Graph Transformation and Pointer Structures

Mike Dodds

Submitted for the degree of
Doctor of Philosophy

The University of York
Department of Computer Science

September 2008

Abstract

This thesis is concerned with the use of graph-transformation rules to specify and manipulate pointer structures. In it, we show that graph transformation can form the basis of a practical and well-formalised approach to specifying pointer properties. We also show that graph transformation rules can be used as an efficient mechanism for checking the properties of graphs.

We make context-sensitive graph transformation rules more practical for specifying structures, by improving their worst-case application time. We define syntactic conditions ensuring faster application of rules, and we show how these conditions improve the application time of sequences of rules. We apply these fast graph transformation systems to the problem of recognising graph languages in linear time, and show that several interesting context-sensitive languages can be recognised using this approach.

We examine the relationship between pointer specification using context-free graph transformation and separation logic, an alternative approach to reasoning about pointers. We show that formulas in a fragment of separation logic can be translated into a restricted class of hyperedge replacement grammars, and vice versa, showing that these two approaches are of equivalent power. This means that our fragment inherits the formal properties of hyperedge-replacement grammars, such as inexpressibility results. We show that several operators of full separation logic cannot be expressed using hyperedge replacement.

We define a C-like language that uses graph transformation rules to ensure pointer safety. This language includes graph transformation constructs for defining and rewriting pointer structures. These constructs can be statically checked for shape safety by modelling them as graph transformation rules. We give both an abstract graph-transformation semantics and a concrete executable semantics for our new constructs, and prove that the semantics correspond.

Contents

I Introduction and preliminaries **13**

1 Introduction **14**

1.1 Background and motivation 15
1.2 Contribution 18
1.3 Thesis structure 21
1.4 Publication history 22

2 Preliminaries **23**

2.1 Graphs and morphisms 23
2.2 Double-pushout graph rewriting 26
2.3 Hyperedge-replacement graph rewriting 29
2.4 Graph signatures 32

II Fast graph transformation and recognition **36**

3 Fast graph transformation **37**

3.1 The problems of graph transformation 38
3.2 Fast left-connected graph transformation 42
3.3 Fast rooted graph transformation 50
3.4 Multi-step graph transformation 54

4 Efficient graph recognition **65**

4.1 Recognition by rooted reduction 66
4.2 Non-context-free RGRS languages 76
4.3 Recognition by left-connected reduction 91
4.4 Comparison between LGRSs and RGRSs 95
4.5 Developing and validating GRSs 96

5 Other approaches tofast graph transformation **99**
5.1 Efficient direct derivations 99
5.2 Efficient multi-step derivation 101
5.3 Efficient recognition and special reduction systems 103

III Graph grammars and separation logic **108**

6 Semantics of formulas and grammars **109**
6.1 Separation logic syntax and semantics 111
6.2 Flattening separation logic formulas 122
6.3 Heap-graphs and mapping between domains 127
6.4 Heap-graph grammars and source normalisation 129

7 Mapping between formulas and grammars **137**
7.1 Intuitive relationship 137
7.2 Mapping from formulas to grammars 139
7.3 Proving the correctness of mapping g 145
7.4 Mapping from grammars to formulas 156
7.5 Proving the correctness of mapping s 158

8 Consequences and limitations **164**
8.1 Inexpressible separation logic operators 164
8.2 Extending the heap model 169
8.3 Consequences of the correspondence 171
8.4 Other related work 175

IV A language for shape safety **176**

9 CGRS: A language for shape safety **177**
9.1 Safe pointers by graph transformation 178
9.2 CGRS: a language for safe pointers 182
9.3 Example: tree insertion and rebalancing 193
9.4 Code size in CGRS 201

10 Semantics of CGRS and shape safety **205**
10.1 Extraction of GRSs and rules from CGRS 205
10.2 Translating CGRS to C 208

10.3 Syntax and semantics of μC 221
10.4 Translating from memory states to graphs 229
10.5 Correctness of translations 234
10.6 Shape safety guarantees in CGRS 242
10.7 Implementing and optimising CGRS 244

11 Other approaches to shape safety 245

11.1 Shape types and structured gamma 245
11.2 Specifying structures using logic 246
11.3 Shape analysis 249
11.4 Specifying structures by graph transformation 252

V Conclusion 255

12 Conclusions and further work 256

12.1 Thesis summary 256
12.2 Further work 260

A Balanced binary trees are not MS-expressible 264

Bibliography 267

List of Figures

2.1 Commutative diagrams defining a pushout. 26
2.2 Commutative diagrams defining pullback. 26
2.3 Two pushouts defining a derivation. 27

3.1 Rooted rule removing an element from a linked list. 52

4.1 RGRS CL for rooted cyclic lists. 70
4.2 Balanced binary tree. 76
4.3 Accepting graph and rules for the rooted GRS RBB. 79
4.4 Accepting graph and rules for the rooted GRS RBB (cont). . 80
4.5 Example reduction sequence for a member of L(RBB). 81
4.6 BBT including three root-pointer-predecessors. 82
4.7 Rooted grid. 85
4.8 Grid RGRS rules and accepting graph. 86
4.9 Rooted binary DAG. 87
4.10 Linear RGRS recognising rooted binary DAGs. 89
4.11 Rules and accepting graph for LGRS BB. 94

5.1 A special reduction system for Stars. 105

6.1 Simple heaps illustrating separation logic satisfaction. 114
6.2 Definition of satisfaction for separation logic. 117
6.3 Flattening function *flat*. 124
6.4 Rewriting function *lift*. 125
6.5 Heap and corresponding heap-graph 129
6.6 Grammar producing non-heap graphs. 130

7.1 Mapping from formulas to grammars. 143
7.2 Transforming a predicate into a grammar. 146

7.3 Transforming a predicate into a grammar (cont). 147
7.4 Mapping from grammars to formulas. 157
7.5 Transforming a grammar into a formula. 159

8.1 Definition of satisfaction for omitted operators. 165

9.1 Rooted GRS defining the language of binary trees. 180
9.2 Graph transformation rule Insert. 181
9.3 Syntax for signatures and shapes. 186
9.4 CGRS signature and shape declarations. 188
9.5 Syntax for transformers and reducers. 189
9.6 Textual syntax for transformer functions. 192
9.7 Reducer `branchleaf`. 194
9.8 Unbalanced tree with AVL labels 195
9.9 GRS for unbalanced trees with AVL labels. 196
9.10 Balanced AVL tree with stack. 196
9.11 Signature and shape declarations for the `avl` shape. 197
9.12 Transformer `rup` and corresponding rule 199
9.13 Transformer `rdouble` and corresponding rule 200
9.14 Transformer `relim` and corresponding rule 201
9.15 Transformer `rsingle` and corresponding rule 202

10.1 Transformer `tree_insert` and corresponding rule. 208
10.2 Translation to C types for CGRS signatures and types. 210
10.3 Source code produced from `bin` and `tree`. 211
10.4 Translation to C for CGRS shapes. 214
10.5 Translation from CGRS to C for transformers. 217
10.6 Source code produced from `tree_goleft`. 218
10.7 Abstract syntax of μC. 222
10.8 SOS semantics of μC control flow. 226
10.9 SOS semantics of μC assignment and dereferencing. 227
10.10SOS semantics of μC memory handling. 229
10.11Large σ-structure 232
10.12Correctness requirement for translation function. 236

11.1 Graph type for a doubly-linked list node. 247

Index of Definitions

WS, 133
WS', 133
  example, 134
WS_0, 133
$\mathbb{C}$-preserving, 29
Σ-rule, 34
  example, 35
Σ_σ, 206
  example, 207
$\_^\bullet$, 165
  example, 166
$\_^\circ$, 166
  example, 166
α, 128
  example, 129
α_t, 141
  example, 141
β_σ, 233
  example, 233
$\bowtie$, 141
  example, 141
κ, 148
  example, 148
$\leq_p$, 118
  example, 118
$\lfloor - \rfloor$, 131
$\mathbb{L}'_R$, 95
$\mathbb{L}_L$, 95
$\mathbb{L}_R$, 95
μC state, 224
μC type schema, 223
μC type schema
  example, 224
$\rightharpoonup$, 25
σ node-pair, 230
  example, 231
σ-schema, 230
  example, 230
σ-structure, 231
  example, 231
$\Downarrow$, 140
  example, 140
ϱ-rooted Σ-rule, 67
  example, 67
flat, 124
  example, 124
lift, 124
  example, 124

abstract declaration, 206
  example, 206
attachment node, 29

balanced binary tree, 76
  example, 76
branching factor, 46
  example, 46

C-like rooted graph, 183
  example, 183

C-like rooted rule, 183
  example, 183
compatible predicate, 132
  example, 133
complete binary tree, 104
Condition 1, 48
Condition 2, 48
Condition 3, 49
Condition R1, 53
Condition R2, 53
Condition R3, 54
confluence, 71
  example, 72
connected component, 24
connected graph, 24
connected nodes, 24
correspondence, 150
  example, 150
corresponding graph signature, 206
  example, 207

dangling condition, 27
dangling location, 112
degree, 23
derivation (double-pushout), 27
  example, 28
derivation (hyperedge replacement), 30
descendant edge, 74
  example, 74
descendant node, 74
  example, 74
direct derivation (double-pushout), 27
direct derivation (hyperedge replacement), 30

edge, 23
edge enumeration, 43
  example, 43
equation system, 149
  example, 149
expose, 142
  example, 142
external node, 29

fast Σ-rule, 91
  example, 91
flat formula, 122
  example, 122

graph, 23
  example, 24
graph class, 29
graph environment, 148
  example, 148
graph evaluation, 148
  example, 148
graph matching problem, 42
graph morphism, 25
  example, 25
graph properties, 24
  example, 24
graph reduction specification, 179
  example, 180
graph signature, 33
  example, 33
GRS type, 178

handle, 29
heap, 112
  example, 112
heap fusion, 117
  example, 118
heap-graph, 128

example, 128
heap-graph grammar, 129
HR grammar, 31
hyperedge, 29
hyperedge replacement, 30
example, 31
hyperedge replacement grammar, 31
example, 32
hypergraph, 29
example, 30

image (heap), 112
in-degree, 23
inclusion, 25
initial graph, 31
injective morphism, 25
interface graph, 26
inverse (rule), 26
irreducible, 27
isomorphic, 25

join, 141
example, 141

label alphabet, 23
language (hyperedge replacement), 31
language family, 31
left-connected graph reduction specification (LGRS), 92
left-connected rule, 42
left-hand side, 26
LGRS recognition problem, 92
linear LGRS, 92
example, 93
linear RGRS, 71
example, 71

matching morphism, 27
morphism correspondence, 237
example, 237
morphism extension, 43
example, 44
multi-step graph transformation problem, 55

natural pushout, 26
node (graph), 23
node (hypergraph), 29
node-pair, 230
example, 231

out-degree, 23

partial morphism, 25
polynomially terminating GRS, 179
example, 180
predicate interpretation, 116
example, 116
preserves undefinedness, 25
production, 30
projection (hyperedge), 132
pullback, 26
pushout, 26

reachable node, 24
replacement, 30
result morphism, 27
RGRS recognition problem, 69
right-hand side, 26
root label, 51
root node, 51
root-pointer-predecessor, 80
example, 80
rooted binary DAG, 87
example, 87
rooted cyclic list

example, 70
rooted graph reduction specification (RGRS), 68
rooted grid, 84
example, 84
rooted rule, 51
example, 52
rule, 26
example, 28
rule application problem, 40

separating edge, 24
shape safe location, 242
shape safety preservation, 242
shape-safe rule, 181
singleton hypergraph, 29
size (rule), 26
source, 29
source normalisation, 131
example, 136
string heap-graph, 166
example, 166
string-graph, 165
example, 166
strong V-structure, 100
example, 100
subgraph, 23
surjective morphism, 25

tagged graph, 139
example, 139
track-morphism, 27
tuple-heap, 169
example, 170
tuple-type schema, 169
example, 170

unify tags, 140
example, 140

variable interpretation, 115
example, 115
vertex (graph), 23
vertex (hypergraph), 29

well-sourced, 128
well-typed state, 224
example, 224

Acknowledgements

I am grateful to my supervisor, Detlef Plump, for his advice, support and guidance during this research, and for giving me the benefit of his invaluable technical understanding.

I would like to thank my examiners, Arend Rensink and Colin Runciman, whose clear and helpful comments have improved this thesis immensely. I would also like to thank Colin for his advice and assistance during my PhD, and for getting me interested in mathematical computer science during my time as an undergraduate.

I would like to thank the University of York Department of Computer Science for acting as my home for the past eight years, and for providing a friendly, intellectually stimulating environment in which to do research. I would like to thank the members of the Programming Languages and Systems group, and especially Neil Mitchell, Sandra Steinert, Greg Manning, and Matthew Naylor, for their help, advice and friendship.

I would like to thank all of the people, including but not limited to those mentioned above, who have commented on various aspects of this work, and especially those who have pointed out my mistakes. This thesis would be much poorer without you.

Finally, I would like to thank my partner Lindsay, my parents Alison and Dave, my family, and my friends.

This thesis is dedicated to Jess, Grace, Abby and Darcy. Sleep well girls.

Author's declaration

I declare that all the work in this thesis is my own, except where attributed and cited to another author. Several sections of this thesis have been published previously. For details, please see the end of Chapter 1.

Part I

Introduction and preliminaries

Chapter 1

Introduction

This thesis is concerned with the use of graph transformation rules to specify and manipulate pointer structures. It aims to show that graph transformation can form the basis of a practical and well-formalised approach to specifying pointer properties, and that graph transformation rules can be used as an efficient mechanism for checking the properties of graphs.

In this thesis we address three main questions. First, how can graph transformation rules be made an efficient mechanism for checking the properties of graph structures? While graph transformation rules are well-understood and semantically clean, their worst-case execution time is too high for many applications, including checking whether a graph is a member of a graph language. We define syntactic conditions ensuring fast application of rules, and apply these conditions to give efficient graph recognition systems.

Second, how do approaches based on graph transformation rules relate formally to other approaches to specifying pointer properties? By examining the relationship between graph transformation and other approaches we can better understand the formal properties of both graph grammars and of other approaches. In this thesis we show that context-free graph grammars are closely related to separation logic [69], probably the most active current approach to specifying pointer properties.

Third, how can graph grammars be used to specify the shape of pointer structures in a practical C-like programming language? Pointer structures are widely used in programming languages, but their properties have proved difficult to specify and verify. We develop a language that uses graph grammars to specify the properties of pointer structures. Graph grammars have previously been used to abstractly specify classes of pointer structures, but applying them to a practical language remains a challenge.

This chapter gives an introduction and overview of the research presented in the thesis. Section 1.1 provides context and motivates our work. Section 1.2 describes the general approach and major research contributions of the thesis. Section 1.3 describes in detail the structure of the thesis. Finally, Section 1.4 describes the prior publication history of the work presented in this thesis.

1.1 Background and motivation

1.1.1 Graph grammars and graph transformation

Graph transformation rules define rewrites over graphs. They generalise to graphs the string rewrite rules familiar from context-free and context-sensitive string grammars. Several different formulations have been proposed (see [71] for an overview), but most can be classed as either context-free or context-sensitive, depending on their power. A major advantage of graph transformation rules is that they have been the subject of a large amount of research into their properties, and they are consequently very well understood formally.

Graph transformation rules can be used to specify grammars in the same way that string rewrite rules specify string grammars. Graph grammars define languages of graphs with common properties, by applying a set of productions to a finite initial element. The various kinds of graph grammar form a powerful and general framework for specifying the properties of graphs.

This thesis makes considerable use of the double-pushout approach to graph transformation, a context-sensitive approach. Rules in this approach consist of a left and right-hand side graph, with an interface between them. Derivations match the left-hand side in the target graph and rewrite it to correspond to the right-hand side. Grammars based on double-pushout rules are generally powerful, in the sense that they can define any recursively enumerable language of graphs.

A major disadvantage of double-pushout graph transformation is that each rule derivation has a polynomial worst-case time complexity given fixed rules. This is a result of the high cost of matching a rule's left-hand side to a subgraph in the target graph. As a consequence, graph grammars and graph recognition systems based on context-sensitive rules have a polynomial worst-case complexity. In this thesis, we address this problem by defining syntactic conditions ensuring improved application time.

This thesis also makes use of the hyperedge-replacement approach to graph transformation, a context-free approach. Hyperedge replacement productions replace single nonterminal edges with graphs. The hyperedge replacement approach is less expressive than the double-pushout approach, in that grammars based on double-pushout rules can express languages that are inexpressible by hyperedge replacement grammars. However, hyperedge replacement grammars have attractive theoretical properties, such as decidable language membership.

1.1.2 Pointer structures and shape safety

Software, unlike most other engineered artifacts, rarely does exactly what we want. The designers of bridges or car engines can generally ensure that their designs will work as intended. In contrast, even well-tested software often behaves in unexpected ways, often by crashing or corrupting information. The consequences of defective software can be severe, including millions of pounds of economic damage and substantial numbers of deaths.

The aim of verification is to ensure that software behaves as we expect. Programs cannot simply be tested to ensure correct behaviour; there are simply too many possible inputs. Instead we must reason formally about programs to show that they conform to specification.

Programming languages such as Java and C store complex data as so-called pointer structures. Programs that use pointers are, however, amongst the most difficult to verify, because pointers permit aliasing, which means that local operations can have global effects. While some progress has been made in verifying other programs, verifying pointer manipulating programs is one of the major outstanding challenges in computer science.

For a pointer program to behave according to its specification the program's pointer structures must have the properties that are expected of them: that they are trees, or are acyclic, or are lists of lists of cyclic lists, and so on. We call these large-scale properties the *shape properties* of a structure, and we call verifying that these properties always hold the problem of *shape safety.* Verifying shape safety is a necessary precondition for verifying the correctness of most pointer programs.

Shape safety verification is currently a highly active area of research. One active approach is verification based on separation logic [69]. This is a recently-developed logic for specifying the properties of heaps that extends normal first-order logic with a so-called *separating* conjunction. This allows a formula to specify the spatial relationships between assertions in the heap. Recent work based on separation logic has made considerable progress in verifying pointer-manipulating programs [20].

An important measure for evaluating shape-safety approaches is the expressiveness of the approach used for specifying structures – that is, the properties that can be expressed by the approach. The expressiveness of graph-transformation based systems are well-understood, but surprisingly little work has been done on the expressiveness of other shape-safety approaches, including separation logic.

1.1.3 Specifying pointer properties using graph grammars

Graph grammars can be used as a mechanism for abstractly specifying and checking the shape properties of pointer structures. The Safe Pointers by Graph Transformation (SPGT) project [4, 3] is a recent approach to ensuring the safety of pointer manipulations. This approach uses double-pushout graph transformation rules to define graph reduction systems (GRS), a variant on graph grammars. A GRS consists of a terminal element and a set of reduction rules. Structures are members of the language of a GRS if they can be reduced to the terminal element by the reduction rules.

The SPGT approach uses GRSs to specify the shape properties of pointer structures. Pointer rewrites are then modelled as graph transformation rules. An algorithm has been developed that checks the safety of pointer rewrites expressed as graph transformation rules against a given GRS.

Reduction systems have the desirable property that they come with an implied algorithm for membership checking, which consists of simply applying the reduction rules to the input graph. However, the high cost of applying double-pushout rules means that this checking algorithm can at best achieve a polynomial worst-case termination time, even when reduction is confluent.

Prior work on the SPGT approach focused on abstractly specifying and checking shape properties [4, 3]. Shape checking in this work was defined over graphs resembling pointer structures, rather than over pointer structures. No attempt was made in this prior work to implement the approach with real executable programs.

1.2 Contribution

In this section we describe the major research contributions of this thesis. These fall into three areas: the development of a framework for efficient graph transformation and recognition; the definition of a relationship between hyperedge replacement grammars and a fragment of separation logic; the development of a new language for ensuring shape safety.

1.2.1 Fast graph transformation and recognition

The SPGT project defines the properties of pointer structures using reduction systems based on graph transformation rules. Context-sensitive graph transformation rules in general require at worst polynomial time for each individual application. This has consequences for their use in some practical applications where polynomial time may be too expensive. In particular, as a result of the high worst-case cost of graph transformation, reduction systems generally require polynomial time or worse for graph recognition, which makes run-time checking of shape properties expensive.

The first objective of this thesis is to show that graph transformation rules can be an efficient approach to checking the properties of graphs. To achieve this objective we first develop a approach to graph transformation that improves the application time of individual rules. The syntactic approach we develop is not just suitable for specifying classes of pointer structures; it is a general approach to faster graph transformation.

Our approach is based on using syntactic conditions to restrict the possible search-space for a rule match. We first describe the semantic properties of languages and rule-sets that permit efficient rewriting, and then develop syntactic conditions that ensure that these semantic properties hold. Our work on syntactic conditions continues and generalises the work of Dörr on strong V-structures [25, 26]. (See §5.1 for a comparison between this work and ours).

Our conditions break down into two classes. First, rooted conditions, which require the existence of a uniquely-identified root. These ensure constant time derivation, given fixed rules. Second, left-connected conditions, which do not require roots. These ensure linear-time derivation, again with fixed rules.

We show that rooted rules can be used for efficient multi-step graph transformation. We also show that by sharing information between derivation steps, we can achieve linear-time multi-step derivation even with rules that require linear time individually. To do this we modify an algorithm of Arnborg and Courcelle [1] to amortise the cost of application over the whole derivation.

We then apply this work to the problem of graph recognition by reduction, and define rooted and left-connected approaches to linear-time reduction. We present several examples of complex context-sensitive languages that can be recognised in linear time, including grid graphs, balanced binary trees, and rooted binary DAGs. We also prove some interesting results about the expressiveness of fast graph transformation systems.

We have treated graph transformation entirely abstractly, without applying it to a particular problem. Our work has defined a class of rules / rule systems with an improved worst-case termination behavior. However, heuristic approaches may perform better than our approach in practice for particular problem domains. In §12.2.1 we discuss some of our objectives for testing our system in practice.

1.2.2 Graph transformation and separation logic

The second objective of this thesis is to examine the relationship between shape specification using graph transformation and other approaches to pointer verification. We have chosen to focus on separation logic, and have shown that a close relationship exists between hyperedge replacement grammars and separation logic. We describe two effective translations between restricted hyperedge replacement grammars and formulas in a fragment of separation logic. These translations preserve the semantics of formulas and grammars.

The translations exist because (1) the recursive definitions commonly used in separation logic closely resemble hyperedge replacement productions, and (2) the separating property enforced by separating conjunction corresponds to the context-free property of hyperedge-replacement grammars. The translations demonstrate that formulas in our fragment of separation logic are of corresponding expressive power to HR grammars under our restrictions.

As a consequence, formal results for hyperedge replacement languages, such as inexpressibility results, can be imported into the fragment of separation logic. For example, the languages of red-black trees, balanced binary trees, grid graphs are all known to be HR-inexpressible. Consequently, they are also inexpressible in our fragment of separation logic.

We have proved that the operators omitted from our fragment of separation logic cannot be simulated in general by a corresponding hyperedge replacement grammar. Notably conjunction corresponds to language intersection, and negation to language complement, both of which are known to be HR-inexpressible.

1.2.3 A new language for shape safety

The third objective of this thesis is to show that graph transformation rules can be used to specify the properties of pointer structures in a practical C-like language. Our foundation is the work of the SPGT project on abstractly specifying pointer structures using graph reduction systems. We have defined the language CGRS that applies SPGT checking to the C programming language.

Our approach in the design of CGRS is to extend C with constructs that explicitly correspond to those used in SPGT-style checking. In this, we generalise the approach of Fradet and Le Métayer [31], who define a shape-safety approach based on context-free graph transformation and extend C with constructs corresponding to their shape safety approach. (See §11.1 for a comparison between Fradet and Le Métayer's work and CGRS).

Pointer structures used in the CGRS program have a declared *shape* that specifies the possible form of the pointer structure. Shapes are declared using a textual syntax that is deliberately close to the abstract definition of a GRS. Shape structures in CGRS are manipulated by *transformers*. These have a textual syntax corresponding closely to graph transformation rules.

CGRS deliberately adopts C conventions for its constructs, for example by requiring that transformers are deterministic. CGRS has been designed to work as an extension to C, and CGRS constructs can be used along-side conventional C pointer structures. In addition, the fact that CGRS uses a syntax close to graph transformation rules means that large rewrites can be clearly expressed in CGRS.

The constructs of CGRS have two semantics. First, they have an abstract semantics defined by mapping transformers to graph transformation rules, and shape declarations to GRSs. Second, constructs have a concrete executable semantics, defined by mapping them in-place to chunks of C code.

We have proved that the concrete implementation corresponds to the abstract graph-transformation model. To do this, we define a concrete semantics for a fragment of C, and prove that the implementation of a construct corresponds to its abstract model. This proof means that the abstract semantics of CGRS constructs can be used to check the shape safety of the concrete implementation of the constructs.

In this thesis we have focused on the correctness of the modelling and semantics. We have defined an operational semantics for CGRS and proved the correctness of this semantics with respect to the abstract semantics. However, we have not yet developed a complete implementation of CGRS. In §10.7 we discuss some of our future objectives for implementing and testing CGRS.

1.3 Thesis structure

The thesis structure is as follows. Chapter 2 gives the basic definitions of graph transformation used in the rest of the thesis. Then the thesis is broken into three parts, reflecting our three areas of research.

Part II examines fast graph transformation and efficient recognition. Chapter 3 presents syntactic conditions ensuring fast execution of graph transformation rules, both individually and in sequence. Chapter 4 presents two classes of fast recognition systems with linear-time membership tests. It also presents several example recognition systems including systems recognising balanced binary trees and grid graphs. Chapter 5 compares our work to other approaches to fast graph transformation and efficient recognition.

Part III examines the correspondence between hyperedge-replacement grammars and separation logic. Chapter 6 introduces separation logic and hyperedge-replacement graph grammars, defines a semantics to our fragment of separation logic, and defines a translation between separation logic states and heap-graphs. Chapter 7 describes a translation from separation logic to hyperedge replacement and back, and proves that the translations are semantics-preserving. Chapter 8 proves that some constructs must be omitted from our fragment of separation logic and describes the consequences of our translation for both separation logic and hyperedge replacement.

In Part IV we describe CGRS, our language for shape safety. Chapter 9 gives the syntax and an informal semantics for CGRS. Examples are given of CGRS programs. Chapter 10 gives a concrete and an abstract semantics for CGRS constructs, and proves that the two correspond. The shape safety guarantees implied by this correspondence are described. Chapter 11 compares CGRS with other approaches to shape safety.

The thesis concludes with Chapter 12, which describes our major contributions and suggests possible areas of future work.

1.4 Publication history

Portions of the material on fast graph transformation and efficient recognition presented in Part II appear in my paper *Graph Transformation in Constant Time*, written with Detlef Plump. This paper presents the rooted syntactic conditions ensuring fast single-step derivations and efficient recognition. It does not include the work on left-connected derivation and recognition. This paper was presented to the 2006 International Conference on Graph Transformation [23].

A overview of the material presented in Part III is given in the extended abstract *From Separation Logic to Hyperedge Replacement and Back.* This short paper omits almost all of the technical detail given in this thesis, including the definitions of the translation functions and the proofs of correctness. It was presented to the Doctoral symposium at the 2008 International Conference on Graph Transformation [21]. A long version written with Detlef Plump will be published in the Doctoral Symposium proceedings [24].

An early version of the material presented in Part IV appears in my paper *Extending C for Shape Checking Safety*, written with Detlef Plump. This paper includes the syntax of our language CGRS, but omits most of the semantics and proofs of correctness. It was presented to the 2005 workshop on Graph Transformation for Verification and Concurrency [22].

Chapter 2

Preliminaries

This chapter defines the basic technical background used in the rest of the thesis.

2.1 Graphs and morphisms

In this section we define the notion of a graph, which is used in double-pushout graph transformation.

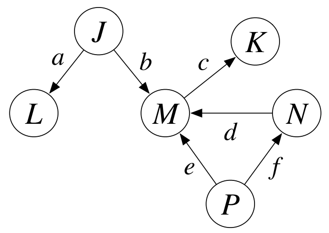
(In §2.3 we define the notion of a *hypergraph*, for use in hyperedge-replacement rewriting. In general a (labelled) graph is a (labelled) hypergraph with edges of arity exactly two. It would therefore be possible to define only the single notion of a hypergraph. However, the notations commonly used in double-pushout rewriting and hyperedge replacement research are quite different, and the two approaches are used in entirely disjoint sections of the thesis. For this reason, we define the two notions separately.)

Definition 2.1 (label alphabet)**.** A *label alphabet* is a pair $\mathcal{C} = \langle \mathcal{C}_V, \mathcal{C}_E \rangle$ of finite sets $\mathcal{C}_V$ and $\mathcal{C}_E$. The elements of $\mathcal{C}_V$ and $\mathcal{C}_E$ serve as node labels and edge labels, respectively. For this section and the next, we assume a fixed alphabet $\mathcal{C}$.

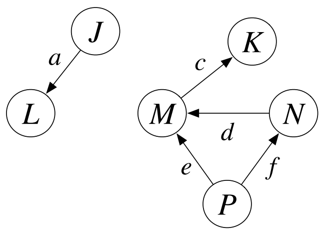
Definition 2.2 (graph)**.** A *graph* $G = \langle V_G, E_G, s_G, t_G, l_G, m_G \rangle$ over $\mathcal{C}$ consists of a finite set V_G of *nodes* (or *vertices*), a finite set E_G of *edges*, source and target functions $s_G, t_G \colon E_G \to V_G$, a partial node labelling function $l_G \colon V_G \to \mathcal{C}_V$ and an edge labelling function $m_G \colon E_G \to \mathcal{C}_E$. The *size* of G, denoted by $|G|$, is the number of nodes plus the number of edges. The *out-degree* of a node v, denoted by $\text{outdeg}_G(v)$, is the number of edges with source v, while the *in-degree*, denoted by $\text{indeg}_G(v)$ is the number of edges with target v. The *degree* of a node v, denoted $\text{deg}_G(v)$, is equal to $\text{indeg}_G(v) + \text{outdeg}_G(v)$. We write $\text{outlab}_G(v, l)$ for $\{e \in E_G \mid s_G(e) = v \wedge m_G(e) = l\}$, the set of edges with label l outgoing from v, and $\text{inlab}_G(v, l)$ for $\{e \in E_G \mid t_G(e) = v \wedge m_G(e) = l\}$. A graph H is a *subgraph* of G if $V_H \subseteq V_G$ and $E_H \subseteq V_G$ and $s_G = s_H$, $t_G = t_H$, $l_G = l_H$ and $m_G = m_H$ for the nodes and edges in H.

Definition 2.3 (graph properties)**.** A node v' is *reachable* from a node v if $v = v'$ or if there are edges $e_1, \ldots, e_n$ such that $s_G(e_1) = v$, $t_G(e_n) = v'$ and for $i = 1, \ldots, n-1$, $t_G(e_i) = s_G(e_{i+1})$. A graph is *reachable* from some node v if every node in V_G is reachable from v. A pair of nodes v and v' are *connected* if $v = v'$ or if v is reachable from v' or v' is reachable from v. A graph is *connected* if every pair of nodes $v, v' \in V_G$ are connected. A *connected component* is a maximal subgraph such that every pair of nodes in the subgraph are connected and no node outside the component is connected to a node in the component. We write $G \setminus l$ for the graph G with edge l removed. An edge l is *separating* in G if $G \setminus l$ has more connected components than G.

Example 2.1 (graph, graph properties)**.** The diagram given below shows a graph G with node-set $V_G = \{v_J, v_K, v_L, v_M, v_N, v_P\}$, edge-set $E_G = \{e_a, e_b, e_c, e_d, e_e, e_f\}$, and node and edge-labelling functions corresponding to the given node and edge subscripts.



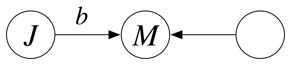
The node v_M has in-degree 3, out-degree 1, and degree 4. The node v_K is reachable from v_P, via the path e_f, e_d, e_c. The edge e_b is separating in G, because removing it results in a graph with two connected components.



The diagram above shows the graph G' resulting from the removal of the separating edge e_b from G'. This graph contains two connected components. The nodes v_M,v_K,v_N and v_P and their intersecting edges form one connected component. The nodes v_L and v_J form the other.

Definition 2.4 (graph morphism)**.** A *graph morphism* $g\colon G \to H$ between two graphs G and H consists of two functions $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ that preserve sources, targets and labels: $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, and $l_H(g_V(v)) = l_G(v)$ for all v in $\mathrm{dom}(l_G)$, and $m_H(g_E(e)) = m_G(e)$ for all $e \in E_G$. A morphism g is *injective* (*surjective*) if g_V and g_E are injective (surjective); it *preserves undefinedness* if $l_H(g(v)) = \perp$ for all v in $V_G - \mathrm{dom}(l_G)$. Morphism g is an *isomorphism* if it is injective, surjective and preserves undefinedness. In this case G and H are *isomorphic*, which is denoted by $G \cong H$. Furthermore, g is an *inclusion* if $g(x) = x$ for all nodes and edges x in G. (Note that inclusions need not preserve undefinedness.) A *partial graph morphism* $h\colon G \rightharpoonup H$ is a graph morphism from some subgraph of G to H.

Example 2.2 (graph morphism)**.** Suppose we have the following small graph G'' with node set $V_{G''} = \{v_{J'}, v_{M'}, v_\perp\}$, edge set $E_{G''} = \{e_{b'}, e_\perp\}$, and the labelling functions corresponding to the node and edge subscripts.



Then the functions $g_V = \{v_{J'} \mapsto v_{J'}, v_{M'} \mapsto v_{M'}, v_\perp \mapsto v_\perp\}$, $g_E = \{e_{b'} \mapsto e_{b'}, e_{e'} \mapsto e_\perp\}$ form an isomorphism (injective, surjective and preserving undefinedness) between G'' and itself.

