graph in terms of these variables, and shows what substitutions are required to obtain the query templates in the parallel tool.

To produce these graphs using these queries we need to determine the \textit{query-variables} for any data model. The natural place to record this information is within the refinement combinators. However, to avoid complicating the combinator expressions, the generic tool determines all the possible query-variable combinations using the \( \mathcal{TF} \) rules in Figure 9.5, and asks the user to select one of them.
select distinct a1.start, a1.end, t1.thrPrim
from a1 act, t1 thr, r1 rel
where a1.actPrim = r1.actPrim
and r1.thrPrim = t1.thrPrim
order by a1.start, t1.thrPrim

Figure 9.6: The query template to produce a thread activity graph.

Example 60 Applying $\mathcal{TF}$ to the final GranSim schema in Section 8.2.1 generates the following possible query-variable combinations:

<table>
<thead>
<tr>
<th>thr</th>
<th>thrPrim</th>
<th>act</th>
<th>actPrim</th>
<th>start</th>
<th>end</th>
<th>rel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>ThreadId</td>
<td>Activity</td>
<td>ActivityNo</td>
<td>AcStart</td>
<td>AcEnd</td>
<td>has</td>
</tr>
<tr>
<td>Thread</td>
<td>ThreadId</td>
<td>Activity</td>
<td>ActivityNo</td>
<td>AcEnd</td>
<td>AcStart</td>
<td>has</td>
</tr>
<tr>
<td>Processor</td>
<td>ProcessorNo</td>
<td>Activity</td>
<td>ActivityNo</td>
<td>AcStart</td>
<td>AcEnd</td>
<td>hosts</td>
</tr>
<tr>
<td>Processor</td>
<td>ProcessorNo</td>
<td>Activity</td>
<td>ActivityNo</td>
<td>AcEnd</td>
<td>AcStart</td>
<td>hosts</td>
</tr>
</tbody>
</table>

If the third combination is selected then the per-thread graph query template is:

select distinct a1.AcStart, a1.AcEnd, a1.Selector, t1.ProcessorNo, a1.ActivityNo
from a1 Activity, t1 Processor, r1 hosts
where a1.ActivityNo = r1.ActivityNo
and r1.ProcessorNo = t1.ProcessorNo
order by t1.ProcessorNo, a1.AcStart

Viewing the activity from a different perspective

An activity graph in a parallel profile shows a breakdown of all the activities at any given time. The generic tool also incorporates another type of activity graph which shows all the activities associated to each threading component. Figure 9.6 shows the query template. To avoid confusion we call the original activity graph the state activity graph, the new activity graph the thread activity graph, and where the distinction is not required simply, the activity graph.

Example 61 Using the final GranSim data model: The following query can display a graph showing which threads are running across the computation:
select distinct a1.AcStart, a1.AcEnd, t1.ThreadId
from a1 Activity, t1 Thread, r1 has
where a1.ActivityNo = r1.ActivityNo
and r1.ThreadId = t1.ThreadId
and a1.Selector = 'run'
order by a1.AcStart, t1.ThreadId

Generating the graphs

The resulting tables generated from queries based on the query templates are post-processed in order to be displayed as graphs.

Activity graphs: Both types of activity graphs are processed in the same way. The first two columns of the result (start and end) define the x-axis start and end points for a block representing the third column (thrPrim for thread activity or Selector for state activity). Where there is more than one block for a given value of the third column and a given x-axis position the blocks are vertically juxtaposed.

Example 62 The following result table:

<table>
<thead>
<tr>
<th>a1.start</th>
<th>a1.end</th>
<th>a1 Selector</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>X</td>
</tr>
</tbody>
</table>

is post-processed to generate the graph (a) in Figure 9.7.

Per-thread graphs: The first two columns (start and end) define the x-axis start and end points respectively. The third column defines the type of activity
(State/Thread) Activity:

\[ \forall i \in \pi(t) \bullet \rho(t, \text{start}, i) \leq \rho(t, \text{end}, i) \]

Per-Thread:

\[ \forall i \in \pi(t) \bullet \rho(t, \text{start}, i) \leq \rho(t, \text{end}, i) \]
\[ \forall i, j \in \pi(t) \bullet (\rho(t, \text{start}, i) < \rho(t, \text{end}, j)) \land (\rho(t, \text{start}, j) < \rho(t, \text{end}, i)) \]
\[ \Rightarrow (\rho(t, \text{thrPrim}, i) \neq \rho(t, \text{thrPrim}, j)) \]

Figure 9.8: Given query result table \( t \) and query-variables \( \text{start} \), \( \text{end} \) and \( \text{thrPrim} \), the above equations define the conditions under which the graphs can be produced. The meaning of \( \pi \) and \( \rho \) is defined in Section 6.5.2.

to associate the bar to and hence the colour (or shade) of the bar, and the fourth column defines the y-axis position (each different value of \( \text{thrPrim} \) is assigned one position on the y-axis). The fifth column is used for identification purposes when the user clicks on the graph.

**Example 63** The following result table:

<table>
<thead>
<tr>
<th>( a_1.\text{start} )</th>
<th>( a_1.\text{end} )</th>
<th>( a_1.\text{Selector} )</th>
<th>( t_1.\text{thrPrim} )</th>
<th>( a_1.\text{actPrim} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>X</td>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Y</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>X</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>X</td>
<td>b</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>Y</td>
<td>b</td>
<td>4</td>
</tr>
</tbody>
</table>

*is post-processed to generate the graph (b) in Figure 9.7.*

**Displaying the graphs**

The \( \mathcal{T} \mathcal{F} \) rules select possible query-variable combinations, but do not necessarily select correct combinations. For example, the second and fourth combinations in Example 60 use the attributes \( \text{AcStart} \) and \( \text{AcEnd} \) as the \( \text{end} \) and \( \text{start} \) variables, respectively. Even if \( \mathcal{T} \mathcal{F} \) generates valid query-variable combinations, this gives no guarantee that a graph can be produced from the result of a query. For example, it is not possible to view a per-thread graph with \( \text{Processor} \) as the threading component because a per-thread graph relies on the activity blocks not overlapping and a processor may host more than one activity at a time.

Figure 9.8 defines the cases that must be satisfied for a query result to be displayed as a graph. It is not possible to encode this restriction in the query
as it requires analysis of adjacent rows which is not possible in SQL³. So, before
displaying a graph, the conditions defined in Figure 9.8 are checked. If these are
not met then a simple error message is displayed.

In the specialized tool a maximum of six different states, and therefore six
different types of bars on a graph, are possible. This is not true for a general
data model where, in the new activity graph, there could be thousands of different
states. In common with the Runciman and Wakeling's original heap profiling
graphs [90] the generic tool shows the activities that account for the largest
amount of computation⁴ individually, and groups the remaining activities as one
block.

Keys and colours

As the set of states produced by the graphs is not fixed, we cannot statically
allocate the colours (or grey-scales) with an appropriate colour as in the special-
ized tool (running: green, runnable: amber, and so on). Instead the colours are
allocated dynamically. However, the user is able to choose the colours that they
wish to represent the different activities. These persist across view and database
changes. Not only does it make it easier to compare different graphs produced at
different view levels, but also removes the chance of odd graphical features, such
as running threads represented by the colour red.

Interactive profiling

Interactive profiling is achieved in the same way as the specialized tool. The user
clicks a bar within the graph and the value of the query-variable representing
actPrim is feed into the SQL editor.

Locking: is it harmful?

As with the specialized tool, the generic tool allows the user to filter what is
displayed in a graph by modifying the template query. To prevent the user from
writing queries that are not in the required format to generate the graphs, some
fields are locked (can not be modified). Figure 9.9 shows the query box for creating
a thread activity graph with a number of locked fields.

