Electronic Communications of the EASST
Volume 16 (2009)

Proceedings of the Doctoral Symposium at the International Conference on Graph Transformation (ICGT 2008)

From Hyperedge Replacement to Separation Logic and Back

Mike Dodds and Detlef Plump

18 pages

Guest Editors: Andrea Corradini, Emilio Tuosto
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

From Hyperedge Replacement to Separation Logic and Back

Mike Dodds[1] **and Detlef Plump**[2]

[1]md466@cl.cam.ac.uk
University of Cambridge, UK

[2]det@cs.york.ac.uk
University of York, UK

Abstract: Hyperedge-replacement grammars and separation-logic formulas both define classes of graph-like structures. In this paper, we relate the different formalisms by effectively translating restricted hyperedge-replacement grammars into formulas of a fragment of separation-logic with recursive predicates, and vice versa. The translations preserve the classes of specified graphs, and hence the two approaches are of equivalent power. It follows that our fragment of separation-logic inherits properties of hyperedge-replacement grammars, such as inexpressibility results. We also show that several operators of full separation logic cannot be expressed using hyperedge replacement.

Keywords: hyperedge replacement, separation logic, graph grammars, program verification, graph transformation

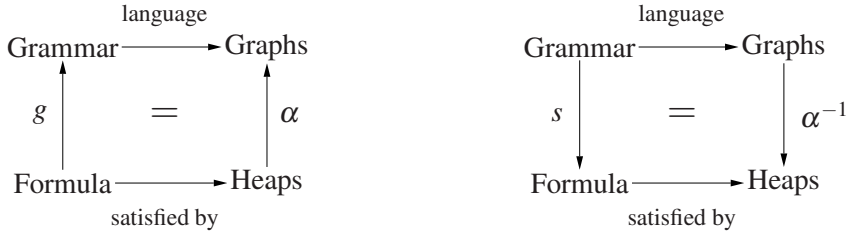
1 Introduction

Hyperedge replacement grammars and separation-logic formulas both define classes of graph-like structures. Hyperedge-replacement grammars [DHK97] are a natural extension of context-free string grammars to the world of hypergraphs. Separation logic [Rey02] is a recently-developed logic which allows compositional reasoning about heap structures using so-called spatial assertions.

In this paper, we identify a class of heap-graphs corresponding to separation logic heaps, and show that hyperedge-replacement grammars generating languages of heap-graphs correspond to formulas in a fragment of separation-logic with recursively defined predicates. To show that this correspondence exists, we define a mapping g from separation logic formulas to grammars generating heap-graphs, and a mapping s from grammars to formulas.

Our major results are as follows:

1. Both g and s are semantics-preserving with respect to a mapping α from states to heap-graphs. That is, the following diagrams commute.



2. As a consequence of (1), our fragment of separation logic is of equivalent expressive power to grammars generating heap-graphs.

3. As a consequence of (2), results for hyperedge replacement languages, such as inexpressibility results, can be imported into our fragment of separation logic.

4. The logical operators $\wedge$ and $\neg$ and the primitive formula **true** are omitted from our fragment. We have proved that a formulas in a fragment with these operators cannot in general be simulated by heap-graph grammars.

This paper is based on Part III of the PhD thesis [Dod08]. It is structured as follows. Sections 2 and 3 define the semantics of separation logic and hyperedge replacement respectively. Section 4 defines the mapping from formulas to grammars and back. Section 5 describes separation logic constructs that cannot be modelled using hyperedge replacement. Section 6 explores the consequences of the correspondence and describes the relationship of our work to other research.

2 Separation logic

Separation logic [Rey02] extends first-order logic with spatial assertions that permit local reasoning. Formulas are interpreted over heaps.

Assumption 1. Loc is a countably infinite set of *locations*, and nil is a distinct value not in Loc. The set of *elements*, Elem is defined as $\mathrm{Loc} \cup \{\mathsf{nil}\}$.

Definition 1 A *heap* h: $\mathrm{Loc} \rightharpoonup \mathrm{Elem} \times \mathrm{Elem}$ is a finite partial function mapping locations to pairs of elements. We write HE for the set of all possible heaps.

We write $img(h)$ for the set of locations held in pairs in the heap, that is $img(h) = \{v \in \mathrm{Loc} \mid \exists v'.\, h(v') = (v,\_) \vee h(v') = (\_,v)\}$. Note that in general $img(h) \setminus \mathrm{dom}(h)$ may be nonempty, meaning elements in h refer to locations outside the heap. The fusion $h_1 \odot h_2$ of two heaps is defined only if $\mathrm{dom}(h_1)$ and $\mathrm{dom}(h_2)$ are disjoint, when it is the union of the two functions.

In our heap model, each location maps to a pair of elements. This model is simple enough to easily define our translation, but rich enough to define interesting structures such as trees and lists. Other heap models, such as the one used in [Sim06], allow integer values in the heap, but prohibit pointer arithmetic. Here we abstract away to deal only with the structure of the heap.

Separation logic extends first-order logic with several constructs permitting reasoning about heaps. The most basic of these is the *points-to assertion* $a \mapsto e_1, e_2$, which asserts that the heap has a singleton domain $\{a\}$ and that it maps a to (e_1, e_2).

Separation logic also introduces the *separating conjunction* $P_0 * P_1$, which asserts that P_0 and P_1 hold in disjoint sections of the heap. This prohibits sharing. For example, the formula $\exists xyz.\,((x \mapsto z,y) * (y \mapsto z,x))$ asserts that the heap associates location x with the pair (z,y), and location y with the pair (z,x), and that x and y are distinct. This is satisfied by heap h_1 in Figure 1, but not by heap h_2 due to sharing.

Our mapping between separation-logic formulas and hyperedge replacement grammars operates over a fragment of the full separation logic. The syntax for the fragment is given below.

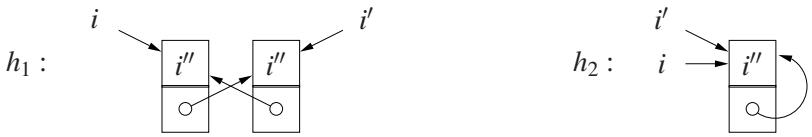


Figure 1: Heap h_1 satisfies the formula $\exists xyz.((x \mapsto z,y) * (y \mapsto z,x))$ with x instantiated with i, y with i' and z with i''. Heap h_2 does not satisfy the formula, due to sharing.

Variables $x, x_1, \ldots, x_n$ are members of a countably infinite set of variable names Var, and σ is a member of a countably infinite set of predicate names Pred. We use SL to stand for the class of all formulas in this fragment.

$$\begin{array}{lcl} E & ::= & \mathsf{Var} \mid \mathsf{nil} \\ F & ::= & \mathsf{emp} \mid x \mapsto E,E \mid F * F \mid \exists x.F \mid F \vee F \mid \sigma(E,\ldots,E) \mid \ \mathsf{let}\ \Gamma\ \mathsf{in}\ F \\ \Gamma & ::= & \sigma(x_1,\ldots,x_n) = F \mid \Gamma,\Gamma \end{array}$$

The only construct in our fragment which does not appear in [Rey02] is let, which is used to recursively define predicates. A let-expression let Γ in F takes as its arguments a list of predicate definitions Γ and a separation logic formula F, over which the definitions scope.