Now consider the graph G defined in Example 2.1. The functions $g'_V = \{v_{J'} \mapsto v_J, v_{M'} \mapsto v_M, v_\perp \mapsto v_P\}$, $g'_E = \{e_{b'} \mapsto e_b, e_\perp \mapsto e_e\}$ form an injective morphism between G'' and G. The pair of functions $g''_V = \{v_{J'} \mapsto v_J, v_{M'} \mapsto v_M, v_\perp \mapsto v_J\}$, $g''_E = \{e_{b'} \mapsto e_b, e_{e'} \mapsto e_b\}$ form a non-injective morphism from G'' to G, as $g''_V(v_{J'}) = g''_V(v_\perp)$ (meaning $v_{J'}$ is shared).

$$\begin{tikzcd}
Z \arrow[r, "g"] \arrow[d, "f"'] & Y \arrow[d, "i_1"] \\
X \arrow[r, "i_2"'] & P
\end{tikzcd}
\qquad
\begin{tikzcd}
Z \arrow[r, "g"] \arrow[d, "f"'] & Y \arrow[d, "i_1"'] \arrow[ddr, "j_2"] & \\
X \arrow[r, "i_2"] \arrow[drr, "j_1"'] & P \arrow[dr, dashed, "u"] & \\
& & Q
\end{tikzcd}$$

Figure 2.1: Commutative diagrams defining a pushout.

$$\begin{tikzcd}
P \arrow[r, "p_2"] \arrow[d, "p_1"'] & Y \arrow[d, "f"] \\
X \arrow[r, "g"'] & Z
\end{tikzcd}
\qquad
\begin{tikzcd}
Q \arrow[drr, "q_2"] \arrow[ddr, "q_1"'] \arrow[dr, dashed, "u"] & & \\
& P \arrow[r, "p_2"'] \arrow[d, "p_1"] & Y \arrow[d, "f"] \\
& X \arrow[r, "g"'] & Z
\end{tikzcd}$$

Figure 2.2: Commutative diagrams defining pullback.

Definition 2.5 (pushout, pullback)**.** The *pushout* of two graph morphisms $f\colon Z \to X$ and $g\colon Z \to Y$ with a common domain consists of a graph P and two morphisms $i_1\colon X \to P$ and $i_2\colon Y \to P$ for which the left-hand diagram in Figure 2.1 commutes. It must be true of the pushout (P, i_1, i_2) that for any other such triple (Q, j_1, j_2) there exists a unique morphism $u\colon P \to Q$ for which the right-hand side diagram in Figure 2.1 commutes.

A *pullback* is the inverse notion to a pushout. The pullback of $f\colon G \to Z$ and $g\colon Y \to Z$ consists of a graph P and two morphisms $p_1\colon P \to X$ and $p_2\colon P \to Y$ for which the left-hand diagram in Figure 2.2 commutes. It must be true of the pullback (P, p_1, p_2) that for any other such triple (Q, q_1, q_2) there exists a unique morphism $u\colon Q \to P$ for which the right-hand side diagram in Figure 2.2 commutes. We call a pushout *natural* if it is also a pullback.

2.2 Double-pushout graph rewriting

In this section we review basic notions of the double-pushout approach to graph transformation, using a version that allows unlabelled nodes in rules [43].

Definition 2.6 (rule, direct derivation)**.** A *rule* $r = \langle L \leftarrow K \rightarrow R \rangle$ consists

$$\begin{tikzcd}
L \arrow[d, "g"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "b"'] \arrow[r] \arrow[d, "d"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "k"] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Figure 2.3: Two natural pushouts defining a direct derivation from graph G to graph H.

of two inclusions $K \to L$ and $K \to R$ such that (1) for all $v \in V_L$, $l_L(v) = \bot$ implies $v \in V_K$ and $l_R(v) = \bot$, and (2) for all $v \in V_R$, $l_R(v) = \bot$ implies $v \in V_K$ and $l_L(v) = \bot$. L is the *left-hand side*, R the *right-hand side* and K the *interface* of r. The *size* of a rule r, denoted by $|r|$, is equal to $\max(|L|, |R|)$. The *inverse* of a rule is obtained by swapping left- and right hand sides together with the inclusion morphisms.

Definition 2.7 (derivation)**.** A *direct derivation* from a graph G to a graph H via a rule $r = \langle L \leftarrow K \rightarrow R \rangle$, denoted by $G \Rightarrow_{r,g} H$ or just $G \Rightarrow_r H$, consists of two natural pushouts, arranged as shown in Figure 2.3, where $g\colon L \to G$ is injective.

A direct derivation via a set of rules $\mathcal{R}$, denoted by $G \Rightarrow_{\mathcal{R}} H$, is defined if for any rule r in $\mathcal{R}$ there exists a direct derivation $G \Rightarrow_r H$. If no direct derivation $G \Rightarrow_r H$ exists, then G is *irreducible* with respect to $\mathcal{R}$. A *derivation* $G \Rightarrow^*_{\mathcal{R}} H$ is defined if there exists a sequence of derivations $G \cong I_0 \Rightarrow_{\mathcal{R}} I_1 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} I_n \cong H$.

Morphism g in Figure 2.3 is called the *matching morphism* while k is called the *result morphism.* The *track-morphism* for a direct derivation $G \Rightarrow H$, denoted $tr_{G \Rightarrow H}$ is the partial morphism between the elements of G and H that are preserved by the derivation.

Lemma 2.1 (proved in [43]) gives a characterisation of the natural pushouts in a direct derivation.

Lemma 2.1 (characterisation of natural pushouts [43])**.** *Given two graph morphisms $b\colon K \to L$ and $d : K \to D$ such that b is injective, pushout (1) in Figure 2.3 is natural if and only if for all $z \in K$, $l_K(z) = \bot$ implies $l_L(b(z)) = \bot$ or $l_D(d(z)) = \bot$.*

Definition 2.8. dangling condition In [43] it is shown that for a given rule r and injective morphism g, there exists such a direct derivation if and only if

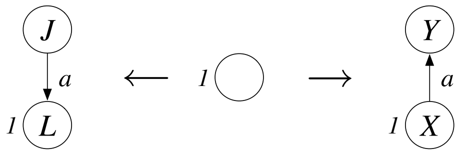
g satisfies the *dangling condition*: no node in $g(L) - g(K)$ must be incident to an edge in $G - g(L)$.

If the dangling condition is satisfied, then r and g determine D and H uniquely up to isomorphism and H can be constructed (up to isomorphism) from G as follows: (1) Remove all nodes and edges in $g(L)-g(K)$, obtaining a subgraph D'. (2) Add disjointly to D' all nodes and edges from $R - K$, keeping their labels. For $e \in E_R - E_K$, $s_H(e)$ is $s_R(e)$ if $s_R(e) \in V_R - V_K$, otherwise $g_V(s_R(e))$. Targets are defined analogously. (3) For each node $g_V(v)$ in $g(K)$ with $l_L(v) \neq l_R(v)$, $l_H(g_V(v))$ becomes $l_R(v)$.

Note that in the construction, D differs from D' in that nodes are unlabelled if they are the images of unlabelled nodes in K that are labelled in L. We do not need D to transform G into H though.

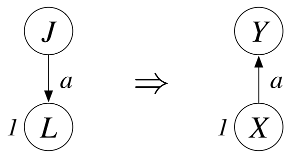
Rules with unlabelled nodes in the interface graph allow the relabelling of nodes. In addition, rules with unlabelled nodes in their left and right-hand sides represent sets of totally labelled rules because unlabelled nodes in the left-hand side can act as placeholders for arbitrarily labelled nodes.

Example 2.3 (rule, derivation)**.** The following pair of morphisms define a rule r that matches a pair of nodes labelled J and L, removes the J-labelled node and constructs a new Y-labelled node with an attached a-labelled edge.

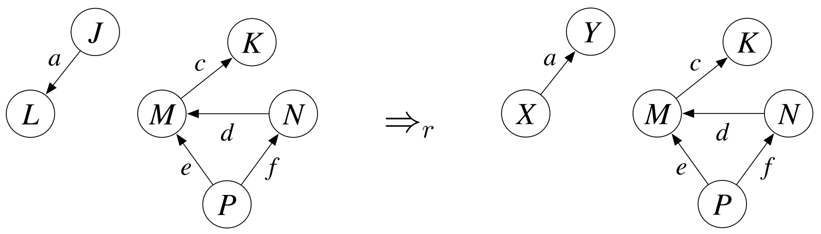


Note that in all three graphs one of the nodes have associated subscript '1'. The morphisms map the 1-subscripted node in the interface graph to the 1-subscripted nodes in the left and right-hand sides. As the J-labelled node on the left-hand side does not have a corresponding node in the interface, it will be deleted by the rule. For the same reason, a new Y-labelled node will be constructed by the rule.

We often omit the interface graph, and define it by implication using node subscripts. The diagram given below shows the same rule written using this notation.



Applying this rule to the graph G' defined in Example 2.1 gives the following direct derivation.



This rule cannot apply to the graph G defined in Example 2.1, even though there exists a matching morphism between the left-hand side of the rule and the graph. This is because the match would remove the J-labelled node, leaving dangling edges and so violating the dangling condition.

Definition 2.9 (graph class, $\mathbb{C}$-preserving)**.** To keep this definition independent of any type system imposed on graphs, we introduce abstract graph classes and rules preserving such classes. A *graph class* over a label alphabet $\mathcal{C}$ is a set $\mathbb{C}$ of graphs over $\mathcal{C}$. A rule r is $\mathbb{C}$-*preserving* if for every direct derivation $G \Rightarrow H$, $G \in \mathbb{C}$ implies $H \in \mathbb{C}$.

2.3 Hyperedge-replacement graph rewriting

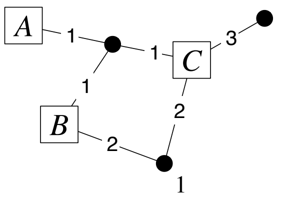
In this section we review the basic notions of the hyperedge-replacement approach to graph transformation. This section is based on the definitions given in [27, 38].

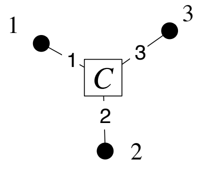
For this section we assume we have a fixed label alphabet C. We also assume a fixed *arity function* $\mathrm{ari}\colon C \to \mathbb{N}$.

Definition 2.10 (hypergraph)**.** A *hypergraph* H over C and ari is a tuple $H = \langle V, E, att, l, ext\rangle$. V and E are, respectively, finite sets of *vertices* (or *nodes*) and *hyperedges* (often just referred to as edges). $l\colon E \to C$ assigns an edge label to each edges. $att\colon E \to V^*$ assigns to edges a sequence of *attachment* nodes, with $|att(e)| = \mathrm{ari}(l(e))$ for all $e \in E$. The first element of $att(e)$ is the *source* of hyperedge e, if it exists. We denote by $att(e)[i]$ the ith attachment point of e. $ext \in V^*$ defines a sequence of pairwise-distinct *external* nodes. A hypergraph H such that $|ext_H| = n$ is an n-hypergraph.

Given a hypergraph H and set $X \subseteq C$ of labels we denote by E^H_X 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 $\mathcal{H}_C$. A graph $H \in \mathcal{H}_C$ is said to be a *singleton* if the members of V_H are all members of 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 $A^\bullet$.

Example 2.4 (hypergraph)**.** The left diagram below shows a hypergraph H over label-set $\{A, B, C\}$ and arity function $\{A \mapsto 1, B \mapsto 2, C \mapsto 3\}$.





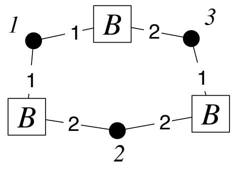
This hypergraph has three vertices and four nodes. The second attachment point of the B and C-labelled edges is the only external node, represented by the subscript 1. The source (and sole attachment point) of the A-labelled hyperedge is also the source for the B and C-labelled edges.

The right diagram shows a singleton hypergraph H'. This hypergraph is also the handle $C^\bullet$, because the external nodes are the same as the list of attachment points for the single C-labelled edge.

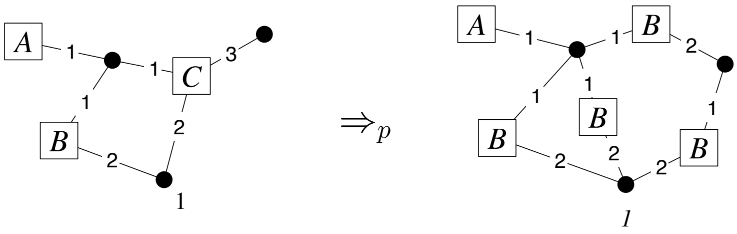
Definition 2.11 (hyperedge replacement)**.** Let $H \in \mathcal{H}_C$ be a hypergraph, $B \subseteq E_H$ be a set of hyperedges to be replaced. Let $repl\colon B \to \mathcal{H}_C$ be a mapping with $|ext_{repl(e)}| = \text{ari}(e)$ for all $e \in B$. Then the *replacement* of B in H by *repl* yields the hypergraph $H[repl]$ by removing B from E_H, adding the nodes and hyperedges of $repl(e)$ for each $e \in B$ disjointly, and fusing the i-th external node of $repl(e)$ with the i-th attachment node of e for each $e \in B$ and $i = 1, \ldots, \text{ari}(e)$. Note that the replacement involves merging nodes in the hypergraph $repl(e)$ if $att(e)$ contains repeated nodes. If $B = \{e_1, \ldots, e_n\}$ and $repl(e_i) = R_i$ for $i = 1, \ldots, n$ then we also write $H[e_1/R_1, \ldots, e_n/R_n]$ instead of $H[repl]$.

Let $N \subseteq C$ be a set of *non-terminals.* A *production* over N is an ordered pair $p = (A, R)$ with $A \in N$, $R \in \mathcal{H}_C$, and $|ext_A| = |ext_R|$. Let H be a graph in $\mathcal{H}_C$ and let P be a set of productions. Let $e \in E_H$ and $(lab_H(e), R) \in P$. Then H *directly derives* H', denoted by $H \Rightarrow_P H'$ if $H' \cong H[e/R]$. A direct derivation via a set of productions P, denoted by $G \Rightarrow_P H$, is defined if for any production p in P there exists a direct derivation $G \Rightarrow_p H$. A *derivation* $G \Rightarrow_P^* H$ is defined if there exists a sequence of derivations $G \cong I_0 \Rightarrow_P I_1 \Rightarrow_P \ldots \Rightarrow_P I_n \cong H$.

Example 2.5 (hyperedge replacement)**.** Suppose we have the set of non-terminals $N = \{A, C\}$, and production $p = (C, R)$, where R is the following graph.



Then applying p to the graph H defined in Example 2.4 gives the following derivation.

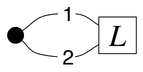


Definition 2.12 (hyperedge replacement grammar)**.** A *hyperedge-replacement grammar* (or *HR grammar*) G over C is a tuple $G = \langle T, N, P, Z\rangle$ where $T \subseteq C$ and $N \subseteq C$ are sets of terminal and non-terminal symbols respectively. P is a set of productions over N. Z is the set of *initial* graphs. The *language* of the grammar, written $\mathrm{L}(G)$ is the set of all graphs $H \in \mathcal{H}_T$ such that there exists a derivation $I \Rightarrow_P^* H$ for some $I \in Z$. The *language family* $L\colon N \mapsto \mathcal{H}_T$ generated by G is given by $L(A) = \{G \in \mathcal{H}_T \mid A^\bullet \Rightarrow_P^* H\}$.

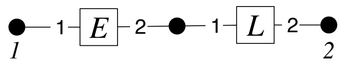
Note that in this document we define hyperedge replacement grammars with a finite initial *set* of graphs. In contrast, the standard definition featured in e.g. [27, 38] (and many others) has a *single* initial graph. This change to the definition makes no difference to the theoretical properties of hyperedge-replacement grammars, because each definition can be simulated by the other. A set of initial graphs $\{G_1, \ldots, G_n\}$ can be trivially simulated in the single-graph definition by replacing the initial set with a graph containing a single I-labelled hyperedge of arity 0 (where I does not occur in the given grammar), and adding n productions (I, G_i) for $1 \leq i \leq n$.

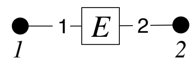
Example 2.6 (hyperedge replacement grammar)**.** Suppose we have a set of terminal labels $T_{\mathrm{CL}} = \{L, E\}$, arity function $\mathrm{ari}_{\mathrm{CL}} = \{L \mapsto 2, E \mapsto 2\}$, and set of nonterminal symbols $N_{\mathrm{CL}} = \{L\}$. Then we can define a hyperedge replacement grammar $\langle T_{\mathrm{CL}}, N_{\mathrm{CL}}, P_{\mathrm{CL}}, Z_{\mathrm{CL}} \rangle$ which generates the language of cyclic E-labelled lists.

This grammar has a single graph in its initial set. The initial graph, shown below, consists of a single L-labelled edge with both attachment points attached to the same node.

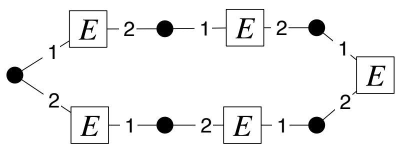


The grammar has two productions (L, R_1) and (L, R_2). The production based on graph R_1, shown on the left below, produces an L-labelled edge and an E-labelled terminal edge. The production based on R_2, shown on the right, terminates the derivation by producing only a single terminal edge.





The language of graphs for this grammar is the class of all E-labelled cyclic lists. For example, the following hypergraph is a terminal member of the language containing five edges.



2.4 Graph signatures

All of the notation given in previous sections is standard in the literature on graph transformation. We now define the notion of a *graph signature*, which is not standard. A graph signature defines a class of conformant graphs that restrict the number of edges with a particular label that can be attached to a node with a particular label.

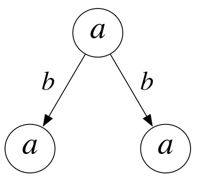
Signatures are used in later chapters to conceptually separate local patterns of node and edge attachment from more complex language membership questions.

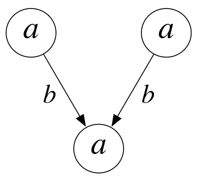
Definition 2.13 (graph signature)**.** A *graph signature* $\Sigma = \langle \mathrm{L}, in, out \rangle$ consists of a label alphabet $\mathrm{L} = \langle \mathrm{L}_V, \mathrm{L}_E \rangle$ and a pair of partial functions $in, out\colon \mathrm{L}_V \times \mathrm{L}_E \to \mathbb{N}$. A graph G over $\mathcal{C}$ is a Σ*-graph* (or *conforms to* Σ) if for every node v and edge label k, (1) if $out(l_G(v), k)$ is defined, then there are at most $out(l_G(v), k)$ edges with source v and label k, and (2) if $in(l_G(v), k)$ is defined, then there are at most $in(l_G(v), k)$ edges with target v and label k.

A graph is Σ*-total* if for every node v and edge label k, (1) if $out(l_G(v), k)$ is defined, then there are *exactly* $out(l_G(v), k)$ edges with source v and label k, and (2) if $in(l_G(v), k)$ is defined, then there are *exactly* $in(l_G(v), k)$ edges with target v and label k.

Note that no restrictions are placed on the number of incoming and outgoing edges with a particular label if in or out are equal to $\perp$ (undefined) for the node and edge label.

Example 2.7 (graph signature)**.** Let Σ be a signature $\langle \mathrm{L}, in, out \rangle$ such that $\mathrm{L} = \langle \{a\}, \{b\} \rangle$, $in(a, b) = 1$ and $out(a, b) = \perp$.





The left-hand graph above conforms to the signature, because the in-degree of every a-labelled node does not exceed 1. The out-degree is unbounded, as $out(a, b) = \perp$. However, the right-hand graph above does not conform to the signature, because the in-degree of the lowest node is 2, which is greater than $in(a, b)$.

The notion of a graph signature defined here is similar to the widely-used notion of a multiplicity label. This notion is used in type graphs [73, 68], and in UML class diagrams [62]. A signature Σ where $out(l) = n$ corresponds to a multiplicity of $0 \dots n$ on an outgoing edge from an l-labelled node. If $out(l) = \perp$, then the corresponding multiplicity would be $*$. The same is true for in.

However, our notion of a signature is designed to be an entirely local, syntactic property. In contrast, type graphs and class diagrams use multiplicity annotations as part of much more expressive systems for specifying larger-scale graph properties. Type graphs and class diagrams specify the permitted relationships *between* nodes. In contrast, our notion of a signature only specifies the permitted multiplicities of node and edge combinations, and places no other restriction on the structure of a graph.

Proposition 2.2 (signature conformance complexity)**.** *Checking whether a graph G is a Σ-graph requires at worst time $O(|G|)$.*

Proof. Each node v must be checked for signature conformance. The node label $l_V(v)$ and the set of labels l of edges attached to v are first checked to confirm that they are in $\mathcal{L}$. Under our general assumption about the time complexity of accessing information about a graph (Assumption 3.1 in §3.1) given a particular edge label l and node v, the number of outgoing edges from v with label l can be retrieved in time $O(1)$. This value can then be compared with the signature-value $out(l_v(v), l)$ to check the signature holds for this particular pair. The same $O(1)$ bound holds for incoming edges with label l. Our graphs have fixed, finite label-sets, so for any node v the signature can be checked for *all* edge labels in time $O(1)$. To check signature conformance for the whole graph requires that each node is checked in this way, so the overall time complexity is $O(|G|)$. □

We now define a sufficient syntactic condition on double-pushout graph transformation rules ensuring that they preserve conformance to a signature.

Definition 2.14 (Σ-rule)**.** A rule $r = \langle L \leftarrow K \rightarrow R\rangle$ is a Σ-*rule* if L, K and R are Σ-graphs and for each node v in K and node label $l \in \mathcal{C}_V$,[1]

(1a) $l_L(v) = \bot = l_R(v)$ implies $|\text{outlab}_L(v, l)| \geq |\text{outlab}_R(v, l)|$.

(1b) $l_L(v) = \bot = l_R(v)$ implies $|\text{inlab}_L(v, l)| \geq |\text{inlab}_R(v, l)|$.

(2a) $l_L(v) \neq \bot$ implies $out(l_L(v), l) - |\text{outlab}_L(v, l)| + |\text{outlab}_R(v, l)| \leq out(l_R(v), l)$.

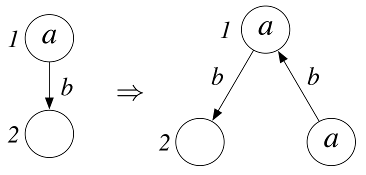
(2b) $l_L(v) \neq \bot$ implies $in(l_L(v), l) - |\text{inlab}_L(v, l)| + |\text{inlab}_R(v, l)| \leq in(l_R(v), l)$.

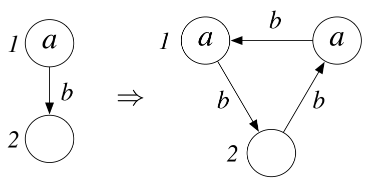
These two conditions ensure that Σ-graphs are preserved by Σ-rules. Conditions (1a) and (1b) ensure that number of outgoing and incoming edges with a particular label cannot increase if the node's label is not fixed by the

[1]To simplify notation, we let the node v stand for its image in the left and right-hand side graphs in some cases. Which is meant will be obvious from the domains of the functions.

rule. Conditions (2a) and (2b) ensure that the number of edges added by the right-hand side combined with the number already in the graph cannot exceed the maximum number mandated by the signature.

Example 2.8 (Σ-rule)**.** Let Σ be a signature $\langle L, in, out\rangle$ such that $L = \langle\{a\}, \{b\}\rangle$, $in(a, b) = 1$ and $out(a, b) = 3$.





The left-hand side rule given above is a Σ-rule, as the new edges respect the signature for the labelled nodes, and do not intersect with the unlabelled node. The right-hand rule is not a Σ-rule, because a new edge is created intersecting an unlabelled node, and because the edges intersecting with the labelled node may violate the signature.

Proposition 2.3 (Σ-rules preserve Σ-graphs)**.** *Let $G \Rightarrow_r H$ be a derivation such that G is a Σ-graph and r a Σ-rule. Then H is a Σ-graph.*

Proof. Consider a node v in V_H and label $l \in \mathcal{L}_E$. If v was not matched by r then the number of l-labelled nodes incoming and outgoing from v are unchanged from G and by assumption the node must conform to the signature. If v was matched by an unlabelled node in r, then by conditions (1a) and (1b) the derivation can only decrease the number of incoming and outgoing edges labelled l, so signature conformance is preserved from G. If v was matched by a labelled node, then conditions (2a) and (2b) mean arithmetically that the increase in the number of attached incoming and outgoing l-labelled edges cannot exceed the maximum number of edges mandated by the signature. $\square$

Part II

Fast graph transformation and recognition

Chapter 3

Fast graph transformation

Graph transformation rules under the double-pushout approach are a powerful mechanism for formulating graph rewriting systems. However, a major obstacle to using these rules in practical computation systems is the time required for rule application. Finding a match for a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ in a graph G requires at worst time $O(|G|^{|L|})$. This is too expensive for some applications, even if r is fixed (meaning that $|L|$ is constant).

For example, in Part IV we propose the language CGRS that extends the C programming language with graph-transformation constructs to enable the safe manipulation of pointers. We argue in §9.2.1 that to make such a language acceptable for programmers, individual rule constructs must be applicable in constant time.

Other applications would also benefit from faster graph transformation. Languages based on graph-transformation, such as the GP language [65, 57], and the GROOVE tool [66, 67], are based on the sequential application of graph-transformation rules. For programs written in such languages to execute in a reasonable amount of time, individual rules must apply quickly.

Constant-time rule application is achieved in CGRS by using a restricted form of graph transformation characterised by (1) a requirement that all nodes in the host-graph have distinctly-labelled out-edges, and (2) the presence of uniquely-labelled *root nodes* in rules and host graphs. These roots serve as entry points for the matching algorithm and ensure, under further assumptions on left-hand sides and host graphs, that all matches of a rule can be found in time independent of the size of the host graph.

In this chapter we take the CGRS approach and present a more general theory of fast graph transformation in the setting of the double-pushout approach, encompassing both linear-time and constant-time rewriting. The rooted graph transformation used in CGRS is included in this general theory as a special case. Our approach is based on syntactic restriction of rules and graphs that result in an improved worst-case application time.

The structure of this chapter is as follows. Section 3.1 defines the basic concepts used in the rest of the chapter, and defines the fundamental problems of graph matching and graph transformation that are solved by our algorithms. Section 3.2 presents a characterisation of so-called left-connected graph transformation rules, and gives an algorithm for applying these rules. The semantic conditions under which these rules apply in linear time are characterised, and syntactic conditions ensuring these properties hold are given. Section 3.3 presents so-called rooted graph transformation rules, and defines the semantic conditions under which such rules can be applied in constant time. Once again we give syntactic conditions that guarantee that these semantic conditions hold. Finally, Section 3.4 defines the problem of multi-step graph transformation, that is, the problem of applying a set of rules in a sequence of rewrites. This section presents conditions on graph transformation systems ensuring that multi-step rule-applications of bounded length terminate in linear time.

3.1 The problems of graph transformation

This section defines formally the basic problems of graph transformation. Here we look at the problem of applying a single rule once only; in §3.4 we extend this to look at applying sets of rules and applying rules in sequence.

Assumption 3.1. We assume in the rest of this thesis that graphs are stored in a format such that the time complexities of various problems are as given in the table below.

Input	Output	Time
label l	Set Z of nodes with label l.	$O(\|Z\|)$
node v	Values $\deg(v)$, $\mathrm{indeg}(v)$, $\mathrm{outdeg}(v)$.	$O(1)$
node v, label l	No. nodes with source v and label l.	$O(1)$
node v, label l	No. nodes with target v and label l.	$O(1)$
node v, label l	Set Z of nodes with source v and label l.	$O(\|Z\|)$
node v, label l	Set Z of nodes with target v and label l.	$O(\|Z\|)$
graph G	$\|V_G\|$ and $\|E_G\|$	$O(1)$

These assumptions are satisfied by a graph structure where each node stores a set of references to incoming and outgoing edges for each edge label, and also records the size of each of these sets.

Constructing a matching morphism for a rule $r = \langle L \leftarrow K \rightarrow R\rangle$ into a graph G requires at worst time $O(|G|^{|L|})$. This time bound is achieved by even the simple algorithm that constructs a partial matching morphism of increasing size by adding nodes and edges in arbitrary order, then backtracks when no extension is possible. As there are at most $|G|^{|L|}$ distinct total morphisms from L to G, the size of the search-tree of partial morphisms is similarly bounded.

In general, we consider the rule for which we are constructing a derivation fixed, rather than as an input to the problem. This fixed-rule viewpoint fits well with the application of this work to graph recognition and programming by graph transformation. In both of these applications we have a rule-set of fixed size from which rules are applied to variable-size graphs, which can be arbitrarily large. This viewpoint also fits with the standard view of computational complexity for computer programs: programs are considered fixed, and data-structures are of variable size.

A consequence of the fixed-rule viewpoint is that the size of the left-hand side is also fixed, and so the time complexity $O(|G|^{|L|})$ is a *polynomial.*[1] The exponent may however be very large, depending on the size of the matched rule.

The time complexity of applying a rule $\langle L \leftarrow K \rightarrow R\rangle$ to a graph G is dominated by the cost of finding the matching graph morphisms $L \rightarrow G$.

[1]If we consider both the left-hand side L and host-graph G as part of the input, the problem of constructing a matching morphism becomes the *subgraph isomorphism problem.* This is the problem to decide whether there exists an injection from L to G. This problem is known to be NP-complete [35]; In the worst case, if there is no subgraph isomorphism, solving it is as expensive as finding all solutions.

This is because for each of these morphisms, checking the dangling condition and transforming G into H can be done in time independent of the size of G (under the assumptions about graph representation given in Assumption 3.1). We isolate the cost of matching by examining first the rule application problem.

Rule Application Problem (RAP).

Given:	A graph class $\mathbb{C}$ and a $\mathbb{C}$-preserving rule $r = \langle L \leftarrow K \rightarrow R \rangle$.
Input:	A graph $G \in \mathbb{C}$ and an injective morphism $g \colon L \rightarrow G$.
Output:	Either a graph H such that $G \Rightarrow_{r,g} H$, or if no such graph exists, **fail**.

We have factored out the dangling condition from the rule application problem, meaning that the dangling condition must be checked for a complete morphism after matching. An alternative algorithm would be to check the dangling condition for each node incrementally at the point it was added to the matching morphism. Such an incremental algorithm would result in an improved application time in many cases by ruling out candidate matches earlier. This improvement makes no difference to the worst-case application time, however.

We have chosen to factor out the dangling condition despite this potential inefficiency in order to clearly separate concerns between matching and all other aspects of rule application. By focusing on matching exclusively we can more clearly explain our solutions to the high cost of graph transformation. In addition, the algorithms we give below can be simply modified to include incremental checking of the dangling condition without disturbing our improved time complexity results. Finally, an incremental algorithm may give a worse application time than the staged algorithm in cases with many candidate but few complete matches.

To solve the rule application problem we define the pair of auxiliary algorithms Dangle and Apply. These algorithms are used in the definition of several of the later algorithms of this chapter.

Algorithm 3.1 (Dangle)**.** The algorithm Dangle(G, r, h) takes as its argument a graph G, a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ and a candidate morphism $g : L \rightarrow G$. It returns **pass** if g satisfies the dangling condition, or **fail** otherwise. Dangle is implemented by degree comparison. The dangling condition holds if and only if for all nodes v in $V_L - V_K$, $\deg_L(v) = \deg_G(h(v))$.

The dangling condition can therefore be checked by comparing $\deg_L(v)$ with $\deg_G(h(v))$ for all nodes v in $V_L - V_K$.

Algorithm 3.2 (Apply)**.** The algorithm Apply(G, r, h) takes as its arguments a graph $G \in \mathbb{C}$, $\mathbb{C}$-preserving rule $r = \langle L \leftarrow K \rightarrow R\rangle$ and injection $g : L \rightarrow G$ satisfying the dangling condition. The algorithm constructs the unique (up to isomorphism) double pushout required for application of r, as given by the following diagram. It returns the uniquely-determined resulting graph H.

$$\begin{tikzcd}
L \arrow[d, "g"'] & K \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

The algorithm constructs the double pushout for the derivation $G \Rightarrow_{g,r} H$ by first constructing the intermediate graph D, by deleting nodes matched to $L - K$ in the morphism, and removing the labels of nodes that are the images of nodes unlabelled in K. Edges not present in K are also removed. The algorithm then constructs the result graph H. New nodes are constructed, labels are inserted, and new edges are added, so that the elements added to D correspond to the graph $R - K$.

Proposition 3.1 (complexity of RAP)**.** *The rule application problem for a rule $r = \langle L \leftarrow K \rightarrow R\rangle$ can be solved in time $O(|r|)$.*

Proof. The rule application problem can be solved using the algorithms Dangle and Apply. Given an injection $h \colon L \rightarrow G$, checking the dangling condition requires $O(|V_L|)$ comparisons of node degrees. We have assumed a graph representation such that the degree of any node can be retrieved in constant time, so all of the degree comparisons required by a single run of Dangle can be checked in time $O(|V_L|)$.

Apply must terminate in time $O(|L| - |K| + |R| - |K| + |V_K|)$. Given a morphism g, nodes and edges can be removed in time $O(|L| - |K|)$. New nodes and edges can be inserted in time $O(|R| - |K|)$. Relabelling nodes requires time $O(|V_K|)$ at most.

Thus we obtain the time bound for the RAP of $O(|V_L| + |L| - |K| + |R| - |K| + |V_K|)$. As $|V_L|$, $|L|$, $|R|$, $|K|$ and $|V_K|$ are all bounded by $|r|$, the problem complexity can be estimated as $O(|r|)$. □

Having solved the rule application problem, this now leads us to the core problem: the construction of matching morphisms.

Graph Matching Problem (GMP).

Given:	A graph class $\mathbb{C}$ and a $\mathbb{C}$-preserving rule $r = \langle L \leftarrow K \rightarrow R \rangle$.
Input:	A graph G in $\mathbb{C}$.
Output:	The set $\{g \colon L \rightarrow G \mid g \text{ is injective}\}$.

As discussed above, on page 38, solving the graph matching problem requires time $O(|G|^{|L|})$ in the worst case – better algorithms are not known. In the following sections we examine syntactic restrictions on graph transformation that improve this worst-case performance.

3.2 Fast left-connected graph transformation

This section introduces conditions sufficient to ensure that rules can be executed in linear time (with fixed rules, as under the graph matching problem). Intuitively, for each node in the target graph, these conditions place a constant bound on the number of potential matches which include this node. Because the number of nodes in the graph is bounded by the size of the graph, this results in a search space that grows linearly.