Although this ensures that the format of the result is not corrupted this may
stop the user from generating some valid graphs. The need to introduce the thread
activity graph as well as the state activity graph exemplifies this problem. They

³SQL queries work like a map across rows with no interaction between them. This high-
lights one of the weaknesses of SQL not containing basic language features such as iteration or
recursion.
⁴Or whatever the activities represent.
both generate the same form of graph but require different queries. A possible way of getting both safety and freedom, is to analyse the queries to see if they meet certain criteria required to generate correct results. An example criterion would be that an activity graph query must select exactly three columns (or four if a count is used). The generic tool only supports the three graphs and the fixed locking. However, an interesting extension would be to introduce more general criteria for graphical queries.

### 9.3 Application: parallel profiling

This section shows how, after making the tool generic, it can still be applied to the GranSim parallel data model in Chapter 8. It starts with a description of what is required to use the generic tool on an example compiled by GranSim, breaking this down into the three stages identified in the previous section. This is followed by a comparison of both the features and performance of the generic and specialized parallel tools.

#### 9.3.1 Stage 1: Data modelling

For the purposes of data modelling we use the GranSim chain of data models (see Section 8.2.1). Its associating refinement combinators are submitted to the view compiler. In addition to the views and table definitions, the integrated view compiler produces a minimal DTD for the parallel data model. The full DTD is given in Appendix A.3.1.
9.3.2 Stage 2: Building the databases

Modifying the minimal DTD

Inside LieOn the type of LieOnId is changed to string. This is because the identity key is derived by concatenating the SparkId and the start and end times to create a unique key. Although this results in a number, it may be larger than can be stored in any of the numeric types in the database.

<element key="LieOnId" type="string"/>

The ParSite attribute is also changed to type string to reflect that the par-site is a source code reference.

<element key="ParSite" type="string"/>

The XML files fed to the generic tool are not designed for use by other special-purpose tools so no extra fields or records are added to the DTD.

Converting GranSim log-files to XML

GranSim uses its own log-file format. Rather than modifying GranSim to produce different output we convert GranSim log-files to XML format. Example output of the log-file convertor is given in Figure 9.10. The reason for the unjustified output is because the standard pretty printing library used by HaXml has heap problems and so simple printing routines are used instead (see Section 9.5.1).

9.3.3 Stage 3: Using the browser

Applying the generic browser to the GranSim data model results in a browser with a similar look and functionality to the specialized parallel browser, as shown
Figure 9.11: The generic browser displaying the Thread Activity graph for the sumEuler example from Chapter 4, with the processor as the threading component.

in Figure 9.11. The main item missing from the generic browser is the more detailed labelling of the axis. In the specialized browser the x-axis is labelled No. of clock cycles. Although the axis still represents time, the unit of time differs between different profiling tools. On the other hand, making the browser more generic has the side-effect of introducing new features not available in the specialized tool that are useful in the parallel setting\(^5\):

- By clicking on the box in the top left-hand corner a view selecting window such as the one in Figure 9.12 appears. This allows the user to select their required data model from the chain, and view the different entities, relationships, and attributes required to construct queries.

\(^5\)These features could of course be implemented in the specialized parallel tool.
• The generic tool provides the new Thread Activity graph. In Figure 9.11 this is used to display a breakdown of number of running threads on each processor across the program’s evaluation. This is in contrast to the state activity graph where it is only possible to view the number of running threads on a given processor.

• The user can select the threading component. In Example 60 we saw that for the most refined GranSim model there is an extra threading component Processor (using relationship hosts), as well as the threading component Thread (using relationship has). This is used in Figure 9.11 to view the computation in a processor oriented way. This may help the user to improve the way that work is distributed among processors.

• The bars on the graphs, representing the different states, are sorted (largest bars at top, smallest at the bottom), whereas in the specialized tool the order is fixed. Also the smaller bars are grouped together under Other. This can make comparing graphs between different programs runs more difficult because when the proportions of each of the states change the order of the bars may also change.

9.3.4 Experimental results

We now ask whether performance of the generic applied to parallel profiling is any different from the specialized tool. Section 8.2.6 compared the length and speed of queries when using a database which integrates the output generated by
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.41</td>
<td>9.05</td>
<td>50.40</td>
<td>4.02</td>
<td>5.20</td>
<td>27.04</td>
</tr>
<tr>
<td>2</td>
<td>0.67</td>
<td>1.09</td>
<td>1.80</td>
<td>1.66</td>
<td>1.22</td>
<td>1.74</td>
</tr>
<tr>
<td>3</td>
<td>1.60</td>
<td>2.61</td>
<td>5.05</td>
<td>1.30</td>
<td>2.09</td>
<td>3.18</td>
</tr>
<tr>
<td>5</td>
<td>6.56</td>
<td>6.56</td>
<td></td>
<td>4.74</td>
<td></td>
<td>4.74</td>
</tr>
<tr>
<td>Total</td>
<td>9.67</td>
<td>19.31</td>
<td>83.26</td>
<td>6.98</td>
<td>13.23</td>
<td>49.68</td>
</tr>
</tbody>
</table>

Table 9.1: The time (in seconds) taken to build databases using: the specialized tool; specialized tool integrating views; and the generic tool. The blank entries indicate that the task is not required.

the view compiler. As the final format and contents of a database are the same when using the generic tool, this section does not include tests relating to the length and speed of queries. Instead it summarises experiments which compare the database build times and the time taken to display the graphs. All tests were performed using the same examples, methodology and architecture as used for the tests in Section 8.2.6.

Table 9.1 shows the time taken to build the different databases for the two examples. The stages correspond to the following activities:

1. Converting the GranSim log-file to XML format.
2. Build tables from input file.
3. Submitting table definitions to database.
4. Submitting tables to database.
5. Submitting view definitions.

The table shows that the generic tool takes over seven times longer to build the database than the specialized tool, for both examples. The majority of this time is taken converting the GranSim log-file to XML form and building the database from the XML file. If GranSim is modified to generate an XML log-file, this would relieve the need for log-file conversion, which takes about one quarter of the time taken to build the databases. However, the major bottleneck is the time spent building the database tables from the XML file. Section 9.5.1 discusses the reasons for this and possible alternatives.
Table 9.2: The time (in seconds) to display the per-thread and activity graphs in the specialized tool and generic tool.

<table>
<thead>
<tr>
<th></th>
<th>BarnesHut</th>
<th></th>
<th>SumEuler</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Specialized</td>
<td>Generic</td>
<td>Specialized</td>
<td>Generic</td>
</tr>
<tr>
<td>Per-Thread</td>
<td>22.2</td>
<td>27.9</td>
<td>11.7</td>
<td>14.1</td>
</tr>
<tr>
<td>Activity</td>
<td>8.1</td>
<td>6.9</td>
<td>5.6</td>
<td>4.9</td>
</tr>
<tr>
<td>Total</td>
<td>30.4</td>
<td>38.8</td>
<td>17.3</td>
<td>19.0</td>
</tr>
</tbody>
</table>

Table 9.2 compares the time taken to display the two graphs, and shows the minimal difference between times for the generic and specialized tools. The per-thread graphs take longer to produce because the activity graph is optimized in C (see Section 9.5.2).

### 9.4 Application: heap profiling

The previous section applied the generic tool to a data model which has already been used in the specialized tool, giving empirical proof that when the generic tool is applied to parallel profiling it retains all of the available features. This section provides evidence that the tool is indeed generic by applying it to a form of heap profiling.