Predicates definitions have the form $\sigma(x_1,\ldots,x_n) = R$. Here σ is a predicate name, $x_1,\ldots,x_n$ are variable arguments, and R is the predicate body. Intuitively, a formula let $\sigma(x_1,\ldots,x_n) = R$ in F states that any (recursive) invocation of the predicate σ in F is satisfied by every heap satisfying R. Predicates defined in Γ scope over the definitions of Γ, so let-statements can define mutually-recursive predicates. We call a formula let-*free* if it does not contain a let-expression.

Example 1. The following formula is satisfied by any heap containing a complete binary tree with nil leaves. We abstract away from a particular root node by existentially quantifying the variable x.

$$\mathsf{let}\ bt(x_1) = (x_1 \mapsto \mathsf{nil},\mathsf{nil}) \vee (\exists x_2,x_3.(x_1 \mapsto x_2,x_3) * bt(x_2) * bt(x_3))\ \ \mathsf{in}\ \exists x.bt(x)$$

In this predicate each branch of the tree holds a pair of values. The leaves of the tree consist of pairs of nil-values.

2.1 Fragment semantics

Satisfaction of a formula in our semantics is defined for a heap h, a variable interpretation i and a predicate interpretation η.

Definition 2 A *variable interpretation* i: Var $\rightharpoonup$ Elem is a partial function mapping variables to heap elements. A variable interpretation i defines an interpretation function $[\![\_]\!]i$: Var $\cup \{\mathsf{nil}\} \rightharpoonup$ Elem, where $[\![\mathsf{nil}]\!]i = \mathsf{nil}$ and $[\![v]\!]i = i(v)$ for all $v \in$ Var.

Definition 3 A *predicate interpretation* η : Pred $\rightharpoonup$ $Pow(\mathrm{Loc}^* \times \mathsf{HE})$ is a partial function mapping predicate names to (possibly infinite) sets of pairs, consisting of a sequence of locations and a heap.

$$
\begin{array}{lll}
h,i,\eta \models \mathsf{emp} & \text{iff} & \mathrm{dom}(h) = \emptyset \\
h,i,\eta \models (x \mapsto E_1, E_2) & \text{iff} & \mathrm{dom}(h) = \{[\![x]\!]i\} \text{ and } h([\![E]\!]i) = ([\![E_1]\!]i, [\![E_2]\!]i) \\
h,i,\eta \models P * Q & \text{iff} & \text{there exist } h_0, h_1 \text{ such that } h_0 \odot h_1 = h \\
 & & \quad \text{and } h_0,i,\eta \models P \text{ and } h_1,i,\eta \models Q \\
h,i,\eta \models P \vee Q & \text{iff} & h,i,\eta \models P \text{ or } h,i,\eta \models Q \\
h,i,\eta \models \exists x.P & \text{iff} & \text{exists } v \in \mathrm{Loc} \text{ such that } h,i[x \to v],\eta \models P \\
h,i,\eta \models \sigma(E_1,\ldots,E_n) & \text{iff} & ([\![E_1]\!]i,\ldots,[\![E_n]\!]i),h) \in \eta(\sigma) \\
h,i,\eta \models \mathsf{let}\ \Gamma\ \mathsf{in}\ P & \text{iff} & h,i,\eta\,[\beta \mapsto k(\beta)]_{\beta \in \mathrm{dom}(\Gamma)} \models P \\
\quad \text{where } k = \ \mathrm{fix} & \lambda k_0.\lambda\sigma \in \mathrm{dom}(\Gamma). & \\
 & \text{bind } \sigma(\vec{x}) = Q \text{ to } \Gamma(\sigma) & \\
 & \text{in } \{(\vec{l},h) \mid h, [\vec{x} \mapsto \vec{l}], \eta\,[\beta \mapsto k_0(\beta)]_{\beta \in \mathrm{dom}(\Gamma)} \models Q\} &
\end{array}
$$

Figure 2: Definition of satisfaction for separation logic.

The definition of the satisfaction relation $\models$ is given in Figure 2. We make use of several extra concepts in this definition. $\vec{x}$ stands for a sequence of variables $x_1,\ldots,x_n$ for some n. Similarly $\vec{l}$ stands for a sequence of values. Square brackets are used to denote the function update operator, so $f' = f[x \mapsto y]$ is defined so $f'(a) = f(a)$ when $a \neq x$ and $f'(a) = y$ otherwise. We use $[\vec{a} \mapsto \vec{b}]$ to stand for $f_\perp[\vec{a} \mapsto \vec{b}]$, where $f_\perp$ is the empty function with $\mathrm{dom}(f_\perp) = \emptyset$.

Given a predicate definition Γ, $\mathrm{dom}(\Gamma)$ is defined as the set of names of predicates defined in Γ. We write 'bind $\sigma(\vec{x}) = R$ to $\Gamma(\sigma)$ in F' to associate $\vec{x}$ and R with the variables and right-hand side of the syntactic definition of σ given in Γ.

Predicates are defined using a recursive let-statement. A formula let Γ in P is satisfied by a heap h in variable interpretation i and predicate interpretation η if h satisfies P in the *new* interpretation defined by Γ. The new interpretation is defined as the least fixed-point of the following function $f_{\eta,\Gamma}$.

$$
\begin{array}{l}
f_{\eta,\Gamma} = \lambda k_0.\lambda\sigma \in \mathrm{dom}(\Gamma). \\
\qquad \text{bind } (\sigma(\vec{x}) = Q) \text{ to } \Gamma(\sigma) \\
\qquad \text{in } \{(\vec{l},h) \mid h, [\vec{x} \mapsto \vec{l}], \eta\,[\beta \mapsto k_0(\beta)]_{\beta \in \mathrm{dom}(\Gamma)} \models Q\}
\end{array}
$$

Intuitively $f_{\eta,\Gamma}$ applies a single iteration of the recursive definitions in Γ. Let $\sigma(\vec{x}) = Q$ be a definition in Γ. The function $f_{\eta,\Gamma}$ is defined so that $f_{\eta,\Gamma}(k_0,\sigma)$ is the set of pairs that satisfy Q in the predicate interpretation k_0. The fixed-point is defined with respect to $\leq_p$, the partial order on predicate interpretations defined by set-inclusion for each predicate name. That is, $\eta_1 \leq_p \eta_2$ if for every predicate symbol σ in the domain of η_1, it holds that $\eta_1(\sigma) \subseteq \eta_2(\sigma)$.

Lemma 1 (Existence of least fixed-point) *Let η and Γ be an arbitrary predicate interpretation and predicate definition respectively. There exists a least fixed-point for $f_{\eta,\Gamma}$ with respect to $\leq_p$.*