Definition 3.1 (left-connected rule)**.** A graph transformation rule $r = \langle L \leftarrow K \rightarrow R \rangle$ is *left-connected* if the left-hand side graph L is connected.

Assumption 3.2. For the rest of this section, let $\mathbb{C}$ be a graph class and $r = \langle L \leftarrow K \rightarrow R \rangle$ a fixed $\mathbb{C}$-preserving left-connected rule. Let n always refer to the number of edges in L.

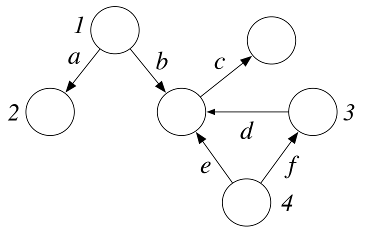
Our matching algorithm for left-connected rules is based on the auxiliary algorithm Search, which takes a node s in the left-hand side of a rule and a node v in the target graph and constructs all matching morphisms that map s to v.

The algorithm starts with the set A_0 consisting of all partial injections that associate only s to v. Each iteration of the algorithm then extends the set of injections in the previous working set with a single edge and its target node, or a single edge and its source node, or a single edge only. Injections that cannot be extended are removed. The algorithm terminates when all of the remaining injections in the set are total, or the set of injections is empty. When an iteration of the algorithm adds some left-hand side node or edge to the domain of an injection, we speak of the node or edge being *matched.*

A pre-processed enumeration of the edges of the left-hand side ensures the algorithm matches edges in order, so that when an edge is matched, its source or target must have been matched in some previous iteration.

Definition 3.2 (edge enumeration)**.** An edge sequence $e_1, \dots, e_n$ is an *edge enumeration* of L if $E_L = \{e_1, \dots, e_n\}$ and for $i = 2, \dots, n$, e_i is incident with the source or target of some edge in $e_1, \dots, e_{i-1}$. The ith member of enumeration E_L is referred to by $E[i]$. A node is an *initial node* of the enumeration if it is the source or target of edge e_1.

Example 3.1 (edge enumeration)**.** Suppose we have the following graph G. In this graph labels are used to uniquely identify graph edges.



The sequence $[a, b, c, e, d, f]$ is a valid edge enumeration for this graph, with initial nodes 1 and 2. The sequence $[f, d, c, b, a, e]$ is also valid, with initial nodes 3 and 4. However the sequence $[a, c, b, e, d, f]$ is not valid, as edge c is included before one of its incident nodes has been incident with an earlier edge. The sequence $[a, b, d]$ is also not valid, because it omits some of the graph's edges.

Every connected graph L possesses at least one edge enumeration. An enumeration can be constructed by picking an arbitrary start edge, and then picking edges non-deterministically from the set of edges adjacent to those already in the enumeration. The connectedness of L ensures that all edges must be either attached to an initial node or adjacent to an edge earlier in the enumeration.

Definition 3.3 (morphism extension)**.** Given partial morphisms $g, g' : G \rightharpoonup H$ and edge $e \in E_G$, we write $g >_e g'$ (pronounced 'g extends g' by e'), if $e \notin \text{dom}(h'_E)$ and $\text{dom}(h_E) = \text{dom}(h'_E) \cup \{e\}$ and $\text{dom}(h_V) = \text{dom}(h_V) \cup$

$\{s_G(e), t_G(e)\}$, and for all $v \in \mathrm{dom}(h'_V)$, $h(v) = h'(v)$, and for all $e \in \mathrm{dom}(h'_E)$, $h(e) = h'(e)$.

Example 3.2 (morphism extension)**.** Consider the graph G in Example 3.1. Let v_1 and v_2 be the nodes with tags 1 and 2, and let e_a be the a-labelled edge. Suppose we have a partial morphism $h\colon G \to G$ such that $h_V = \{v_1 \mapsto v_1\}$ and $h_E = \emptyset$. Then the morphism h' extends h by e_a if $h'_V = \{v_1 \mapsto v_1, v_2 \mapsto v_2\}$ and $h'_E = \{e_a \mapsto e_a\}$.

Algorithm 3.3 (Search)**.** The algorithm Search(j, v, G, E) assumes the fixed rule r and it takes as its inputs a left-hand side start node $j \in V_L$, and a target graph start node $v \in V_G$, a graph $G \in \mathbb{C}$, and edge enumeration $E = e_1, \ldots, e_n$ with j as an initial node. It returns as its output a (possibly empty) set of all injections $h : L \to G$ such that $h_V(j) = v$.

```
Search($j, v, G, E$)
1  $A_0 \leftarrow \{g\colon L \rightharpoonup G \mid g_V(j) = v_G \wedge \mathrm{dom}(g_V) = \{j\} \wedge \mathrm{dom}(g_E) = \emptyset\}$
2  for $i \leftarrow 1$ to $n$
3      do $A_i \leftarrow \{g\colon L \rightharpoonup G \mid g$ is injective $\wedge\, \exists g' \in A_{i-1}.\, g >_{E[i]} g'\}$
4  return $A_n$
```

Proposition 3.2 (correctness of Search)**.** *Let $G \in \mathbb{C}$ be a graph. Let $E = e_1, \ldots, e_n$ be an edge enumeration of L and let $j \in V_L$ be an initial node of E. Let $v \in V_G$ be a node in G. Then* Search(j, v, G, E) *returns the set of all injections $h : L \to G$ such that $h_V(j) = v$.*

Proof. Termination is guaranteed by the fact that a given morphism can only be extended to a finite number of distinct new morphisms, bounded by the number of nodes in the target graph.

Soundness. The sets A_i for $0 \leq i \leq n$, must by the definition of Search consist only of partial injections from L to G, so the same holds for the returned set A_n. For all injections $h \in A_0$ it holds that $h_V(j) = v$, so by the definition of morphism extension, this must also hold for all morphism in all other sets A_i for $i \leq n$.

All that remains then is to show that the morphisms in A_n are total. It follows by induction on the definition of morphism extension that all morphisms in set A_i are partial injective morphisms with edge-domain $e_1, \ldots, e_i$

and node-domain of the sources and targets of $e_1, \ldots, e_i$. By the requirement for left-hand side connectedness, all nodes in the graph must be the source or target of some edge e_i. Therefore, the returned set A_n contains only injective morphisms with edge-domain E_L and node-domain V_L, i.e. *total* injective morphisms.

Completeness. For $i = 1, \ldots, n$, let L_i be the subgraph of L consisting of the edges $e_1, \ldots, e_i$ and their incident nodes. Also, let L_0 be the subgraph consisting of just the start node i. A straightforward induction on i shows that for $i = 0, \ldots, n$,

$$\{g \colon L \rightharpoonup G \mid g_V(j) = v \text{ and } h \text{ is injective and } \mathrm{dom}(h) = L_i\} \subseteq A_i.$$

Since $L_n = L$ by the structure of L, it follows that A_n contains all total injections from L to G. □

We now define the algorithm FAST-MATCH (Algorithm 3.4), which solves the graph matching problem for left-connected graph transformation rules.[2] This algorithm solves the graph matching problem by searching, for each node in the host graph, for all matching morphisms that include the identified node in their range.

Algorithm 3.4 (Matching algorithm FAST-MATCH)**.** The algorithm works for the fixed rule r and an input graph $G \in \mathbb{C}$, as stated in the graph matching problem. It assumes an edge enumeration $E = e_1, \ldots, e_n$ of L, and a start node $j \in V_L$ such that j is an initial node for E.

```
FAST-MATCH($j, G, E$)
1  $A \leftarrow \emptyset$
2  for each $v$ in $V_G$
3     do $H \leftarrow$ SEARCH($j, v, G, E$)
4        $A \leftarrow A \cup H$
5  return $A$
```

Proposition 3.3 (correctness)**.** *Algorithm* FAST-MATCH *solves the graph matching problem.*

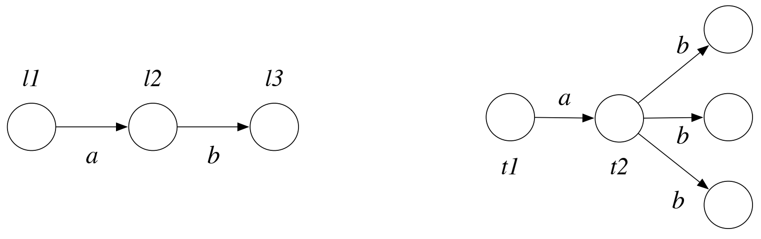
[2]It would be easy to merge the algorithm SEARCH into FAST-MATCH. The algorithms are given separately rather than merged because SEARCH is reused in several later algorithms.

Proof. We have shown in Proposition 3.2 that the auxiliary algorithm Search constructs the set of all morphisms $h : L \to G$ that matches the left-hand side start node $j \in V_L$ to a particular host graph start node $v \in V_G$. FastMatch applies Search in turn to every node $v \in V_G$. Soundness therefore follows immediately from the correctness of Search. Completeness follows from the fact that each morphism h must include some node as the value of $h_V(j)$; applying Search to this as the start node will construct the corresponding set of morphisms. □

We now define the branching factor for an edge enumeration of r's left-hand side over a target graph. The branching factor determines the worst-case performance of algorithm Search, and so determines the conditions under which it terminates in constant time.

Definition 3.4 (branching factor)**.** Let $G \in \mathbb{C}$ be a graph. Let $E = e_1, \ldots, e_n$ be an edge enumeration of L. Let $h \colon L \rightharpoonup G$ be a partial injection such that $\mathrm{dom}(h_E) = \{e_1, \ldots, e_k\}$ for some k such that $1 \leq k < n$, and for any node v, $v \in \mathrm{dom}(h_V)$ if and only if v is the source or target of an edge in $\mathrm{dom}(h_E)$. The *out-branching factor* of h_k is the number of edges $e \in E_G$ with label $l(e_{k+1})$ and source $h_V(s(e_{k+1}))$. The *in-branching factor* is the number of edges $e \in E_G$ with label $l(e_{k+1})$ and target $h_V(t(e_{k+1}))$. The *local branching factor* for h_k is the minimum of the out- and in-branching factors, if they are defined. The *branching factor* of E over graph G is the maximum local branching factor for any h_k with $1 \leq k < n$.

Example 3.3 (branching factor)**.** Left: a left-hand side graph L. Right: a graph $G \in \mathbb{C}$. We can use the edge labels of L to stand for its edges as they are distinct for distinct edges. We assume the edge enumeration $[a, b]$ and a partial injection h such that $\mathrm{dom}(h_E) = \{a\}$. The injection h_V maps node $l1$ to $t1$ and node $l2$ to $t2$.



The out-branching factor for h is 3, as the next edge in the enumeration is labelled b, and there are three edges in the host graph with label b outgoing from node $t2$. The in-branching factor is undefined, as $l3$ is not in the domain of the morphism. The local branching factor is therefore 3. This is the maximum of any of the partial morphisms from L to G so the overall branching factor is also 3.

Note that, if there is a maximal node degree d in $\mathbb{C}$, every edge enumeration of rule r has a branching factor $b \leq d$ over all $G \in \mathbb{C}$.

Proposition 3.4 (complexity of SEARCH)**.** *If edge enumeration $e_1, \ldots, e_n$ of the left-hand side graph L has branching factor b over graph $G \in \mathbb{C}$ then algorithm* SEARCH *terminates in time at most $O(\sum_{i=0}^{n} b^n)$. The maximum size for the resulting set of morphisms is b^n.*

Proof. The initialisation of the algorithm constructs the set A_0, which contains at most one partial morphism, and which (under Assumption 3.1) can be constructed in constant time.

A single run of the algorithm involves n iterations of the main loop, each of which attempts to extend all morphisms h in A_{i-1} by edge e_i. By the definition of an edge enumeration, h must be defined for either $s_L(e_i)$, or $t_L(e_i)$, or both. By the definition of branching factor there can be at most b distinct edges with the same label as e_i attached to either $h_V(s_L(e_i))$, or $h_V(t_L(e_i))$, meaning that there can be at most b distinct new injections h' constructed from h. It follows $|A_i| \leq b|A_{i-1}|$. Under Assumption 3.1, the set of extensions to each h can be identified time $O(b)$, so the time required to update A_{i-1} to A_i is $O(b|A_{i-1}|)$.

Thus we obtain the following upper bound for the overall running time: $O(1+b|A_0|+b|A_1|+\ldots+b|A_{n-1}|)$. By recursively expanding each term $|A_i|$ to its maximal size, we arrive at the expression $O(1 + b + b^2 + \ldots + b^n) = O(\sum_{i=0}^{n} b^i)$. Because the final term of the expansion also gives the maximal size of the resulting set, this also shows that maximal size of A_n is b^n. □

Theorem 3.5 (complexity of FAST-MATCH)**.** *If edge enumeration $E = e_1, \ldots, e_n$ of the left-hand side graph L has branching factor b over graph $G \in \mathbb{C}$, then* FAST-MATCH *requires time $O(|V_G| \sum_{i=0}^{n} b^i)$ at most. The maximal size for the resulting set A is $|V_G| b^n$.*

Proof. We have shown in Proposition 3.4 that the time bound for algorithm SEARCH given a branching factor of b is $O(\sum_{i=0}^{n} b^n)$, and that it constructs at most b^n distinct morphisms. The algorithm FAST-MATCH applies SEARCH to each node v in V_G in turn. We can therefore conclude that the algorithm requires time $O(|V_G| \sum_{i=0}^{n} b^i)$ at most, and that the maximal size for the resulting set of morphisms A_n is $|V_G|b^n$. $\square$

Note that n is constant according to the graph matching problem. Hence, if b is bounded then the time bound given in Theorem 3.5 is linear.

The time complexity results of Proposition 3.4 and Theorem 3.5 require the existence of an edge enumeration with a bounded branching factor over all members of $\mathbb{C}$. This is a semantic condition, meaning that a complex proof is required to show that the condition holds for a given graph class.

We now present several simple syntactic conditions on rule r and class of graphs $\mathbb{C}$ that ensure that an edge enumeration with bounded branching factor over all graphs in $\mathbb{C}$ can be constructed. These conditions have the main advantage that it is simple to check graph rules and graph classes for conformance to the conditions.

Condition 1 (Bounded degree).

There exists an integer $d \geq 0$ such that for every graph G in $\mathbb{C}$, the degree of every node $v \in V_G$ is bounded by d.

Lemma 3.6 (Condition 1 branching factor)**.** *If Condition 1 is satisfied, all edge enumerations of r have branching factor of at most d over all graphs in $\mathbb{C}$.*

Proof. The degree-bound ensures that for any node in a graph $G \in \mathbb{C}$ and any edge label, there can be no more than d attached edges with this label. Therefore, because the branching factor is defined by the number of attached edges with a particular label, the branching factor b for any edge enumeration over any graph in $\mathbb{C}$ must be less than or equal to d. $\square$

In the next condition we restrict only the out-degree of nodes. This has the interesting consequence that the in-degree of the node can be unbounded, but matching still only requires linear time.

Condition 2 (Bounded out-degree).

(1) There exists a node s in L such that every node in L is reachable from s.

(2) There exists an integer $d \geq 0$ such that for every graph G in $\mathbb{C}$, the out-degree of every node $v \in V_G$ is bounded by d.

Lemma 3.7 (Condition 2 branching factor)**.** *If Condition 2 is satisfied, there exists an edge enumeration $E = e_1, \ldots, e_n$ with branching factor at most d over all graphs in $\mathbb{C}$.*

Proof. The reachability property ensures that an edge enumeration $E = e_1, \ldots, e_n$ can be constructed such that for $i = 2, \ldots, n$, the source of e_i is the source or target of any of the edges of $e_1, \ldots, e_{i-1}$. As a result of the out-degree bound, the local branching factor for any morphism h and edge e_i with the source of e_i already matched in h cannot be larger than d. The structure of the edge enumeration ensures that any partial morphism h constructed from a partial edge enumeration $e_1, \ldots, e_i$ must include the source node for e_{i+1}. Therefore, the branching factor b over any $G \in \mathbb{C}$ of such an enumeration must be less than or equal to d. □

Proposition 3.8. *If either Condition 1 or Condition 2 is satisfied, the graph matching problem for rule r and graph $G \in \mathbb{C}$ can be solved in time at most $O(|V_G| \sum_{i=0}^{n} d^i)$, or linear time. The resulting set contains at most $|V_G| d^n$ injections.*

Proof. Immediate consequence of Theorem 3.5 and Lemmas 3.6 and 3.7. □

The next condition requires that out-edges are distinctly labelled. This results in an improved time complexity compared to the results proved in Proposition 3.8 for Conditions 1 and 2.

Condition 3 (Distinctly-labelled out-edges).

(1) There exists a node s in L such that every node in L is reachable from s.

(2) For every graph G in $\mathbb{C}$, distinct edges outgoing from the same node have distinct labels.

This condition is satisfied by the graphs and graph transformation rules we use in Chapters 9 and 10 to model pointer structure rewriting. This is because data-structures can include only a single outgoing field of each type (corresponding to a single out-edge with each label), but can be the target of an unbounded number of other data-structure fields (corresponding to an unbounded in-degree). As a result, the time complexity bound that we prove for this condition holds in our model of pointer rewriting.

Note that Condition 3 implies Condition 2, which can be seen by choosing bound d as the size of $\mathcal{C}_E$. The converse does not hold in general. Condition

1 is incomparable with Conditions 2 and 3, because while its restriction on the degree of nodes is stronger, its restriction on the left-hand sides of rules is weaker.

Lemma 3.9 (Condition 3 branching factor)**.** *If Condition 3 is satisfied, there exists an edge enumeration $E = e_1, \ldots, e_n$ with branching factor* 1 *over all graphs in $\mathbb{C}$.*

Proof. As with Condition 2, the reachability ensures that an enumeration $e_1, \ldots, e_n$ can be constructed such that for $i = 2, \ldots, n$, the source of e_i is the source or target of one of the edges of $e_1, \ldots, e_{i-1}$. Due to the distinctness of outgoing edge labels the local branching factor for any morphism h and edge e_i with the source of e_i already matched in h cannot be larger than 1. As a result, such an edge enumeration must have an overall branching factor of 1. ☐

Proposition 3.10. *If Condition 3 is satisfied, the graph matching problem can be solved in time $O(n|V_G| + |V_G|)$, or linear time. The resulting set contains at most $|V_G|$ injections.*

Proof. Immediate consequence of Theorem 3.5 and Lemma 3.9. ☐

Propositions 3.8 and 3.10 show that the graph matching problem can be solved in linear time under Conditions 1, 2, and 3. This is a substantial improvement on the polynomial time required to solve the graph matching problem in general.

Conditions 1, 2 and 3 do not alone guarantee that an application of rule r preserves the constraints on $\mathbb{C}$ expressed in the conditions. To preserve Condition 1, it suffices that for each node v in K, $\deg_R(v) \leq \deg_L(v)$. For properties (1) and (2) of Condition 2, it suffices to require that for each node v in K, $\mathrm{outdeg}_R(v) \leq \mathrm{outdeg}_L(v)$. To ensure the preservation of properties (1) and (2) of Condition 3 it suffices to require that for each node v in K and label l there exists an l-labelled node in R with source v only if there exists such an edge in L.

3.3 Fast rooted graph transformation

We now present conditions on graph transformation systems which ensure that rules can be executed in constant time. Intuitively, these systems conform to the conditions which ensure that left-connected rules apply in linear time, and the new requirement that each rule has a root node which can be matched to only a single node in the target graph. Under these conditions, the overall space of possible matches is of constant size, which means that constant time matching can be achieved.

Constant time rule application through the use of roots is of interest to us for two reasons, aside from the intrinsic merit of improving the time complexity of rule application. Firstly, rooted rules useful for modelling of pointer rewriting, as we do in Part IV of this thesis. Second, in Chapter 4 rooted rules are used as the basis for efficient language recognition systems. In these rooted recognition systems, the root is explicitly used to control the order of application of reduction rules.

Definition 3.5 (root label, root node, rooted rule)**.** A node label ϱ in graph G is a *root label* if G contains exactly one ϱ-labelled node. A node v in a graph G is a *root* if it has label ϱ. A graph G is *ϱ-rooted* if ϱ is a root label for G. A rule $r = \langle L \leftarrow K \rightarrow R\rangle$ is a *ϱ-rooted rule* if (1) r is a left-connected rule, and (2) L is ϱ-rooted.

Assumption 3.3. For the rest of this section, let ϱ be a node label and $\mathbb{C}$ a graph class such that ϱ is a root label for every graph in $\mathbb{C}$. Let $r = \langle L \leftarrow K \rightarrow R\rangle$ be a fixed $\mathbb{C}$-preserving ϱ-rooted rule.

Any matching morphism between the target graph $G \in \mathbb{C}$ and the rule r graph must match the ϱ-labelled nodes in L and G together. This can cut down substantially the space of possible matches. As the two ϱ labelled nodes must be matched together, they can be used as the start nodes in a single call to Search (Algorithm 3.3), which removes the need for the algorithm Fast-Match.

We now prove that a single call to Search suffices to solve the graph-matching problem. We also show that the existence of an edge enumeration with a the ϱ-labelled node as an initial node and bounded branching factor suffices to ensure constant time termination for Search. (See Proposition 3.3 and Theorem 3.5 for the analogous results for left-connected graph transformation rules).

Proposition 3.11 (correctness under rootedness)**.** *Let $E = e_1, \ldots, e_n$ be an edge enumeration such that the ϱ-labelled node in L is an initial node for*

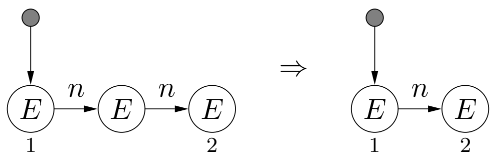


Figure 3.1: Rooted rule removing an element from a linked list.

E. Let j be the ϱ-labelled node in L and v be the ϱ-labelled node in G. Then the call Search(j, v, G, E) *solves the graph matching problem for rule r.*

Proof. By the definition of a root label, there exists exactly one node j in L and exactly one node v in G such that $l(j) = l(v) = \varrho$. For this reason, for any morphism $h : L \to G$ it must be true that $h_V(j) = v$. We have shown in Proposition 3.2 that the call Search(j, v, G, E) returns the set of all total injective morphisms $h : L \to G$ such that $h_V(i) = v$. Thus the Search solves the graph matching problem. □

Proposition 3.12 (complexity under rootedness)**.** *If the enumeration $E = e_1, \ldots, e_n$ has a branching factor of b over graph G, and the root node labelled ϱ in L is an initial node for E, then the graph matching problem can be solved by algorithm* Search *in time $O(\sum_{i=0}^{n} b^i)$. The maximal size of the resulting set of matching morphisms is b^n.*

Proof. Immediate consequence of Proposition 3.11 and Proposition 3.4. □

Note once again that n is constant according to the graph matching problem. Hence, if b is bounded then the time bound of Prop. 3.12 is constant – albeit a possibly large one.

Example 3.4 (rooted rule)**.** The root node in a rooted graph transformation rule can be seen as a handle into the graph, used to select an application area for the rule. For example, the rule shown in Figure 3.1 removes a single element from a singly-linked list. The root node, shown as a small grey node, identifies the preceding node from the list element which will be removed.

We now give sufficient syntactic conditions on rule r and class of graphs $\mathbb{C}$ to ensure that an edge enumeration with bounded branching factor over all graphs in $\mathbb{C}$ can be constructed. These conditions are based on Conditions 1, 2, and 3 given in §3.2. We begin by modifying Condition 1 to give Condition R1.

Condition R1 (Bounded degree with roots).

There exists an integer $d \geq 0$ such that for every graph G in $\mathbb{C}$, the degree of every node $v \in V_G$ is bounded by d.

Lemma 3.13 (Condition R1 branching factor)**.** *If Condition R1 is satisfied, there exists an edge enumeration with the root of L as an initial node and branching factor of at most d over all $G \in \mathbb{C}$*

Proof. By Lemma 3.6 we know under Condition 1 all edge enumerations have branching factor less than d over graphs in $\mathbb{C}$. The general assumption that rule r is left-connected ensures that an edge enumeration exists with a root-labelled node as its initial node. Condition R1 implies Condition 1, so this enumeration must have a bounded branching factor over all $G \in \mathbb{C}$. □

Conditions 2 and 3 are modified in a similar way to give conditions R2 and R3. While Conditions 2 and 3 require that every node in L is reachable from some arbitrary node, Conditions R2 and R3 require that every node is reachable from the root-labelled node in L.

Condition R2 (Bounded out-degree with roots).

(1) There exists a node s in L with label ϱ such that every node in L is reachable from s.

(2) There exists an integer $d \geq 0$ such that for every graph G in $\mathbb{C}$, the out-degree of every node $v \in V_G$ is bounded by d.

Lemma 3.14 (Condition R2 branching factor)**.** *If Condition R2 is satisfied, then there exists an edge enumeration with branching factor at most d over all graphs in $\mathbb{C}$.*

Proof. The reachability property ensures that an edge enumeration $E = e_1, \ldots, e_n$ exists such that the root-labelled node is an initial node, and for $i = 2, \ldots, n$, the source of e_i is the source or target of any of the edges of $e_1, \ldots, e_{i-1}$. By the same argument as used in Lemma 3.7, this enumeration must have a branching factor less than or equal to d over all $G \in \mathbb{C}$. □

Proposition 3.15. *If either Condition R1 or Condition R2 is satisfied, the graph matching problem can be solved in time at most $O(\sum_{i=0}^{n} d^i)$. The resulting set contains at most d^n injections.*

Proof. Immediate consequence of Proposition 3.12 and Lemmas 3.13 and 3.14. □

Condition R3 (Distinct out-labels with roots).

(1) There exists a node s in L with label ϱ such that every node in L is reachable from s.

(2) For every graph G in $\mathbb{C}$, distinct edges outgoing from the same node have distinct labels.

Lemma 3.16 (Condition R3 branching factor)**.** *If Condition R3 is satisfied, there exists an edge enumeration $E = e_1, \ldots, e_n$ with branching factor 1 over all graphs in $\mathbb{C}$.*

Proof. The reachability property ensures that an edge enumeration $E = e_1, \ldots, e_n$ exists such that the root-labelled node is an initial node, and for $i = 2, \ldots, n$, the source of e_i is the source or target of any of the edges of $e_1, \ldots, e_{i-1}$. By the same argument used in Lemma 3.9, this enumeration must have branching factor 1 over all graphs in $\mathbb{C}$. □

Proposition 3.17. *If Condition R3 is satisfied, the graph matching problem can be solved in time $O(n)$. The resulting set contains at most one injection.*

Proof. Immediate consequence of Proposition 3.12 and Lemma 3.16. □

Propositions 3.15 and 3.16 demonstrate that Conditions R1, R2 and R3 are sufficient to ensure termination in constant time. Note that this is a considerable improvement on the polynomial time complexity of unrestricted graph transformation, and also an improvement on the linear termination time possible under Conditions 1, 2, and 3.

Preservation of Conditions R1, R2 and R3 is largely similar to the requirement for preservation given at the end of chapter 3 for the corresponding unrooted conditions. To preserve the ϱ-labelled node as a root node, it suffices to require that exactly one ϱ-labelled node is present in both L and R.

3.4 Multi-step graph transformation

In many applications of graph transformation rules are not applied individually and in isolation. Rather they are applied in sets and in sequence: the graph resulting from one rule application becomes the input graph for the next rule application. So far in this chapter we have discussed conditions that improve the application time of individual graph transformation rules. This section discusses the efficient sequential application of sets of graph-transformation rules. We examine two approaches to efficient multi-step graph transformation, one rooted and one unrooted, both of which ensure linear time termination.

We first define formally the multi-step application problem.

Multi-Step Graph Transformation Problem (MGTP).

Given: A Graph class $\mathbb{C}$ and a set of left-connected, $\mathbb{C}$-preserving rules $\mathcal{R}$.

Input: A Graph $G \in \mathbb{C}$ and a length $l \geq 0$.

Output: A Graph $H_i \in \mathbb{C}$ which is the result of derivation sequence $G \Rightarrow_{\mathcal{R}} H_1 \Rightarrow_{\mathcal{R}} H_2 \ldots \Rightarrow_{\mathcal{R}} H_i$ where either $i = l$, or $i < l$ and H is irreducible w.r.t $\mathcal{R}$.

Intuitively, in this problem we are given a set of rules, an initial graph, and a sequence length bound. We must construct an arbitrary sequence of rule applications of up to the given length, starting from the given graph, using the rules in the given set. Any derivation shorter than the given length is also acceptable if it results in an irreducible graph. This avoids the need for backtracking by allowing ‘stuck’ configurations to be valid results.

Backtracking must be avoided if we want to achieve linear-time termination, because even quite heavily restricted graph transformation systems with back-tracking have an exponential time complexity. For example, if each iteration of the derivation sequence produces two distinct possible results, then there are at most 2^l distinct derivations of length l. If all these fail on the final step of the derivation sequence, the search for a valid derivation sequence of length l requires the construction of $O(2^l)$ graphs, even without the cost of matching.

In this section we consider both rooted and unrooted multi-step graph transformation rules. The rooted case is a simple application of the results given in §3.3, while the unrooted case requires a new application algorithm. For this reason we consider the simpler rooted case first.

We have shown in Proposition 3.12 that a rooted rule can be applied to a graph in constant time given a rooted edge enumeration with a bounded branching factor over the graph. As any derivation sequence consists of a sequence of single rule applications, linear-time termination can be achieved by applying this result directly to the multi-step graph transformation problem. This still holds when we construct the sequence by choosing non-deterministically from a set of rules, as the size of the rule-set is fixed under the multistep graph transformation problem.

Assumption 3.4. For the rest of this section, let ϱ be a node label and $\mathbb{C}_\varrho$ a graph class such that ϱ is a root label for every graph in $\mathbb{C}_\varrho$. Let $\mathcal{R}_\varrho$ be a fixed set of $\mathbb{C}$-preserving ϱ-rooted rules.

In the following lmax_ϱ refers to the number of nodes in the largest left-hand side of the rules in $\mathcal{R}$, and $\mathsf{rulemax}_\varrho$ refers to sum of the number of nodes in the left- and right-hand sides of the largest rule in $\mathcal{R}$.

Proposition 3.18 (complexity of the MGTP with rooted rules)**.** *If there exists for each rule $r = \langle L \leftarrow K \rightarrow R\rangle$ in $\mathcal{R}_\varrho$ an edge enumeration E_r with the ϱ-labelled root of L as an initial node and branching factor less than b over all graphs G in $\mathbb{C}_\varrho$, then the MGTP can be solved in time $O(l(|\mathcal{R}_\varrho| \sum_{i=0}^{\mathsf{lmax}_\varrho} b^i + \mathsf{rulemax}_\varrho |\mathcal{R}_\varrho| b^{\mathsf{lmax}_\varrho}))$.*

Proof. By applying Proposition 3.11 we know that we can search for a match for any rule $r \in \mathcal{R}_\varrho$ or show that no such match exists in time at most $O(\sum_{i=0}^{\mathsf{lmax}_\varrho} b^i)$. In the worst case we have to do this for every rule, for a time complexity of $O(|\mathcal{R}_\varrho| \sum_{i=0}^{\mathsf{lmax}_\varrho} b^i)$. Once the set of potential matches has been constructed, by Proposition 3.1 we can attempt to apply each match in time $O(\mathsf{lmax}_\varrho)$. As each rule can result in at most $b^{\mathsf{lmax}_\varrho}$ morphisms, this requires time $O(\mathsf{lmax}_\varrho |\mathcal{R}_\varrho| b^{\mathsf{lmax}_\varrho})$ in total. A derivation sequence of length l requires at most l such searches. The time bound $O(l(|\mathcal{R}_\varrho| \sum_{i=0}^{\mathsf{lmax}_\varrho} b^i + \mathsf{lmax}_\varrho |\mathcal{R}_\varrho| b^{\mathsf{lmax}_\varrho}))$ follows immediately. □

If b is a constant, under the multi-step graph transformation problem, $C = |\mathcal{R}_\varrho| \sum_{i=0}^{\mathsf{lmax}_\varrho} b^i + \mathsf{rulemax}_\varrho |\mathcal{R}_\varrho| b^{\mathsf{lmax}_\varrho}$ is a constant. Substituting in C gives time bound $O(C.l)$, or linear time.

In §3.3 we gave Conditions R1, R2 and R3 which ensure the existence of an enumeration with a bounded branching factor for a single rooted rule and graph class. These conditions can also be applied in the context of multi-step graph transformation. We say that a set of rules satisfies one of the conditions if every rule in the set satisfies the condition.

Proposition 3.19. *If $\mathcal{R}_\varrho$ and $\mathbb{C}_\varrho$ satisfy Condition R1, R2 or R3, then the MGTP can be solved in time $O(l)$.*

Proof. Immediate consequence of Lemmas 3.7 and 3.9, and Proposition 3.18. □

3.4.1 Amortising the cost of matching

We now consider an algorithm that permits fast multistep graph transformation for left-connected rules.

Assumption 3.5. For the rest of this section, let $\mathcal{R}$ be a set of left-connected, $\mathbb{C}$-preserving rules. In the following lmax refers to the number of nodes in the largest left-hand side and rmax the number of nodes in the largest right-hand side of the rules in $\mathcal{R}$. $\mathsf{rulemax}$ refers to sum of the number of nodes in the left- and right-hand sides of the largest rule in $\mathcal{R}$.

We have shown in §3.2 that left-connected rules can be applied in linear time given the existence of an edge enumeration with bounded branching factor. Intuitively this would seem to imply that solving the multi-step graph transformation problem must require polynomial time, as each of the l steps must require at worst linear time.

However, if for every node in the left-hand side of every rule in $\mathcal{R}$ there exists an edge enumeration with a bounded branching factor and the left-hand side node as its initial node, then the MGTP can in fact be solved in linear time. Condition 1, described in §3.3, ensures the existence of such an enumeration bound, and so ensures linear time termination of our new algorithm MULTISTEP-APPLY .

The advantage of this algorithm lies in its time complexity, which is considerably improved over a naïve algorithm that applies each rule in isolation and shares no information between steps in the derivation sequence. For comparison, we now define an example of such a naïve algorithm, making use of the algorithm FAST-MATCH (Algorithm 3.4).