In the same way that we restricted our application of the generic tool to parallelism by applying it to the ghc profiler GranSim, this section applies the generic tool to heap profiling by using the heap profiler from nhc98 as its basis.

#### 9.4.1 Stage 1: Data modelling

For the purpose of data-modelling we use the heap profiling by producer chain in Section 8.3. The integrated view compiler generates the minimal DTD and view definitions, in Appendix B.3.1 and Appendix B.2, for the heap profiling combinator expression (see Section 8.3.1).

#### 9.4.2 Stage 2: Building the databases

**Modifying the minimal DTD**

The only change made to the minimal DTD is the setting of the type of source-code reference to string:
The final DTD used is given in Appendix B.3.2.

Converting a heap profile log-file to XML

We use a conversion tool, hp2xml, to translate the log-files generated by the heap profiler into XML format. Sample output of hp2xml is given in Figure 9.13.

9.4.3 Stage 3: Using the browser

The look-and-feel of the generic browser when applied to heap profiling is the same as when it is applied to parallel profiling, as is shown in Figure 9.14. There is only one possible set of query variables: the threading entity is Component, with relationship produces, and activity entity Heap. If it was possible to produce a single database which also contained heap constructor and retainer information (as used suggesting in Figure 8.4) then further combinations of query variables would be available.

To display the activity graphs the query template needs to be modified because each record in the Heap table records a population count. Without making the changes, each record would only be counted once. Figure 9.4 shows the query-template with an optional count query variable. When generating graphs from queries that include a count variable, the count variable is used to specify how high each block should be.

The state activity graph shows the heap usage across the computation divided, into heap use, drag, lag and void (in the concrete model). The thread activity graph shows the amount of heap used by different program components across the computation. It is not normally possible to produce per-thread graph heap profiling information because, unlike in parallel profiling, there is no concept
Figure 9.14: The generic browser displaying heap waste by producer nhc98 compiling a quicksort module. The key at the top of the window is too compact to be able to read the entries accurately, so the user is able to bring up a legend with the full details.
of threads of activity. For each program component at any time during the computation there could be heap cells which are in use, dragging, lagging and void.

We now consider some of the features provided by the generic tool when applied to heap profiling, that are not provided by the use of the standard heap profiler and accompanying tools. Although most of the extra features are the same as those gained by using the generic tool for parallel profiling, some features are unique to heap profiling:

- The query interface can extract information not found in the graphs. For example, it is possible to find out the number of times that the heap cells produced by a program component are used.

- It is possible to view the heap usage as only waste and use. For example, Figure 9.14 shows the amount of heap waste when nhc98 compiles a quicksort module.

- Both the state activity and thread activity graphs can be generated from only one program run. In nhc98 heap profiling tools it is necessary to run the program twice to extract this information.

However, the most useful features are those shared with parallel profiling: query interface allowing efficient extraction of information; queries able to filter information displayed in graphs; dynamic graphs.

### 9.5 Implementation

This section describes the implementation of the parts of the generic profiler that differ from the specialized tool, the implementation of which is described in Section 4.4.

#### 9.5.1 The Builder

The aim of the Builder is to construct the database tables. It does this by first reading the (possibly) modified DTD and selecting only the definitions required. The DTD file is then used to read the XML file which constructs the database. A description of the implementation of each of these phases follows.

**Parsing and reading the DTD file**

Aiming to implement as much in Haskell as possible, Wallace and Runciman’s XML combinator library HaXml [110] is used to extract the information from the DTD. A brief explanation of how the table definitions are extracted follows.
The table names are found by searching for the `elementType` tags within the `s:schema` tags, and selecting the `id` attribute.

```haskell
(showattr "id") 'o' (deep $ tag "elementType")
```

The fields for each table are extracted. A mapping function with threaded state builds up a lookup table to handle references.

```haskell
mapST (getFields dtfile) [] (getTableNames dtfile)
```

The fields are extracted from both the `element` tags and the `foreignKey` tags.

```haskell
getFields' = cat [getElementParts, getFKeyParts]
```

The key and type fields are extracted for each `element`.

```haskell
getElementParts = map (getKeyType o showAttrs) o
  ((deep $ tag "element") 'o' findTable)

getKeyType as = (getAttrV $ lookup "key" as,
  getAttrV $ lookup "type" as)

findTable = (attrval ("id",makeAttrV tn)) 'o'
  (deep $ tag "elementType")
```

The range, key and label fields are extracted for each `foreignKey`. Where a field has a label this is used, otherwise the key is used as the field name. The type of the field derived by looking up the reference in the state.

```haskell
getFKeyParts = map (getKeyType' o showAttrs) o
  ((deep $ tag "foreignKey") 'o' findTable)

getKeyType' as
  (getKey label key,lookupType range key)
  where
  (range,key,label) = (getAttrV $ lookup "range" as,
    getAttrV $ lookup "key" as,
    lookup "label" as)
```

### Filtering the DTD

Filtering of the DTD is required because the implementor is allowed to add any extra records of fields that they require. The Builder uses the information obtained directly from the view compiler to determine should be kept. In fact, the only reason that the DTD needs to be read is to pick up the changes that the implementor makes to the types of the fields. If these changes were not allowed then the Builder could obtain the entire format of the tables directly from the view compiler.

---

6The references are enclosed within quotes, and are not XML references, so it is not possible to use the combinator library’s handling of XML references.
### Parsing and reading the XML file

After retrieving the format of the log-files, the next step is to extract the tables from the XML files. Again, the HaXml library is used. A natural way of reading the XML file given the table's format, is to map a table retrieving function (getTableData) across the list of table definitions:

```haskell
getTables = map (getTableData xmlFile) (getTableDefsFromDtd dtdFile)
```

The problem with this is that during the first application of `getTableData` a data structure representing the XML file stays in the heap. None of it can be garbage collected as it is required for the further applications of `getTableData`. The XML files can be very large and so this scheme would be expensive in memory. One way to get round this is to force the IO monad to re-read the XML file each time. However, this is expensive in processing time.

To read the XML file in an efficient way there should be only one pass through the XML file and little heap drag, so that only a small proportion of the file is in memory at one time. It is not possible to get this effect by using only the HaXml combinators, as a combinator expression corresponds to a single query and we require multiple queries (one for each table). However, we can take advantage of what we know about the structure of the XML file to achieve our goal:

- The implementor may modify the DTD so that the log-files tags may be anywhere within the XML file, so we first need to search for the log-file.

- Each of the top-level elements within the log-file is a table row entry, with the tag name referring to the table name, and its children, the field entries.

The `makeTables` function in Figure 9.15 uses information to extract the tables data. First it searches for the log-file\(^7\). Then the different fields are selected for each of the child elements that correspond to the required tables. The result is a list of pairs containing the table name and a list of field values representing a row.

### Changes made to HaXml

Changes needed to be made to standard version of HaXml [110] so that it could handle large XML files:

- HaXml normally uses Peyton Jones's [45] modified version of Hughes's pretty-printing combinator library [44]. This needed to be replaced as a heap-profiling analysis showed that it caused major space leaks. Instead simple printing routines that generate unjustified output are used.

\(^7\)In practice, if the log-file is not the top-level definition, then the program is likely to run out of heap space.
• The type of Parser is changed to remove all the symbol table information. Without this change, the entire file is kept in memory in order to check the XML references. As we do not use XML references it is safe to remove this table. An alternative way to solve the problem, where references are required, is to add strictness information where necessary.

• The document parser is changed so that it does not check whether the top-level tags match. Again, without this modification the XML file is retained in memory until the closing top-level tag, at the end of the file, is checked.