Proof. $\leq_p$ defines a complete lattice, and the fragment SL does not include $\neg$ or $\Rightarrow$, so $f_{\eta,\Gamma}$ is

$$
\begin{array}{rcl}
\mathit{lift}(\exists x.\,\mathsf{let}\ \Gamma\ \mathsf{in}\ P) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma\ \mathsf{in}\ \exists x.\,P \\
\\
\mathit{lift}((\mathsf{let}\ \Gamma\ \mathsf{in}\ P) * Q) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma\ \mathsf{in}\ (P * Q) \\
\mathit{lift}(P * (\mathsf{let}\ \Gamma\ \mathsf{in}\ Q)) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma\ \mathsf{in}\ (P * Q) \\
\mathit{lift}((\mathsf{let}\ \Gamma_1\ \mathsf{in}\ P) * (\mathsf{let}\ \Gamma_2\ \mathsf{in}\ Q)) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma_1, \Gamma_2\ \mathsf{in}\ (P * Q) \\
\\
\mathit{lift}((\mathsf{let}\ \Gamma\ \mathsf{in}\ P) \vee Q) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma\ \mathsf{in}\ (P \vee Q) \\
\mathit{lift}(P \vee (\mathsf{let}\ \Gamma\ \mathsf{in}\ Q)) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma\ \mathsf{in}\ (P \vee Q) \\
\mathit{lift}((\mathsf{let}\ \Gamma_1\ \mathsf{in}\ P) \vee (\mathsf{let}\ \Gamma_2\ \mathsf{in}\ Q)) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma_1, \Gamma_2\ \mathsf{in}\ (P \vee Q) \\
\\
\mathit{lift}(\mathsf{let}\ \Gamma_1\ \mathsf{in}\ (\mathsf{let}\ \Gamma_2\ \mathsf{in}\ P)) & \stackrel{\text{def}}{=} & \mathsf{let}\ \mathit{lift}(\Gamma_1), \Gamma_2\ \mathsf{in}\ P \\
\mathit{lift}(\mathsf{let}\ \Gamma\ \mathsf{in}\ P) & \stackrel{\text{def}}{=} & \mathsf{let}\ \mathit{lift}(\Gamma)\ \mathsf{in}\ P \\
\\
\mathit{lift}(\Gamma_1, \Gamma_2) & \stackrel{\text{def}}{=} & \mathit{lift}(\Gamma_1),\ \mathit{lift}(\Gamma_2) \\
\mathit{lift}(\sigma(\vec{x}) = \mathsf{let}\ \Gamma\ \mathsf{in}\ P) & \stackrel{\text{def}}{=} & \Gamma, \sigma(\vec{x}) = P
\end{array}
$$

Figure 3: Rewriting function *lift* used by the flattening function *flat*.

monotonically increasing with respect to $\leq_p$. Consequently the existence of a least fixed-point is guaranteed by Tarski's fixed-point theorem [Tar55]. □

2.2 Flattening formulas

Mapping directly from our fragment of separation logic to a graph grammar is difficult, because the potential nesting of let-constructs makes the definition of productions complex. For this reason, our mappings operate over the restricted domain of flat formulas.

Definition 4 A formula $F \in \mathsf{SL}$ is *flat* if it is a let-statement $\mathsf{let}\ \Gamma\ \mathsf{in}\ P$ where Γ and P are let-free. We write SLF for the class of all flat formulas in SL.

Every formula in our fragment can be rewritten as an equivalent flat formula. To show this we define a function *flat* which takes as its input an arbitrary formula in our fragment and produces a sematically-equivalent flattened formula. A formula is flattened by incrementally 'promoting' let-expressions outwards, merging together predicate definitions, until only the outermost let remains. We assume input formulas are *conflict-free*, meaning that predicate names are not reused between distinct predicate definitions.

The flattening is performed by the function *lift*, which takes as its argument a formula F with flat sub-expressions and produces a flattened formula $\mathit{lift}(F)$. *lift* is defined in Figure 3. Here, the variables P and Q always stand for let-free formulas. All unintroduced cases are defined as the identity transformation.

The flattening function *flat* is defined by a bottom-up transformation of the structure of a formula. Function *flat* recursively applies *flat* to the sub-expressions of the formula, and then

applies the rewriting function *lift* to the resulting formula.

Proposition 1 *For every heap h, variable interpretation i and predicate interpretation η, $h,i,\eta \models F$ if and only if $h,i,\eta \models \mathit{flat}(F)$.*

Proof. By appeal to the correctness of *lift*. See [Dod08] for full details. □

3 Hyperedge replacement

This section reviews hyperedge-replacement graph transformation, based on the definitions given in [DHK97]. We assume a fixed label alphabet C and a fixed *arity function* $\mathrm{ari}\colon C \to \mathbb{N}$.

A *hypergraph* H over C and ari is a tuple $H = \langle V_H, E_H, att_H, l_H, ext_H \rangle$. Here V_H and E_H are, respectively, finite sets of *vertices* (or *nodes*) and *hyperedges* (often just referred to as *edges*), $l_H\colon E \to C$ assigns an edge label to each edges, and $att_H\colon E \to V^*$ assigns to edges a sequence of *attachment* nodes, with $|att_H(e)| = \mathrm{ari}(l(e))$ for all $e \in E_H$. The first element of $att_H(e)$ is the *source* of hyperedge e, if it exists. We denote by $att_H(e)[i]$ the ith attachment point of e. Finally, $ext_H \in V^*$ defines a sequence of pairwise-distinct *external* nodes.

Given a hypergraph H and set $X \subseteq C$ of labels we denote by E_X^H the set $\{e \in E_H \mid l(e) \in X\}$ of hyperedges of H with labels in X. The class of all hypergraphs over C is denoted by H_C. A graph $H \in \mathsf{H}_C$ is said to be a *singleton* if $V_H = ext_H$ and $|E_H| = 1$. A singleton H with $E_H = \{e\}$ and $att(e) = ext_H$ is a *handle*. The unique handle for a label A is denoted $A^\bullet$.

Let $H \in \mathsf{H}_C$ be a hypergraph, $e \in E_H$ a hyperedge to be replaced, and $R \in H_C$ a hypergraph with $|ext_R| = \mathrm{ari}(l(e))$. Then the *replacement* of e in H by R yields the hypergraph $H[R/e]$ by removing e from E_H, adding R disjointly, and merging the i-th external node of R with $att(e)[i]$ for $i = 1,\ldots,|ext_R|$.

A *production* over a set of labels X is a pair (A,R) such that $A \in X$, $R \in H_C$ and $\mathrm{ari}(A) = |ext_R|$. Let H and H' be graphs in H_C and P a finite set of productions over X. Then H *directly derives* H', denoted by $H \Rightarrow_P H'$, if there are an edge e in H and a production $(l(e),R)$ in P such that $H' \cong H[R/e]$. (Here $\cong$ denotes isomorphism of hypergraphs). A *derivation* $H \Rightarrow_P^* H'$ is a sequence of direct derivations $H \cong H_0 \Rightarrow_P H_1 \Rightarrow_P \ldots \Rightarrow_P H_n \cong H'$, where $n \geq 0$. Given a derivation $R \Rightarrow^* H$ and edge $e \in E_R$, the *projection* of e in H, written $H(e)$ is the subgraph of graph H resulting from edge e in the derivation.

A *hyperedge-replacement grammar* (or *HR grammar*) G over C is a tuple $G = \langle T,N,P,Z \rangle$ where $T \subseteq C$ and $N \subseteq C$ are sets of terminal and non-terminal symbols respectively, P is a finite set of productions over N, and Z is a finite set of initial graphs. The language of graphs $\mathrm{L}(G)$ *generated* by G is the set of all graphs $H \in \mathsf{H}_T$ such that there exists a derivation $I \Rightarrow_P^* H$ for some $I \in Z$. The *system of equations* $EQ_P : N \to \mathsf{H}_{N \cup T}$ associated with G is defined as $EQ_P(A) = \{R \mid (A,R) \in P\}$.

Note that we define hyperedge replacement grammars with a finite initial set of graphs, compared to the standard definition which has a single initial graph. This change makes no difference to the properties of the grammars.