Algorithm 3.5 (naïve multistep application algorithm)**.** This algorithm takes arguments in the same format as MULTISTEP-APPLY.

```
MULTISTEP-NAÏVE($H_0, R, l, E$)
1  for $i \leftarrow 1$ to $l$
2      do for $r = \langle L \leftarrow K \rightarrow R \rangle$ in $\mathcal{R}$
3             do pick $s$ from $L$
4                $M \leftarrow M \cup$ FAST-MATCH($s, H_{i-1}, E[r, s]$)
```

```
5       if $M = \emptyset$
6          then return $H_{i-1}$
7          else pick $(h, r)$ from $M$
8                $H_i \leftarrow$ APPLY$(H_{i-1}, r, h)$
9   return $H_l$
```

The termination time of MULTISTEP-NAÏVE depends on the maximum size of the intermediate graphs constructed during the derivation. The maximum possible size of any graph constructed on a derivation of length l from the graph $H_0 \in \mathbb{C}$ is given by the function $\mathsf{maxsize}(H_0, l) = |H_0| + l.\mathsf{rmax}$. This function is correct because any application of a rule in $\mathcal{R}$ can add at most rmax nodes to the existing graph.

Even assuming an array of edge enumerations with branching factor bounded by b over all graphs in $\mathbb{C}$, MULTISTEP-NAÏVE has worst-case termination time of $O(l.|\mathcal{R}|.\mathsf{maxsize}(H_0, l). \sum_{i=0}^{\mathsf{lmax}} b^i)$ (consequence of Theorem 3.5). Under Condition 1, this gives polynomial time termination. Our more sophisticated algorithm MULTISTEP-APPLY improves on this because it permits linear time termination (assuming an enumeration branching factor bound).

This result holds even when each individual rule application requires at worst linear time for application. It is achieved by sharing information between iterations of the derivation by maintaining a set of unsearched nodes. This amortises the high cost matching over the whole derivation. Sharing this information between steps requires new application algorithm MULTISTEP-APPLY, which is based on the algorithm of [1].

MULTISTEP-APPLY maintains a search set consisting of a set of unsearched host graph nodes. In each iteration of the algorithm an unsearched node in the host graph is first selected. Then for each rule in $\mathcal{R}$ the set of matches that include the selected node are constructed. If a match is discovered then the particular rule is applied, and the algorithm begins again on the new graph. If no matches that include the selected node exist for any rule in $\mathcal{R}$, then the node is removed from the search set and a new node is selected.

When the graph is updated, any match between a rule in $\mathcal{R}$ and the new graph not already present in the previous graph must include some of the nodes accessed by the update, so these are added to the search-set.

The algorithm is guaranteed to terminated in a linearly-bounded number of iterations, even without a bound on the enumeration branching factor. This is because each rewrite can add only add a bounded number of new nodes to the graph, and so the search-set can only increase by a bounded amount in each iteration. However a constant bound on the branching factor is required to ensure that the algorithm SEARCH, called by MULTISTEP-APPLY, terminates in constant time, which ensures overall linear time termination.

We first define the auxiliary algorithm APPLY*, which extends the rule application algorithm APPLY. APPLY* differs from in APPLY in that that it returns two values rather than one. In addition to the derived graph, APPLY* also constructs the set of nodes which constitute the image of the rule's right-hand side in the new graph. This set is used in the algorithm MULTISTEP-APPLY to locate the nodes accessed by each iteration of the algorithm.

Algorithm 3.6 (algorithm APPLY*)**.** The algorithm APPLY*(G, r, h) extends the algorithm APPLY (Alg. 3.2). As with APPLY, the algorithm takes as its arguments a graph $G \in \mathbb{C}$, $\mathbb{C}$-preserving rule $r = \langle L \leftarrow K \rightarrow R \rangle$ and injection $g : L \rightarrow G$ satisfying the dangling condition. The algorithm constructs the unique (up to isomorphism) double-pushout required for application of rule r, as given by the following diagram. It is implemented in exactly the same way as described for APPLY.

$$\begin{tikzcd}
L \arrow[d, "g"'] & K \arrow[l] \arrow[r] \arrow[d] & R \arrow[d, "k"] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

APPLY* returns a pair of values: (1) the uniquely-determined resulting graph H, and (2) the set $\text{ran}(k_V)$ of nodes in H matched to R by the result morphism k.

We now give algorithm MULTISTEP-APPLY, which solves the multi-step graph transformation problem for left-connected rules. This algorithm uses the general approach of the algorithms given in the work on so-called special recognition systems in [1] and [11], but applies it to our domain of general rewrite systems based on left-connected rules, rather than just the domain of graph-language recognition. The differences between our algorithm and the algorithm of [1] are described in detail in §5.3.

Algorithm 3.7 (Multistep-Apply)**.** This algorithm takes as its input a nonempty graph $H_0 \in \mathbb{C}$, a set of left-connected, $\mathbb{C}$-preserving rules $\mathcal{R}$, a derivation-sequence length $l \geq 0$, and an array of edge enumerations $E[r, s]$, where index values r and s consist of a rule $r \in \mathcal{R}$ and a start node s in the left-hand side of r.

```
Multistep-Apply($H_0, R, l, E$)
 1  $i \leftarrow 1$
 2  $U \leftarrow V_{H_0}$
 3  while $U \neq \emptyset$ and $i \neq l$
 4      do $M \leftarrow \emptyset$
 5          pick $v$ from $U$
 6          for $r = \langle L \leftarrow K \rightarrow R \rangle$ in $\mathcal{R}$
 7              do for $s$ in $V_L$
 8                      do $S \leftarrow$ Search($s, v, H_{i-1}, E[r, s]$)
 9                          $M \leftarrow M \cup \{(h, r) \mid h \in S \wedge$ Dangle($H_{i-1}, r, h$)$\}$
10          if $M = \emptyset$
11             then $U \leftarrow U \setminus \{v\}$
12             else  $i \leftarrow i + 1$
13                   pick $(h, r)$ from $M$
14                   $U \leftarrow U \setminus \mathrm{dom}(h_V)$
15                   $(H_i, P) \leftarrow$ Apply*($H_{i-1}, r, h$)
16                   $U \leftarrow U \cup P$
17  return $H_i$
```

The definition of Multistep-Apply we give here first picks a node in the set of candidate nodes, and then constructs all possible matches for all rules in $\mathcal{R}$ that can be discovered from this location. Only then does the algorithm check the dangling condition for any of the matches. This algorithm is less efficient in many cases than the alternative version that checks the dangling condition for each match as it is constructed. This backtracking version of Multistep-Apply eliminates candidate matches earlier, and thus often finds a successful match more quickly.

In addition, the version of Multistep-Apply given here often finds matches more than once while searching from different starting points. The backtracking version of Multistep-Apply avoids some of this duplication, although failed partial matches can still appear several times.

However, the backtracking version of Multistep-Apply is considerably harder to understand intuitively, and the version of Multistep-Apply given here is in any case a large and complex algorithm. The worst-case time complexity result given below in Theorem 3.23 holds for both versions of the algorithm. To make our explanation simpler we have therefore given the comparatively inefficient version of the algorithm. Our aim in giving the algorithm is to communicate the approach embodied in it, rather than present the most highly optimised version possible.

The time complexity results given below depend on the assumption that insertion and removal of nodes from the working set U requires only constant time. To achieve this, we assume that the nodes of the graph include in their representation both 'next' and 'previous' fields, in addition to the representation of graph edges. The working set of graph-nodes U can then be stored as a circular doubly-linked list embedded in the graph itself, with each graph-node storing pointers to two adjacent nodes. Such a list permits both constant time insertion and constant-time removal of any node. Such a node representation can be constructed in time $O|H_0|$ if the graph is not already in this format, so this assumption does not affect linear time termination results given Theorem 3.23.

Intuitively, the following lemma expresses the fact that, given a derivation $G \Rightarrow H$, any morphisms between the left-hand side of some rule r' and H which weren't present between r and G must have been generated by the derivation. Direct derivations under the double-pushout approach are local, in the sense that only the portion of the graph matched by the left-hand side is modified. As a result, any 'new' morphism must include in its range a node matched to the right-hand side graph of the rule in the derivation.

Lemma 3.20 (new morphisms)**.** *Let $r = \langle L \leftarrow K \rightarrow R\rangle$ and $r' = \langle L' \leftarrow K' \rightarrow R'\rangle$ be rules. Let $G \Rightarrow_r H$ be a derivation for r, and let $k : R \rightarrow H$ be the result morphism for the derivation. Let $tr_{G\Rightarrow H}$ be the corresponding track-morphism. Let $h'\colon L' \rightarrow H$ be a morphism such that $h' \neq tr_{G\Rightarrow H} \circ h$ for all $h : L' \rightarrow G$. Then then there must exist a v in $\mathrm{ran}(h'_V)$ such that v is also in $\mathrm{ran}(k_V)$.*

Proof. Suppose no node exists in $\mathrm{ran}(h'_V)$ that is also in $\mathrm{ran}(k_V)$, so all nodes in $\mathrm{ran}(h'_V)$ are also in $\mathrm{ran}(tr_{G\Rightarrow H})$. As the relationships between nodes are preserved outside of the domain of the right-hand side, we can construct an h such that $h' = tr_{G \Rightarrow H} \circ h$ by simply applying the inverse of $tr_{G \Rightarrow H}$ to h', which contradicts our assumption that $h' \neq tr_{G \Rightarrow H} \circ h$ for all $h : L' \to G$. $\square$

Theorem 3.21 (correctness)**.** *The set of possible results for the nondeterministic algorithm* MULTISTEP-APPLY *is the set of possible results for the multi-step graph transformation problem.*[3]

Proof. Algorithm soundness follows trivially from the soundness of algorithms SEARCH (proved in Proposition 3.2) and DANGLE and APPLY*.

To prove completeness we must show that if a graph is a solution to the multi-step graph transformation problem, then it is a possible result for the algorithm. We prove this by showing that the set U contains enough nodes to ensure that any matching morphism can be discovered by applying SEARCH to some node in U.

We prove this by showing that the algorithm maintains the following invariant I. Here $i \geq 1$ is the current iteration in MULTISTEP-APPLY, where the working graph H_i is the result of derivation sequence $H_0 \Rightarrow_{\mathcal{R}} H_1 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} H_i$.

> I: For every rule $r = \langle L \leftarrow K \rightarrow R \rangle$ in $\mathcal{R}$ and every morphism $h : L \to H_{i-1}$ there must exist a node in U that is a member of $\text{ran}(h_V)$.

We now show that I is preserved by the main loop of MULTISTEP-APPLY. When the algorithm starts, $i = 1$ and H_0 is the input graph and $U = V_{H_0}$. Therefore by construction the property holds. Now consider the inductive case. An iteration of the main loop begins by selecting some node $v \in U$. It then executes the algorithms SEARCH and DANGLE on this node for all rules and left-hand side initial nodes, and stores the results in set M. There are two possible results for the search:

1. $M = \emptyset$. By the completeness of SEARCH and DANGLE, no pair (h, r) of a rule $r \in \mathcal{R}$ and a morphism h exists such that $v \in \text{ran}(h)$ and h is a matching morphism for r in H_i which satisfies the dangling condition.

[3]Note that this is a slightly stronger result than simple deterministic correctness. As well as knowing that MULTISTEP-APPLY will produce *some* correct solution to the MGTP, we want to know that any of the possible correct results for the problem can be reached non-deterministically by the algorithm.

The node v is removed from U, which preserves the invariant because no morphism exists starting from the selected node.

2. $M \neq \emptyset$. By the correctness of SEARCH and DANGLE, the set M consists of all pairs (h, r) of a rule $r \in \mathcal{R}$ and a morphism h such that $v \in \text{ran}(h)$ and h is a matching morphism for r in H_i which satisfies the dangling condition. The algorithm selects an arbitrary member $(h, r) \in M$ and constructs derivation $H_i \Rightarrow_{r,h} H_{i+1}$ using algorithm APPLY*.

   APPLY* also constructs the result morphism k for the derivation and adds the nodes $\text{ran}(k_V)$ to U. By Lemma 3.20, any matching morphism for rule $r' \in \mathcal{R}$ in H_{i+1} not present between r' and H_i must include at least one node in $\text{ran}(k_V)$. Therefore, adding these nodes to U preserves the invariant.

As a consequence of invariant I and the correctness of SEARCH, at any iteration of the algorithm, any morphism between rule r and the current graph H_i can be discovered by non-deterministically applying SEARCH to some node in U. Any node in U can be selected non-deterministically, so by induction on the length of a derivation sequence any derivation up to length l can be constructed. This means that any solution to the MGTP can be constructed, and so completes the proof. □

Theorem 3.22 (complexity). *Let H_0 be a graph in $\mathbb{C}$, and let l be a fixed sequence length. If the algorithm* MULTISTEP-APPLY *is called with an array of edge enumerations E such that each edge enumeration has a branching factor less than b over every $G \in \mathbb{C}$, then it terminates in time $O((|V_{H_0}| + l.\mathsf{rmax} + l)(|\mathcal{R}|.\mathsf{lmax}.\sum_{i=0}^{\mathsf{lmax}} b^i) + l(\mathsf{rulemax} + \mathsf{rmax}))$.*

Proof. Each iteration of the main loop either removes a node from U or increments the counter i and adds at most rmax nodes to U. As i must be less than l, over the algorithm execution at most $l.\mathsf{rmax}$ nodes can added to U. Hence there can be at most $|V_{H_0}| + l.\mathsf{rmax} + l$ iterations of the main loop.

Each algorithm iteration applies algorithm SEARCH to graph H_{i-1} for some node $v \in U$ and for every left-hand size node of every rule $r \in \mathcal{R}$. The maximal size of any left-hand side is lmax. If the edge enumerations used by SEARCH have a branching factor of at most b, by Proposition 3.4 each iteration of the algorithm requires time $O(|\mathcal{R}|.\mathsf{lmax}.\sum_{i=0}^{\mathsf{lmax}} b^i)$ to perform all the applications of SEARCH.

In addition, at most l iterations of the loop use a morphism constructed by SEARCH to construct a derivation using algorithm APPLY*. By Proposition 3.1 each application of a rule r requires time $O(|r|)$, for a maximum time over the whole algorithm execution of $O(l.\mathsf{rulemax})$. Each of these rule-applying iterations also insert a maximum of rmax nodes into the set U, which in the worst case requires time $O(\mathsf{rmax})$.

Combining all of these results together produces an overall algorithm time bound of $O((|V_{H_0}|+l.\mathsf{rmax}+l)(|\mathcal{R}|.\mathsf{lmax}.\sum_{i=0}^{\mathsf{lmax}} b^i)+l.(\mathsf{rulemax}+\mathsf{rmax}))$. □

Theorem 3.23. *If $\mathbb{C}$ and $\mathcal{R}$ satisfy Condition 1, then the multi-step graph transformation problem can be solved in linear time.*

Proof. Under the specification of the multistep graph transformation problem the terms $C_1 = (|\mathcal{R}|.\mathsf{lmax}.\sum_{i=0}^{\mathsf{lmax}} b^i)$ and $C_2 = (\mathsf{rulemax} + \mathsf{rmax})$ are constant. As a result, the time bound $O((|V_{H_0}|+l.\mathsf{rmax}+l)C_1+l.C_2)$ given by Theorem 3.22 is equivalent to $O(l)$, or linear time.

By Lemma 3.6, if Condition 1 is satisfied for all rules in $\mathcal{R}$ and graph class $\mathbb{C}$, then every edge enumeration of a rule in $\mathcal{R}$ has a branching factor less than the constant bound d over any graph in $\mathbb{C}$. As a consequence of Theorem 3.22, Condition 1 therefore allows the construction of an array of edge enumerations which ensures that algorithm MULTISTEP-APPLY terminates in linear time. □

Chapter 4

Efficient graph recognition

One application of graph transformation rules is in the definition of graph languages by so-called *reduction systems*. These recognise languages by reduction; language members are reduced to a final accepting graph by repeated application of a set of reduction rules. Reduction systems are the converse notion to conventional graph grammars, in that a grammar constructs a language by derivation from an initial graph. For any reduction system an equivalent graph grammar can be constructed, and *vice versa*. Reduction systems are used in Part IV of this thesis to specify the properties of pointer structures.

One advantage of reduction systems over grammars is that a reduction system defines an implied algorithm that checks language membership. Membership can be checked for a graph by repeatedly applying reduction rules, although backtracking may be required. However, this membership checking algorithm is expensive even for recognition systems that avoid the need for backtracking, because (as discussed in the previous chapter) each rule application requires at worst polynomial time.

In the previous chapter we examined several approaches that improved the worst-case application time of graph transformation rules. In this chapter we show that these approaches can be applied to reduction systems, allowing us to define fast reduction systems that have an efficient membership checking algorithm.

Two kinds of fast reduction system are described – rooted and unrooted – corresponding to the two approaches to solving the multi-step graph transformation problem given in §3.4. Despite the restricted form of these fast reduction systems, they are surprisingly expressive, allowing linear-time recognition of both context-free and context-sensitive graph languages.

The structure of the chapter is as follows. Section 4.1, introduces rooted graph reduction specifications (RGRSs) that define graph languages of rooted graphs. We identify a class of linear RGRSs that permit a linear membership test. Section 4.2 defines several examples of linear RGRSs recognising non-context-free languages. Section 4.3 then introduces left-connected graph reduction specifications (LGRSs), and identifies a class of linear LGRSs permitting a linear membership test. Section 4.4 compares the expressive power of LGRSs and RGRSs. Section 4.5 discusses the process of developing and validating GRSs.

4.1 Recognition by rooted reduction

This section defines a class of rooted graph reduction systems which consist of rooted rules, as described in Chapter 3. We give syntactic conditions which ensure that the graph languages defined by such reduction systems have a linear-time membership test. This is in sharp contrast to the situation for general graph grammars, where even context-free languages can have an NP-complete membership test [27].

Graph reduction systems are defined by adapting the approach of [4] to the setting of fast graph transformation. The notion of a *graph signature* was defined in Chapter 2 as a means for defining basic constraints on the labelling and shape of graphs.

Assumption 4.1. For the rest of this section $\Sigma = \langle \mathcal{L}, in, out \rangle$ is an arbitrary but fixed graph signature.

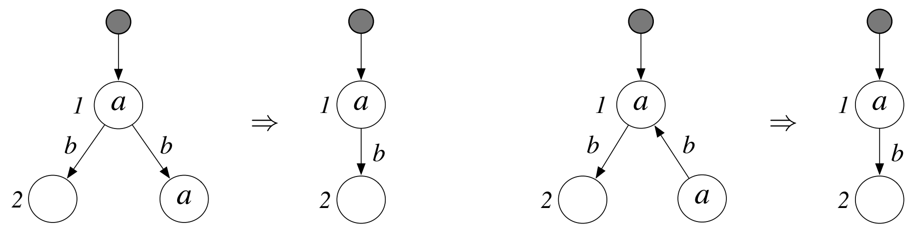
We now define a 'rooted' version of the graph reduction specifications of [4]. Recall from §3.3 that a node is described as a *root* if its label ϱ appears exactly once in the graph and that A graph is described as *ϱ-rooted* if it contains a single ϱ-labelled root.

We now define a general notion of a ϱ-rooted Σ-rule. This definition syntactically restricts rules and graphs in a similar way to Conditions R1, R2 and R3 in §3.3. However, our objective with Conditions R1, R2 and R3 was to define simple, easily understood restrictions ensuring fast graph transformation. In contrast, the definition ϱ-rooted Σ-rule is defined in terms of the individual edge-labels restricted by the signature Σ, and so is considerably more complex. Checking the definition is also in general more difficult than checking Conditions R1, R2 and R3, as checking requires the construction of an edge enumeration conforming to its restrictions.

We have define this more complex condition to provide a common framework for defining other syntactic conditions. The definition of a ϱ-rooted Σ-rule is more general than the syntactic Conditions R1, R2, and R3 given in §3.3 (proved in Lemma 4.1), and all of our examples conform to one of the simpler conditions. The common syntactic condition also allows us to reason generally about the expressive power of RGRSs (see §4.1.1).

Definition 4.1 (ϱ-rooted Σ-rule)**.** We call a Σ-rule $r = \langle L \leftarrow K \rightarrow R\rangle$ *ϱ-rooted* if (1) L and R each include a single ϱ-labelled node, (2) r is left-connected, and (3) there exists an edge enumeration $e_1, \dots, e_n$ of L with the ϱ-labelled node v as an initial node such that every edge e_i either (A) $s_L(e_i)$ is either v or the source or target of some edge e_j where $j < i$, and $(l_L(s_L(e_i)), m_L(e_i)) \in \mathrm{dom}(out)$, or (B) $t_L(e_i)$ is either v or the source or target of some edge e_j where $j < i$, and $(l_L(t_L(e_i)), m_L(e_i)) \in \mathrm{dom}(in)$.

Example 4.1 (ϱ-rooted Σ-rule)**.** Let Σ be a signature $\langle L, in, out\rangle$ such that $L = \langle\{a, \varrho\}, \{b, p\}\rangle$, $out(\varrho, p) = 1$, $out(\varrho, b) = 0$, $out(a, p) = 0$ and $out(a, b) = 2$. We define $in(\_, \_) = \bot$ for all labels.



The left-hand side rule given above is a ϱ-rooted Σ-rule. Here the ϱ-labelled root shown as a small grey node, and the p-labelled root edge is unlabelled. An edge enumeration for this rule must begin with the root edge, and then include the left and right-hand side b-labelled edges in arbitrary order.

The right-hand rule is not a ϱ-rooted Σ-rule, because no valid edge enumeration exists conforming to Def. 4.1. This is because no enumeration exists that has the root as an initial node where the source of the b-labelled edge e_b is the target of any edge earlier in the numeration than e_b. Every enumeration must include the e_b, but $in(b, a) = \bot$, meaning any enumeration violates Def. 4.1.

The definition of a ϱ-rooted Σ-rule is similar to the definition of a V-structure avoiding rule under Dörr's framework [26]. In §5.1 we discuss the separating properties between this approach and ours.

Lemma 4.1. *If a rule r and class of Σ-graphs $\mathbb{C}_\Sigma$ conform to Condition R1, R2 or R3, then r is a rooted Σ-rule.*

Proposition 4.2 (ϱ-rooted Σ-rule time complexity)**.** *A ϱ-rooted Σ-rule can be applied in constant time to a ϱ-rooted Σ-graph.*

Proof. The definition of a ϱ-rooted Σ requires the existence of an edge enumeration restricted by the signature. This edge enumeration has a bounded branching factor over all Σ-graphs, because either the in-branching factor or out-branching factor for each edge in the enumeration is bounded by the signature. By appeal to Definition 3.4, the branching factor must therefore be bounded by some fixed bound d. As a consequence of this and the existence of a uniquely-labelled root, we can apply Proposition 3.12 to show that the rule can be applied in constant time. $\square$

We now define the notion of a rooted graph reduction specification. This consists of a signature Σ, a root label used to ensure constant-time graph rewriting, a set of reduction rules, and an accepting graph. The language defined by it is the class of Σ-graphs that can be reduced to the accepting graph by repeated application of the reduction rules.

Our notion of a graph reduction specification is derived from [4], where graph reduction specifications are used for defining classes of pointer structures. Our notion of a reduction system is in one way more general, because our notion of a signature is much less restrictive than the one used in [4]. Their notion of a signature only permits graphs with uniquely-labelled out edges. However, their notion of a reduction system does not require that rules are left-connected, or that they include a root label, and does not require the existence of an edge enumeration with the root-labelled node as an initial node.

Definition 4.2 (rooted graph reduction specification)**.** A *rooted graph reduction specification* $S = \langle \Sigma, \varrho, \mathcal{C}_N, \mathcal{R}, Acc \rangle$ consists of a signature $\Sigma =$

$\langle \mathcal{C}, in, out \rangle$, a root label $\varrho \in \mathcal{C}_V$, a set $\mathcal{C}_N \subseteq \mathcal{C}_V$ of *nonterminal* labels, a finite set $\mathcal{R}$ of ϱ-rooted Σ-rules and an $\mathcal{R}$-irreducible ϱ-rooted Σ-graph Acc, the *accepting graph.* The graph language specified by S is

$$\mathrm{L}(S) = \{G \mid G \text{ is a } \Sigma\text{-graph and } G \Rightarrow^*_{\mathcal{R}} Acc \text{ and } l_G(V_G) \cap \mathcal{C}_N = \emptyset\}.$$

We often abbreviate 'rooted graph reduction specification' by RGRS. The *recognition problem* (or *membership problem*) for RGRS languages is defined as follows:

RGRS Recognition Problem.

Given:	An RGRS $S = \langle \Sigma, \varrho, \mathcal{C}_N, \mathcal{R}, Acc \rangle$.
Input:	A ϱ-rooted Σ-graph G.
Output:	*Yes* if G belongs to $\mathrm{L}(S)$, and *No* otherwise.

Note that in this formulation of the recognition problem the signature Σ is fixed, and the input graph G is assumed to conform to Σ. Our later claim of linear-time graph recognition therefore ignores the cost of checking the conformance of G to Σ. Proposition 2.2 shows that conformance to any given signature can be checked in time $O(|G|)$, by simply examining each graph node in turn. For this reason, if the complexity of membership checking is linear or worse, it makes no difference to the *order* of the complexity result whether the input graph is a Σ-graph or an unrestricted graph.

We use a signature because some restriction on the class of input graphs is a necessary precondition for the existence of a static bound on the branching factor of an edge enumeration. As discussed in §3.2 and §3.3 the existence of such a bounded enumeration improves rule application times. Definition 4.1 gives sufficient conditions on rules and signatures to ensure improved application time.

We could achieve similar results without a signature by including a depth bound as a parameter to the matching algorithm, and searching only a bounded space of possible matches. However this approach would require a more complex correctness proof to show that all matches exist within the depth bound.

Several problems are also simplified by the assumption that input graphs conform to a signature. The reduction rules that define an RGRS are generally simpler under this assumption. This is because rules can assume more about the form of the graph. Without a signature, many more rule cases

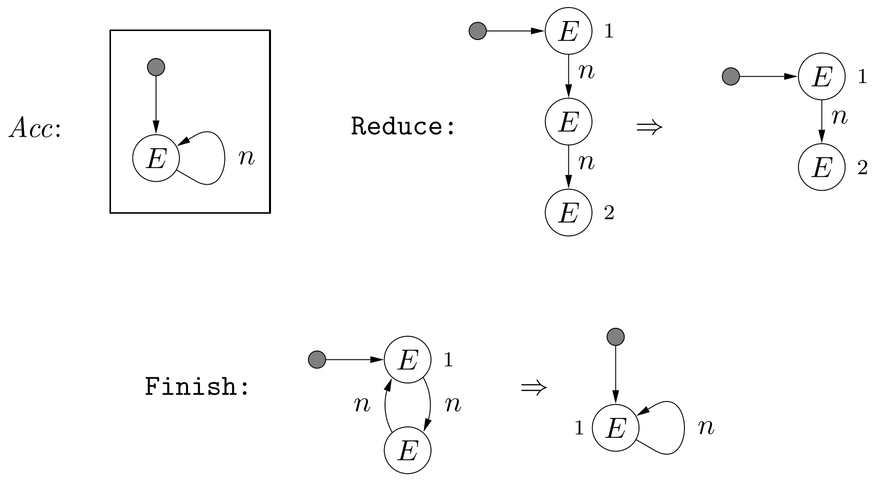


Figure 4.1: RGRS CL for rooted cyclic lists.

need to be enumerated to eliminate malformed graphs. Proofs of correctness and linear termination for RGRSs also need to consider fewer cases under the assumption of a signature. Without a signature, even malformed graphs need to be considered in these proofs.

The following simple example of an RGRS specifies the language of rooted cyclic lists. In addition to the normal nodes and edges forming a cyclic list, language members also include a root node.

Example 4.2 (rooted cyclic list)**.** The RGRS CL $= \langle \Sigma_{\mathrm{CL}}, \varrho, \emptyset, \mathcal{R}_{\mathrm{CL}}, Acc_{\mathrm{CL}} \rangle$ has the signature $\Sigma_{\mathrm{CL}} = \langle \langle \{\varrho, E\}, \{p, n\} \rangle, in_{\mathrm{CL}}, out_{\mathrm{CL}} \rangle$, where $in_{\mathrm{CL}}(l, l') = \bot$ for all $l \in \{\varrho, E\}$ and $l' \in \{p, n\}$.

$$out_{\mathrm{CL}}(l, l') = \begin{cases} 1 & \text{if } l = \varrho \text{ and } l' = p, \text{ or } l = E \text{ and } l' = n. \\ 0 & \text{otherwise.} \end{cases}$$

The accepting graph Acc_{CL} and the rules $\mathcal{R}_{\mathrm{CL}}$ are shown in Figure 4.1. The unique ϱ-labelled node is drawn as a small grey-filled node and the label p of its outgoing edge is omitted.

Proposition 4.3 (CL correctness)**.** *The language* L(CL) *is the set of all rooted cyclic lists.*

Proof. We have to show soundness (every graph in L(CL) is a cyclic list) and completeness (every cyclic list is in L(CL)).

Soundness follows from the fact that for every inverse r^{-1} of a rule r in $\mathcal{R}_{\mathrm{CL}}$, and every cyclic list G, $G \Rightarrow_{r^{-1}} H$ implies that H is a cyclic list. Every reduction $G \Rightarrow^* Acc$ via $\mathcal{R}_{\mathrm{CL}}$ gives rise to a corresponding derivation $Acc \Rightarrow^* G$ via $\mathcal{R}_{\mathrm{CL}}^{-1}$. Acc is a cyclic list, and therefore by induction G must be a cyclic list.

Completeness is shown by induction on the number of E-labelled nodes in cyclic lists. The cyclic list with one E-labelled node is Acc_{CL}, which belongs to L(CL). If G is a cyclic list with at least two E-labelled nodes, then there is a unique injective morphism from the left-hand side of either `Reduce` (if G has more than two E-labelled nodes) or `Finish` (if G has exactly two E-labelled nodes) to G. Hence there is a step $G \Rightarrow_{\mathcal{R}_{\mathrm{CL}}} H$, and it is easily seen that the resulting graph H is a cyclic list that is smaller than G. Hence, by induction, there is a derivation $H \Rightarrow^*_{\mathcal{R}_{\mathrm{CL}}} Acc$ and thus $G \in \mathrm{L(CL)}$. □

We now define a class of RGRSs whose languages can be recognised in linear time.

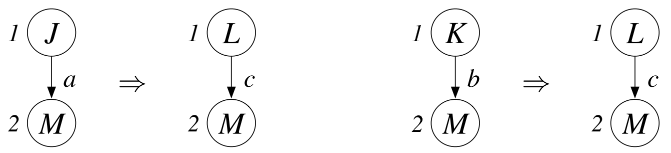
Definition 4.3 (linear RGRS)**.** An RGRS $\langle \Sigma, \varrho, \mathcal{C}_N, \mathcal{R}, Acc \rangle$ is *linearly terminating* if there is a natural number c such that for every derivation $G \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} G_n$ on ϱ-rooted Σ-graphs, $n \leq c|G|$. It is *closed* if for every step $G \Rightarrow_{\mathcal{R}} H$ on ϱ-rooted Σ-graphs, $G \Rightarrow^*_{\mathcal{R}} Acc$ implies $H \Rightarrow^*_{\mathcal{R}} Acc$. A linearly terminating and closed RGRS is a *linear* RGRS.

Example 4.3 (linear RGRS)**.** The RGRS CL given in Example 4.2 is a linear RGRS. Reduction sequences must terminate in at most $|V_G|$ steps, because all of the reduction rules reduce the number of nodes in the graph. Consequently the termination measure is 1. Reduction is also closed (in fact it is deterministic), as ϱ-rooted graphs must contain a unique root and the left-hand sides of the two rules cannot match overlapping areas of any graph in Σ_{CL}.

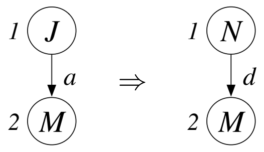
The closedness of an RGRS is a semantic property of the reduction system; that is, it cannot be checked from the simple syntactic structure of the rules. A sufficient condition for closedness is *confluence*.

Definition 4.4 (confluence)**.** An RGRS $\langle \Sigma, \varrho, \mathcal{C}_N, \mathcal{R}, Acc \rangle$ is confluent if for any ϱ-rooted Σ-graph G and pair of derivations $G \Rightarrow^*_{\mathcal{R}} H$, $G \Rightarrow^*_{\mathcal{R}} H'$, there exists some graph G' such that $H \Rightarrow^*_{\mathcal{R}} G'$ and $H' \Rightarrow^*_{\mathcal{R}} G'$. Confluence implies closedness, although the converse does not hold.

Example 4.4 (confluence)**.** The following pair of rules form a confluent RGRS, as the left-hand sides of the rules cannot overlap.



However, adding the following extra rule makes the GRS non-confluent.



This is because the left-hand sides of the first and third rules overlap, meaning that any graph isomorphic to the rule's left-hand side can be derived into two possible graphs, neither of which are reducible.

Confluence is also a semantic property, and furthermore is known to be undecidable, even for terminating graph rewriting systems [64]. However, the approach of [63] gives a sufficient syntactic condition that ensures that the rewriting system is confluent through the identification of so-called *critical pairs*. Critical pair analysis is sufficient to show that all of the examples given below are are confluent.

Theorem 4.4 (linear recognition)**.** *The recognition problem is decidable in linear time for a linear RGRS.*

Proof. Let $S = \langle \Sigma, \varrho, \mathcal{C}_N, \mathcal{R}, Acc \rangle$ be a linear RGRSs and G a ϱ-rooted Σ-graph. Membership of a Σ-graph G in $\mathrm{L}(S)$ can be tested as follows: (1) Check that G contains no node labels from $\mathcal{C}_N$. (2) Apply the rules of $\mathcal{R}$ in sequence (non-deterministically) until a graph is reached that is irreducible with respect to $\mathcal{R}$. (3) Check whether the resulting graph is isomorphic to *Acc*.

Correctness of this testing procedure follows immediately from the fact that S is closed. If the testing procedure reaches an irreducible graph, closedness ensures that the resulting graph is isomorphic to *Acc* if and only if the input graph can be reduced to Acc. Because S is linearly terminating, the testing procedure must eventually reach an irreducible graph.