• The sanityCheck function is updated so that it does not check for data beyond the parsed document.

Bottlenecks
In Section 9.3.4 we saw that time taken to build the database tables is the the major bottleneck for the generic tool. Although this might be expected, it is more surprising that it takes about five times longer to complete this stage than when the specialized tool is used on exactly the same data (in a different format).

What is the cause of this reduction in speed? Most of the time is spent extracting the data from the XML file. The main reason why this takes longer than with the original log-files is because the XML files are longer and so require more time to read and parse. Both the GranSim and nhc98 heap-profiling log-files are at least four times longer when converted into XML format. This gain is because each data item has an XML start and end tag. Indeed, the log-file portion in Figure 9.10 illustrates the proportion of the file used by the tags. A
Table 9.3: The time taken to display the graphs for the sumEuler example, for the specialized parallel tool incorporating view information, and for the unoptimized generic tool applied to the GranSim model.

```
<table>
<thead>
<tr>
<th>Version</th>
<th>Per-Thread</th>
<th>State Activity</th>
<th>Thread Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specialized</td>
<td>2.4</td>
<td>1.9</td>
<td>2</td>
</tr>
<tr>
<td>Generic</td>
<td>2.5</td>
<td>2</td>
<td>2.1</td>
</tr>
</tbody>
</table>
```

binary representation of XML such as that suggested by Jarvis et al. [48] for XML-Data, should reduce the size of the files and hence parsing time.

### 9.5.2 The front-end

The front-end requires modest modifications to handle more generic data. Changes are made to generalize the datatypes and to add the features explained in Section 9.2.4. Table 9.3 shows how the new modifications to the tool have little effect on the speed at which the parallel graphs are displayed. However, the time taken to display the heap-profiling graphs is unacceptable. Lots of examples cause the GUI, to run out of heap when attempting to display a graph, and even small examples can take a several minutes to display.

Table 9.4 gives a breakdown of the time taken on each task that is required to generate the activity graphs for the spec example. The majority of the time is spent in the GUI reading the result, processing it and displaying it. More surprisingly, over one third of the time is spent sending the result to the GUI. The query evaluator returns its result in a C data structure. This is then converted into Haskell data using GreenCard marshalling functions, and then written to a socket descriptor. To send data down the socket stream the Haskell data is automatically converted back into C data. The penalty for converting the data into Haskell is high. This highlights a known speed deficiency with nhc98 compiled code, where reading and writing files is considerably slower than other Haskell implementations.

```
main = readFile in >>= writeFile out
```

\*\*The following example that reads the file in, and writes the file out takes about 25 seconds when compiled with nhc98, and a maximum of four seconds in hbc, ghc, and hugs, where the in file is 700KBs: \*\*
Table 9.4: A breakdown of the times taken to display the standard graphs for the spc example.

<table>
<thead>
<tr>
<th>Task</th>
<th>State Activity</th>
<th>Thread Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database to process query</td>
<td>&lt;1 (0%)</td>
<td>2 (5%)</td>
</tr>
<tr>
<td>Query evaluator to send result to GUI</td>
<td>2 (40%)</td>
<td>16 (40%)</td>
</tr>
<tr>
<td>GUI to display graph</td>
<td>3 (60%)</td>
<td>22 (55%)</td>
</tr>
<tr>
<td>Total time</td>
<td>5</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 9.5: A breakdown of the times taken to display the standard graphs for the spc example, in the final version of the tool, where the processing of the query results into graph descriptions is done in C.

<table>
<thead>
<tr>
<th>Task</th>
<th>State Activity</th>
<th>Thread Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database to process query</td>
<td>&lt;1s (0%)</td>
<td>2s (67%)</td>
</tr>
<tr>
<td>Query evaluator to create graph description</td>
<td>1s (100%)</td>
<td>1s (33%)</td>
</tr>
<tr>
<td>Query evaluator to send result to GUI</td>
<td>&lt;1s (0%)</td>
<td>&lt;1s (0%)</td>
</tr>
<tr>
<td>GUI to display graph</td>
<td>&lt;1s (0%)</td>
<td>&lt;1s (0%)</td>
</tr>
<tr>
<td>Total time</td>
<td>1s</td>
<td>3s</td>
</tr>
</tbody>
</table>

the resultant C data structure. Another GreenCard function uses this pointer to send data down the socket stream. The time taken to send the result to the GUI using this method is negligible, taking less than 0.1 second when producing the same graphs as those produced in Table 9.4.

The other bottleneck identified is where the data is read, processed, and displayed in the GUI. Of these three activities the most consuming of time and space is the processing of the query result to create a description of the graph. To speed this up, the data is processed in C before being sent down the socket to the GUI to create the description. The processing is done before being sent to the GUI since there is no nice way of calling C functions when using the Fudget compiler, hbc.

A side effect of using C to implement the graph description processing is that it is now possible to sort the bars into size order, and group the smaller bars under the single label of Other. This was not possible in the original Haskell version because of the excessive heap and time required.

Table 9.5 gives a breakdown of the times taken on each task when creating the two activity graphs for the spc example, with these changes made. As the table shows, making these changes drastically speeds up the displaying of the graphs.
9.5.3 Implementation experience

One motivation of this thesis is to demonstrate the use of Haskell as a language for writing profiling tools. We now summarise two problems experienced in implementing the majority of the tools in this chapter in Haskell.

No problems were experienced when working with small data sets, but whenever large data sets were used, heap problems were encountered. One major problem was that most of the heap problems were in third party libraries. All such libraries used had heap usage problems of some kind. Heap profiling helped to make some improvements but forced a lot of time to be spent trying to understand the internal workings of the libraries. This is partly because the libraries are the result of research projects and are not of production standard. However, the ease at which a small heap problem in a self-contained library can be exaggerated to consume all resources counts against Haskell.

Another problem experienced using Haskell is the difficulty encountered in storing large data structures in memory with current Haskell implementations. The entire result of some queries could not fit into the heap even when allocated 100MBs worth. Generally this was not a problem because the individual data items were only used once and only for a short amount of time, so garbage collection ensured ample free heap space. However, the graph producing code required most of the table to be in memory at once in order to process the data items. Our solution was to write some parts of the processing in C. Another possible solution that may help is to use binary heap compression [109].

Despite these problems the advantages that Haskell offers are significant. The language was particularly useful for rapid-prototyping and testing the view compilation rules. In fact the Haskell code in the view compiler looks very similar to the V compilation rules.

HaXml

The overall experience of using the HaXml combinator library [110] was positive. Parsing, printing and filtering XML is simple. However, the library has primarily been designed to allow data extraction and manipulation of an XML file, with the intent to produce a new XML or HTML file. The database builder requires the intermediate Haskell data structure representing the extracted data, just before pretty printing. Although it is easy to get to these intermediate data structures, extracting data from them is a contorted process.
9.6 Related work

Much of the related work for the generic tool is similar to that of the specialized tool summarised in Section 4.6. As a result this section only compares the features that differ between the two tools.

There has been very little generic profiling in the functional programming community. Previously, a new form of profiling often meant writing new tools to visualize the data.

Halstead’s Vista [29] system, that aims to provide a generic browser for analysing performance related data, inspired some of the features provided by the generic toolkit described in this chapter. In particular, it uses a standard log-file format and allows multiple threading fields so that the computation can be viewed in different ways. Halstead [30] shows how a parallel computation can be viewed in a thread or processor oriented way, in same way as in Section 9.3.

Commercial database products, such as Microsoft™ Access, provide tools for visualizing query results using general data visualization techniques. Although database views are supported by most commercial databases, the user is often expected to define the views themselves from the concrete data model. The onus is on the user to determine whether the views they create have a theoretical justification.