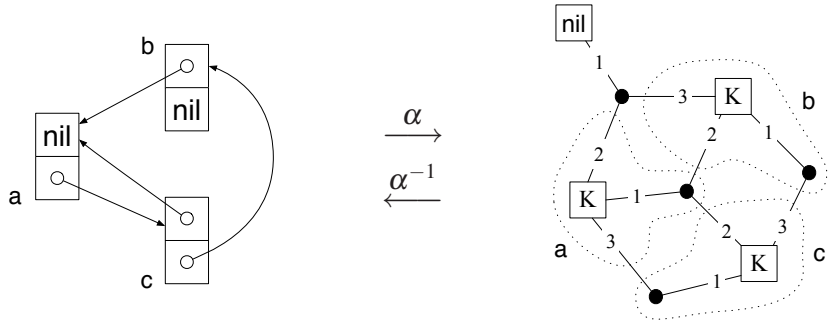


Figure 4: Left: A heap h with three locations. Right: The corresponding heap-graph $\alpha(h)$.

3.1 Heap-graphs and heap-graph grammars

We define heap-graphs as hypergraphs corresponding to separation logic heaps. A location l in a heap h is modelled in a heap-graph as a node to which the first attachment point of a single hyperedge of arity 3 is attached. This edge is labelled with the distinguished terminal label 'K'. The second and third attachment points of the edge correspond to the pair of heap elements $(l_1, l_2) = h(l)$. The nil value in the heap is modelled by a nil-labelled edge of arity 1.

Definition 5 Consider the label alphabet $C = \{K, \mathsf{nil}\}$ with $\mathrm{ari}(K) = 3$ and $\mathrm{ari}(\mathsf{nil}) = 1$. A hypergraph H over C is a *heap-graph* if (1) H contains exactly one nil-labelled edge and (2) each node is the source of at most one edge. We write HG for the set of all heap-graphs.

We define a representation function α mapping heaps to corresponding heap graphs. The function constructs a node for each location with a defined value and an additional unique node for the atomic value nil.

Definition 6 The function $\alpha\colon \mathsf{HE} \to \mathsf{HG}$ is defined as follows. Given a heap h, the heap-graph $\alpha(h)$ has nodes $V = \mathrm{dom}(h) \cup \mathit{img}(h) \cup \{v_{\mathsf{nil}}\}$ and edges $E = \mathrm{dom}(h) \cup \{e_{\mathsf{nil}}\}$. For each $k \in E - \{e_{\mathsf{nil}}\}$, $l(k) = K$ and $\mathit{att}(k) = k, k_1, k_2$ where $h(k) = (k_1, k_2)$. Moreover, $l(e_{\mathsf{nil}}) = \mathsf{nil}$ and $\mathit{att}(e_{\mathsf{nil}}) = v_{\mathsf{nil}}$.

In the following we consider heaps and heap-graphs distinct only up to isomorphism. Under this assumption, α is a bijection, and hence there exists an inverse function $\alpha^{-1} : \mathsf{HG} \to \mathsf{HE}$ that is also a bijection.

The domain of the heap on the left-hand side of Figure 4 contains three locations. The resulting heap-graph, shown on the right of Figure 4, contains three K-labelled hyperedges, corresponding to locations defined by the heap function, and four nodes. The heap locations are labelled a, b, c, and the corresponding nodes and edges in the graph are identified by dashed regions. The extra node corresponds to the nil-element of the heap.

We call hyperedge-replacement grammars that generate languages of heap-graphs *heap-graph grammars*, and use HRH to refer to the class of all heap-graph grammars.

3.2 Source normalisation and compatible properties

When constructing the grammar that corresponds to a separation-logic formula, we must be careful that we construct a heap-graph grammar. It is simple to ensure syntactically that the first clause of Definition 5, that exactly one nil-labelled edge exists, is respected by every graph generated by a grammar. The second clause, that each node is the source of at most one edge, is more tricky to ensure.

We say that a graph is *well-sourced* if each node is the source for at most one hyperedge. Whether all graphs generated by a grammar are well-sourced is a property of the whole grammar, rather than a simple syntactic property of individual productions. A derivation sequence violating this requirement may involve many rules, and removal of a single rule may convert the grammar into a valid heap-graph grammar.

Definition 7 Given hyperedge-replacement grammars G and G', G' is a *source normalisation* of G if $\mathrm{L}(G')$ is the set of all well-sourced graphs in $\mathrm{L}(G)$.

In [DHK97] it is proved that for any hyperedge-replacement grammar G and predicate P expressing a *compatible property* ([DHK97], Definition 2.6.1), a HR grammar G_P can be constructed so that $\mathrm{L}(G_P) = \{H \in \mathrm{L}(G) \mid P(H)\}$.

Intuitively, a property is compatible if it can be tested for a graph G derived from graph R just by testing the property for the projections in G of the nonterminal edges of R. Restriction of a grammar to a compatible property works because the property can be ensured statically by embedding property information into the grammar's nonterminal symbols.

We now state formally the definition of a compatible predicate. This definition is taken verbatim from [Hab92], with minor updates to conform to our notation.

Definition 8 Let C be a class of HR grammars, I a finite set, called the index set, $PROP$ a decidable predicate defined on pairs (H,i), with $H \in \mathsf{H}_C$, and $i \in I$, and $PROP'$ a decidable predicate on triples $(R,assign,i)$ with $R \in \mathsf{H}_C$, a mapping $assign\colon E_R \to I$, and $i \in I$. Then $PROP$ is called $(C,PROP')$-*compatible* if for all $HRG = \langle N,T,P,Z\rangle \in \mathsf{C}$ and all derivations $A^\bullet \Rightarrow R \Rightarrow^* H$i with $A \in N \cup T$ and $H \in \mathsf{H}_T$, and for all $i \in I$, $PROP(H,i)$ holds if and only if there is a mapping $assign\colon E_R \to I$ such that $PROP'(R,assign,i)$ holds and $PROP(H(e),assign(e))$ holds for all $e \in E_R$.

A predicate $PROP_0$ on H_C is called C-*compatible* if predicates $PROP$ and $PROP'$ and an index i_0 exist such that $PROP$ is $(\mathsf{C},PROP')$-compatible and $PROP_0 = PROP(-,i_0)$. ($PROP(-,i_0)$ denotes the unary predicate given by $PROP(-,i_0)(H) = PROP(H,i_0)$ for all $H \in \mathsf{H}_C$.)

This definition requires the existence of three predicates $PROP$, $PROP'$ and $PROP_0$. Property information is held using index values. $PROP$ is the property defined without any auxilliary information recorded for non-terminal nodes. $PROP'$ records auxiliary information for nonterminal edges using the assignment function $assign$. $PROP_0$ eliminates the auxilliary information from $PROP$ by just requiring that $PROP$ holds for some index.

We now define a predicate WS_0 expressing well-sourcedness.

Definition 9 Let WS' be a predicate defined on triples $(R,assign,I)$, with $R \in \mathsf{H}_C$ a hypergraph

over label-set C, $assign : E_R^N \to Pow(\mathbb{N})$ a mapping from non-terminal hyperedges to sets of natural numbers, and $I \subseteq \mathbb{N}$ a finite set of natural numbers. The set *assign* records the configuration of terminal edges which can replace the non-terminal edges, while I records the edges attached to the external nodes of R.

$WS'(R, assign, I)$ holds if and only if: (1) For each node $v \in V_R$ there exists at most one edge $e \in E_R$ such that either $l(e)$ is terminal and $att(e)[1] = v$, or $l(e)$ is non-terminal and there exists $n \in assign(e)$ such that $att(e)[n] = v$. (2) For each $i \in I$ there exists exactly one edge e such that either $l(e)$ is terminal and $s(e) = ext_R(i)$, or $l(e)$ is non-terminal and there exists $n \in assign(e)$ such that $att(e)[n] = ext_R(i)$.