The time complexity of the problem can be broken down into the three phases. Phase (1) requires time $O(|G|)$ time to examine each node for non-terminal symbols. Phase (2) of this procedure requires at most $c|G|$ reduction steps, as S is linearly terminating. As a consequence of Proposition 4.2, each step can be performed in constant time. So the time needed for phases (1) and (2) is $O(|G|)$. Lemma 4.5 shows that checking the existence of an isomorphism between a Σ-graph and a fixed Acc requires only constant time. Therefore the whole testing procedure requires only linear time. $\square$

Lemma 4.5 (checking isomorphism to Acc)**.** *Let Acc be a fixed accepting graph. For any Σ-graph G we can test whether an isomorphism h exists between G and Acc in constant time.*

Proof. Under the general assumptions about graph representation (given in Assumption 3.1) the number of nodes and edges in a graph can be determined in constant time, meaning that graphs G with different numbers of nodes and edges to Acc can be eliminated in constant time. Only graphs of the same size need to be checked. The complexity of checking that two graphs size S are isomorphic is $O(S^S)$ (application of the standard subgraph isomorphism time complexity [35]). Under the RGRS recognition problem, Acc is fixed, which means $S = |Acc|$ is constant. The cost of checking isomorphism with Acc is therefore also constant. $\square$

4.1.1 RGRS expressive power

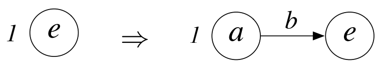
Proposition 4.6 (rgrs languages have a bounded number of connected components)**.** *Let S be an RGRS. The number of connected components in any graph G in* $\mathrm{L}(S)$ *must be at most b, the number of connected components in Acc.*

Proof. If G is a member of $\mathrm{L}(S)$ there must exist a reduction sequence $G \Rightarrow^* Acc$ via $\mathcal{R}$, and a corresponding inverse derivation $Acc \Rightarrow^* G$ via $\mathcal{R}^{-1}$. Each rule r in $\mathcal{R}$ is left-connected, so each inverse rule r^{-1} in $\mathcal{R}^{-1}$ has a connected right-hand side. Each rule in r has a ϱ-labelled node in its right-hand side, so each r^{-1} must have a nonempty left-hand side. Given inverse derivation $H \Rightarrow_{r^{-1}} H'$, graph H' must therefore have no more connected components than H. The accepting graph has a fixed number of connected components, so any graph produced by an inverse derivation $Acc \Rightarrow^*_{\mathcal{R}^{-1}} G$ must have the same number or fewer. Therefore the number of components in any graph in $\mathrm{L}(S)$ must be at most b, the number of connected components in Acc. □

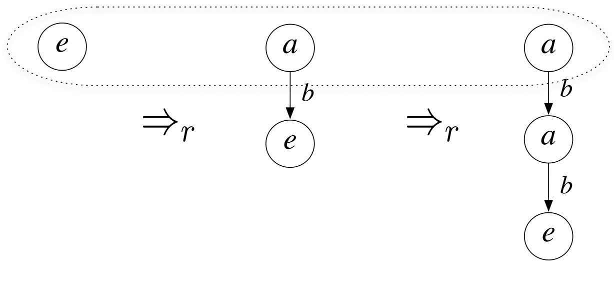
We now use this result to prove a stronger result about languages definable by RGRSs modulo the removal of the root node and incident edges.

Definition 4.5 (descendant node, descendant edge)**.** Let $G \Rightarrow H$ be a derivation and $tr_{G \Rightarrow H}$ the track morphism for the derivation. For any node v in $\mathrm{dom}(tr_{G \Rightarrow H})$, the node $v' = tr_{G \Rightarrow H}(v)$ is the *descendant node* of v. Similarly for any edge e in $\mathrm{dom}(tr_{G \Rightarrow H})$, the node $e' = tr_{G \Rightarrow H}(e)$ is the *descendant edge* of e.

Example 4.5 (descendant node, descendant edge)**.** Assume the following simple rule r over e-labelled nodes.



The following graphs form a derivation sequence.



The nodes identified by the dotted area form a sequence of descendant nodes, reading from left to right. The b-labelled edges attached to these nodes also form a sequence of descendant edges.

Lemma 4.7 (preservation of separating edges)**.** *Let $\mathcal{R}$ be a set of left-connected rules and let G, H be graphs such that $G \Rightarrow_{\mathcal{R}} H$. Let e be an edge in E_G such that e is separating and that there exists a descendant edge e' in E_H. If e is not matched by the derivation then edge e' is still separating in H.*

Proof. The result follows from the left-connectedness of the rules in $\mathcal{R}$. To make e' non-separating, the derivation must add an extra edge between the two connected components resulting from the removal of e. But because all rules are connected, to match both components, a left-connected rule must also match e. □

Proposition 4.8. *Let S be an RGRS and G a graph in $\mathrm{L}(S)$. Let G' be G with the ϱ-labelled root node and all incident edges removed. There must exist a bound b such that any G' contains less than b connected components.*

Proof. From Proposition 4.6 we know that there must exist a bound b' such that any G has less than b' connected components. Therefore it suffices to show that the number of separating edges attached to the root-labelled node in G must be bounded.

Suppose the converse holds; there exists an RGRS S such that for any b, we can pick a graph $G \in \mathrm{L}(S)$ with more than b separating edges attached to the root. As the label-set is finite, there must exist a label l such we can pick a graph with more than b l-labelled separating edges attached to the root.

As there can be an unbounded number of these edges, some of them must be removed by the reduction rules. By Lemma 4.7 these edges must remain separating until they are matched by some rule. Removing any of the b separating edges therefore requires a rule with one or more separating l-labelled edge attached to the root in its left-hand side. However, by the definition of the language the signature Σ cannot bound the number of l-labelled edges attached to the root, so such a rule violates the condition on edge enumerations in the definition of a rooted Σ-rule (Definition 4.1). Therefore no such language exists, which proves the result. □

This means that a RGRS cannot recognise quite simple languages such as the language of forests, or even the language of graphs consisting just of unconnected nodes. Other than this quite elementary restriction, we have not been able to clarify the limits of the expressive power of rooted graph reduction systems. For example, we do not know whether RGRSs are of equivalently expressive to unrooted GRSs over languages of connected graphs.

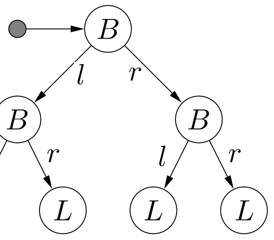


Figure 4.2: Balanced binary tree.

4.2 Non-context-free RGRS languages

To demonstrate the expressive power of linear RGRSs, in this section we present several more complex examples of graph reduction systems. All three of the examples presented here are non-context-free languages that are definable by a rooted RGRS.[1] It is interesting that we are able to recognise such languages in linear time, as there exist context-free graph languages with an NP-complete membership problem, even among languages of bounded node degree [51].

In §4.5 below we discuss the construction and validation of such large graph reduction systems.

4.2.1 Balanced binary trees

The first example in this section is the non-context-free graph language of balanced binary trees, which can be specified by a linear RGRS.

Definition 4.6 (balanced binary tree)**.** A graph is a *balanced binary tree*, BBT for short, if it consists of a binary tree built up from nodes labelled B, U and L such that all paths from the tree root to leaves have the same length. Nodes labelled with B have two children pointed to by edges labelled l and r; nodes labelled with U have one child pointed to by a c-labelled edge; nodes labelled with L have no children.

Example 4.6 (balanced binary tree)**.** Fig. 4.2 shows a balanced binary tree of depth 2.

[1]The graph languages are non-context-free in the sense of both hyperedge replacement grammars [27] and node replacement grammars [29].

In addition to the normal tree nodes, rooted BBTs include a uniquely-labelled root node (not to be confused with the *tree root*) with a single outgoing edge called the root pointer that can point to any tree-node. As in Example 4.2, we draw the root of a BBT as a small grey node, and omit the label of its unique p-labelled outgoing edge.

The RGRS for rooted balanced binary trees is RBB $= \langle \Sigma_{\mathrm{RBB}}, \{B', U'\}, \varrho, \mathcal{R}_{\mathrm{RBB}}, Acc_{\mathrm{RBB}} \rangle$. The reduction rules and accepting graph are is shown in Figure 4.3. The signature is $\Sigma_{\mathrm{RBB}} = \langle \{\varrho, B, B', U, U', L\}, \{p, l, r, c\}, out_{\mathrm{RBB}}, in_{\mathrm{RBB}} \rangle$, where $in_{\mathrm{RBB}}(l, l') = 1$ for all l, l', and

$$out_{\mathrm{RBB}}(l, l') = \begin{cases} 1 & \text{if } l = \varrho \text{ and } l' = p \\ 1 & \text{if } l \in \{B, B'\} \text{ and } l' \in \{l, r\} \\ 1 & \text{if } l \in \{U, U'\} \text{ and } l' = c \\ 0 & \text{otherwise.} \end{cases}$$

The rules in $\mathcal{R}_{\mathrm{RBB}}$ satisfy the syntactic conditions given in §4.1 ensuring that rules they are Σ_{RBB}-preserving. The rules in $\mathcal{R}_{\mathrm{RBB}}$ and class of Σ_{RBB}-graphs conform to our syntactic Condition R1 given in §3.3, meaning by Lemma 4.1 the rules are Σ_{RBB}-rules. As consequence of Proposition 4.2 rule application can be therefore be completed in constant time.

This rooted GRS RBB is a rooted modification of the *un*-rooted reduction system for balanced binary trees given in [4] (the unrooted version is used later in this chapter, in §4.3, as an example for unrooted fast reduction). Intuitively, the root in RBB behaves as a proxy for a left-hand side match in the unrooted version. Rather than rely on the left-hand side matching process to locate the areas for reduction, the root is explicitly moved by the rules of the RGRS to possible application areas, and then rooted versions of the original reduction rules are applied.

Note that this RGRS omits the size 1 graph containing a single leaf node from its language. This is to remove the need for a special case rule, and so reduce the number of rules. We do the same in the rooted binary DAG and unrooted balanced binary tree examples (§4.2.3 and Example 4.11 respectively).

The rooted BBT RGRS has a larger set of reduction rules than the unrooted version, and this increase is mostly a result of the extra cases required to allow traversal of the graph by the root. This is because the root controls the application for the rules. In the unrooted version, the automatic process of matching takes the place of explicit root moving.

The size of the RGRS is also increased by the number of nonterminal symbols. These are used to ensure termination when the RGRS is applied to malformed BBTs. Nodes are marked when they are visited by the root. Without node marking, the RGRS would not terminate when applied to (for example) a circular list of U-nodes.

The RGRS operates by collapsing subtrees together using the `R2` and `R3` rules to form a new subtree of the same height consisting entirely of U-labelled nodes. The root is moved in a depth-first manner, down to the lowest point at which one of these rules can be applied. The root then alternately collapses subtrees, and searches up the tree for the next subtree that can be collapsed. Figure 4.5 shows a sequence of reductions for a member of L(RBB).

We now show that L(RBB) is the set of all rooted balanced binary trees, by proving RBB sound (Proposition 4.9) and complete (Proposition 4.13).

we call a graph an NBBT (for *nonterminal* BBT) if it can be obtained from a rooted BBT by relabelling any number of B-nodes into B'-nodes and any number of U-nodes into U'-nodes. We describe the single edge with a ϱ-labelled node as its source as the *root pointer.*

Proposition 4.9 (RBB soundness)**.** *Let G be a Σ_{RBB}-graph. If G is a member of* L(RBB)*, then G is a rooted balanced binary tree.*

Proof. We will show that every Σ_{RBB}-graph reducible by $\mathcal{R}_{\mathrm{RBB}}$ to Acc_{RBB} is an NBBT, implying that every graph in L(RBB) is a BBT. Clearly, Acc_{RBB} is an NBBT. By the same argument as used in Proposition 4.3, it suffices to show that the inverses of the rules in $\mathcal{R}_{\mathrm{RBB}}$ preserve membership in the class of NBBTs.

The inverses of the rules `Up`, `D1` and `D2` are clearly NBBT-preserving, as they only relabel nonterminal into terminal nodes and redirect the root pointer to some other node in the tree. The dangling condition ensures that the inverse of rules `R1-U` and `R1-B` can only be applied at the tree root, because the rule application deletes either a U-labelled or B-labelled node without an ingoing tree edge. Hence the inverted rule just inserts a new U- or U'-labelled tree root on top of the old one, which preserves balance and so NBBT membership. The inverses of rules `R2-L` and `R2-R` are also NBBT-

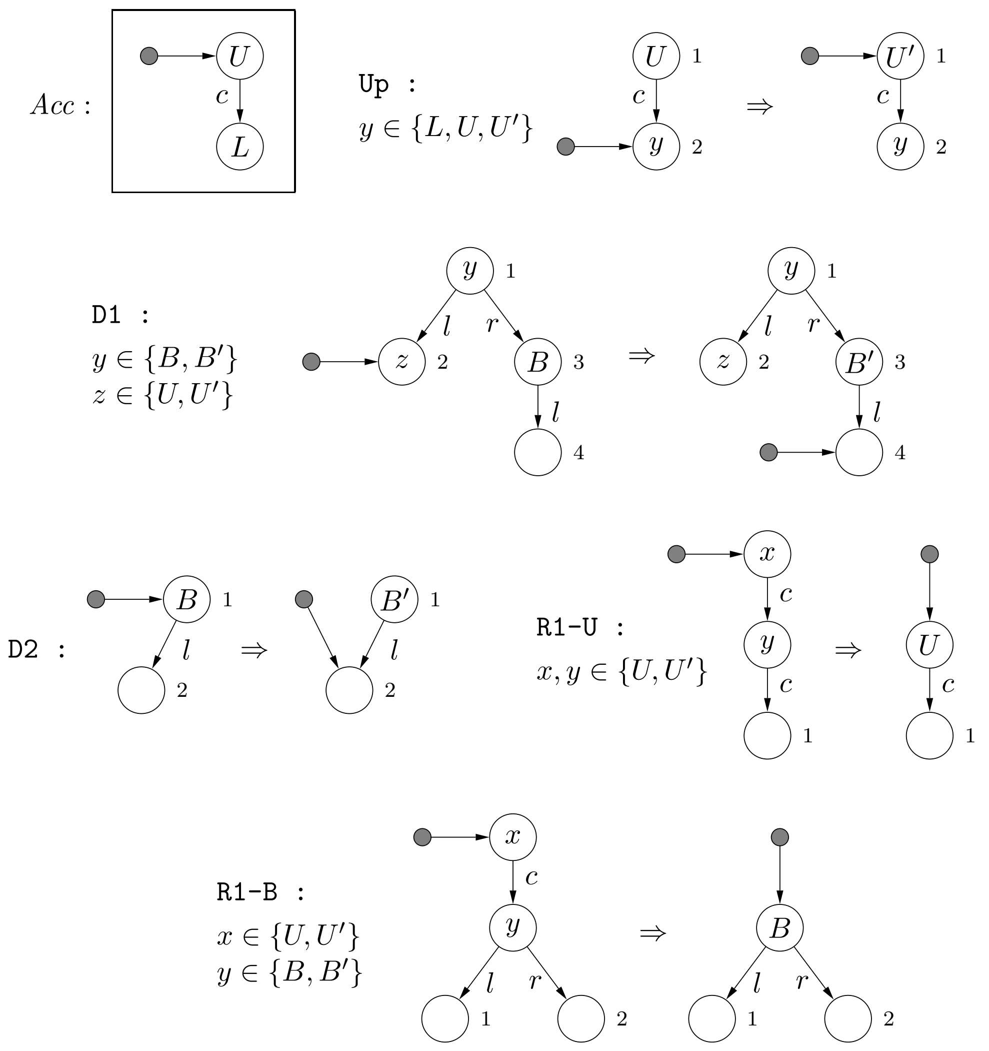
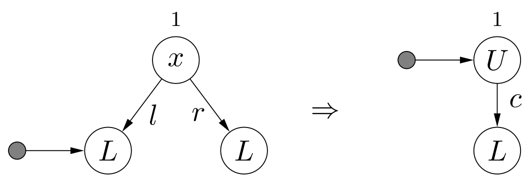
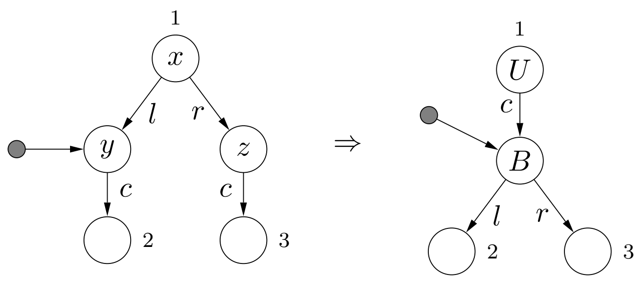


Figure 4.3: Accepting graph and rules for the rooted GRS RBB.

`R2-L :`
$x \in \{B, B'\}$



`R3-L :`
$x \in \{B, B'\}$
$y, z \in \{U, U'\}$



`R2-R`: as `R2-L`, but with labels l and r swapped
`R3-R`: as `R3-L`, but with the left-hand root pointing to the z-node

Figure 4.4: Accepting graph and rules for the rooted GRS RBB (cont).

preserving: replacing a U-node pointing to a leaf with a B-node pointing to two leaves preserves balance. Similarly, the inverses of `R3-L` and `R3-R` preserve balance and the other NBBT properties. □

Definition 4.7 (root-pointer-predecessor)**.** Given an NBBT in which the root pointer does not point to the tree root, we call a node a *root-pointer-predecessor* if it is on the unique path from the tree root to the parent of the root pointer's target.

Example 4.7 (root-pointer-predecessor)**.** In the graph shown in Fig. 4.6, the two B'-labelled nodes and the L-labelled target of the root edge are root-pointer-predecessors.

We call a graph an IBBT (for *invariant* BBT) if it is an NBBT satisfying the following conditions: (1) root-pointer-predecessors are not labelled U' and (2) all nodes labelled B' are root-pointer-predecessors. Note the class of terminal IBBTs is the class of BBTs.

Proposition 4.10. *RGRS* RBB *is linear.*

Proof. Closure can easily be checked for RBB by the critical pair approach of [63].

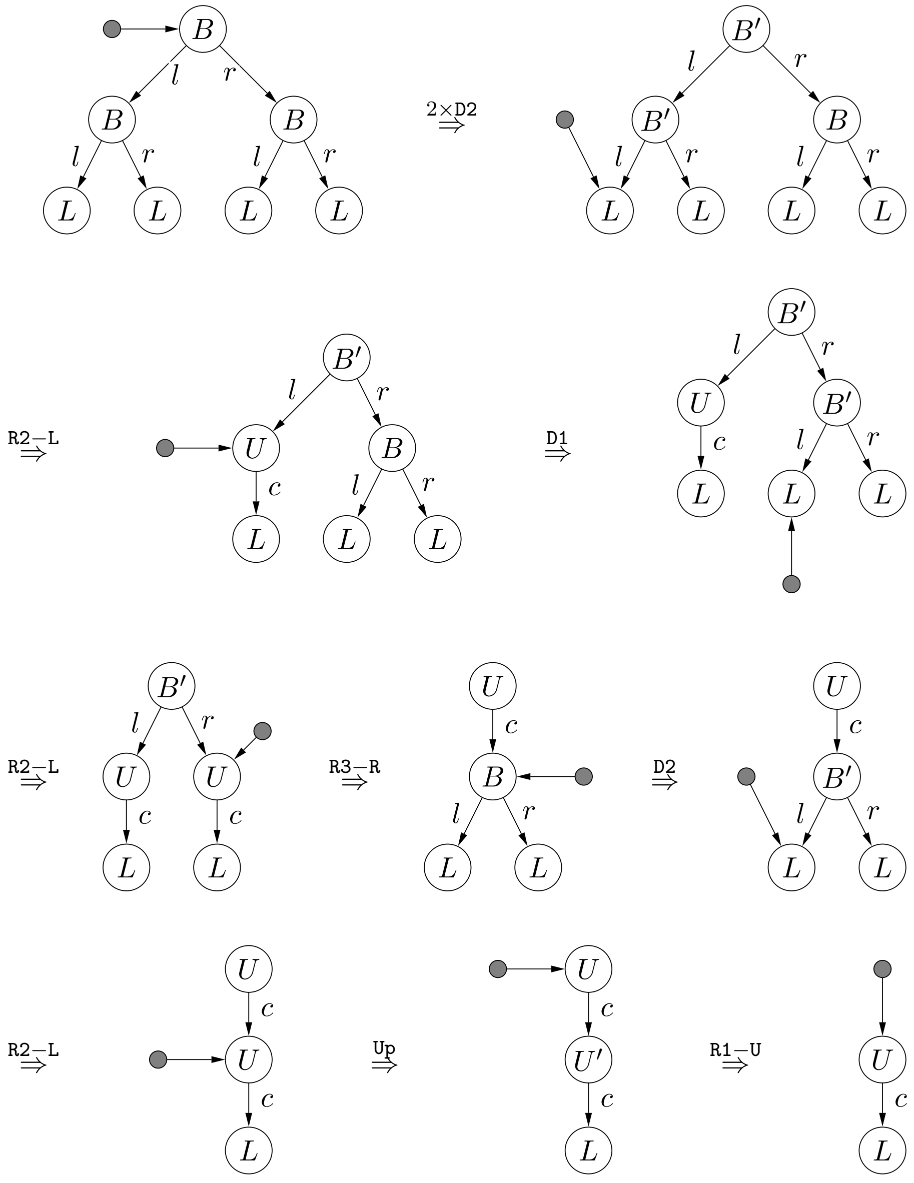


Figure 4.5: Example reduction sequence for a member of L(RBB).

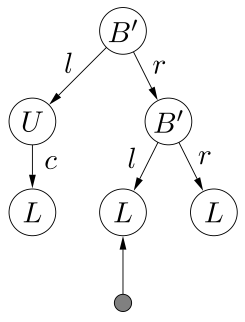


Figure 4.6: BBT including three root-pointer-predecessors.

To show that RBB is linearly terminating we define a termination measure T. For every Σ_{RBB}-graph G the termination measure $T(G) = 2|G| + |l_G^{-1}(\{B, U, L\})|$. We show by simple examination of the rules in $\mathcal{R}_{\text{RBB}}$ that for any derivation $G \Rightarrow_{\mathcal{R}_{\text{RBB}}} H$ on Σ_{RBB}-graph G, $T(G) > T(H)$.

Rules `Up`, `D1` and `D2` preserve the size of graphs but decrease the number of terminally labelled nodes, hence they decrease T's value. Rules `R1-U` and `R1-B` increase the number of terminals by at most one, but also decrease the graph size by two, so T decreases. Rules `R2-L` and `R2-R` decrease size without increasing the number of terminal node labels, so they decrease T's value too. Rules `R3-L` and `R3-R` decrease the graph size by two and increase the number of terminal node labels by at most two, so they also decrease T's value.

The length of any derivation $G \Rightarrow^*_{\mathcal{R}_{\text{RBB}}} H$ given a Σ_{RBB}-graph G must therefore be less than the value of the termination measure. But $T(G)$ is at most $3|G|$ so RBB is linearly terminating. $\square$

Lemma 4.11 (non-*Acc* IBBTs are reducible)**.** *For every IBBT G, either $G \cong Acc_{\text{RBB}}$, or G is reducible by some rule in $\mathcal{R}_{\text{RBB}}$.*

Proof. Clearly Acc_{RBB} is irreducible by $\mathcal{R}_{\text{RBB}}$. We show that every non-Acc_{RBB} IBBT is reducible by $\mathcal{R}_{\text{RBB}}$ by enumerating possible cases.

Case 1: The root pointer points to a U or U'-node. Then we know from Σ_{RBB} that it must have an outgoing edge labelled c. If it has no incoming edges, it is the treeroot and we can reduce it using either rule `R1-B` or `R1-U` depending on the target of the c-labelled edge. If this node has an incoming edge, it must be labelled c, l or r. If c, from the signature and the definition of an IBBT we know that the source must be a U node, and so rule `Up` applies. If the incoming edge is labelled l or r, its source must be a B or B' node and it must have a sibling r or l edge. From the balance property of an IBBT, the other edge must point to another node with an outgoing edge – either a U, U', B, or B'. B' is excluded by the definition of an IBBT. If U or U', rule `R3-L` or `R3-R` applies. If a B, by Σ_{RBB} it must have outgoing l and r edges, and so rule `D1` applies.

Case 2: The root pointer points to a B-node. Then by Σ_{RBB} it must have an outgoing edge labelled l, so the `D2` rule applies. By the definition of an IBBT, the root pointer cannot point to a B'-node.

Case 3: The root pointer points to an L-node. Then there must exist a single incoming edge labelled l, r, or c. If c, by Σ_{RBB} and definition of IBBT the source must be labelled U and the graph can be reduced by the `Up` rule. If the incoming edge is labelled l or r, its source must be labelled B and it must have a sibling edge labelled r or l. We know by the balance property that the target of this edge must be another L-node, so either `R2-L`, or `R2-R` applies. This completes the proof that every IBBT apart from Acc_{RBB} can be reduced. □

Lemma 4.12 ($\mathcal{R}_{\mathrm{RBB}}$ preserves IBBTs)**.** *For every derivation $G \Rightarrow_{\mathcal{R}_{\mathrm{RBB}}} H$, if G is an IBBT then H is an IBBT.*

Proof. We show that all rules preserve IBBT membership by examination of each rule in turn. Rule `Up` moves the root pointer from its current position to the node's parent and relabels this parent from U to U'. As the rest of the graph is preserved, this preserves IBBT condition (1). Rules `D1` and `D2` move the root pointer and relabel a single B-node to B'. In both cases the B'-node is a root-pointer-predecessor, so IBBT condition (2) is satisfied. No U' nodes are added, so IBBT condition (1) is satisfied. Rules `Up`, `D1` and `D2` only relabel nodes and move the root pointer, so balance is preserved.

Rules `R1-U` and R1-B delete a U or U'-node and move the root pointer to the child of the current position. We know from the IBBT conditions that this child cannot be labelled B', and so the IBBT conditions are satisfied. Rule `R1-U` and R1-B can only delete the treeroot, as this is the only non-L-node that can be safely deleted under the dangling condition, and so it preserves balance.

Rules `R2-L` and `R2-R` replace a B or B'-node with a U-node and move the root pointer. Replacing a B-node and two leaves with a U-node and one leaf preserves balance. The IBBT conditions are satisfied, as the new target of the root pointer is a U-node and the root-pointer-predecessors are otherwise preserved.

Rules `R3-L` and `R3-R` replace two U or U'-nodes with a B node. The rule preserves balance because the distance from the 'top' of the rule to the 'bottom' is preserved. These rules replace a single B or B'-node on the path to the treeroot with a U-node, and the root-pointer-predecessors are otherwise unaltered. No B'-nodes are added, so both IBBT conditions are satisfied. This completes the proof that all rules in $\mathcal{R}_{\mathrm{RBB}}$ are IBBT-preserving. □

Proposition 4.13 (RBB completeness)**.** *Let G be a Σ_{RBB}-graph. If G is a rooted balanced binary tree, then G is a member of* L(RBB)*.*

Proof. We show that every IBBT is reducible to Acc_{RBB} by $\mathcal{R}_{\mathrm{RBB}}$, implying that every BBT is in L(RBB). In Proposition 4.10 we show that every $\mathcal{R}_{\mathrm{RBB}}$-derivation sequence terminates, so it suffices to show in Lemma 4.11 that Acc_{RBB} is the only $\mathcal{R}_{\mathrm{RBB}}$-irreducible IBBT and in Lemma 4.12 that applying any rule in $\mathcal{R}_{\mathrm{RBB}}$ to an IBBT results in another IBBT. □

4.2.2 Rooted grids

We now give an RGRS for the language of rooted grids, based on the grid graph GRS given in [4].

Definition 4.8 (rooted grid)**.** A graph is a *rooted grid* if it is constructed from nodes labelled B and L, where nodes labelled B have two outgoing edges labelled d and r (intuitively these edges are 'down' and 'right'), and nodes labelled L have no outgoing edges. A grid graph consists of a $k \times l$ grid of B-labelled nodes – k rightward and l downward – with the downward and rightward edges of the grid terminated by L-labelled nodes. A *rooted* grid includes an extra root node that points to the single B-labelled node without any incoming edges (the 'upper left corner' of the grid).

Example 4.8 (rooted grid)**.** Fig. 4.7 shows a rooted grid of size 3×2.

The RGRS for rooted grids is $\mathrm{GG} = \langle \Sigma_{\mathrm{GG}}, \{C\}, \varrho, \mathcal{R}_{\mathrm{GG}}, Acc_{\mathrm{GG}} \rangle$. The reduction rules and accepting graph for GG are shown in Figure 4.8. The

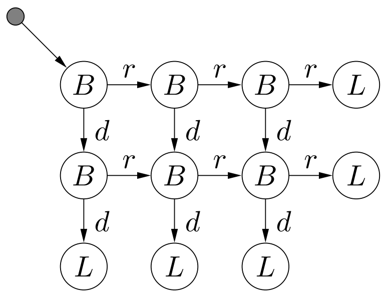


Figure 4.7: Rooted grid.

signature is $\Sigma_{\mathrm{GG}} = \langle \{\varrho, B, C, L\}, \{l, r, p\}, out_{\mathrm{GG}}, in_{\mathrm{GG}} \rangle$, where $in_{\mathrm{GG}}(v, e) = \perp$ for all v, e, and

$$out_{\mathrm{GG}}(v, e) = \begin{cases} 1 & \text{if } v = \varrho \text{ and } e = p, \text{ or } v \in \{B, C\} \text{ and } e \in \{d, r\} \\ 0 & \text{otherwise.} \end{cases}$$

GG operates by incrementally reducing the height of the grid. The `Del` rule removes B-labelled nodes, while checking that each row-element can be paired up with an element in the next row, until the whole row is deleted. The grid structure of the graph is ensured by the fact that when each row is removed, the rules ensure that the following row is of the same width. Following the deletion of a row, the rules `Next`, `Next'` and `Next''` select the next row. Once all the rows but one have been removed, the bottom row is deleted by the `DelBot` rule.

The rules and class of all Σ_{GG}-graphs conform to Condition R2 given in §3.3, meaning by Lemma 4.1 that the rules are Σ_{RBB}-rules.

Proposition 4.14 (GG correctness)**.** *The language* L(GG) *is the set of all rooted grids.*

Proof. The same approach can be used here as used in proving Propositions 4.9 and 4.13. Completeness is proved by showing that all members of the class of partially-reduced grids can be reduced. Soundness is proved by showing that inverse rules preserve membership of the class of partially-reduced grids. In both cases, the proof works by constructing possible cases for the root node to point to, and then showing that the two properties hold in all cases. ☐

Proposition 4.15. *RGRS* GG *is linear.*

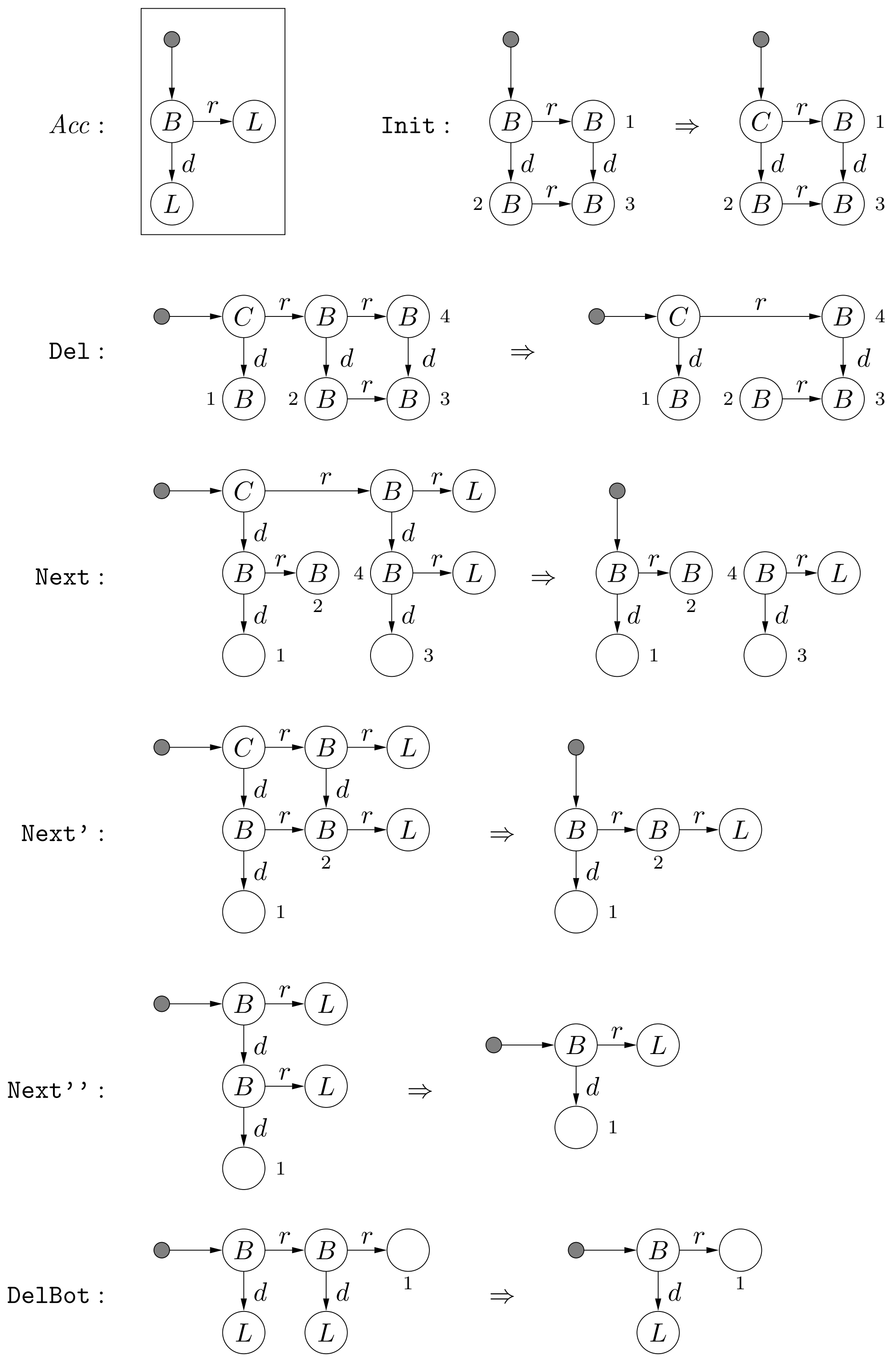


Figure 4.8: Accepting graph and reduction rules for the rooted grid RGRS GG.