The novel feature of the generic tool is its ability to present multiple abstraction levels to the user. Although different abstractions are provided by the different graphs that are generated by other profiling browsers, they generally offer a similar level of abstraction. The generic tool allows the implementor to create an abstraction level for every refinement made to create the final data model. So the user can view the computation using any level of complexity, from the most abstraction model to the concrete details of the computation. This feature is not present in previous profiling tools.

9.7 Summary

In this chapter we have looked various extensions and adaptations to make a generic profiling tool. We summarize what is required by both the implementor and the user to utilize the whole tool. First the implementor must design a chain of one or more E-R schemas to describe the underlying data model of a profiling tool. This is expressed using the refinement combinators, which are compiled using the view compiler. This generates a minimal DTD, which the implementor modifies to change types and introduce new definitions, to create the final DTD. The implementor then either adapts their profiling tool to generate XML files that conform to this DTD, or produces a log-file convertor.
The user then runs the implementor’s profiling tool (and perhaps a log-file converter) to create an XML file which describes the computation. The user submits the XML file with the final DTD (provided by the implementor) to the database Builder. They are then able to interact with the front-end to:

- submit queries;
- view static, and dynamic graphs;
- and view the data using different levels of data model complexity;

to help them understand their program.

This chapter has shown that the techniques developed in Chapter 4 and Part II are not restricted to parallel profiling. However, we have only applied them to one other form of profiling. One area of further work would be to test the tool on other types of profiling. The least generic part of the application is the graphical visualization system. We have have described initial research into generating graphical views but only a limited set of queries can be displayed graphically. Another area for further work would be to integrate standard visualization techniques into the system.
Chapter 10

Conclusions

In Chapter 1 we noted that, although profiling has become an important part of writing functional programs, little explicit data modelling has taken place in the construction of tools. Making the data models explicit is useful for both the implementor and the programmer: implementors can identify inconsistencies between the implicit data model used by their tools, the data recorded in the log-files, and the information displayed by visualization tools; it is clear to programmers exactly what data model is used. This lead to the overall aim of the thesis:

- to make functional languages more usable by improving the quality of functional language profiling using data modelling techniques.

In this chapter we assess the extent to which this aim is met.

Section 10.1 summarises the core of the thesis: data model refinement, and Section 10.2 summarises the effect of applying data modelling techniques to profiling. Section 10.3 assesses the strengths and weaknesses of the work, which leads to the suggestions for further work in Section 10.4.

10.1 Data model refinement

There is only a small amount of literature on refinement in E-R modelling, despite it being the de facto standard for designing databases. Most of this work is concerned with syntax for describing refinement for documentation purposes only. The remaining literature that does consider the semantics of data model refinement, does so to define correct refinements. There is no previous literature showing how to extract data associated with an abstract data model from a database based on a refined data model.

In Part II we presented a method for realizing refinement in E-R modelling. Chapter 5 set up the framework for describing refinements in a non-diagrammatic
way, using a set of low-level primitives. The meanings of the primitives are formally defined, unlike most primitive libraries found in the literature where the meaning is often ambiguous.

Chapter 6 presents compilation rules for constructing a database capable of handling queries over both an abstract data model, and a refined data model. The queries over the abstract data model are handled by database views. Informal proofs show that generated views act as valid retrieve functions, and consequently that the primitives are sound.

In Chapter 7 we described a view compiler, based on a combinator library, that extends the work described in Chapter 6 to handle a chain of refined data models. A data model chain is defined as a high-level combinator expression, that when evaluated generates definitions to allow a user to submit queries over any of the data models.

10.2 Profiling and data modelling

The work described in this thesis was originally inspired by experiences gained using two different parallel profiling tools. Their behaviour is typical of most profiling tools: they gather information during a program run, and record this in a log-file for later conversion into graphs. The log-files are generally too long for direct use by the programmer, and the graphs often display too little information for them to understand their program. Even so, for large computations the graphs are too crowded. If the information needed to confirm some hypothesis is not displayed in the graphs the programmer may have to resort to analysis of the log-file.

To improve on this situation we built an interactive tool that allows efficient extraction of data by using a query interface and dynamic graphs. In Chapter 3 we described the data modelling process that formed an important part of the tool’s design. We showed that explicit data modelling allowed us to:

- compare different parallel profiling tools by analysing the class of queries answerable using their data model;
- evaluate the profiling tools by comparing the data models describing the log-files, the graphs, and what we expect;
- design the format of a database allowing extraction of the full information stored in the log-files.

Chapter 4 outlines the features of the tool and shows how data can be extracted efficiently using queries.
Data model refinement

In Chapter 8 we applied the view compiler, described in Part II, to the parallel data model chain of Chapter 3. We showed how to integrate the resulting views into the interactive tool, so that queries can be based on any data model in the chain. Experiments showed that queries over the more abstract data models were both more concise and evaluated faster than queries over the concrete model. Yet, the concrete model was still required for some queries.

We also showed that the techniques described in this thesis are not restricted to parallel profiling. In Chapter 8 we applied the view compiler to a form of heap profiling, whilst in Chapter 9 we presented initial research, describing a generic browser tool that can handle any type of profiling data. The browser was applied to both parallel and heap profiling.

10.3 Criticism

Taking a step back, we now ask have we improved the quality of profiling using data modelling techniques? Let us start by looking at the evidence that the goal has been achieved:

- Chapter 3 showed how explicit data modelling can help identify inconsistencies in profiling tools. Implementors can use this information to improve their tools.

- The interactive tool provides a query interface that allows much more efficient extraction of data than using current methods, as discussed in Section 4.5.

- This thesis presents theoretically sound methods that provide the facility where a user can view and query profile data using different levels of abstraction. This in contrast with other profiling tools that give a fixed view of abstraction.

We now list the thesis’s potential areas for improvement, dividing it into improvements associated with the thesis goal, and others that apply to data model refinement in general. We start with those directly related to the thesis goal:

- The view compiler can only handle a chain of refinements. This forces the implementor to select one path through the possible refinement lattice.

- The data models used for the heap and parallel profilers are based on already existing profiling tools. We have not considered the effect of defining a data model chain before building a profiling tool around it. We might expect the data models to include more relationships, but at what cost?
• Although this thesis has discussed several examples, a larger study is required to make firm conclusions about the usefulness of both the generic profiling tool and the refinement process.

• The generic tool is only able to display graphs for a limited set of queries.

• The view compiler does not handle pre-conditions on the use of the $\mathcal{V}$ rules. Also, we do not show how to propagate pre-conditions when refinements chained together.

The following areas for improvement do not effect the thesis goal, but do concern general data model design:

• The view compilation rules require a non-standard format to store relationships. This leads to inevitable redundancy which is a considered bad in database design as it leads to larger, less efficient databases.

• Restrictions are made on the type of E-R models that are allowed. In particular, the primary key can only be comprised of one attribute.

• The refinement primitives are not complete enough to handle all non-binary relationships.

• The refinement framework does not handle attribute refinement found in the literature [26, 15].

We consider how to address these improvement areas in the next section.

### 10.4 Future Work

Motivated by the areas for improvement outlined in the previous section, the following directions for future work are identified:

• Extend the view compiler to handle a lattice of data models.

• Develop profiling tools *after* designing a data model chain. This may result in more consistent and richer data models allowing a greater range of queries.

• Examine the use of the generic profiling system on a wider range of programs and pre-existing forms of profiling.

• Integrate standard data visualization techniques into the generic profiler.