The predicate WS is defined as $WS(H, I) = WS'(H, emp, I)$. The predicate $WS_0(H)$ holds if $WS(H, I)$ holds for some $I \subseteq \mathbb{N}$.

Lemma 2 *WS_0 holds for a graph H if and only if H is well-sourced.*

Proof.
If. If graph H is well-sourced, then the first requirement for WS' must be satisfied by definition, because every edge is terminal and every node has at most a single E-labelled edge with the node as its first attachment point. The second requirement is automatically satisfied for the same reason; as every node has either zero or one edge with the node as its source, any appropriate I can always be constructed.

Only-if. The condition on terminal edges ensures that the graph satisfies the restriction on E-labelled edge sources. The definition of a heap-graph (Definition 5) places no restriction on the external nodes of the graph, meaning that any I for the external nodes will result in a well-sourced graph. □

Lemma 3 *WS_0 is a compatible predicate in the sense of Def. 8.*

Proof. We must show that for any derivation $R \Rightarrow^* H$, $WS(H, I)$ holds if and only if there is a mapping $assign : E_R^N \to Pow(\mathbb{N})$ such that $WS'(R, assign, I)$ and $WS(H(e), assign(e))$ holds for all $e \in E_R^N$.

If. Assume that the mapping $assign : E_R^N \to Pow(\mathbb{N})$ exists. If we consider a node $n \in N_H$, then there must be at most one attached edge with n as a source. This is because the function *assign* records the attached edges for the external nodes of the subgraphs $H(e)$, so if we replace the non-terminals with the terminal graphs, $WS(H, I)$ must hold.

Only-if. Assume that $WS(H, I)$ holds. Then there must exist a function *assign* such that $WS'(R, assign, I)$ and $WS(H(e), assign(e))$ holds for all $e \in E_R^N$. Assignment *assign* can be constructed by removing the subgraphs $H(e)$, constructing a set I such that $WS(H(e), I)$ holds, and assigning to $assign(e)$ the set I. The values of I here can be defined by simple examination of the projections $H(e)$ with respect to the function WS. By construction the predicates must then satisfy our requirements. □

As an immediate consequence of Lemma 3 and the result of [DHK97], a source normalisation can be constructed for any hyperedge-replacement grammar. We write $\lfloor G \rfloor$ for the source normalisation of a grammar G.

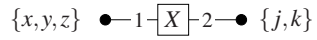
4 Mapping from formulas to grammars and back

We now define the function $g : \mathsf{SLF} \to \mathsf{HRH}$ that maps from flat separation logic formulas to hyperedge-replacement grammars, and the function $s : \mathsf{HRH} \to \mathsf{SLF}$ that maps from grammars to separation-logic formulas.

Graphs in the mappings are constructed by 'gluing together' small graphs into larger graphs. Nodes are identified and merged using *tags*.

Definition 10 Let X be a countably infinite set of tags. A *tagged graph* $R = \langle G, t \rangle$ consists of a hypergraph G and a partial tagging function $t : V_G \to Pow(X)$ that associates tags sets to nodes. The *tag-set* for a node v is $t(v)$. $\mathrm{tag}(R)$ gives the union of all tag-sets in R. A tagged graph is *well-tagged* if the tag-sets of the nodes are pairwise disjoint. The tagged graph $R \setminus x$ is the graph R with the tag x removed from all tag-sets.

We show tags visually as label-sets next to graph nodes. In the following two-node graph, the first and second attachment points of the X-edge have tag-sets $\{x,y,z\}$, and $\{j,k\}$ respectively.



Definition 11 Let h be a heap and i a variable interpretation. Then $\alpha_{\mathsf{tag}}(h,i)$ is the tagged graph $\langle G,t \rangle$. Here $G = \alpha(h)$, and the tagging function t is defined so that for any node $v \in V_G$ and any variable $x \in \mathrm{Var}$, $x \in t(v)$ if $i(x) = l$ and l is the unique heap location in h mapped to v by α.

Definition 12 Let $H = \langle G,t \rangle$ be a tagged graph. A unification step on R is constructed by fusing any pair of nodes v_1 and v_2 where $t(v_1) \cap t(v_2) \neq \emptyset$. The resulting node is tagged with $t(v_1) \cup t(v_2)$. The *tag unification* of R denoted by $\Downarrow R$, is the well-tagged graph constructed by applying unification steps as long as possible.

A tag unification is guaranteed to exist for any graph because unification steps are size-reducing and can be applied to any non-well-tagged graph. The tag unification is unique because unification steps are strongly confluent.

Definition 13 The *join* of two tagged graphs R_0 and R_1, denoted by $R_0 \bowtie R_1$, is the tag unification of the disjoint union of R_0 and R_1: $\mathsf{R}_0 \bowtie R_1 = \Downarrow R_0 \uplus R_1$.

Tags are used by the expose operator to attach external nodes to a graph.

Definition 14 Let $R = \langle H,t \rangle$ be a well-tagged graph, and $\vec{x} = x_1,\ldots,x_n$ be a sequence of tags such that each x_i belongs to $\mathrm{tag}(R)$. Then $expose(H,\vec{x})$ is identical to H, except (1) it has n external nodes, and (2) the ith external node is the node in H tagged with x_i.

4.1 Mapping from formulas to grammars

Function g: $\mathsf{SLF} \to \mathsf{HRH}$, defined in Figure 5, maps a formula F in SLF to a set of initial graphs Z and a set of productions P. A minimal set N of non-terminal labels and T of terminal labels

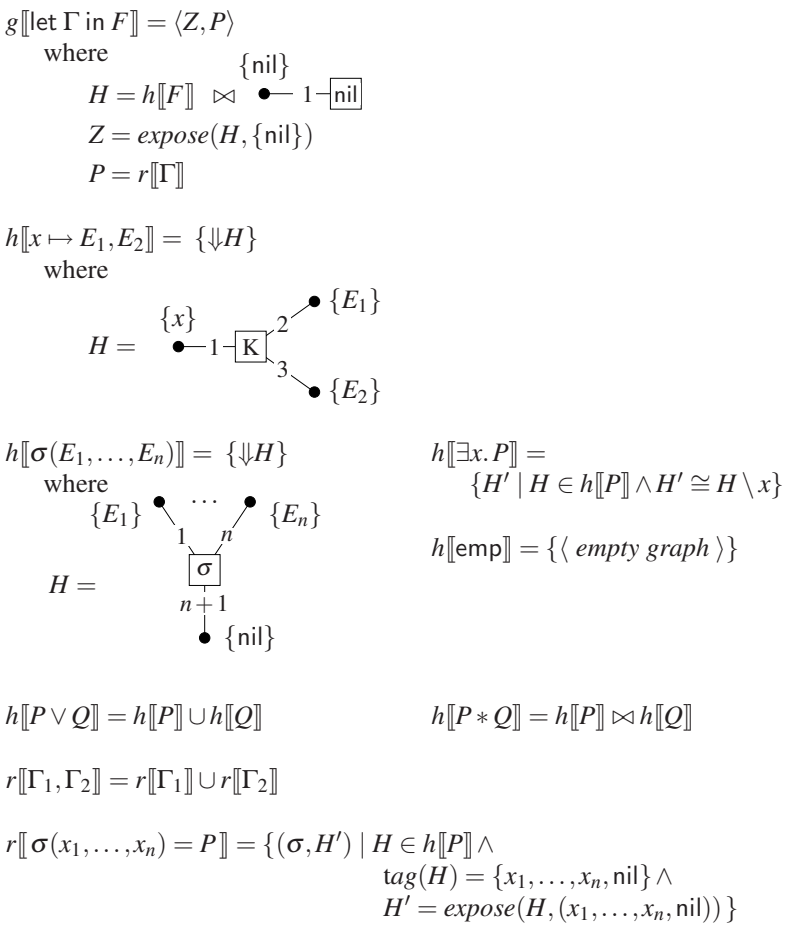


Figure 5: Mapping g from separation logic formulas to hyperedge-replacement grammars.

can be inferred from the labels used in the productions and the initial graphs. This suffices to define a heap-graph grammar $H = \langle N, T, P, Z \rangle$.