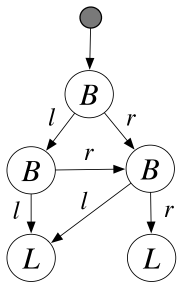


Figure 4.9: Rooted binary DAG.

Proof. Closure can easily be checked for GG by the critical pair approach of [63]. To show GG is linearly terminating, we define for all Σ_{GG}-graphs the termination measure $T(G) = |l_G^{-1}(B)|$, where $|l_G^{-1}(B)|$ is the number of B-labelled nodes in G. All rules of $\mathcal{R}_{\mathrm{GG}}$ reduce the number of B-labelled nodes. Therefore, for any derivation $G \Rightarrow_{\mathcal{R}_{\mathrm{GG}}} H$ where G is a Σ_{GG}-graph, $T(G)$ decreases. As a consequence, no derivation from G can be longer than $|G|$, as $|G| > |l_G^{-1}(B)|$ so GG is linearly terminating. □

4.2.3 Rooted binary DAGs

Our final example is a linear RGRS recognising *rooted binary DAGs*, a class of structures with nodes of unbounded in-degree. This is in contrast to our previous examples, which have all defined languages of graphs with a bounded overall degree.

Definition 4.9 (rooted binary DAG)**.** A *rooted DAG* is a directed and acyclic graph where there exists some node from which all other nodes are reachable (we will call this node the *DAG root*, to distinguish it from our ϱ-labelled graph root). A rooted *binary* DAG is a rooted DAG constructed from nodes labelled B and L, where nodes labelled B have two outgoing edges labelled l and r, and nodes labelled L have no children.

Example 4.9 (rooted binary DAG)**.** Fig. 4.9 shows a rooted binary DAG.

The language of rooted binary DAGs is recognised by the RGRS RBD = $\langle \Sigma_{\mathrm{RBD}}, \{B', B'', L', s, b, n\}, \mathcal{R}_{\mathrm{RBD}}, Acc_{\mathrm{RBD}} \rangle$. The signature is $\Sigma_{\mathrm{RBD}} = \langle \{\varrho, B, B', B'', L, L'\}, \{l, r, s, t, n\}, \varrho, out_{\mathrm{RBD}}, in_{\mathrm{RBD}} \rangle$, where $in_{\mathrm{RBD}}(v, e) = \bot$ for all v,e and out_{RBD} is defined as

$$out_{\mathrm{RBD}}(v,e) = \begin{cases} 1 & \text{if } v = \varrho \text{ and } e \in \{s,t\} \\ 1 & \text{if } v \in \{B, B', B''\} \text{ and } e \in \{l,r\} \\ 1 & \text{if } v = L' \text{ and } e = n \\ 1 & \text{if } e = b \\ 0 & \text{otherwise} \end{cases}$$

The rules and accepting graph of the RGRS are shown in Figure 4.10. Dashed edges stand for the edges labelled b that are used by the reduction system to record the path taken through the graph. As in previous examples, we draw the root of the rooted binary DAG as a small grey node and back pointers as dashed arrows, omitting labels in both cases.

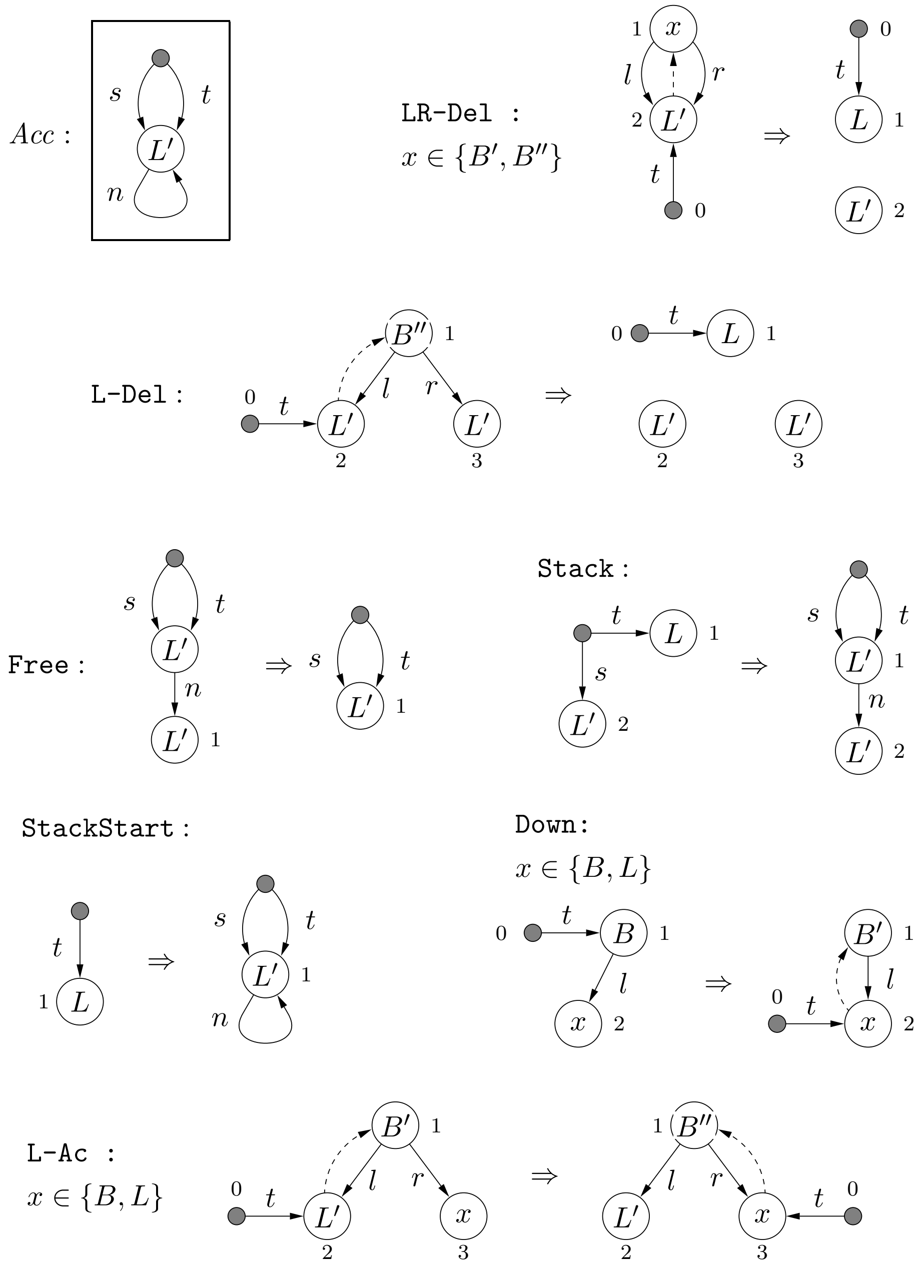
The RGRS RBD works by moving the t-labelled root-edge through the graph in a depth-first manner, converting B-labelled nodes into L-labelled nodes. The resulting L-labelled nodes are then deleted.

As rules in the reduction system are all rooted, any disconnected graph elements cannot be deleted. The RGRS avoids the possibility of garbage nodes by attaching nodes to a stack as they are encountered. This stack is constructed by pushing an L-node onto the stack when one of its parent B-nodes is relabelled to an L-node. The top of the stack is pointed to by the s-labelled ('stack') root-edge. This s-labelled root edge is a nonterminal edge that is added to the graph by the stack initialisation rule `StartStack`.

As the graph is not a tree, the t root-edge may encounter the same L-node several times during the depth-first traversal. The reduction system prevents the same node appearing several times on the stack by relabelling L-nodes to the nonterminal label L' once they are placed on the stack. Nodes are only deleted from the stack when they have no non-stack incident edges, as ensured by the dangling condition.

The rules in $\mathcal{R}_{\mathrm{RBD}}$ and the class of Σ_{RBD}-graphs conform to Condition 3, and so by lemma 4.1 we can conclude that the rules of the RGRS are Σ-rules.

We describe a graph G as an *IRBD* (for *invariant* RBD) if it is a Σ_{RBD}-graph satisfying the following conditions: (1) Graph G is acyclic. (2) There exists a none-ϱ-labelled *top node* from which all non-L'-nodes are reachable, if any exist. (3) There exists a path of l- and r-edges from the top node to the target of the root t-edge where all nodes on the path are B' or B''-labelled and have a b-labelled edge pointing to their immediate parent on the path.



R-Del : as L-Del, but with labels l and r swapped
R-Ac : as L-Ac, but with labels l and r swapped

Figure 4.10: Linear RGRS recognising rooted binary DAGs.

(4) Nodes not on the path are not B'- or B''-labelled. (5) Each B''-node must have at least one L'-labelled child. (6) Each L'-node must have either one L'-node parent or through the s root-edge a ϱ-labelled parent. (7) An L'-node's outgoing n-edge must either be a self-loop or point to another L'-node. (8) If no non-L'-nodes exist, then root-edge t points to the target of root-edge s.

Proposition 4.16. *RGRS* RBD *is linear.*

Proof. Closure can be checked by the critical pair method. The termination function for every Σ_{RBD}-graph G is defined as $T(G) = |G| + 2|l_G^{-1}(L)| + 2|l_G^{-1}(B'')| + |l_G^{-1}(B')|$. It can easily be checked by inspecting the rules of $\mathcal{R}_{\text{RBD}}$ that for every step $G \Rightarrow^*_{\mathcal{R}_{\text{RBD}}} H$ on Σ_{RBD}-graphs, $T(G) > T(H)$. This result implies that the length of any derivation $G \Rightarrow^*_{\mathcal{R}_{\text{RBD}}} H$ on Σ_{RBD}-graphs can be at most $|G| + 2|V_G|$, since $2|l_G^{-1}(L)| + 2|l_G^{-1}(B'')| + |l_G^{-1}(B')| \leq 2|V_G|$. □

Proposition 4.17 (RBD soundness)**.** *Let G be a Σ_{RBD}-graph. If G is a member of* L(RBD)*, then G is a rooted binary DAG.*

Proof. We show that every Σ_{RBD}-graph reducible to Acc_{RBD} is an IRBD. As the set of rooted binary DAGs is exactly the set of terminal IRBDs this is sufficient to prove soundness. As in Proposition 4.9 we show by case-analysis that the inverses of rules in $\mathcal{R}_{\text{RBD}}$ preserve membership in the class of IRBDs.

The inverse of rules `Down`, `L-Ac` and `R-Ac` clearly preserve IRBDs because they only move the path from the root and relabel a node. Inverse rules `StackStart` and `Stack` reorganise the stack in a way compatible with IRBDs. The inverse of rule `Free` just increases the stack length by one. The inverses of the deletion rules `LR-Del`, `L-Del`, `R-Del` convert L-nodes to B'' or B' nodes, preserving the path-labelling requirement. □

Proposition 4.18 (RBD completeness)**.** *Let G be a Σ_{RBD}-graph. If G is a rooted binary DAG, then G is a member of* L(RBD)*.*

Proof. We show that every IRBD can be reduced to Acc_{RBD}. As the set of rooted binary DAGs is exactly the set of terminal IRBDs this is sufficient to prove completeness. As in Proposition 4.13 we show that Acc_{RBD} is the only

$\mathcal{R}_{\text{RBD}}$-irreducible IRBD and that applying any rule in $\mathcal{R}_{\text{RBD}}$ to an IRBD results in another IRBD.

The reducibility of any IRBD is shown by case analysis. If t points to a B-labelled node then `Down` must apply. If it points to an L-labelled node either `Stack` or `StartStack` applies. If an L'-labelled node then the node may be the target of a B' or B'' node, in which case one of the `Ac` or `Del` rules must apply. Finally, if the L' node is not the target of any other edge, then the node must be the top of the stack, and so the rule `Free` applies.

IRBD preservation is demonstrated by case analysis of possible graph configurations matching the rule applications. Most of these cases are clear. The difficult cases are the deletion rules `LR-Del`, `L-Del`, `R-Del`. In these rules the existence of the stack of L'-nodes ensures that the graph stays connected. □

4.3 Recognition by left-connected reduction

The previous sections describes an approach to linear-time recognition which depends on the presence of uniquely-labelled roots. In this section we describe so-called *left-connected graph reduction systems* that achieve linear-time reduction without the need for roots.

Application of a left-connected graph reduction system uses the Multi-step-Apply algorithm described in §3.4. This algorithm permits the derivation of an application of a set of rules $\mathcal{R}$ to a graph $G \in \mathbb{C}$ in linear time *if* an edge enumeration with bounded branching factor can be constructed for each node in the left-hand side of each rule.

The basic definitions used in defining left-connected graph reduction systems are quite similar to those used in Section 4.1 to define RGRSs.

Definition 4.10 (fast Σ-rule)**.** Let Σ be a graph signature. We call a Σ-rule $r = \langle L \leftarrow K \rightarrow R\rangle$ *fast* if for every node v in the left-hand side L there exists an edge enumeration $e_1, \ldots, e_n$ with initial node v such that every edge e_i either (A) $s_L(e_i)$ is either v or the source or target of some edge e_j where $j < i$, and $(l_L(s_L(e_i)), m_L(e_i)) \in \text{dom}(out)$, or (B) $t_L(e_i)$ is either v or the source or target of some edge e_j where $j < i$, and $(l_L(t_L(e_i)), m_L(e_i)) \in \text{dom}(in)$.

Example 4.10 (fast Σ-rule)**.** The definition of a fat Σ-rule is quite similar to the definition of a ϱ-rooted Σ-rule – the major difference is the lack of a root label. Consequently the rules given in Example 4.1 also serve as illustration for fast Σ-rules. The left-hand rule given in this example is a fast Σ-rule, while the right-hand rule is not, for the same reasons given in that example.

Lemma 4.19. *If a rule r and class of Σ-graphs $\mathbb{C}_\Sigma$ conform to Condition 1 (given in §3.2), then r is a fast Σ-rule.*

Lemma 4.20. *For every node v in the left-hand side of a fast Σ-rule there exists an edge enumeration with v as an initial node and bounded branching factor over all Σ-graphs.*

Proof. Simple generalisation of the proof given for Proposition 4.2. □

Definition 4.11 (left-connected graph reduction specification)**.** A *left-connected graph reduction specification* $S = \langle \Sigma, \mathcal{C}_N, \mathcal{R}, c, Acc \rangle$ consists of a signature $\Sigma = \langle \mathcal{C}, in, out \rangle$, a set $\mathcal{C}_N \subseteq \mathcal{C}_V$ of *nonterminal* labels, a finite set $\mathcal{R}$ of fast Σ-rules, a *termination factor* c, and an $\mathcal{R}$-irreducible Σ-graph Acc, the *accepting graph*. The graph language specified by S is $\mathrm{L}(S) = \{G \in \mathcal{G}(\Sigma) \mid G \Rightarrow^*_{\mathcal{R}} Acc \text{ and } l_G(V_G) \cap \mathcal{C}_N = \emptyset\}$.

Note that the termination factor c is included as part of the definition of an LGRS to conform to Multistep-Apply, which requires a derivation length as one of its arguments.

We often abbreviate 'left-connected graph reduction specifications' by LGRS. The recognition problem for LGRS languages is defined as follows:

LGRS Recognition Problem.

Given:	An LGRS $S = \langle \Sigma, \mathcal{C}_N, \mathcal{R}, Acc \rangle$.
Input:	A Σ-graph G.
Output:	Does G belong to $\mathrm{L}(S)$?

Definition 4.12 (linear LGRS)**.** An LGRS $\langle \Sigma, \mathcal{C}_N, \mathcal{R}, c, Acc \rangle$ is *linearly terminating* if for every derivation $G \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} G_n$ on Σ-graphs, $n \le c|G|$. It is *closed* if for every step $G \Rightarrow_{\mathcal{R}} H$ on Σ-graphs, $G \Rightarrow^*_{\mathcal{R}} Acc$ implies $H \Rightarrow^*_{\mathcal{R}} Acc$. A linearly terminating and closed LGRS is a *linear* LGRS.

Theorem 4.21 (linear LGRS time complexity)**.** *The recognition problem is decidable in linear time for a linear LGRS.*

Proof. By Theorem 3.21, if we have a initial graph H_0, set of left-connected rules $\mathcal{R}$, depth bound l, and array of edge enumerations for the left-hand sides of $\mathcal{R}$, then the algorithm MULTISTEP-APPLY defined in §3.4 will return some graph H such that either (1) H is irreducible with respect to set of rules $\mathcal{R}$, and H is the product of a derivation of length $k \leq l$, or (2) H is the product of a derivation of length l.

Let $S = \langle \Sigma, \mathcal{C}_N, \mathcal{R}, c, Acc \rangle$ be a linear LGRS, and G a Σ-graph. By the definition of a linear LGRS, any derivation must be shorter than $c|G|$. Due to the fact that a linear LGRS is closed, any irreducible graph derivable from G will be isomorphic to Acc if and only if $G \in \mathrm{L}(S)$.

We can therefore solve the recognition problem for linear LGRSs by calling the algorithm MULTISTEP-APPLY, with the graph G as the initial graph, the set of LGRS rules $\mathcal{R}_S$ as the reduction rules, and $c|G|$ as the depth bound. The algorithm will return graph Acc if and only if the graph G is in $\mathrm{L}(S)$, ensuring correctness of the algorithm.

By Theorem 3.22, if MULTISTEP-APPLY is called with Σ-preserving rules and an array of enumerations such that each enumeration has a bounded branching factor over all Σ-graphs, then the algorithm terminates in linear time. All Σ-rules preserve membership of the class of Σ-graphs. By Lemma 4.20, the definition of a fast Σ-rule ensures the existence for each left-hand side node of an edge enumeration with the node as its initial node, and a bounded branching factor. Therefore, MULTISTEP-APPLY solves the recognition problem for linear LGRSs in linear time. $\square$

The following example of an LGRS specifies the set of *unrooted* balanced binary trees. This example is one of the GRSs appearing in [4] (with minor alterations to fix an error discovered in the original version).

Example 4.11 (unrooted balanced binary tree GRS)**.** The LGRS BB $=$ $\langle \Sigma_{\mathrm{BB}}, \emptyset,$
$\mathcal{R}_{\mathrm{BB}}, 1, Acc_{\mathrm{BB}} \rangle$ defines the language of unrooted balanced binary trees. The signature is $\Sigma_{\mathrm{BB}} = \langle \{B, U, L\}, \{l, r, c\}, out_{\mathrm{BB}}, in_{\mathrm{BB}} \rangle$, where $in_{\mathrm{BB}}(v, e) = 1$ for all v, e, and

$$out_{\mathrm{BB}}(v, e) = \begin{cases} 1 & \text{if } v = B \text{ and } e \in \{l, r\}, \text{ or } v = U \text{ and } e = c \\ 0 & \text{otherwise} \end{cases}$$

The rule-set $\mathcal{R}_{\mathrm{BB}}$ and accepting graph Acc_{BB} are shown in Figure 4.11. The accepting graph has been slightly changed from the original version where

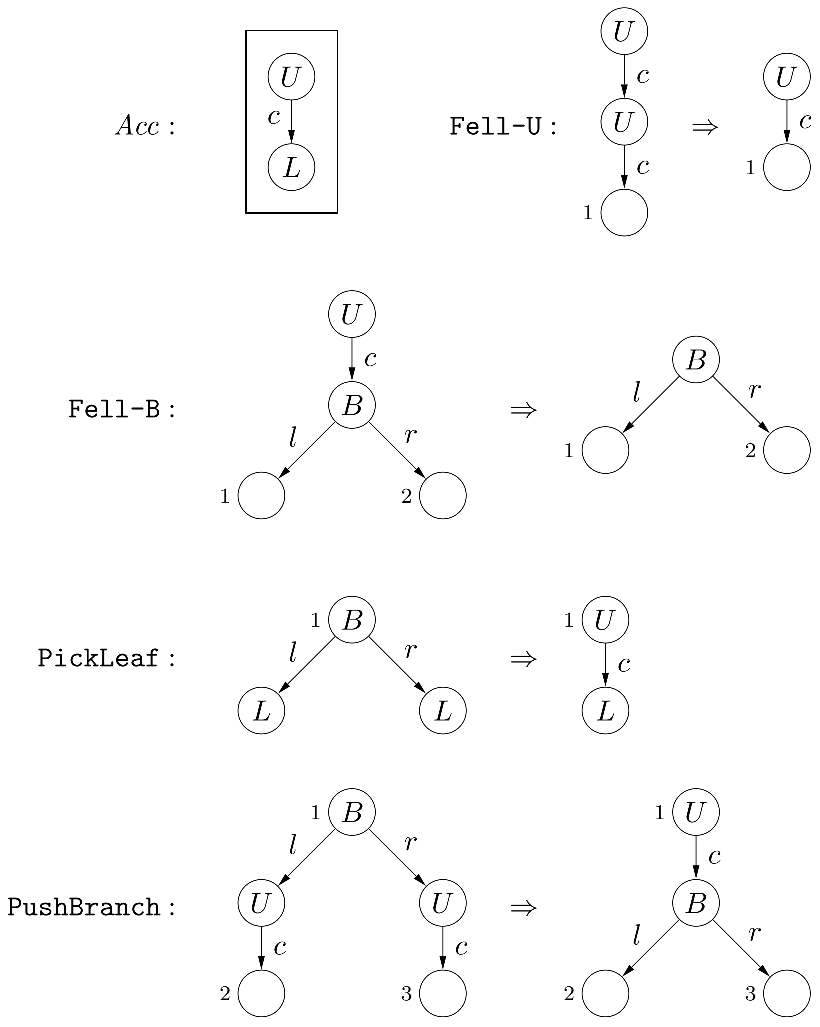


Figure 4.11: Rules and accepting graph for LGRS BB.

it consisted of a single L-node; The original could be restored by adding a special case rule that reduced only the current accepting graph.

Lemma 4.22. *The LGRS* BB *is linear.*

Proof. Given in [4]. □

It is interesting to note that in the source paper [4] the original balanced binary tree GRS was presented simply as an example of a GRS requiring a linear number of reductions to terminate. It is only now, after we have defining the conditions ensuring linear termination for left-connected reduction systems, that it has become clear that the membership problem for this LGRS can be solved in linear time.

4.4 Comparison between LGRSs and RGRSs

We now compare the LGRS and RGRS approaches to reduction. Note that, while our major results up to this point have been about *linear* LGRSs and RGRSs, in this section we compare LGRSs and RGRSs *in general* for expressive power. This is because we have not been able to characterise the classes of linear LGRS and RGRS languages precisely enough to compare their expressive powers.

Definition 4.13 ($\mathbb{L}_R$, $\mathbb{L}_L$, $\mathbb{L}'_R$)**.** $\mathbb{L}_R$ stands for the class of languages for which an RGRS exists and $\mathbb{L}_L$ for the class of languages for which an LGRS exists. $\mathbb{L}'_R$ stands for the class of languages such that for each $L' \in \mathbb{L}'_R$ there exists an $L \in \mathbb{L}_R$ where L' is the language of graphs in L with root-labelled node and attached edges removed.

Clearly the languages in $\mathbb{L}_L$ and $\mathbb{L}_R$ differ trivially in that members of a language in $\mathbb{L}_R$ must include a uniquely-labelled root node.

Proposition 4.23. *There exists a language* L *in* $\mathbb{L}_L - \mathbb{L}'_R$

Proof. As shown in Proposition 4.6, for any RGRS-recognisable language there exists a bound b such that no graph in the language contains more than b connected components. By Proposition 4.8, this result holds even if the single root-labelled node and attached edges are removed. LGRS languages in contrast can contain of an arbitrarily large number of connected components. For example, the language of single nodes (that is, an unbounded number of nodes with no incoming or outgoing edges) can be recognised by an LGRS with a single reduction rule and an empty graph as the accepting graph. ☐

The distinction between the two kinds of GRSs is a result of two major differences between LGRSs and RGRSs: (1) LGRS rules do not have to be applied at the location of a uniquely-labelled root node, and (2) LGRS rules do not have a root node that must be preserved. As a result, even though the accepting graph and the left-hand sides of rules must be connected in an LGRS, connected components can be deleted by reduction.

The converse result also holds: there exist languages recognisable by some rooted GRS that cannot be recognised by any left-connected GRS. This is true because in an RGRS only the root needs to be the initial node of an enumeration satisfying the signature condition. In an LGRS each left-hand side node must be the source of such an enumeration. Intuitively, the existence of a root in the rules of an RGRS restricts the *order of search*, and so prevents some searches that require an unbounded amount of time.

Proposition 4.24. *There exists a language L in $\mathbb{L}_R - \mathbb{L}_L$.*

Proof. Our counter-example is the language of rooted binary DAGs, originally defined in §4.2.3. As shown in that section, this language can be expressed using a linear rooted GRS. It therefore suffices to show that no LGRS exists recognising the language of RBDs.

Let us assume the converse; we have an LGRS RBD_L recognising the language of rooted binary DAGs, as defined in §4.2.3. We call an l-labelled edge that points to an L-labelled node with in-degree one an *l-singleton.* An l-singleton is separating for the singleton's source and target in an RBD. As an L-labelled node in an RBD can have an unbounded number of incoming l-labelled edges, in the signature Σ_{RBD_L} it must hold that $in(L, l) = \perp$.

As a consequence of Lemma 4.7, an l-singleton can only be removed or modified by a rule that contains an l-singleton. We can pick a RBD with an unboundedly large number of l-singletons, meaning some must be removed to reach the fixed-size accepting graph. Consequently RBD_L must include a reduction rule that includes an l-singleton on its left-hand side. Any enumeration over this rule with the L-labelled node as an initial node must include the l-singleton edge. However, as $in(L, l) = \perp$ in the signature, this violates the condition on edge enumerations in the definition of a fast Σ-rule (Definition 4.10). Consequently, no such LGRS RBD_L exists, which proves our result. □

4.5 Developing and validating GRSs

In this chapter we have given several RGRSs and LGRSs, some of which are quite complex. In this section we sketch the process we have used for designing and validating new GRSs.

In most cases, the GRSs presented in this thesis were developed from other preexisting GRSs. The only GRS designed entirely from scratch was the most complex, the rooted binary DAG RGRS given in §4.2.3.

Designing a GRS entirely from scratch was essentially the process of designing an algorithm for reduction. We first sketched a number of potential strategies for reduction. We then tried to implement the strategies as GRSs. Inevitably some of the approaches turned out to be unworkable, or very difficult to understand, or to require a very large GRSs. The most successful strategy was debugged and developed further.

Our major design objectives for a GRS were as follows, roughly in order of importance from most to least.

- Minimise the complexity of the reduction approach.
- Minimise the number of reduction rules.
- Minimise the number of reduction steps to termination.

The construction of a nonterminal class of intermediate graphs goes hand in hand with the development of a GRS. A set of reduction rules define a nonterminal class, so the nonterminal class can be seen as capturing the approach to reduction in the RGRS. In several the of the GRSs presented in this thesis, we designed the class of nonterminal graphs before defining the appropriate rules to reduce the class.

For many LGRSs, a corresponding RGRS can be constructed by augmenting the reduction rules with a root node, and adding extra 'search' rules to move the root to application areas. The RGRSs GG and RBB from §4.2, and CL from Example 4.2 are rooted versions of unrooted GRSs appearing in [4]. In all of these case adding roots resulted in a much larger number of reduction rules.

The major general design difference between the two kinds of GRS is the level of operational control of the reduction process given by the rules. RGRSs are rooted, and the root is the only valid application area. Consequently RGRSs have to explicitly move the root between application areas. This means that the reduction system gives quite a specific ordering on the sequence of reductions applied to the graph. In contrast, in LGRSs the matching algorithm finds an application area automatically, and this often means that the choice of application area is much more non-deterministic.

The strategy for moving the root can be quite complex (see for example RBD, §4.2.3), as the direction of search can be controlled to a fine degree using labels. The main challenge with adding explicit movement of the root is ensuring termination. Just adding arbitrary root-moving operations will often result in non-termination in ill-formed graphs with cycles. In RBB and several other examples termination is achieved by marking visited nodes.

We found it difficult in general to judge informally whether a GRS was correct, as correctness often depended on subtle rule interactions. We have made use of a general methodology for validating RGRS and LGRS against a given graph language. This consists of defining the class of nonterminal intermediate graphs and then showing correctness and linearity as follows:

- Show soundness by showing that the class is closed under the inverses of the reduction rules.
- Show completeness by showing that every member of the nonterminal class can be reduced by some reduction rule.
- Show linear termination by showing that some termination measure exists that is reduced by every reduction rule.

Termination generally was simple to check, as our termination measures depend on simple structural properties of the rules. Validation of soundness involves considering all possible cases for a rule match to show that the nonterminal class is closed under the rule. Similarly completeness involves breaking down the nonterminal class by case to check that every member can be reduced.

In the case of soundness and correctness checking rules is quite tedious, as a large number of cases have to be checked. This suggests that it may be possible to automate this step of the validation process.

Chapter 5

Other approaches to fast graph transformation

In this chapter we compare our work on fast graph transformation and reduction systems to other related approaches. The approach to fast graph transformation we present in Chapter 3 and Chapter 4 improves the worst-case application time of rules or systems of rules. For this reason, we focus on approaches that improve the worst-case application time, and mention heuristic-based approaches only when they relate closely to our work.

The chapter is structured as follows. §5.1 discusses the approaches that have been developed for improving the time complexity of single derivations. Section 5.2 discusses the approaches developed for efficient multi-step derivations. In Section 5.3 we describe other approaches to the problem of efficient graph language recognition, and specifically discuss the relationship of our fast recognition systems to the special reduction systems of [11, 1].

5.1 Efficient direct derivations

Many graph rewriting systems are based on local search algorithms. These begin matching from a single node and then extend the matching morphism to neighbouring nodes and edges. For example, the matching algorithms of GrGen [36, 59] and GP [57] work in this way. Both of these approaches use heuristics to select appropriate start nodes. In the case that the heuristics make a good selection of start node and search order, these approaches achieve the time-bounds given for left-connected and rooted graph transformation.

However, these approaches do not ensure linear-time or constant-time application. The major result of our work on fast graph transformation, given in Chapter 3, has been to take the local search approach and define the restrictions on graphs and rules necessary to ensure linear and constant worst-case time bounds.

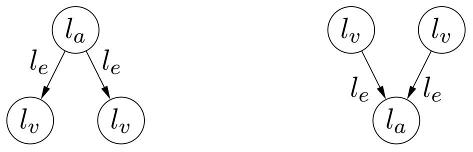
The heuristic approaches often attempts to identify uniquely-labelled nodes to use as start nodes. For example, GrGen [36, 59] attempts to approximate the frequency of node labels, and use less frequent labels as start nodes. In this way, these approaches make use of root nodes.

Other works have considered using explicitly-identified root nodes to enable fast rule application. For example, [33, 34] consider classes of host-graphs representing infinite trees constructed by 'unrolling' from some root. This approach permits linear-time matching with rules as inputs. However, host graphs are severely restricted by the requirement that they are tree-structures.

The approach closest to ours is the work of Dörr [25, 26]. This is a rooted approach based on constructing edge enumerations that avoid so-called strong V-structures.

Definition 5.1 (strong V-structure)**.** A *strong V-structure* is defined as by a triple of graph labels $\langle l_a, l_e, l_v\rangle$. A strong V-structure corresponding to $\langle l_a, l_e, l_v\rangle$ is two edges e_1, e_2 and three nodes v_0, v_1, v_2 where (1) e_1 and e_2 have label l_e, (2) v_1 and v_2 have label l_v, (3) v_0 has label l_a, and (4) either v_0 is the source of e_1, e_2 and v_1, v_2 are their targets, or the reverse.

Example 5.1 (strong V-structure)**.** A strong V-structure corresponding to $\langle l_a, l_e, l_v\rangle$ consists of a subgraph isomorphic one of the following two cases.



Suppose we have a set T of label triples such that all V-structures in the host graph correspond to members of T. Then an edge enumeration is *bypassing* with respect to T if the source label, target label, and edge label of an edge in the enumeration matches no triple in T.

Dörr's application algorithm is a simple enumeration-based matching algorithm, very similar to the algorithm Fast-Match given in §3.3. If the input enumeration bypasses all strong V-structures then the application algorithm terminates in linear time.

Our notion of an enumeration with a bounded branching factor can be seen as a generalisation of Dörr's notion of bypassing enumeration. A bypassing enumeration has a branching factor of 1 under our approach. Dörr also requires unique root nodes. For this reason, rules with a bypassing enumerations can be applied in constant time using the rooted application algorithm given in §3.3.

Our notion of branching is more general because rules under the V-structure approach are deterministic, while ours may be non-deterministic. Furthermore, all host-graphs under Dörr's approach are assumed to belong to a graph grammar which can be analysed for possible V-structures. We don't require any generation mechanism for host graphs, and instead have developed simple syntactic conditions that ensure the existence of an edge-enumeration bound.

We have extended the bounded enumeration approach in ways that Dörr does not consider. Dörr's requirement for constant time application means that he considers only rooted systems, while we have extended the notion of enumerations with a bounded branching factor to unrooted left-connected systems. By modifying the algorithm of [11], we have also shown that sequences of such unrooted rules can be applied in linear time, even if individual rule applications also require at worst linear time.

Finally, Dörr differs from our work in the uses he makes of his rules with improved application times. He applies rooted rules to general computation systems based on graph transformation (his example in [26] is the implementation of a functional programming language), while we have applied our work on fast graph transformation to the recognition problem for graph transformation systems.

5.2 Efficient multi-step derivation

Our algorithm Multistep-Apply, which solves the multi-step graph transformation problem for unrooted rules, is derived from the multi-step algorithm given by Bodlaender and de Fluiter in [11].[1]

Bodlaender and de Fluiter's version of the algorithm maintains a list of search locations in the same way that ours does. However, it ensures that each individual search terminates in bounded time by bounded adjacency search. The graph is represented by an adjacency list representation, and the algorithm searches only within a bounded distance from previously-matched edges to find more matching edges. This means that the correctness of the algorithm depends on the assumption that a bounded search is guaranteed to find a match.