• Extend the view compiler so that *threading components* are recorded in the data modelling stage.
• Implement the pre-conditions on the use of primitives and the view compilation rules within the view compiler. This partially requires determining how pre-conditions are inherited in multiple refinements.

• Formally prove the cases where the primitives are sound.

• Develop view compilation rules that use the standard methods for converting an E-R schema into database tables. This will require extending the primitives and schema representation to include the optionality of and degree of relationships.

• Extend the view compilation rules to handle n-ary relationships.

• Extend the framework to handle attribute refinement.

10.5 Concluding Thought

Previously, profiling tools were developed with little consideration to data modelling. Although a lot of data was recorded by the profiling tools it was often not possible to access this in an efficient way. This thesis has shown how data modelling techniques can be successfully used to improve profiling tools for functional languages and has exemplified the need for proper data modelling when designing these tools. We have shown how a query based interface can be integrated into a graphical environment to enable efficient access to information previously hidden from the user. We have also shown how to present multiple abstraction levels to the user alleviating the common problem of developing one profiling tool that is suitable both for extraction of low and high level information about a computation. However, the full potential of the ideas presented in this thesis will only be realized when profiling tools are developed with data modelling in mind from their conception.
Appendix A

Parallel profiling example data

This appendix shows the various outputs generated by the view compiler for the parallel data model chain described by the combinator expression Figure 8.3.

Appendix A.1 shows the primitive encoding of the parallel data model chain defined in Figures 8.1 and 8.2. Appendix A.2 shows the view definitions generated by the view compiler for the combinator expression in Figure 8.3. Appendix A.3 shows the minimal DTD generated by the view compiler for the parallel data model chain, and the final DTD as used in Section 9.3.

A.1 Data model chain: primitive encoding

[[EGeneration "Thread" "ThreadId",
AGeneration "Thread" Nothing ["ParSite","ThStart","ThEnd"],
RGeneration "sparks" [ArcLabel ("Thread", Just "creator"),
ArcLabel ("Thread", Just "created")]]
AGeneration "Thread" (Just "ActivityNo") [],
E Decomposition "Thread" "has"
  [EAttribute ("Thread", "ThreadId", ["ParSite","ThStart","ThEnd"]),
EAttribute ("Activity", "ActivityNo", [])
[(ArcLabel ("sparks", Just "created"), "Thread")],
AGeneration "Activity" Nothing ["AcStart","AcEnd"]
],[HDecomposition "Activity" "Run" [ArcLabel ("sparks", Nothing)],
HDecomposition "Activity" "Block" []]
,[HEGeneration "Activity" "Queue"
],[HEGeneration "Activity" "Migrate", HEGeneration "Activity" "Fetch"]
,[EGeneration "Processor" "ProcessorNo",
RGeneration "hosts" [ArcLabel ("Processor", Nothing),
ArcLabel ("Activity", Nothing)],
RGeneration "migrateTo" [ArcLabel ("Processor", Nothing),
ArcLabel ("Migrate", Nothing)]]
R Generation "fetchFrom" [ArcLabel ("Processor", Nothing),
   ArcLabel ("Fetch", Nothing)]

, [A Generation "Thread" Nothing ["SparkId"],
   E Decomposition "Thread" "becomes"
   [E PAtribute ("Thread", "ThreadId", ["ThStart", "ThEnd"]),
    E PAtribute ("Spark", "SparkId", ["ParSite"])]
   [(ArcLabel ("sparks", Nothing), "Spark")],
   A Generation "Spark" Nothing ["CreatedAt"],
   R Generation "liesOn" [ArcLabel ("Spark", Nothing),
   ArcLabel ("Processor", Nothing)]
   , [A Generation "liesOn" Nothing ["LieOnId", "LieFrom", "LieTo"],
   R Decomposition "liesOn" "LieOn" ["LieOnS", "LieOnP"]]

A.2 Views

%level 1; %view 0

create view sparks1_0 as
   select distinct  t1.ThreadId as creator, t2.ThreadId as created
   from sparks1_1, Thread1_1 t1, Thread1_1 t2
   where sparks1_1.creator = t1.ActivityNo
   and sparks1_1.created = t2.ActivityNo

%view 1

create view Thread1_0 as
   select distinct  Thread1_1.ThreadId, Thread1_1.ParSite,
   Thread1_1.ThStart, Thread1_1.ThEnd
   from Thread1_1

%view 2

create view sparks1_1 as
   select distinct  sparks6_1.ActivityNo as creator, t2.ActivityNo as created
   from sparks6_1, has8_0, Thread6_0
   where t2.ThreadId = s2.ThreadId
   and sparks6_1.ThreadId = s2.ThreadId

%view 3

create view Thread1_1 as
   select distinct  Thread6_0.ThreadId, Thread6_0.ParSite,
   Thread6_0.ThStart, Thread6_0.ThEnd, Activity1_2.ActivityNo
   from has8_0, Thread6_0, Activity1_2
   where Thread6_0.ThreadId = has8_0.ThreadId
   and Activity1_2.ActivityNo = has8_0.ActivityNo

%view 4
create view Activity\textsubscript{1,2} as
    select distinct Activity\textsubscript{2,0}.ActivityNo from Activity\textsubscript{2,0}
\%level 2; \%view 0
create view Activity\textsubscript{2,0} as
    select distinct Activity\textsubscript{2,1}.ActivityNo, Activity\textsubscript{2,1}.AcStart, Activity\textsubscript{2,1}.AcEnd
    from Activity\textsubscript{2,1}
\%view 1
create view Activity\textsubscript{2,1} as
    select distinct Activity\textsubscript{3,0}.ActivityNo, Activity\textsubscript{3,0}.AcStart, Activity\textsubscript{3,0}.AcEnd
    from Activity\textsubscript{3,0}
\%level 3; \%view 0
create view Activity\textsubscript{3,0} as
    select distinct Activity\textsubscript{4,0}.* from Activity\textsubscript{4,0}
    where Activity\textsubscript{4,0}.Selector <> 'queue'
\%level 4; \%view 0
create view Activity\textsubscript{4,0} as
    select distinct Activity\textsubscript{4,1}.* from Activity\textsubscript{4,1}
    where Activity\textsubscript{4,1}.Selector <> 'migrate'
\%view 1
create view Activity\textsubscript{4,1} as
    select distinct Activity\textsubscript{8,0}.* from Activity\textsubscript{8,0}
    where Activity\textsubscript{8,0}.Selector <> 'fetch'
\%level 5
\%level 6; \%view 0
create view Thread\textsubscript{6,0} as
    select distinct Thread\textsubscript{6,1}.ThreadId, Thread\textsubscript{6,1}.ParSite, Thread\textsubscript{6,1}.ThStart, Thread\textsubscript{6,0}.ThEnd
    from Thread\textsubscript{6,1}
\%view 1
create view \textit{sparks}_{6,1} as
    select distinct \textit{sparks}_{8,0}.ActivityNo, \textit{t2}.ThreadId
    from \textit{sparks}_{8,0}, \textit{becomes}_{8,0} \textit{t2}, \textit{Spark}_{6,2} \textit{s2}
    where \textit{t2}.SparkId = \textit{s2}.SparkId
    and \textit{sparks}_{8,0}.SparkId = \textit{s2}.SparkId
\%view 2
create view Thread\textsubscript{6,1} as
    select distinct Thread\textsubscript{8,0}.ThreadId, Thread\textsubscript{8,0}.ThStart, Thread\textsubscript{8,0}.ThEnd, 
    \textit{Spark}_{6,2}.SparkId, \textit{Spark}_{6,2}.ParSite
    from \textit{becomes}_{8,0}, Thread\textsubscript{8,0}, \textit{Spark}_{6,2}
    where Thread\textsubscript{8,0}.ThreadId = \textit{becomes}_{8,0}.ThreadId
    and \textit{Spark}_{6,2}.SparkId = \textit{becomes}_{8,0}.SparkId
\%view 3
create view $Spark_{6,2}$ as
    select distinct $Spark_{8,0}.SparkId, Spark_{8,0}.ParSite$ from $Spark_{8,0}$
\%level 7; \%view 0
create view $liesOn_{7,0}$ as
    select distinct $liesOn_{7,1}.SparkId, liesOn_{7,1}.ProcessorNo$ from $liesOn_{7,1}$
\%view 1
create view $liesOn_{7,1}$ as
    select distinct $LieOnS_{8,0}.SparkId, LieOnP_{8,0}.ProcessorNo,$
    $LieOnS_{8,0}.LieOnId, LieOnS_{8,0}.LieFrom, LieS_{8,0}.LieTo$
    from $LieOnS_{8,0}, LieOnS_{8,0}, LieOnP_{8,0}$
    where $LieOnS_{8,0}.LieOnId = LieOnS_{8,0}.LieOnId$
    and $LieOnP_{8,0}.LieOnId = LieOnS_{8,0}.LieOnId$