The function is defined using two subsidiary functions: h, which takes as its argument a let-free formula and recursively constructs a sets of tagged graphs, and r, takes as its argument let-formulas and constructs sets of productions.

The base cases for h construct edges from points-to assertions and predicate calls. Terminal K-labelled hyperedges are constructed from points-to assertions, with attached nodes tagged according to the assertion's arguments. Non-terminal hyperedges are constructed from calls to predicates. Attachment points are tagged corresponding to the predicate's variable arguments, with the addition of a nil-tagged node needed to track the common nil. In both cases, after generating an edge the tag unification operator $\Downarrow$ is applied. This ensures that only a single node exists with any tag x in its variable-set.

The recursive cases for h merge graphs using the graph join operator. Disjunction merges by union the graph-sets constructed for each subexpression of the formula. Intuitively, a heap satis-

fies the disjunction if it satisfies either the left-hand subexpression or right-hand subexpression, so set-union models the intuitive semantics. The separating conjunction applies the graph-join operator $\bowtie$ pairwise to the two sets constructed from its subexpressions. This fuses any nodes that are referred to by the same variable.

The function r is called by g to construct productions from predicate definitions. The right-hand sides of productions are constructed by applying h to the right-hand sides of the predicate definitions and then apply the expose operator to attach the appropriate external nodes.

Consider the following small formula, which asserts that the heap contains a linked list.

$$\mathsf{let}\ ll(x_1) = (x_1 \mapsto \mathsf{nil}, \mathsf{nil}) \vee (\exists x_2.\,(x_1 \mapsto \mathsf{nil}, x_2) * ll(x_2))\ \mathsf{in}\ \exists x.\,ll(x)$$

Applying g to the two subexpressions $(x_1 \mapsto \mathsf{nil}, x_2)$ and $ll(x_2)$ gives the following pair of single-element graph sets.

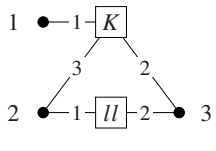
$$h[\![x_1 \mapsto \mathsf{nil}, x_2]\!] \quad = \quad \left\{ \begin{array}{l} \{x_1\} \bullet\!\!-1-\boxed{K}-2-\!\!\bullet \{\mathsf{nil}\} \\ \qquad\qquad\quad 3 \\ \qquad\qquad\quad \bullet \{x_2\} \end{array} \right\}$$

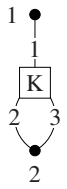
$$h[\![ll(x_2)]\!] \quad = \quad \left\{ \{x_2\} \bullet\!\!-1-\boxed{ll}-2-\!\!\bullet \{\mathsf{nil}\} \right\}$$

The fusion of these two graphs gives the following single graph. Note that the label x_2 has been removed as per the mapping of existentially quantified variables.

$$h[\![\exists x_2.\,(x_1 \mapsto \mathsf{nil}, x_2) * ll(x_2)]\!] \quad = \quad \left\{ \begin{array}{l} \{x_1\} \bullet\!\!-1-\boxed{K} \\ \qquad\qquad\quad 3 \qquad 2 \\ \qquad\quad \bullet\!\!-1-\boxed{ll}-2-\!\!\bullet \{\mathsf{nil}\} \end{array} \right\}$$

When combined with the other disjunct in the predicate definition, this results in the following production definitions.

$ll \Rightarrow$ 



When combined with an initial graph constructed from $\exists x.\,ll(x)$, these productions give a grammar corresponding to the formula.

4.2 Mapping from grammars to formulas

We now define the function $s : \mathsf{HRH} \to \mathsf{SLF}$, which maps a pair $\langle Z, P \rangle$ consisting of a set of initial graphs Z and set of productions P to a recursive let-statement $\mathsf{let}\ \Gamma_P\ \mathsf{in}\ F_Z$. The definition of s is given in Figure 6.

The definition of s uses the subsidiary functions s_q, s_G, s_g and s_e, which operate over productions, sets of graphs, graphs and edges respectively. s calls s_q to construct the predicate definitions Γ_P and s_G to construct the let-body F_Z.

$$s[\![\langle Z, P\rangle]\!] \stackrel{\text{def}}{=} \text{let } s_q[\![EQ_P]\!] \text{ in } s_G[\![Z]\!]$$

$$s_G[\![\{g_1,\ldots,g_m\}]\!] \stackrel{\text{def}}{=} s_g[\![g_1]\!] \vee \ldots \vee s_g[\![g_m]\!]$$

$$s_g[\![\langle g,t\rangle]\!] \stackrel{\text{def}}{=} \exists y_1 \ldots \exists y_m . s_e[\![e_1,t]\!] * \ldots * s_e[\![e_n,t]\!] * \mathsf{emp}$$
where
$\{e_1,\ldots,e_n\} = \{e \in E_g \mid l(e) \neq \mathsf{nil}\}$
$\{y_1,\ldots,y_m\} = \{t(v) \mid v \in V_g \wedge v \notin ext_g \wedge t(v) \neq \mathsf{nil}\}$

$$s_e[\![e,t]\!] \stackrel{\text{def}}{=} \begin{cases} t(v) \mapsto t(v_0), t(v_1) & \text{if } l_E(e) = K \\ \quad \text{where } att(e) = (v, v_0, v_1) & \\ \sigma(t(v_1),\ldots,t(v_n)) & \text{if } l_E(e) \in N \\ \quad \text{where } \sigma = l_E(e) & \\ \qquad att(e) = (v_1,\ldots,v_n) & \end{cases}$$

$$s_q[\![EQ_\mathsf{R}]\!] \stackrel{\text{def}}{=} \sigma(x_1,\ldots,x_n,\mathsf{nil}) = s_G[\![EQ_\mathsf{R}(\sigma)]\!],\ s_q[\![EQ_\mathsf{R}/\sigma]\!]$$
where $n = \text{ari}(\sigma)$

Figure 6: Mapping from heap-graph grammars to separation logic formulas.

The mapping needs to associate each node with a particular variable name (or nil). To do this, we assume that every initial graph and production right-hand side G in an input grammar has been replaced with a corresponding well-tagged graph $\langle G, t_G\rangle$. For each graph G, we assume that t_G is an arbitrary but fixed injective tagging function satisfying the following formula. Here n is the number of external nodes in G.

$$t(v) = \begin{cases} \{x_i\} & \text{if } v \in ext_G \text{ and } pos_G(v) = i. \\ \{\mathsf{nil}\} & \text{if a nil-labelled edge is attached to } v. \\ \{r\} \text{ s.t. } r \in Var \setminus \{x_1 \ldots x_n, \mathsf{nil}\} & \text{otherwise.} \end{cases}$$

The tagging function associates internal nodes with arbitrary variable names, while the ith external node of a graph is given the fixed tag 'x_i'. The nil node is tagged with nil.