This assumption is discharged by applying the algorithm only to so-called special reduction systems. However, their definition of such reduction systems is semantic. It cannot be checked by examining the syntactic structure of the rewrite rules. This is the main difference between our version of the algorithm and the version of [11]. We have defined simple sufficient syntactic conditions ensuring that our algorithm terminates in linear time.

Bodlaender and de Fluiter's algorithm is in turn based on the algorithm given in [1]. This version of the algorithm searches only from interface-graph nodes. The conditions on rules which ensure linear-time termination are syntactic, but they require the absence of edges between interface-graph nodes – a severe restriction. In contrast, our algorithm requires only the existence of sufficient edge enumerations with bounded branching factor, as ensured by our syntactic conditions. Our conditions on labels and node degree also seem more natural than prohibiting all edges between interface nodes.

Our algorithm Multistep-Apply shares information between rewriting steps. As such, it is similar to the approach of [77]. Under this approach, all potential matchings between a rule and the current graph are recorded in a database. After rewriting, the matches in the database must be updated with the new structure of the graph.

Our algorithm is also similar to the RETE algorithm [30], an algorithm for matching patterns to objects which can be used for graph matching. As with [77], RETE maintains a structure during a derivation represents all partial matches.

Like our algorithm, both of these approaches use an intermediate struc-

[1]This algorithm and the algorithm of [1] are used in approaches to graph recognition. In the next section we compare these approach to our work on recognition systems.

ture to amortise the cost of matching over a derivation. The intermediate data-structures recorded by these two approaches are quite complex, meaning that the cost of maintaining the structures may be very high. In contrast, our algorithm maintains only a list of potential locations, and updating is therefore guaranteed to be much cheaper.

Unlike our algorithm, these approaches apply to unrestricted graph transformation systems, meaning that they only aim to improve the average performance of graph matching. The results given in [77] are experimental; in contrast, we ensure an improved worst-case time complexity.

5.3 Efficient recognition and special reduction systems

This section compares our fast left-connected and rooted graph reduction systems to other approaches that have been developed for efficient recognition by reduction. Our main comparison is with the approach to language recognition described by Bodlaender et al. in [11] and Arnborg et al in [1]. This work is based on Courcelle's earlier work on monadic second-order logic [17], and the prior work on the treewidth of graphs [10].

The formulations in [11, 1] are slightly different, but both give the following: (1) An algorithm that solves the language recognition problem. (2) A class of so-called special reduction systems which ensure that the recognition algorithm terminates in linear time. (3) A condition on graph languages ensuring the existence of a corresponding special reduction system.

Suppose we have a language of graphs that can be defined as the intersection of (1) the set of graphs of treewidth up to some bound, and (2) a language that can be defined by some formula in monadic second-order logic. Then [11] and [1] show that a special reduction system can be automatically constructed, and consequently that membership of these languages is decidable in linear time.

Our approach can recognise in linear time languages which violate both of these sufficient conditions. The language of balanced binary trees is defined under the RGRS approach in §4.2.1 and LGRS approach in §4.3, but we show in Appendix A that this language is not expressible in monadic second-order logic[2]. The language of grid-graphs is defined using the RGRS

[2]It is not clear in [1] whether their sufficient condition requires monadic second-order

approach in §4.2.2 and can be expressed in a similar way using an LGRS, but it has an unbounded treewidth. This is because an $n \times n$ grid graph has treewidth n [18, p324].

The formulations of a special reduction system given in these two papers are slightly different. Our aim in our work has been to construct syntactic conditions on rules and graphs ensuring improved termination times. For this reason, we focus on the syntactic SRS formulation of [1], rather than the semantic formulation of [11].

On the one hand, special reduction systems as given in both [11] and [1] are of incomparable power to both LGRSs and RGRSs. This is because the notion of a reduction system given in these papers lacks non-terminal labels, which places quite a severe restriction on their expressive power. To prove this, we give an example of a language inexpressible without non-terminals.

Definition 5.2 (complete binary tree)**.** We describe a graph as a *complete binary tree* (CBT) if it is a balanced binary tree where all of the branch nodes are binary.

The language of all complete binary trees can expressed by both LGRSs and RGRSs. The rooted BBT example given in §4.2 and the unrooted BBT example given in §4.3 can both be adapted to recognise CBTs by simply turning the terminal U-label into a non-terminal label.

However, the language of complete binary trees cannot be expressed in a special reduction system, or indeed any reduction system under the formulation in [11] and [1]. We prove this result by a simple size argument. This proof is a restatement of the proof of Theorem 5 in [4].

Proposition 5.1. *No reduction system as defined in [1] exists which can recognise the language of complete binary trees.*

Proof. By the definition of a reduction system in [1], a reduction system R for property P must be 'safe', meaning whenever $G \xrightarrow{R} G'$, then $P(G) \Leftrightarrow P(G')$. Assume we have R_c, a reduction system for CBTs. Let s be the maximum of the sizes of the left-hand sides of rules in R_c. Let G be a CBT of depth k chosen such that $2^{(k-1)} > s$. Each reduction by any rule in R_c can remove at most s nodes from the graph G. However, every CBT smaller

logic with or without an incidence predicate. In Appendix A we prove that balanced binary trees are inexpressible in both versions of MSOL.

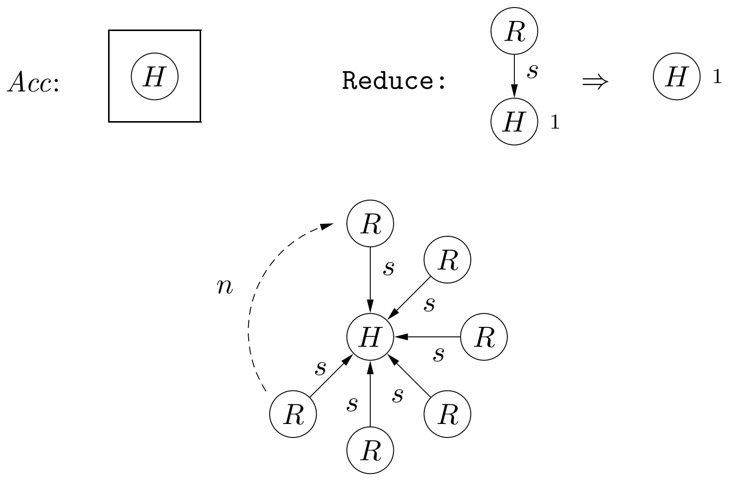


Figure 5.1: A special reduction system which recognises the language of star, and an example star.

than G is at least $2^{(k-1)}$ nodes smaller; every larger CBT is at least 2^k nodes larger. Therefore R_c does not exist. □

On the other hand, the special reduction systems of [1] can also express languages which are inexpressible under both linear and rooted GRSs. We call a graph a *star* if it consists of a single H-labelled 'hub' node, and an unbounded number of R-labelled 'rim' nodes, and for each rim node there exists a single s-labelled 'spoke' edge with the rim node as its source and the unique hub node as its target.

A special reduction system exists which recognises the language of star. An example of a star, and the corresponding special reduction system is shown in Figure 5.1. For clarity we use our normal notation for a rule rather than the notation used in [1], but the two notations are equally expressive.

This language of stars is not in $\mathbb{L}'_R$. This means that it cannot be recognised by an RGRS, no matter where the uniquely-labelled graph root is added to the graph. To prove this result we need the following lemmas.

Lemma 5.2 (moving the root). *Let G be a rooted graph with a separating edge l in the same component as the root. Let v be the node attached to l that is in the same component as the root in $G \setminus l$. Let $G \Rightarrow_r H$ be a derivation such that l is not matched, and let l' be the descendant of l in H. Then l' is separating in H, and the descendant of v is in the same component as the* *root in $H \setminus l'$.*

Lemma 5.3 (roots and separating edges)**.** *Let $r = \langle L \leftarrow K \rightarrow R\rangle$ be a rooted rule with matching enumeration $e_1, \dots, e_n$. Let G be a rooted graph, and let $G \Rightarrow_r H$ be a direct derivation where edge e matched by $l \in E_L$ is separating in G. Let v_u be the node attached to l not in the same component as the root in $G \setminus l$. Let l be the ith element of the edge enumeration. Then there can be no l_j in the enumeration such that $j < i$ and l_j is incident to v_u.*

Proposition 5.4. *No RGRS exists which recognises the language of stars, even with the addition of an arbitrarily-located root node.*

Proof. Suppose we have an RGRS which recognises the language of stars with an arbitrarily-attached ϱ-labelled root node. All rules in the RGRS must satisfy the conditions on rooted Σ-rules given in Def. 4.1.

Now we can pick an arbitrarily-large star H with the root attached to some node. To reach the finite-size accepting graph from graph H, we must therefore be able to delete some of the rim nodes and spoke edges. To delete or modify an edge it must be matched to the left-hand side of some rule during the reduction. However, any such rule is not a rooted Σ-rule. There are two cases.

Case 1: The root is attached to the hub node. All the spoke edges are separating for the hub node and the corresponding rim node. By Lemma 5.2 and Lemma 5.3, to match any of the spoke edges, the spoke edge must be matched in some left-hand side by an edge enumeration which matches from the hub towards the rim. Because the number of edges outgoing from the hub is unbounded, such an edge enumeration must violate the condition on rooted Σ-rules.

Case 2: The root is attached to a rim node. By Lemma 5.2 and Lemma 5.3, to match and remove more than one spoke edge must again require an edge enumeration which extends from the hub to the rim. Once again, this violates definition 4.1. Therefore no such RGRS exists. □

The proof that no LGRS exists for star graphs is easier, because any rule can be applied non-deterministically anywhere in the graph, rather than just at the root.

Proposition 5.5. *No LGRS exists which recognises the language of stars.*

Proof. Suppose we have an LGRS which recognises the language of stars. Then all rules in the LGRS must conform to Def. 4.10 for fast Σ-rules. Let us pick a star G which is larger than the accepting graph, with hub degree $d > b$. To recognise the accepting graph we must therefore remove some of the spoke edges in the graph. By Lemma 4.7 to remove any spoke edge there must exist a rule which includes a separating s-labelled spoke edge. But, by the same argument as used in Proposition 4.24 to argue that the language of RBDs is not LGRS-expressible, any such rule must violate Def. 4.10. Therefore no such LGRS exists. $\square$

The restriction on special reduction systems imposed by the absence of non-terminal symbols can be lifted by modifying the definition of an SRS to permit the use of non-terminals. We denote by $\mathbb{L}_S$ the class of graph languages definable by special reduction systems extended with non-terminal symbols.

Conjecture. We believe that neither of the classes $\mathbb{L}_L$ and $\mathbb{L}_R$ are subsets of the class $\mathbb{L}_S$ of languages definable by special reduction systems extended with non-terminal symbols. This conjecture seems plausible because special reduction systems prohibit edges between interface nodes, which severely restricts the structure of rewrite rules. Intuitively, it seems that this must restrict the formal power of the system. However, at present we have not been able to prove this.

Part III

Graph grammars and separation logic

Chapter 6

Semantics of formulas and grammars

Hyperedge-replacement grammars, which are defined formally in §2.3, are a natural extension of context-free string grammars to the world of hypergraphs. Hyperedge replacement productions are context-free, meaning that they only replace a single non-terminal hyperedge, rather than a subgraph of the target graph. The language defined by such a grammar is the class of graphs derivable from some initial graph.

Separation logic is a recently-developed logic for specifying the properties of heaps. It makes use of a *separating* conjunction to enable local reasoning. Loosely speaking, this separating conjunction allows a logical formula to specify the spatial relationships between assertions in the heap. Formulas can specify properties that are true in disjoint portions of the heap.

A separation-logic formula is normally seen as defining a class of satisfying states, each of which consists of a heap, recording the relation between memory locations, and a stack, recording program variables. States are graph-like structures, with heap locations as 'nodes' and pointers as 'edges'. So both hyperedge replacement grammars and separation logic formulas define classes of graph-like structures.

Separation logic formulas are frequently used with recursively-defined predicates. These formulas with recursive predicates resemble hyperedge replacement grammars quite closely, in that a predicate stands for a class of satisfying heaps attached to the larger separation-logic state through its arguments. Furthermore, the separating property enforced by the separating conjunction resembles the context-free property of hyperedge-replacement grammars, in that separated predicate calls can only share a fixed number of locations. That is to say, predicate calls can't 'overlap'.

In this part of the thesis, we show that this intuition is correct: restricted hyperedge replacement grammars and formulas in a restricted fragment of separation logic are indeed related. We do this by defining mappings in both directions between these restricted grammars and formulas in the fragment. We show that the two mappings g and s are correct in the sense that they are semantics-preserving with respect to a bijective mapping α between separation-logic states and graphs. The preservation of semantics that we are looking for in the mappings can be expressed by the requirement that the following pair of diagrams commute:

$$\begin{tikzcd}
\text{Grammar} \arrow[r, "\text{language}"] & \text{Graphs} \\
\text{Formula} \arrow[u, "g"] \arrow[r, "\text{satisfied by}"'] \arrow[ur, phantom, "="] & \text{States} \arrow[u, "\alpha"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{Grammar} \arrow[r, "\text{language}"] \arrow[d, "s"'] \arrow[dr, phantom, "="] & \text{Graphs} \arrow[d, "\alpha^{-1}"] \\
\text{Formula} \arrow[r, "\text{satisfied by}"'] & \text{States}
\end{tikzcd}$$

Intuitively, if this correctness property holds, then any class of structures that can be expressed in one domain can also be expressed in the other. This means that our fragment of separation logic is of equivalent expressive power to our restricted form of hyperedge-replacement.

This result has the interesting consequence that some of the theoretical results for hyperedge replacement languages, such as the inexpressibility results, can be imported wholesale into our fragment of separation logic. For example, the languages of red-black trees, balanced binary trees and grid graphs are all known to be inexpressible by any hyperedge-replacement grammar. Therefore no formula exists in our fragment specifying any of these structures.

The chapter is structured as follows. In Section 6.1 we define the fragment of separation logic we use and formally define its semantics. In Section 6.2 we show that all formulas in this fragment can be *flattened* into a simpler form without nested recursive definitions. In Section 6.3 we define a class of heap-graphs that correspond to separation-logic heaps, and define a mapping between heap-graphs and heaps. Finally, in Section 6.4 we define a class of graphs corresponding to heaps, and show that any grammar can be normalised to one producing only heap-graphs.

6.1 Separation logic syntax and semantics

Separation logic [46, 69] extends first-order logic with so-called *spatial* assertions that describe the state of the heap. Spatial assertions in separation logic are *local*, meaning that they describe properties that hold in disjoint areas of the heap, and that they restrict the sharing between regions of the heap. Spatial assertions permit *local reasoning*, meaning that assertions can be safely combined without explicit restrictions on sharing.

This section specifies the syntax and semantics of a fragment of separation logic. The semantics for separation logic given in this section are based on those appearing in [74, 52], with modifications that we describe. The definition of satisfaction given in Figure 6.2 is based on [74], while the semantics of the recursive let is based on [52].

6.1.1 Separation logic model

Separation logic formulas primarily express properties of heaps. We use a *cons-model* for the heap, where each location in the heap points to a pair of elements. We have chosen this model because it is simple enough to easily define our translation, but rich enough to define interesting structures such as trees and lists.

Other models can be used as the domain for separation logic. In §8.2 we consider an extended heap-model based on tuples. More complex models are also available, such as the *RAM model*, which records locations as integers and so permits pointer arithmetic [69].

The RAM model is too complex to map to hyperedge replacement in a semantics-preserving manner, because the relationship between adjacent cells is impossible to capture using edges. In order to model it would need to be able to create edges between arbitrary members of the heap, to model locations that were accidentally adjacent. Doing this would require that we maintain nonterminal edges between all heap locations, which would break the locality of hyperedge replacement.

Other heap models such as the one used in [74] allows integer values in the heap, but prohibits pointer arithmetic; Here we abstract away to deal only with the structure of the heap. In this, the model we use is similar to the model used for *symbolic heaps* that form the basis of recent separation-logic-based program analysis [6, 20]. In §8.3.2 we look at the relationship between our fragment of separation logic and the symbolic heaps fragment.

Assumption 6.1. Let Loc be a countably infinite set of *locations*. The set of *elements*, $\mathrm{Elem} = \mathrm{Loc} \cup \{\mathsf{nil}\}$, consists of locations and a special atomic value nil. We also assume an atomic unknown value $\boxslash$.

Definition 6.1 (heap)**.** A *heap* $h\colon \mathrm{Loc} \rightharpoonup (\mathrm{Elem} \times \mathrm{Elem}) \cup \boxslash$ is a finite partial function mapping heap locations to pairs of elements or $\boxslash$. We use $\mathcal{HE}$ to stand for the set of all possible heaps.

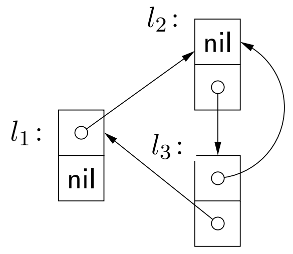
The *image* of a heap h, written $\mathrm{img}(h)$, is the set of locations held in pairs in the heap, that is $\mathrm{img}(h) = \{v \in \mathrm{Loc} \mid \exists v'.\, h(v') = (v, \_) \vee h(v') = (\_, v)\}$. We describe a location l in a heap h as *dangling* if $l \in \mathrm{img}(h)$ and $l \notin \mathrm{dom}(h)$.

Note that in general $\mathrm{img}(h) \setminus \mathrm{dom}(h)$ may be nonempty, meaning elements in h may refer to locations outside the heap.

The inclusion of the unknown value $\boxslash$ is a departure from the standard model given in [69]. In the semantics of formulas, we want to avoid locations that are in the image of a heap but not in the heap domain. Locations that could dangle in this way are avoided by mapping representative locations to the value $\boxslash$. See p. 117 for more on this.

Example 6.1 (heap)**.** The heap h shown below associates three locations, l_1, l_2 and l_3 with pairs of values. Other locations are undefined.

$$\begin{aligned} h = \{\ & l_1 \mapsto \langle l_2, \mathsf{nil}\rangle, \\ & l_2 \mapsto \langle \mathsf{nil}, l_3\rangle, \\ & l_3 \mapsto \langle l_2, l_1\rangle\ \} \end{aligned}$$



We represent this heap by the diagram shown on the right-hand side. Undefined locations are not shown.

We consider heaps as distinct only up to isomorphism. In practice, this means that the underlying set of locations can be substituted for each other without changing the meaning of the heap. This relies on the implicit assumption that only the *structure* of the heap is important, as encoded by the relation between heap locations. This is sensible when we are interested in reasoning about the abstract properties of separation logic. It may break down however if we are dealing with more concrete heaps with distinguishable locations.

6.1.2 Separation logic fragment

Our mapping between separation-logic formulas and hyperedge replacement grammars operates over a fragment of the full separation logic as given in [69][1]. This includes separating conjunction ($*$), the recursive let ($\mathsf{let}\ \Gamma\ \mathsf{in}\ P$), 'points-to' ($\mapsto$), disjunction, existential quantification, elements from the set Elem, predicate names σ from a set Pred, and variable names $x, x_1, .., x_n$ from a set Var.

The class $\mathcal{SL}$ of all separation-logic formulas in this fragment is defined by the following abstract syntax, with initial symbol F. Given a set of variables K we write F_K for a formula that contains only free variables in K – see the discussion on p. 121.

$$
\begin{aligned}
F &::= \mathsf{emp} \mid x \mapsto V, V \mid F * F \mid \exists x.\, F \mid F \vee F \mid \\
&\quad\;\; \sigma(V, \ldots, V) \mid \mathsf{let}\ \Gamma\ \mathsf{in}\ F \\
\Gamma &::= \sigma(x_1, \ldots, x_n) = F_{x_1, \ldots, x_n} \mid \Gamma, \Gamma \\
V &::= x \mid \mathsf{nil}
\end{aligned}
$$

The precedence order for nested operators, given from strongest to weakest, is: $*$, $\vee$, $\exists$, $\mathsf{let}\ \Gamma\ \mathsf{in}\ F$.

The simplest assertion is emp, which asserts that the heap is empty. A heap is empty if the heap's address function is undefined for all addresses. Next, the points-to assertion, written $a \mapsto e_1, e_2$, asserts that the heap-function defines the single address a as pointing to the tuple (e_1, e_2).

Separation logic's most important extension to first-order logic is the separating conjunction, written '$*$'. A heap h satisfies $P_0 * P_1$ if there exist disjoint subheaps satisfying P_0 and P_1, such that h is the composition of the two subheaps.

Separation logic extends first-order logic with several constructs permitting reasoning about heaps. The most basic of these is the *points-to assertion*

[1] We generally describe our logic as a *fragment* of full separation logic. However it is not a true fragment, as our recursive let is more general than the notion of recursion in most formulations separation logics. The exception to this are the separation logics given in [52, 74], both of which include a general operator for recursion.

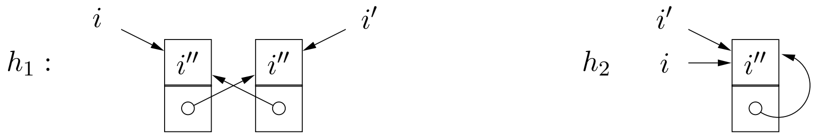


Figure 6.1: Heap h_1 satisfies the formula $\exists xyz.\,((x \mapsto z, y) * (y \mapsto z, x))$ with x instantiated with i, y with i' and z with i''. Heap h_2 does not satisfy the formula, due to sharing.

$a \mapsto e_1, e_2$, 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 6.1, but not by heap h_2 due to sharing.

Most of the constructs in this fragment are either conventional first-order logical operators, or are conventional separation-logic operators as introduced in [69]. The only unusual construct is the let-construct, which is used to define recursive 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 6.2 (binary tree formula)**.** The separation-logic formula for the class of binary trees is given below. It is satisfied by any heap containing a binary tree with nil leaves. The predicate $bt(x)$ defines the class of heaps containing a binary tree with nil-valued leaves and root at x. We abstract away from a particular root node by existentially quantifying x.

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

In this predicate each node of the tree holds a pair of values. The leaves of the tree consist of pairs of nil-values. $bt(x)$ is satisfied if either x points to a location holding a pair of nil-values, or if x points to a pair of locations, both of which also satisfy bt. The separating conjunction $*$ between the branch and the two subtrees differs from conventional conjunction in that it prevents its conjuncts from overlapping in the heap. This enforces the tree property by preventing sharing between the subtrees.

Our fragment of separation-logic has been selected to remove features that are inexpressible in hyperedge-replacement grammars. Notably the conjunction, negation and separating implication ($-\!\!*$) operators are omitted.

Conjunction and negation are omitted because the class of HR-expressible languages is not closed under either language intersection or negation [39]. Negation is also omitted in order to ensure the existence of a least fixed-point in our semantics for let-statements. Separating implication is omitted because a formula with separating implication can be used to 'match' and 'remove' a portion of the heap; operations that are fundamentally context sensitive, and so outside the domain of hyperedge replacement grammars.

Section 8.1 discusses in more detail our reasons for omitting these constructs, and proves that some of the omitted constructs are inexpressible under hyperedge-replacement.

6.1.3 Fragment semantics

To define the semantics of a formula in our fragment, we define the satisfaction relation '$\models$'. Satisfaction is defined for a heap h, a variable interpretation i and a predicate interpretation η.

Assumption 6.2. In the following, let Var be a countably infinite set of logical variable names and let Pred be a countably infinite set of predicate names.

Definition 6.2 (variable interpretation)**.** A *variable interpretation* defines the meaning of variables in a separation logic state. A variable interpretation $i\colon \mathrm{Var} \to \mathrm{Elem}$ is a partial function mapping variables to heap elements. A variable interpretation i defines an interpretation function $[\![-]\!]i\colon\ \mathrm{Var} \cup \{\mathsf{nil}\} \to \mathrm{Elem}$, where $[\![\mathsf{nil}]\!]i = \mathsf{nil}$ and $[\![v]\!]i = i(v)$ for all $v \in \mathrm{Var}$.

Example 6.3 (variable interpretation)**.** Given variables x, y and z and locations l_1 and l_2, we can define the following variable interpretation.

$$i = \{x \mapsto l_1, y \mapsto \mathsf{nil}, z \mapsto l_2\}$$

Definition 6.3 (predicate interpretation)**.** A *predicate interpretation* defines the semantics of a recursively defined predicates as a class of satisfying heaps with attachment points. A predicate interpretation η: Pred $\rightharpoonup$ $Pow(\text{Loc}^* \times H)$ is a partial function mapping predicate names to (possibly infinite) sets of pairs, consisting of a heap and a sequence of locations in the heap.

Example 6.4 (predicate interpretation)**.** Let P_1 and P_2 be names in Pred. Then we can define the following (finite) predicate environment η.

$$\begin{aligned}\eta = \{\ & P_1 \mapsto \{\langle [l_1], \{l_1 \mapsto \langle l_2, \mathsf{nil}\rangle, l_2 \mapsto \langle l_1, \mathsf{nil}\rangle\}\rangle\},\\ & P_2 \mapsto \{\langle [l_1, l_3], \{l_1 \mapsto \langle l_2, \mathsf{nil}\rangle, l_2 \mapsto \langle \mathsf{nil}, l_2\rangle, l_3 \mapsto \langle l_2, l_1\rangle\}\rangle\}\ \}\end{aligned}$$

η associates a single pair with P_1, where the list of locations contains a single locations. It also associates and a single pair with P_2, where the list of locations contains two locations.

The variable interpretation i_0 and predicate interpretation η_0 are *empty* interpretations, meaning that $\text{dom}(i_0) = \text{dom}(\eta_0) = \emptyset$. We say that a heap h *satisfies* a formula F iff $h, i_0, \eta_0 \models F$.

The definition of $\models$ is given in Figure 6.2. We make use of several extra concepts in this definition.

Square brackets are used to denote the function update operator. For example, $f' = f[x \mapsto y]$ is defined so $f'(a) = f(a)$ when $a \neq x$ and $f'(a) = y$ otherwise. If $\vec{a} = a_1, \ldots, a_n$ and $\vec{b} = b_1, \ldots, b_n$ are vectors, we write $f[\vec{a} \mapsto \vec{b}]$ as shorthand for $f[a_1 \mapsto b_1] \ldots [a_n \mapsto b_n]$. Finally, we allow $[\vec{a} \mapsto \vec{b}]$ to stand for $f_\perp[\vec{a} \mapsto \vec{b}]$, where $f_\perp$ is the empty function that has $\text{dom}(f_\perp) = \emptyset$.

In defining satisfaction for the recursive let formula $\mathsf{let}\ \Gamma\ \mathsf{in}\ P$, we need to access the individual predicate definitions in Γ. We use $\text{dom}(\Gamma)$ to stand for a set containing the names of the 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.

In defining the fixed-point function for recursive let formulas, we use a λ notation to define simple functions. The definition $\lambda x. Y$ stands for the

$$
\begin{array}{lll}
h, i, \eta \models \mathsf{emp} & \text{iff} & \mathrm{dom}(h) = \emptyset \\[1ex]
h, i, \eta \models (x \mapsto V_1, V_2) & \text{iff} & \mathrm{dom}(h) = \{[\![x]\!]i, [\![V_1]\!]i, [\![V_2]\!]i\} \text{ and} \\
 & & \quad h([\![x]\!]i) = ([\![V_1]\!]i, [\![V_2]\!]i) \text{ and} \\
 & & \quad h([\![V_1]\!]i) = h([\![V_2]\!]i) = \boxslash \\[1ex]
h, i, \eta \models P * Q & \text{iff} & \text{there exist } h_0, h_1 \text{ such that } h_0 \cdot h_1 = h \\
 & & \quad \text{and } h_0, i, \eta \models P \text{ and } h_1, i, \eta \models Q \\[1ex]
h, i, \eta \models P \vee Q & \text{iff} & h, i, \eta \models P \text{ or } h, i, \eta \models Q \\[1ex]
h, i, \eta \models \exists x.\, P & \text{iff} & \text{exists } v \in \mathrm{Loc} \\
 & & \quad \text{such that } h, i[x \to v], \eta \models P \\[1ex]
h, i, \eta \models \sigma(V_1, \ldots, V_n) & \text{iff} & (([\![V_1]\!]i, \ldots, [\![V_n]\!]i), h) \in \eta(\sigma) \\[1ex]
h, i, \eta \models \mathsf{let}\ \Gamma\ \mathsf{in}\ P & \text{iff} & h, i, \eta\,[\beta \mapsto k(\beta)]_{\beta \in \mathrm{dom}(\Gamma)} \models P
\end{array}
$$

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

Figure 6.2: Definition of satisfaction for separation logic.

function taking a single input value and substituting all instances of x in Y with the input value.

Definition 6.4 (heap fusion)**.** The heap $h_0 \cdot h_1$ denotes the *fusion* of the two heaps h_0 and h_1. This is defined only if the set of defined, non-$\boxslash$ locations for the heaps are disjoint. That is, no location l exists such that both $h_0(l) \in (\mathit{Elem} \times \mathit{Elem})$ and $h_1(l) \in (\mathit{Elem} \times \mathit{Elem})$.[2] The fusion overwrites $\boxslash$-values with known values when fusing the two heaps. Fusion is defined as follows, *if* the two heaps' non-unknown domains of definition are disjoint.

$$
h_0 \cdot h_1(l) = \begin{cases} h_i(l) & \text{if } h_i(l) \in (\mathit{Elem} \times \mathit{Elem}) \wedge i \in \{0, 1\} \\ \boxslash & \text{if } h_i(l) = \boxslash \wedge h_j(l) \in \{\boxslash, \bot\} \wedge i, j \in \{0, 1\} \wedge i \neq j \\ \bot & \text{otherwise} \end{cases}
$$

[2]The semantics presented in [74, 52] define this fusion as simple disjoint union, which our definition reduces to in heaps without $\boxslash$-valued addresses. Our introduction of the unknown value $\boxslash$ makes the definition heap fusion a little more complex however.

Example 6.5 (heap fusion)**.** Let heaps h_1, h_2 and h_3 be defined as follows.

$$\begin{aligned} h_1 &= \{l_1 \mapsto \langle \mathsf{nil}, l_3\rangle, l_3 \mapsto \langle \mathsf{nil}, l_2\rangle\} \\ h_2 &= \{l_2 \mapsto \langle l_3, \mathsf{nil}\rangle, l_3 \mapsto \boxslash\} \\ h_3 &= \{l_1 \mapsto \langle \mathsf{nil}, l_2\rangle\} \end{aligned}$$

Then the fusion of h_1 and h_2 is defined as follows.

$$h_1 \cdot h_2 = \{l_1 \mapsto \langle \mathsf{nil}, l_3\rangle, l_2 \mapsto \langle l_3, \mathsf{nil}\rangle, l_3 \mapsto \langle \mathsf{nil}, l_2\rangle\}$$

Note that the unknown value for l_3 in h_2 is overwritten by the definite value for l_3 in h_1. The fusion of h_1 and h_3 is undefined, as location l_1 has a non-$\boxslash$ value in both heaps.

The fixed-point definition for recursive predicates is defined with respect to the following ordering on predicate interpretations.

Definition 6.5 ($\leq_p$)**.** We define the relation $\leq_p$ over predicate interpretations with a given set of predicate names by simple set-inclusion for each symbol. So $\eta_1 \leq_p \eta_2$ if for every predicate symbol σ in the domain of η_1, it holds that $\eta_1(\sigma) \subseteq \eta_2(\sigma)$.

Example 6.6 ($\leq_p$)**.** Suppose h_1 and h_2 are heaps, σ_1 and σ_2 are predicate names, and l_1, l_2 are locations. We define the predicate interpretations η_1, η_2 and η_3 as follows.

$$\begin{aligned} \eta_1 &= \{\sigma_1 \mapsto \{\langle [l_1, l_2], h_1\rangle\}, \sigma_2 \mapsto \emptyset\} \\ \eta_2 &= \{\sigma_1 \mapsto \{\langle [l_1, l_2], h_1\rangle, \langle [l_1, l_2], h_2\rangle\}, \sigma_2 \mapsto \{\langle [l_1, l_2], h_1\rangle\}\} \\ \eta_3 &= \{\sigma_1 \mapsto \{\langle [l_1, l_2], h_1\rangle\}, \sigma_2 \mapsto \{\langle [l_1, l_2], h_1\rangle, \langle [l_1], h_2\rangle\}\} \end{aligned}$$

Then $\eta_1 \leq_p \eta_2$ holds, but η_2 and η_3 are incomparable.

6.1.4 Differences from standard separation logic

The semantics of most of the non-recursive operators in our fragment are close enough to the intuitive behaviour described in §6.1.2 that we will not explain them on a case-by-case basis. The exceptions are in the handling of dangling values, the semantics of predicate definitions, and the scope of variables, all of which differ substantially from the conventional semantics presented in [74, 52].

Handling of dangling values

Normally in separation logic, dangling values (for example, y and z in the formula $\exists xyz.\, x \mapsto y, z$) are satisfied by any arbitrary value assignment. By contrast, under our semantics a dangling heap locations must contain the special *unknown* heap element $\boxslash$, rather than an arbitrary heap element. This means that a heap that satisfies the formula $x_1 \mapsto x_2, x_3$ should define a value for three locations, two of which consist of $\boxslash$.

The unknown value is included in our semantics to avoid the problem of modelling dangling pointers in a hyperedge replacement grammar. Without the unknown value, a grammar would have to construct every possible instantiation of dangling values by attaching them to arbitrary heap nodes.

The addition of $\boxslash$ does not seem too radical a modification. The semantics is still sufficiently close to the standard semantics to be of general interest. If a formula contains a dangling value, instead of an arbitrary value the value in a satisfying heap must be $\boxslash$. In formulas without dangling variables, the semantics are identical.

Semantics of predicate declarations

Our semantics for recursive predicates differs from the semantics given in [52] (on which it is based) and we will therefore examine it in detail. A predicate interpretation (defined in Def. 6.3) records the meaning of predicates, by associating each predicate name in Pred with a sets of pairs. Each of these pairs consists of a heap, which is a satisfying instance of the predicate, and a sequence of locations, which form the points to which the arguments of the predicate are bound.