A.3 DTDs

A.3.1 The minimal DTD

<s:schema id="991209">
    <elementType id="Activity">
        <element key="ActivityNo" type="int"/>
        <element key="AcStart" type="int"/>
        <element key="AcEnd" type="int"/>
        <element key="Selector" type="string"/>
    </elementType>
    <elementType id="Block">
        <foreignKey range="#Activity" key="#ActivityNo"/>
    </elementType>
    <elementType id="Fetch">
        <foreignKey range="#Activity" key="#ActivityNo"/>
    </elementType>
    <elementType id="LieOn">
        <element key="LieOnId" type="int"/>
        <element key="LieFrom" type="int"/>
        <element key="LieTo" type="int"/>
    </elementType>
    <elementType id="Migrate">
        <foreignKey range="#Activity" key="#ActivityNo"/>
    </elementType>
    <elementType id="Processor">
        <element key="ProcessorNo" type="int"/>
    </elementType>
</schema>
<elementType id="Queue">
  <foreignKey range="#Activity" key="#ActivityNo"/>
</elementType>

<elementType id="Run">
  <foreignKey range="#Activity" key="#ActivityNo"/>
</elementType>

<elementType id="Spark">
  <element key="SparkId" type="int"/>
  <element key="ParSite" type="int"/>
  <element key="CreatedAt" type="int"/>
</elementType>

<elementType id="Thread">
  <element key="ThreadId" type="int"/>
  <element key="ThStart" type="int"/>
  <element key="ThEnd" type="int"/>
</elementType>

<elementType id="LieOnP">
  <foreignKey range="#Processor" key="#ProcessorNo"/>
  <foreignKey range="#LieOn" key="#LieOnId"/>
</elementType>

<elementType id="LieOnS">
  <foreignKey range="#Spark" key="#SparkId"/>
  <foreignKey range="#LieOn" key="#LieOnId"/>
</elementType>

<elementType id="becomes">
  <foreignKey range="#Thread" key="#ThreadId"/>
  <foreignKey range="#Spark" key="#SparkId"/>
</elementType>

<elementType id="fetchFrom">
  <foreignKey range="#Processor" key="#ProcessorNo"/>
  <foreignKey range="#Fetch" key="#ActivityNo"/>
</elementType>

<elementType id="has">
  <foreignKey range="#Thread" key="#ThreadId"/>
  <foreignKey range="#Activity" key="#ActivityNo"/>
</elementType>

<elementType id="hosts">
  <foreignKey range="#Processor" key="#ProcessorNo"/>
  <foreignKey range="#Activity" key="#ActivityNo"/>
</elementType>

<elementType id="migrateTo">
  <foreignKey range="#Processor" key="#ProcessorNo"/>
<foreignKey range="#Migrate" key="#ActivityNo"/>
</elementType>
<elementType id="sparks">
  <foreignKey range="#Run" key="#ActivityNo"/>
  <foreignKey range="#Spark" key="#SparkId"/>
</elementType>
<elementType id="log_file">
  <group groupOrder="OR" occurs="ZEROORMORE">
    <element type="#Activity"/>
    <element type="#Block"/>
    <element type="#Fetch"/>
    <element type="#LieOn"/>
    <element type="#Migrate"/>
    <element type="#Queue"/>
    <element type="#Run"/>
    <element type="#Spark"/>
    <element type="#LieOnP"/>
    <element type="#LieOnS"/>
    <element type="#becomes"/>
    <element type="#fetchFrom"/>
    <element type="#has"/>
    <element type="#hosts"/>
    <element type="#migrateTo"/>
  </group>
</elementType>
</s: schema>

A.3.2 The final DTD

The differences from the minimal DTD are shown in bold.

<s:schema id="991209">
  <elementType id="Activity">
    <element key="ActivityNo" type="int"/>
    <element key="AcStart" type="int"/>
    <element key="AcEnd" type="int"/>
    <element key="Selector" type="string"/>
  </elementType>
  <elementType id="Block">
    <foreignKey range="#Activity" key="#ActivityNo"/>
  </elementType>
  <elementType id="Fetch">
    <foreignKey range="#Activity" key="#ActivityNo"/>
  </elementType>
  <elementType id="LieOn">
    <element key="LieOnId" type="string"/>
    <element key="LieFrom" type="int"/>
    <element key="LieTo" type="int"/>
  </elementType>
</s: schema>
<elementType id="Migrate">
  <foreignKey range="#Activity" key="#ActivityNo"/>
</elementType>

<elementType id="Processor">
  <element key="ProcessorNo" type="int"/>
</elementType>

<elementType id="Queue">
  <foreignKey range="#Activity" key="#ActivityNo"/>
</elementType>

<elementType id="Run">
  <foreignKey range="#Activity" key="#ActivityNo"/>
</elementType>

<elementType id="Spark">
  <element key="SparkId" type="int"/>
  <element key="ParSite" type="string"/>
  <element key="CreatedAt" type="int"/>
</elementType>

<elementType id="Thread">
  <element key="ThreadId" type="int"/>
  <element key="ThStart" type="int"/>
  <element key="ThEnd" type="int"/>
</elementType>

<elementType id="LieOnP">
  <foreignKey range="#Processor" key="#ProcessorNo"/>
  <foreignKey range="#LieOn" key="#LieOnId"/>
</elementType>

<elementType id="LieOnS">
  <foreignKey range="#Spark" key="#SparkId"/>
  <foreignKey range="#LieOn" key="#LieOnId"/>
</elementType>

<elementType id="becomes">
  <foreignKey range="#Thread" key="#ThreadId"/>
  <foreignKey range="#Spark" key="#SparkId"/>
</elementType>

<elementType id="fetchFrom">
  <foreignKey range="#Processor" key="#ProcessorNo"/>
  <foreignKey range="#Fetch" key="#ActivityNo"/>
</elementType>

<elementType id="has">
  <foreignKey range="#Thread" key="#ThreadId"/>
  <foreignKey range="#Activity" key="#ActivityNo"/>
</elementType>
<elementType id="hosts">
  <foreignKey range="#Processor" key="#ProcessorNo"/>
  <foreignKey range="#Activity" key="#ActivityNo"/>
</elementType>

<elementType id="migrateTo">
  <foreignKey range="#Processor" key="#ProcessorNo"/>
  <foreignKey range="#Migrate" key="#ActivityNo"/>
</elementType>

<elementType id="sparks">
  <foreignKey range="#Run" key="#ActivityNo"/>
  <foreignKey range="#Spark" key="#SparkId"/>
</elementType>

<elementType id="log_file">
  <group groupOrder="OR" occurs="ZEROORMORE">
    <element type="#Activity"/> <element type="#Block"/>
    <element type="#Fetch"/> <element type="#LieOn"/>
    <element type="#Migrate"/> <element type="#Processor"/>
    <element type="#Queue"/> <element type="#Run"/>
    <element type="#Spark"/> <element type="#Thread"/>
    <element type="#LieOnP"/> <element type="#LieOnS"/>
    <element type="#becomes"/> <element type="#fetchFrom"/>
    <element type="#has"/> <element type="#hosts"/>
    <element type="#migrateTo"/> <element type="#sparks"/>
  </group>
</elementType>
</s:schema>
Appendix B

Heap profiling example data

This appendix shows the various outputs generated by the view compiler for the heap profiling by producer data model chain described by the combinator expression Figure 8.6.