The function s_G maps a set of tagged heap-graphs to a disjunction, each disjunct of which corresponds to a single graph. These single graphs are constructed by the function s_g, which takes as its input a tagged graph and constructs a separating conjunction, with each conjunct corresponding to a single hyperedge. The conjunction for a graph is wrapped in existential quantifications, which bind all variables corresponding to internal nodes.

Individual conjuncts for a graph are constructed by the function s_e. Terminally labelled K-edges result in points-to assertions, while non-terminal edges result in calls to recursively-defined predicates. The variable arguments for the point-to assertions and predicate calls are recovered from the tagging function.

The function s_q takes as its input a system of equations EQ_R corresponding to a set of productions R, and constructs the predicate definitions for a let-statement. An equation system is used

rather than the original set of productions because s_q needs to recover *all* the right-hand sides for a single symbol.

s_q constructs a single recursive definition for each nonterminal symbol in R. The set of right-hand side graphs for productions over a single symbol are mapped to a right-hand side formula by the function s_G. The arguments to the predicate are fixed as $x_1, \ldots, x_n$, which by the definition of the tagging function form the free variables of the right-hand side formula.

4.3 Mappings preserve semantics

We want the functions g and s to preserve the semantics of formulas and grammars. That is, for every formula $F \in \mathsf{SLF}$ and heap h, we would like that $h, i_0, \eta_0 \models F$ if and only if $\alpha(h) \in \mathrm{L}(g[\![F]\!])$ (where i_0 and η_0 are empty, i.e. $\mathrm{dom}(i_0) = \mathrm{dom}(\eta_0) = \emptyset$). Analogously, for every heap-graph grammar G and graph $H \in H_C$, we would like that $H \in \mathrm{L}(G)$ if and only if $\alpha^{-1}(H), i_0, \eta_0 \models s[\![G]\!]$.

However, the first correspondence holds only for *dangling-free* formulas. These are formulas such that for any satisfying heap h, $img(h) \subseteq \mathrm{dom}(h)$. A separating conjunction of points-to assertions is dangling-free if every variable appearing on the left-hand side of a points-to assertion is either used on the right-hand side of a points-to assertion, or passed as a dangling-free argument to a predicate. Dangling-free formulas can be identified syntactically by tracking sets of dangling variables. Dangling-free arguments to predicates can be identified by recursion over definitions.

Theorem 1 *Let formula $F = \mathsf{let}\ \Gamma\ \mathsf{in}\ P$ be a dangling-free separation logic formula, and let both Γ and P be* let*-free. Let h be a heap. Then $h, i_0, \eta_0 \models F$ if and only if $\alpha(h) \in \mathrm{L}(g[\![F]\!])$.*

Proof. By induction over the structure of a flat formula. See [Dod08]. □

Theorem 2 *Let G be a heap-graph grammar. Let $H \in \mathsf{H}_C$ be a graph. Then $H \in \mathrm{L}(G)$ if and only if $\alpha^{-1}(H), \eta_0 \models s[\![G]\!]$.*

Proof. By incremental proof of the result first for graphs, then sets of graphs, and finally grammars. See [Dod08]. □

Theorem 3 (Equivalent expressive power) *Any class of structures defined by a dangling-free formula in* SL *can be generated by some heap-graph grammar modulo α, and* vice versa.

Proof. Function s is total over heap-graph grammars. The composition $g \circ flat$ of the flattening function *flat* and g is total over formulas in SL. The result follows immediately from the correctness of flattening (Proposition 1) and of the correctness of g and s (Theorem 1 and Theorem 2). □

5 Inexpressible separation logic operators

In this section we show that the logical operators $\wedge$ and $\neg$, and the primitive assertion **true** must be omitted from our fragment.

Lemma 4 *There exists HR-expressible heap-graph languages L_1, L_2 such that $L_1 \cap L_2$ is HR-inexpressible.*

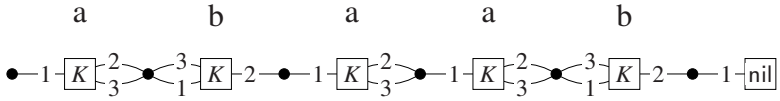
Proof. It is proved in [HK89] that the class of all hyperedge-replacement languages is not closed under intersection. The proof first defines the following pair of string languages.

$$S_1 : \quad \{a^{n_1}b^{n_1}a^{n_2}b^{n_2}\ldots b^{n_{k-1}}a^{n_k}b^{n_k} \mid k \geq 1 \wedge n_1 \ldots n_k \geq 1\}$$

$$S_2 : \quad \{a^{n_1}b^{n_2}a^{n_2}b^{n_3}\ldots b^{n_k}a^{n_k}b^{n_{k+1}} \mid k \geq 1 \wedge n_1 \ldots n_{k+1} \geq 1\}$$

The proof then defines a mapping from any string w to a corresponding string graph $w^\bullet$. S_1 and S_2 are used to define the HR-expressible languages $S_1^\bullet$ and $S_2^\bullet$. The intersection between them gives the language $\{((a^n b^n)^k)^\bullet \mid k \geq 1 \wedge n \geq 1\}$. In [DHK97] it is shown by application of the pumping lemma that this language is HR-inexpressible.

To show the corresponding result for the class of heap-graph languages, we define a mapping from any string $w \in \{a,b\}^*$ to a corresponding string heap-graph w°. A string heap-graph for string w of length n, $w = c_1 c_2 \ldots c_n$ consists of $n+1$ unlabelled nodes $v_1 \ldots v_{n+1}$ and n K-labelled edges $e_1 \ldots e_n$. If character c_n is an 'a', then the edge K has attachment points $[v_n, v_{n+1}, v_{n+1}]$. If c_n is a 'b' then the edge has attachment points $[v_n, v_{n+1}, v_n]$. For the string 'abaab', the corresponding string heap-graph $(\text{abaab})^\circ$ is as follows.



The rest of the proof is then identical to [HK89]. □

Proposition 2 *There exist formulas $F_1, F_2 \in \mathsf{SL}$ such that no grammar $g[\![F_1 \wedge F_2]\!]$ can be defined.*

Proof. Any grammar $g[\![F_1 \wedge F_2]\!]$ must generate the language $\{\alpha(h) \mid h, i_0, \eta_0 \models F_1 \wedge F_2\}$. By Lemma 4 there exist heap-graph grammars G and H such that $\mathrm{L}(G) \cap \mathrm{L}(H)$ is HR-inexpressible. Let us construct formulas $F_1 = s[\![G]\!]$ and $F_2 = s[\![H]\!]$. By the correctness of g and s (Theorems 1 and 2) $L(G) = \{\alpha(h) \mid h, i_0, \eta_0 \models F_1\}$ and $L(H) = \{\alpha(h) \mid h, i_0, \eta_0 \models F_2\}$. By the semantics of conjunction $\{\alpha(h) \mid h, i_0, \eta_0 \models F_1 \wedge F_2\}$ equals $\mathrm{L}(G) \cap \mathrm{L}(H)$, which is HR-inexpressible. Consequently no grammar $g[\![F_1 \wedge F_2]\!]$ can be defined. □

Given a language of graphs L over label alphabet C, we write $\overline{L}$ for the language complement $\{g \in \mathsf{H}_C \mid g \notin L\}$.