Appendix B.1 shows the primitive encoding of the parallel data model chain. Appendix B.2 shows the view definitions generated by the view compiler for the combinator expression. Appendix B.3 shows the minimal DTD generated by the view compiler and the final DTD as used in Section 9.4.

B.1 Data model chain: primitive encoding

[[EGeneration "Component" "name"]
 ,[EGeneration "Heap" "blockNo",
   AGeneration "Heap" Nothing ["startTime","endTime","population"],
   RGeneration "produces" [ArcLabel ("Component", Nothing),
     ArcLabel ("Heap", Nothing)]
 ,[HDecomposition "Heap" "Waste" []
   HDecomposition "Heap" "Use" []]
 ,[HDecomposition "Waste" "Drag" []
   HDecomposition "Waste" "Void" []]
 ,[HDecomposition "Use" "Use1" []
   HDecomposition "Use" "Lag" []
   AGeneration "Use" Nothing ["number"]]
 ]

B.2 Views

%level 1
%view 0
create view Heap1,1 as
    select distinct Heap2,0.blockNo from Heap2,0

%level 2
%view 0
create view Heap2,0 as
    select distinct Heap2,1.blockNo,Heap2,1.startTime,
                Heap2,1.endTime,Heap2,1.population
        from Heap2,1

%view 1
create view Heap2,1 as
    select distinct Heap3,0.blockNo,Heap3,0.startTime,
                Heap3,0.endTime,Heap3,0.population
        from Heap3,0

%level 3
%view 0
create view Waste3,0 as
    select distinct Waste3,1.blockNo from Waste3,1

%view 1
create view Waste3,1 as
    select distinct Waste5,0.blockNo from Waste5,0

%level 4
%view 0
create view Use4,0 as
    select distinct Use4,1.blockNo from Use4,1

%view 1
create view Use4,1 as
    select distinct Use4,2.blockNo from Use4,2

%view 2
create view Use4,2 as
    select distinct Use5,0.blockNo from Use5,0

B.3 DTDs

B.3.1 The minimal DTD

<?xml version='1.0' ?>
<s:schema id="991223">
    <elementType id="Component">
        <elementType key="name" type="int"/>
    </elementType>
<elementType id="Heap">
  <elementType id="Use">
    <elementType id="Void">
      <elementType id="Lag">
        <elementType id="Short">
          <elementType id="Long">
            <elementType id="Use">
              <elementType id="Void">
                <elementType id="Lag">
                  <elementType id="Short">
                    <elementType id="Use">
                      <elementType id="Void">
                        <elementType id="Lag">
                          <elementType id="Short">
                            <elementType id="Use">
                              <elementType id="Void">
                                <elementType id="Lag">
                                  <elementType id="Short">
                                    <elementType id="Use">
                                      <elementType id="Void">
                                        <elementType id="Lag">
                                          <elementType id="Short">
                                            <elementType id="Use">
                                              <elementType id="Void">
                                                <elementType id="Lag">
                                                  <elementType id="Short">
                                                    <elementType id="Use">
                                                      <elementType id="Void">
                                                        <elementType id="Lag">
                                                          <elementType id="Short">
                                                            <elementType id="Use">
                                                              <elementType id="Void">
                                                                <elementType id="Lag">
                                                                  <elementType id="Short">
                                                                    <elementType id="Use">
                                                                      <elementType id="Void">
                                                                        <elementType id="Lag">
                                                                          <elementType id="Short">
                                                                            <elementType id="Use">
                                                                              <elementType id="Void">
                                                                                <elementType id="Lag">
                                                                                  <elementType id="Short">
                                                                                    <elementType id="Use">
                                                                                      <elementType id="Void">
                                                                                      </elementType>
                                                                                  </elementType>
                                                                            </elementType>
                                                                        </elementType>
                                                                      </elementType>
                                                                    </elementType>
                                                                  </elementType>
                                                                </elementType>
                                                              </elementType>
                                                            </elementType>
                                                          </elementType>
                                                        </elementType>
                                                      </elementType>
                                                    </elementType>
                                                  </elementType>
                                                </elementType>
                                              </elementType>
                                            </elementType>
                                          </elementType>
                                        </elementType>
                                      </elementType>
                                    </elementType>
                                  </elementType>
                                </elementType>
                              </elementType>
                            </elementType>
                          </elementType>
                        </elementType>
                      </elementType>
                    </elementType>
                  </elementType>
                </elementType>
              </elementType>
            </elementType>
          </elementType>
        </elementType>
      </elementType>
    </elementType>
  </elementType>
</elementType>

B.3.2 The final DTD

The differences from the minimal DTD are shown in bold.
<?xml version='1.0'?>
<s:schema id="991223">
  <elementType id="Component">
    <element key="name" type="string"/>
  </elementType>

  <elementType id="Heap">
    <element key="startTime" type="int"/>
    <element key="endTime" type="int"/>
    <element key="activityNo" type="int"/>
    <element key="count" type="int"/>
    <element key="Selector" type="string"/>
  </elementType>

  <elementType id="Use">
    <foreignKey range="#Heap" key="#blockNo"/>
    <element key="number" type="int"/>
    <element key="Selector" type="string"/>
  </elementType>

  <elementType id="Waste">
    <foreignKey range="#Heap" key="#blockNo"/>
  </elementType>

  <elementType id="Drag">
    <foreignKey range="#Waste" key="#blockNo"/>
  </elementType>

  <elementType id="Void">
    <foreignKey range="#Waste" key="#blockNo"/>
  </elementType>

  <elementType id="Lag">
    <foreignKey range="#Use" key="#blockNo"/>
  </elementType>

  <elementType id="Use'">
    <foreignKey range="#Use" key="#blockNo"/>
  </elementType>

  <elementType id="produces">
    <foreignKey range="#Component" key="#name"/>
    <foreignKey range="#Heap" key="#blockNo"/>
  </elementType>

  <elementType id="log_file">
    <group groupOrder="OR" occurs="ZEROORMORE">
      <element type="#Component"/>
      <element type="#Drag"/>
      <element type="#Heap"/>
      <element type="#Lag"/>
      <element type="#Use'"/>
      <element type="#Void"/>
      <element type="#Waste"/>
      <element type="#Use"/>
      <element type="#produces"/>
    </group>
  </elementType>
</s:schema>
Bibliography


   http://www.microsoft.com/xml/.


http://www.cee.hw.ac.uk/ gnik/apset/.


http://www.dcs.gla.ac.uk/fp/software/green-card/.


http://www.w3.org/TR/1999/WD-xmlschema-1-19991105/.