Lemma 5 *There exists HR-expressible heap-graph language L such that $\overline{L}$ is HR-inexpressible.*

Proof. Consequence of De Morgan's law $\overline{(\overline{A} \cup \overline{B})} = A \cap B$ and Lemma 4. □

Proposition 3 *There exists a formula $F \in \mathsf{SL}$ such that no grammar $g[\![\neg F]\!]$ can be defined.*

Proof. Corollary of Lemma 5. □

Proposition 4 *No grammar $g[\![\mathbf{true}]\!]$ can be defined.*

Proof. Consequence of the result of [Hab92], Chapter IV, where it is shown that any HR language of simple graphs (meaning graphs without loops or parallel edges) must have a maximum bound on the size of any clique in the language. However, any grammar $g[\![\mathbf{true}]\!]$ must construct the language $\{\alpha(h) \mid h, i_0, \eta_0 \models \mathbf{true}\}$, which is the language of all possible heap-graphs, contradicting this result. □

6 Consequences and related work

We have defined a bijective mapping between the domains of heap-graphs and heaps, and we have shown that the mappings g and s are correct with respect to this mapping. This means that formulas in our fragment and heap-graph grammars can be used interchangeably as methods for defining classes of structures.

An immediate consequence of Theorem 3 is that some results for HR grammars hold for formulas in SL. Most obviously, if a language of graphs is proved to be inexpressible by any hyperedge replacement grammar, then the corresponding class of heaps must also be undefinable in our fragment.

For example, both the languages of grid graphs and balanced binary trees are inexpressible by any hyperedge replacement grammar (consequences of the pumping lemma and linear-growth theorem of [Hab92] respectively) as is the language of all heap-graphs (consequence of the clique-size limit discussed in §5). The most flexible result for proving inexpressibility results is the pumping lemma for hyperedge-replacement languages – see [Hab92] for an extended discussion.

In §3.1 we used the fact that the class of HR languages is closed under intersection with compatible properties to show that we can define a source normalisation operator. This result also holds for any formula in our separation logic fragment. So for any formula in SL, a new formula can be constructed expressing the intersection between the formula and a compatible property.

Some other work has been done on the expressiveness of separation logic. The class of closed separation-logic formula is known to be of equivalent expressive power to the class of formulas in first-order logic without separating operators [Loz04]. However, in [CGZ07] it is shown that the correspondence does not hold for separation logic formulas that include logical parameters standing for formulas, and that it also does not hold between separation logic with a list segment predicate and first-order logic with such a predicate.

Our results regarding expressiveness are incomparable with these results, because our fragment omits several operators from full separation logic but includes a more general notion of recursion. However, our translation gives us a more general framework for deriving expressiveness results, because the class of hyperedge replacement grammars has many well-understood properties.

The work of Lee *et al.* [LYY05] was a major inspiration for our work. In this, a semantics is given to graph grammars by mapping them to separation logic formulas. These grammars are used as the abstract domain in an automatic shape analysis. However, the mapping is only one-way, meaning no correspondence result can be derived. In addition, the grammars used are severely restricted in their expressiveness compared to our heap-graph grammars.

Our fragment SL is quite close to *symbolic heaps* that form the basis of the Space Invader tool [YLB+08]. This suggests that our fragment, while restricted, is still suitable for practical use. Symbolic heaps are a symbolic heap representations defined in a fragment of separation logic (the exact fragment used varies with the paper cited). A symbolic heap $\Pi \mid \Sigma$ as defined in [DOY06] consists of a finite set Π of equalities and a finite set Σ of heap predicates. The elements of Σ are of the form $E \mapsto F$, $\mathsf{ls}(E,F)$, and junk. $\mathsf{ls}(x,y)$ stands for a recursively-defined list segment, while junk can stand for any heap. Semantically, a symbolic heap $\Pi \mid \Sigma$ where $\Pi = \{P_1, P_2, \ldots, P_n\}$ and $\Sigma = \{Q_1, Q_2, \ldots, Q_m\}$ expands to a formula $(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \wedge (Q_1 * Q_2 * \ldots * Q_m)$.

Formulas in our fragment closely resemble the heap portion of a symbolic heap. We include a general notion of recursive predicate, while symbolic heaps use specific predicates defined for particular domains. For example, early work on the Space Invader tool used a simple list fragment predicate $ls(x,y)$, while more recent work has included a nested 'list of lists' predicate. In both cases these predicates can be defined using our let statement.

The predicate junk is equivalent to true, and so by the result given in §5 cannot be expressed by any hyperedge replacement grammar. However, junk is largely used to handle unconnected sections of the heap. Structure in the heap is specified in the junk-free fragment of symbolic heaps, which suggests that the structure-specifying properties of symbolic heaps may be similar to our fragment.

All of the symbolic heap fragments include assertions (such as equality) that describe the relationships between locations. We conjecture that hyperedge replacement as defined in §3 is expressive enough to model equality and inequality. To support this, we refer to [Eng97], where it is proved that for any grammar defined using rules with repetition, a corresponding repetition-free grammar exists that defines the same language. A graph with repetition is one where nodes can appear more than once in the sequence of external nodes. A rule with repetition on its right-hand side will merge some of the attachment points of any nonterminal node to which it is applied. This seems to correspond in some sense to assertions of equality. However, proving this result remains the subject of future work.

Acknowledgements: We would like to thank Matthew Parkinson, Colin Runciman and the anonymous referees for their comments on this work. Work completed while the first author was studying for a PhD at the University of York. Publication supported by EPSRC grant EP/F019394/1.

Bibliography

[CGZ07] C. Calcagno, P. Gardner, U. Zarfaty. Context logic as modal logic: completeness and parametric inexpressivity. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Pp. 123–134. ACM, 2007.

[DHK97] F. Drewes, A. Habel, H.-J. Kreowski. Hyperedge Replacement Graph Grammars. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation. Volume I: Foundations*. Chapter 2, pp. 95–162. World Scientific, 1997.

[Dod08] M. Dodds. *Graph Transformation and Pointer Structures*. PhD thesis, University of York, UK, 2008.
http://www.cl.cam.ac.uk/~md466/publications/phdthesis.mdodds.pdf

[DOY06] D. Distefano, P. W. O'Hearn, H. Yang. A local shape analysis based on separation logic. In *12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science 3920, pp. 287–302. Springer, 2006.

[Eng97] J. Engelfriet. Context-free graph grammars. In *Handbook of formal languages, vol. 3: beyond words*. Pp. 125–213. Springer, 1997.

[Hab92] A. Habel. *Hyperedge Replacement: Grammars and Languages*. Lecture Notes in Computer Science 643. Springer, 1992.

[HK89] A. Habel, H.-J. Kreowski. Filtering Hyperedge-Replacement Through Compatible Properties. In Nagl (ed.), *WG*. Lecture Notes in Computer Science 411, pp. 107–120. Springer, 1989.

[Loz04] E. Lozes. Separation logic preserves the expressive power of classical logic. In *Proceedings of the 2st Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*. 2004. Informal proceedings.

[LYY05] O. Lee, H. Yang, K. Yi. Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis. In *Proceedings of the 14th European Symposium on Programming*. Lecture Notes in Computer Science 3444, pp. 124–140. Springer, April 2005.

[Rey02] J. C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*. 2002.

[Sim06] É.-J. Sims. Extending separation logic with fixpoints and postponed substitution. *Theoretical Computer Science* 351(2):258–275, 2006.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics* 5:285–309, 1955.

[YLB+08] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn. Scalable Shape Analysis for Systems Code. In *International Conference on Computer-Aided Verification*. Lecture Notes in Computer Science 5123, pp. 385–398. Springer, 2008.