The satisfaction relation for an instance of a predicate σ in the predicate interpretation η is defined as follows.

$$h, i, \eta \models \sigma(V_1, \ldots, V_n) \quad \text{iff} \quad ((\llbracket V_1 \rrbracket i, \ldots, \llbracket V_n \rrbracket i), h) \in \eta(\sigma)$$

A predicate $\sigma(x_1, \ldots, x_n)$ is satisfied by a heap h and interpretations i, η if there exists a pair $((l_1, \ldots, l_n), h') \in \eta(\sigma)$ such that (1) $h = h'$ and (2) the sequence of locations $l_1, \ldots, l_n$ correspond to locations $\llbracket x_1 \rrbracket i, \ldots, \llbracket x_n \rrbracket$.

New predicates are defined using a recursive let-statement. A formula $\mathsf{let}\ \Gamma\ \mathsf{in}\ P$ is satisfied by a heap h in variable interpretation i and predicate interpretation η, if h satisfies P in a new predicate interpretation $\eta[\beta \mapsto k(\beta)]$ recording the semantics of the predicates defined in Γ.

The predicate interpretation $\eta[\beta \mapsto k(\beta)]$ is defined over all of the predicate names defined in Γ as the value of newly-constructed predicate interpretation k. This k is defined as the least fixed-point with respect to $\leq_p$ of the following function, which we call $f_{\eta,\Gamma}$.

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

The satisfaction relation for $\mathsf{let}\ \Gamma\ \mathsf{in}\ P$ is then as follows.

$$\begin{aligned} h, i, \eta \models \mathsf{let}\ \Gamma\ \mathsf{in}\ P \quad \text{iff} \quad & h, i, \eta\,[\beta \mapsto k(\beta)]_{\beta\in\mathrm{dom}(\Gamma)} \models P \\ & \text{where } k = \text{fix } f_{\eta,\Gamma} \end{aligned}$$

The function $f_{\eta,\Gamma}$ takes as its argument a predicate interpretation k_0 and returns a predicate interpretation $f_{\eta,\Gamma}(k_0)$. Intuitively $f_{\eta,\Gamma}$ applies a single iteration of the recursive definitions given 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 least fixed-point of $f_{\eta,\Gamma}$ is the least predicate interpretation such that another iteration of the recursion does not alter the predicate interpretation. This fits with our intuitive understanding of a recursive predicate definition.

It is important to show that our modified semantics for separation logic gives a semantics to all formulas in the fragment. This is trivially true for let-free formulas, but we must prove that a fixed-point exists for let formulas. The existence of a least fixed-point has been proved for other versions of the separation-logic semantics with let-statements [52]. However, the semantics given in this chapter have been modified from the ones appearing in these papers, which means that we must re-prove the result.

Lemma 6.1 (existence of fixed-point[3]). *Let $f_{\eta,\Gamma}$ be the function constructed by the satisfaction relation to define the semantics of a* let*-expression. A least fixed-point for $f_{\eta,\Gamma}$ is guaranteed to exist.*

Proof. The existence of a least fixed-point for the function $f_{\eta,\Gamma}$ is guaranteed by Tarski's fixed-point theorem [75]. Let $(L, \leq)$ be some *complete lattice*,

[3]This proof was suggested by Hongseok Yang in personal communication.

meaning that any subset of L has both a least upper bound and greatest lower bound. Let f be a monotone increasing function over L (that is, for $l_1, l_2 \in L$, if $l_1 \leq l_2$, then $f(l_1) \leq f(l_2)$). Then Tarski's theorem states that the set of fixed-points of f is also a complete lattice with respect to $\leq$. This implies there must exist a least fixed-point of f.

It is clear from the definition of $\leq_p$ that every set of predicate interpretations has both a least upper bound and a greatest lower bound, constructed respectively by set union and intersection over each symbol. Therefore the class of predicate interpretations over a given set of predicate names forms a complete lattice.

By Tarski's result, to show that a least fixed-point exists for the function $f_{\eta,\Gamma}$, we need only show that it is monotonic increasing with respect to $\leq_p$. In other words, given the function $f_{\eta,\Gamma}$ defined by the semantics of satisfaction from a particular separation logic formula, we want to show that $\eta_1 \leq_p \eta_2$ implies $f_{\eta,\Gamma}(\eta_1) \leq_p f_{\eta,\Gamma}(\eta_2)$. As our fragment of separation logic is free of negation, it is simple to show by structural induction that for any symbol σ, any pair present in $f_{\eta,\Gamma}(\eta_1)$ must also be in $f_{\eta,\Gamma}(\eta_2)$. This completes the proof. $\square$

Handling of variable scope

Our semantics for recursive predicates differs from the semantics of [52] in its handling of logical variables. In the original semantics, variables scope over predicate definitions, while in our semantics they do not. This means that free variables cannot be meaningfully used in predicate definitions. All variables in the body of a predicate must be bound, either by an existential quantification, or by the argument names for the predicate.

We restrict variable scope in this way to reduce the complexity of mapping predicates to non-terminal hyperedges. Passing variables by scope is difficult to simulate correctly in a hyperedge replacement grammar. To do it successfully we must analyse the formulas to see which variables are used by each predicate's right-hand side and then add extra attachment points to hyperedges to pass nodes corresponding to these variables. Restricting scope makes the mapping considerably easier.

We justify this change to the semantics by observing, that our semantics is identical to the semantics of [52] for formulas without free variables in predicate definitions. We also observe that for any formula defined under the semantics of [52], an equivalent formula can be constructed without free variables in predicate definitions. This is because the set of variable names referred to in a formula is always finite, so variables can always be passed as arguments to a predicate definition, rather than passed implicitly by scope.

For example, consider the following formula, which passes the variable x implicitly by scope.

$$\mathsf{let}\ f(z) = z \mapsto x, \mathsf{nil}\ \mathsf{in}\ f(y)$$

This can be converted to the following formula by converting the variable x into a predicate argument.

$$\mathsf{let}\ f(z_1, z_2) = z_1 \mapsto z_2, \mathsf{nil}\ \mathsf{in}\ f(y, x)$$

As a result, restricting variable scope in our semantics results in no alteration in expressive power.

6.2 Flattening separation logic formulas

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

Definition 6.6 (flat formula)**.** A formula $\mathsf{let}\ \Gamma\ \mathsf{in}\ P \in \mathcal{SL}$ is a *flat* formula if Γ and P are let-free. $\mathcal{SLF}$ is the class of all flat formulas in $\mathcal{SL}$.

Example 6.7 (flat formula)**.** The following formula is flat, as it contains only a single outermost let.

$$\mathsf{let}\ P(x) = x \mapsto \mathsf{nil}, \mathsf{nil}\ \mathsf{in}\ \exists y, z.\, P(y) * P(z)$$

However, the following formula is not flat, as it contains more than one let, and in any case neither of the lets is outermost.

$$(\mathsf{let}\ P(x) = x \mapsto \mathsf{nil}, \mathsf{nil}\ \mathsf{in}\ \exists y.\, P(y)) \vee (\mathsf{let}\ Q(x) = \mathsf{emp}\ \mathsf{in}\ \exists y.\, Q(y))$$

In this section we show that every formula in our fragment can be rewritten as an equivalent flat formula. This means that restricting our mapping to flat formulas does not reduce its power.

Any let-free formula F can be converted to the corresponding flat formula by embedding it in a let-statement with no predicate definitions. Non-let-free formulas must be explicitly flattened.

Before flattening, we must first ensure that predicate names are used unambiguously in formulas. The let-construct in our fragment operates in the same way as variable quantification, in that predicate instances can be either bound or free. A predicate instance is bound if it exists inside a let-expression that defines a meaning for the predicate. Other predicate instances are free.

A formula in our fragment is *conflict-free* if each predicate name is used in the scope of at most one let-expression. This means that different let expressions must define predicates with different names, and also that the names of free predicate instances must not be used in any let-expression's definitions. Any formula in our fragment can be rewritten as an equivalent conflict-free formula.

Lemma 6.2. *Let F be a formula in $\mathcal{SL}$. Then there exists a corresponding conflict-free formula $F' \in \mathcal{SLF}$ such that for any heap h, variable interpretation i and predicate interpretation η, $h, i, \eta \models F \iff h, i, \eta \models F'$.*

Proof. In the following, let $F[a/b]$ stand for formula F with all free instances of the predicate name a replaced by b.

We prove our result by showing that free-variable rewriting alters only the names of the variables in the satisfying predicate interpretation. For any predicate name σ and unused predicate name ς, if $h, i, \eta \models F$, then $h, i, \eta[\sigma \mapsto \bot][\varsigma \mapsto \eta(\sigma)] \models F[\sigma/\varsigma]$. This result follows trivially from the semantics of satisfaction.

We then apply this result to show that for any let-expression $F = \mathsf{let}\,\Gamma\,\mathsf{in}\,P$, $h, i, \eta \models F$ if and only if $(s, h), \eta \models \mathsf{let}\,\Gamma[\sigma/\varsigma]\,\mathsf{in}\,P[\sigma/\varsigma]$. This follows from the fact that replacing Γ with $\Gamma[\sigma/\varsigma]$ result in a predicate interpretation $k[\sigma \mapsto \bot][\varsigma \mapsto k(\sigma)]$, where k is the predicate interpretation defined by Γ.

Once we have this result for let-formulas, it follows by structural induction that we can replace any bound predicate name in a formula with an arbitrary unused predicate name without altering the semantics of the formula. Therefore, any conflicting predicates can be renamed. $\square$

We now define a function that takes as its input an arbitrary conflict-

$$\begin{aligned}
\mathit{flat}(P * Q) &\stackrel{\text{def}}{=} \mathit{lift}(\mathit{flat}(P) * \mathit{flat}(Q)) \\
\mathit{flat}(P \vee Q) &\stackrel{\text{def}}{=} \mathit{lift}(\mathit{flat}(P) \vee \mathit{flat}(Q)) \\
\mathit{flat}(\exists x.\, P) &\stackrel{\text{def}}{=} \mathit{lift}(\exists x.\, \mathit{flat}(P)) \\
\mathit{flat}(\mathsf{let}\ \Gamma\ \mathsf{in}\ P) &\stackrel{\text{def}}{=} \mathit{lift}(\mathsf{let}\ \mathit{flat}(\Gamma)\ \mathsf{in}\ \mathit{flat}(P)) \\
\mathit{flat}(\sigma_1(\vec{x_1}) = R_1 \ldots \sigma_n(\vec{x_n}) = R_n) &\stackrel{\text{def}}{=} \\
\sigma_1(\vec{x_n}) = \mathit{flat}(R_1) &\ldots \sigma_n(\vec{x_n}) = \mathit{flat}(R_n)
\end{aligned}$$

Figure 6.3: Flattening function *flat*.

free 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.

The flattening function *flat* is defined by a bottom-up transformation of the structure of a formula. The flattening itself is performed by the function *lift*, which takes as its argument a formula F with flat sub-expressions and produces a flattened formula $\mathit{lift}(F)$. Function *flat* first applies *flat* to the sub-expressions of the formula, and then applies the rewriting function *lift* to the resulting formula.

Definition 6.7 (*flat*,*lift*)**.** *flat* is defined in Figure 6.3. All of the unintroduced cases for *flat* are the identity function on formulas. The rewriting function *lift* is defined in Figure 6.4. Here, the variables P and Q always stand for let-free formulas. Once again, all unintroduced cases are defined as the identity.

Example 6.8 (flattening function)**.** Consider the following non-flat formula:

$$\mathsf{let}\ (\sigma_1 = (\mathsf{let}\ \sigma_1 = \mathsf{emp}\ \mathsf{in}\ \sigma_1()))\ \mathsf{in}\ (\mathsf{let}\ (\sigma_1 = \mathsf{emp})\ \mathsf{in}\ \sigma_1())$$

We first convert the formula into an equivalent non-conflicting formula by rewriting the predicate names.

$$\mathsf{let}\ (\sigma_1 = (\mathsf{let}\ \sigma_2 = \mathsf{emp}\ \mathsf{in}\ \sigma_2()))\ \mathsf{in}\ (\mathsf{let}\ (\sigma_3 = \mathsf{emp})\ \mathsf{in}\ \sigma_3())$$

Applying *flat* gives the following derivation.

$$\begin{array}{rcl}
\mathit{lift}(\exists x.\, \mathsf{let}\ \Gamma\ \mathsf{in}\ P) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma\ \mathsf{in}\ \exists x.\, P \\[1em]
\mathit{lift}((\mathsf{let}\ \Gamma\ \mathsf{in}\ P) * Q) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma\ \mathsf{in}\ (P * Q) \\
\mathit{lift}(P * (\mathsf{let}\ \Gamma\ \mathsf{in}\ Q)) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma\ \mathsf{in}\ (P * Q) \\
\mathit{lift}((\mathsf{let}\ \Gamma_1\ \mathsf{in}\ P) * (\mathsf{let}\ \Gamma_2\ \mathsf{in}\ Q)) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma_1,\, \Gamma_2\ \mathsf{in}\ (P * Q) \\[1em]
\mathit{lift}((\mathsf{let}\ \Gamma\ \mathsf{in}\ P) \vee Q) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma\ \mathsf{in}\ (P \vee Q) \\
\mathit{lift}(P \vee (\mathsf{let}\ \Gamma\ \mathsf{in}\ Q)) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma\ \mathsf{in}\ (P \vee Q) \\
\mathit{lift}((\mathsf{let}\ \Gamma_1\ \mathsf{in}\ P) \vee (\mathsf{let}\ \Gamma_2\ \mathsf{in}\ Q)) & \stackrel{\text{def}}{=} & \mathsf{let}\ \Gamma_1,\, \Gamma_2\ \mathsf{in}\ (P \vee Q) \\[1em]
\mathit{lift}(\mathsf{let}\ \Gamma_1\ \mathsf{in}\ (\mathsf{let}\ \Gamma_2\ \mathsf{in}\ P)) & \stackrel{\text{def}}{=} & \mathsf{let}\ \mathit{lift}(\Gamma_1),\, \Gamma_2\ \mathsf{in}\ P \\
\mathit{lift}(\mathsf{let}\ \Gamma\ \mathsf{in}\ P) & \stackrel{\text{def}}{=} & \mathsf{let}\ \mathit{lift}(\Gamma)\ \mathsf{in}\ P \\[1em]
\mathit{lift}(\Gamma_1, \Gamma_2) & \stackrel{\text{def}}{=} & \mathit{lift}(\Gamma_1),\ \mathit{lift}(\Gamma_2) \\
\mathit{lift}(\sigma(\vec{x}) = \mathsf{let}\ \Gamma\ \mathsf{in}\ P) & \stackrel{\text{def}}{=} & \Gamma, \sigma(\vec{x}) = P
\end{array}$$

Figure 6.4: Rewriting function *lift*.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{flat}(\ \mathsf{let}\ (\sigma_1 = (\mathsf{let}\ \sigma_2 = \mathsf{emp}\ \mathsf{in}\ \sigma_2()))\ \mathsf{in}\ (\mathsf{let}\ (\sigma_3 = \mathsf{emp})\ \mathsf{in}\ \sigma_3())\ )} \\[0.5em]
\Rightarrow & \mathit{lift}(\mathsf{let}\ \mathit{flat}(\sigma_1 = (\mathsf{let}\ \sigma_2 = \mathsf{emp}\ \mathsf{in}\ \sigma_2())) \\
& \quad \mathsf{in}\ \mathit{flat}(\mathsf{let}\ (\sigma_3 = \mathsf{emp})\ \mathsf{in}\ \sigma_3())) \\[0.5em]
\Rightarrow^* & \mathit{lift}(\mathsf{let}\ (\sigma_1 = \mathit{lift}(\mathsf{let}\ (\sigma_2 = \mathsf{emp})\ \mathsf{in}\ \sigma_2())) \\
& \quad \mathsf{in}\ \mathit{lift}(\mathsf{let}\ (\sigma_3 = \mathsf{emp})\ \mathsf{in}\ \sigma_3())) \\[0.5em]
\Rightarrow^* & \mathit{lift}(\mathsf{let}\ (\sigma_1 = \sigma_2(), \sigma_2 = \mathsf{emp})\ \mathsf{in}\ (\mathsf{let}\ (\sigma_3 = \mathsf{emp})\ \mathsf{in}\ \sigma_3())) \\[0.5em]
\Rightarrow^* & \mathsf{let}\ (\sigma_1 = \sigma_2(), \sigma_2 = \mathsf{emp}, \sigma_3 = \mathsf{emp})\ \mathsf{in}\ \sigma_3()
\end{array}$$

The result is a flat formula.

Termination of *lift* is ensured because the only recursion in *lift* occurs over definitions inside a let-expression. By assumption, the right-hand sides of these predicate definitions are flat, so the recursion only needs at most a single iteration on each definition. Termination of the flattening function *flat* is guaranteed (assuming termination of *lift*) by the fact that *flat* simply traverses a formula.

It is simple to see from the structure of *lift* that each application results in a flat formula, under the assumption that all the input formula's sub-expressions are flat. The formulas constructed by *flat* are guaranteed to be flat by the fact that *lift* produces flat formulas.

It remains for us to prove that *flat* is sound.

Lemma 6.3. *Let F be a separation logic formula, h a heap, i a variable interpretation and η a predicate interpretation such that $h, i, \eta \models F$. Let σ be a predicate name that is unused in F. Then for any value $R \in Pow(Loc^* \times H)$ defining a new predicate interpretation $\eta' = \eta[\sigma \mapsto R]$, it holds that $h, i, \eta' \models F$.*

Proof. This follows directly from the definition of satisfaction. Extending a predicate interpretation by unused predicate symbols leaves the semantics of the existing symbols unchanged. □

We now show that the flattening function *flat* is correct, in the sense that the result of applying *flat* to a formula P is a semantically equivalent formula *flat*(P). To do this, we first show that *lift* is semantics-preserving.

Proposition 6.4. *Let F be a conflict-free formula, h a heap, i a variable interpretation and η a predicate interpretation. Then $h, i, \eta \models F$ if and only if $h, i, \eta \models$ lift(F).*

Proof. First consider the case where we have let-expression

$$\mathsf{let}\ \Gamma_1\ \mathsf{in}\ (\mathsf{let}\ \Gamma_2\ \mathsf{in}\ P)$$

and the set of predicates defined in Γ_1 is disjoint from the set defined in Γ_2. This formula is equivalent to $\mathsf{let}\ \Gamma_1, \Gamma_2\ \mathsf{in}\ P$ because the predicate interpretation k_2 defined by Γ_2 is unaltered by the addition of the predicate interpretation k_1 defined from Γ_1. We can use this to show the correctness of most of the other cases.

Consider the function *lift* applied to a separating conjunction, disjunction, or existential quantification. By the predicate interpretation extension result given in Lemma 6.3, we know that the predicate interpretation can be extended without altering the semantics. Because the function is conflict-free, we can promote a single let expression from a sub-expression to the surrounding expression without altering the semantics. All of the definitions of *lift* over these constructs can be composed from let-fusion and a single promotion, so all of them are safe.

Finally, we need to show that the promotion of rules from the right-hand side of a predicate definition to the surrounding set of definitions is safe. This follows from the assumption that formulas are conflict-free, which ensures that the new fixed-point derived by promoting a definition is the union of the two old fixed-points. □

Note that this result applies only to formulas that are conflict-free. Without this assumption, *lift* can alter the semantics of a formula in two ways. First, a let-expression could be promoted to surround a free variable that is included in the let-expression's rule-set. Second, two let-expressions that define the same predicate name could be merged, resulting in a new definition for both.

We can now prove that any formula can be converted into an equivalent flat formula.

Corollary 6.5 (correctness of *flat*)**.** *Let F be a formula in $\mathcal{SL}$. Then there exists a flat formula F' in $\mathcal{SL}_f$ such that for any heap h, variable interpretation i and predicate interpretation η, $h, i, \eta \models F$ if and only if $h, i, \eta \models F'$.*

Proof. Any let-free formula F can be converted to the corresponding flat formula let in F, that is by embedding it in a let-statement with no predicate definitions. Non-let-free formulas must be flattened using *flat*.

All formula rewrites performed by *flat* are applications of *lift*, so the fact that *flat* is semantics-preserving for conflict-free formulas follows as a direct consequence of Proposition 6.4. Lemma 6.2 states that for any formula, a corresponding conflict-free formula exists, so *flat* can always be safely applied, resulting in a flat formula corresponding to F. □

6.3 Heap-graphs and mapping between domains

Separation-logic formulas and hyperedge-replacement grammars both define sets of graph-like structures. However, the two approaches operate over different domains. For this reason, to define semantics-preserving mappings between formulas and grammars, we must first define a mapping between the domains of the two approaches. Because our mapping should be semantics-preserving, the mapping between domains must be bijective.

To begin with, we identify the two domains. A separation logic formula defines the class of heaps that satisfy the formula. The content of this class is defined by the satisfaction relation given in §6.1.3.

A hyperedge replacement grammar defines a class of graphs. We are interested, however, only in those grammars that define classes of so-called *heap-graphs*. That is, graphs for which there exists a corresponding heap. Intuitively, we model defined locations by nodes, points-to assertions by E-labelled edges of arity 3, and nil by a unique edge of arity 1 pointing to a node modelling nil.

Definition 6.8 (well-sourced)**.** We say that a graph is *well-sourced* if each node is the source for at most one hyperedge.

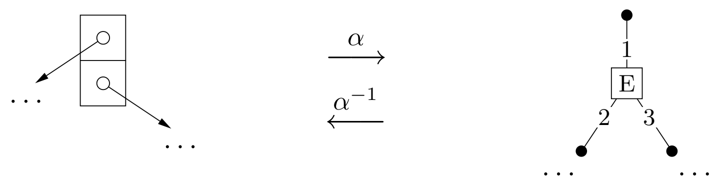
Definition 6.9 (heap-graph)**.** We say that a hypergraph H is a *heap-graph* if (1) The label set is $\{E, \mathsf{nil}\}$, where E has arity 3, and nil arity 1. (2) There exists exactly one nil-labelled edge. (3) The graph is well-sourced. We use $\mathcal{HG}$ to stand for the set of all heap-graphs.

Example 6.9 (heap-graph)**.** The right-hand side of Figure 6.5 shows a heap-graph containing four nodes and four edges.

We now define α, a bijective mapping from heaps to heap-graphs. As α is bijective, this also defines by implication a bijective inverse α^{-1} from heap-graphs to heaps.

The function α constructs a node for each heap location with a defined value and the atomic value nil. The single nil-value results in a single node with an attached nil-labelled hyperedge. Pointers between locations result in E-labelled hyperedges. Locations that have the value $\boxslash$ result in nodes which are not the source of any hyperedge.

For example, the following diagram shows a fragment of a heap consisting of a single location l in a heap h, where $h(l)$ is defined as a pair of locations (a_1, a_2). It also shows the corresponding heap-graph fragment, consisting of a node with an attached E-labelled hyperedge.



Definition 6.10 (α)**.** The function $\alpha\colon \mathcal{HE} \to \mathcal{HG}$ maps from heaps to heap-graphs. Given a heap h, the heap-graph $\alpha(h)$ consists of: (1) A vertex v_k for each heap element $k \in \mathrm{Elem}$ where either $k \in \mathrm{dom}(h)$, or $k = \mathsf{nil}$. (2) A

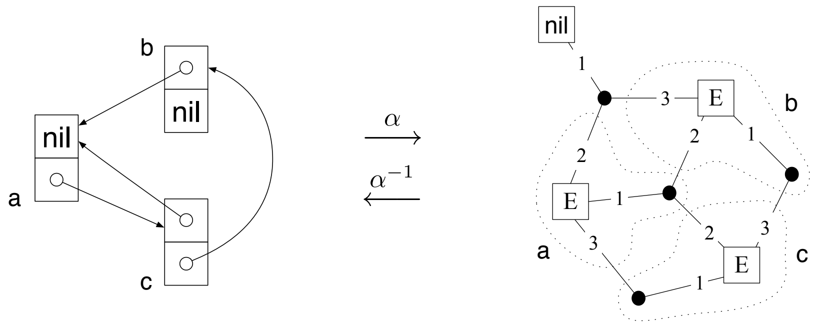


Figure 6.5: Left: A heap h with three locations. Right: The corresponding heap-graph $\alpha(h)$.

hyperedge e_k for each heap location $k \in \text{dom}(h)$ where $h(k) = (k_1, k_2)$, such that $l(e_k) = E$, and $att(e_k) = v_k, v_{k_1}, v_{k_2}$. (3) A hyperedge e_{nil} such that $l(e_{\mathsf{nil}}) = \mathsf{nil}$ and $att(e_{\mathsf{nil}}) = v_{\mathsf{nil}}$.

The mapping α is a bijection (with heaps and graphs considered unique up to isomorphism) because each element of the heap results in sufficient nodes and edges to record its relationship to other elements in the resulting heap-graph. As a result, the heap-graph constructed for a particular heap h is distinct from the heap-graph resulting from any other heap. As α is bijective, the it implicitly defines the inverse bijective mapping $\alpha^{-1}\colon H \to G$ from graphs to heaps.

Example 6.10 (α)**.** The domain of the heap on the left-hand side of Figure 6.5 contains three locations. The resulting heap-graph, shown on the right of Figure 6.5, contains three E-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.

6.4 Heap-graph grammars and source normalisation

Our correspondence operates between formulas in $\mathcal{SL}$ and hyperedge replacement grammars producing only heap-graphs.

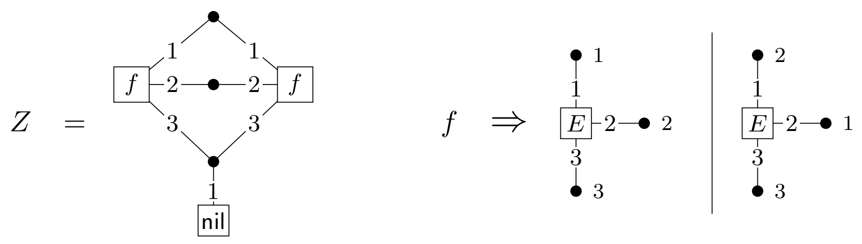


Figure 6.6: Grammar producing non-heap graphs.

Definition 6.11 (heap-graph grammar)**.** We call a grammar G a *heap-graph grammars* if all the graphs in $\mathrm{L}(G)$ are heap-graphs. The set $\mathcal{HGG}$ refers to the class of all possible heap-graph grammars.

When constructing the grammar that corresponds to a separation-logic formula, we must be careful that we construct a heap-graph grammar. Using the intuitive approach described in §7.1, it is easy to inadvertently construct a grammar with a language containing non-heap-graphs. This is because the simple approach does not restrict the sources of edges. The following example illustrates the problem.

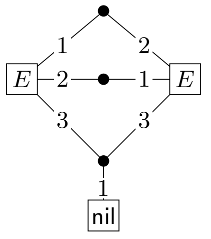
Example 6.11 (naïve grammar construction)**.** Consider the following recursive definition.

$$\mathsf{let}\ f(z_1, z_2) = ((z_1 \mapsto z_2, \mathsf{nil}) \vee (z_2 \mapsto z_1, \mathsf{nil}))\ \mathsf{in}\ \exists x, y.\, f(x, y) * f(x, y)$$

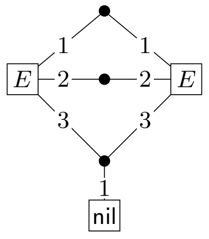
A naïve mapping from formulas to grammars translates the single predicate definition into a pair of productions over the symbol f, and the in-subexpression to an initial graph. This gives the grammar shown in Fig. 6.6.

The language of this grammar includes heap-graphs, such as graph A below, but it also includes non-heapgraphs, such as graph B.

A:



B:



The problem with graph B is that the first attachment points of both E-labelled edges are the same. Clause (3) of the definition of a heap-graph (Def. 6.9) forbids this, because such graphs do not correspond to heaps.

The simplest solution (if it worked) would be to define syntactic conditions on rules which ensure that grammars are heap-graph grammars. It is easy to ensure syntactically that the first two clauses of Def. 6.9 are respected by a grammar. The first is just a restriction on the grammar's terminal label-set, while the second requires the existence of a single nil-labelled edge.

The third clause, that the graph is well-sourced (see Def. 6.8), is more tricky to ensure. A sequence of derivations violating this requirement may involve many rules, and removing any of the rules may make the grammar a heap-graph grammar. Conformance to this requirement is a property of the whole grammar, rather than a simple syntactic property that can be checked on a rule-by-rule basis.

We could simply ignore non-heap-graphs in the languages of grammars constructed by the mapping. This seems inelegant however. Instead we show that for any grammar we can construct a corresponding grammar with all non-heap-graphs removed from its language.

Definition 6.12 (source normalisation)**.** A grammar H' is a *source normalisation* of grammar H if $\mathrm{L}(H')$ is the set of all well-sourced graphs in $\mathrm{L}(H)$.

To make use of source normalisation in our mappings, we define a source normalisation operator $\lfloor - \rfloor$. This takes an input grammar H and constructs a grammar $\lfloor H \rfloor$ such that $\lfloor H \rfloor$ is a source normalisation of H.

To show that we can implement this operator, we need to show that a source normalisation can be constructed for each input grammar. Given a hyperedge-replacement grammar and an arbitrary property, we cannot necessarily construct a new grammar with a language consisting of the intersection between the two.

In [27] it is proved that for any hyperedge replacement grammar H and predicate P expressing a *compatible property* in the sense of [27], definition 2.6.1, a HR grammar H_P can be constructed so that $\mathrm{L}(H_P) = \{g \in \mathrm{L}(H) \mid P(g)\}$.

Intuitively, a compatible property is a property that can be decomposed and checked piecewise. To check such a property for a large hypergraph it suffices to examine the smaller component graphs constructed during the derivation of the large graph. That is, suppose we have a non-terminal hypergraph G and a derivation to a terminal graph H. We can first check a compatible property for the component graphs that replaced G's nonterminal edges, then compose these properties in the initial graph G to check the compatible property for H.

Restriction to a compatible property works because any such property can be checked purely by examination of the constituent graphs formed from non-terminal nodes. Consequently the property can be ensured statically by embedding property information into nonterminal symbols.

To show that for any grammar H, some source normalisation $\lfloor H \rfloor$ can be constructed, it suffices to show that membership of the class of well-sourced graphs is a compatible property. We do this by constructing a *compatible predicate* WS_0 such that $WS_0(H)$ holds for graph H if H is well-sourced.

We now state formally the definition of a compatible predicate. This definition is taken verbatim from [38], with minor updates to conform to our notation.

Definition 6.13 (projection)**.** If we have graphs a derivation $R \Rightarrow^* H$ and edge $e \in E_R$, we use $H(e)$ to stand for the subgraph of graph H resulting from edge e in the derivation. We call this graph the *projection* of e in H.

Definition 6.14 (compatible predicate)**.** Let $\mathcal{C}$ be a class of HR grammars, I a finite set, called the index set, $PROP$ a decidable predicate defined on pairs (H, i), with $H \in \mathcal{H}_C$, and $i \in I$, and $PROP'$ a decidable predicate on triples $(R, assign, i)$ with $R \in \mathcal{H}_C$, a mapping $assign\colon E_R \to I$, and $i \in I$. Then $PROP$ is called $(C, PROP')$-*compatible* if for all $HRG = \langle N, T, P, Z \rangle \in \mathcal{C}$ and all derivations $A^\bullet \Rightarrow R \Rightarrow^* H$i with $A \in N \cup T$ and $H \in \mathcal{H}_T$, 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 $\mathcal{H}_C$ is called $\mathcal{C}$-*compatible* if there exist predicates $PROP$ and $PROP'$ such that $PROP$ is $(\mathcal{C}, PROP')$-compatible and $PROP_0$ holds for a graph H if and only if $PROP(H, i_0)$ holds for some index i_0.

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 auxiliary information recorded for nonterminal nodes. $PROP'$ records auxiliary information for nonterminal edges using the assignment function $assign$. $PROP_0$ eliminates the auxiliary information from $PROP$ by just requiring that $PROP$ holds for some index.

Intuitively, $PROP$ is called $(C, PROP')$-compatible if for every derivation $R \Rightarrow^* H$, if $PROP$ holds for H then there exists an $assign$ such that $PROP'$ holds for R, and the information in $assign$ holds for the projections of the edges in H.

Example 6.12 (compatible predicate)**.** We show that the property '*graph contains at most one X-labelled edge*' is compatible. We make the index set $\{0, 1\}$. The predicate $CX(H, 1)$ holds if H contains a single X-labelled edge, and $CX(H, 0)$ holds if it contains no such edge. For any nonterminal graph R, $CX'(R, assign, 1)$ holds if either (1) R contains an X-labelled edge and $assign$ assigns 0 to all nonterminal edges, or (2) $assign$ assigns 1 to a single nonterminal edge and R contains no X-labelled edges. $CX'(R, assign, 0)$ holds if R contains no X-labelled edges and $assign$ assigns 0 to all nonterminal edges.

CX is CX'-compatible because for any derivation $R \Rightarrow^* H$, if $CX(H, 1)$ holds then the single X-labelled node must either appear in R, or appear in the projection of one of the nonterminal edges in R. Consequently there must exist an assignment $assign$ such that $CX'(R, assign, 1)$ holds. The same holds for $CX(H, 0)$.

Let $\mathcal{HRG}$ be the class of all hyperedge replacement grammars. We now define predicates WS', WS and WS_0, such that WS_0 is $\mathcal{HRG}$-compatible according to Definition 6.14 and WS_0 holds if a graph is well-sourced.

Definition 6.15 (WS', WS, WS_0)**.** Let WS' be a predicate defined on triples $(R, assign, I)$, with $R \in \mathcal{H}_C$ 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 ass 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.

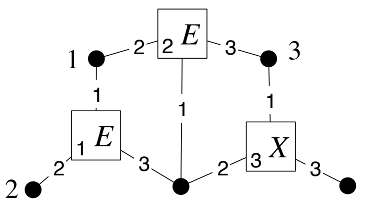
$WS'(R, assign, I)$ holds if and only if: (1) For each node $v \in V_R$ there exists at most one edge $e \in E_R$ such that either $l(e)$ is terminal and

$att(e)[1] = v$, or $l(e)$ is non-terminal and there exists an $n \in assign(e)$ 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 an $n \in assign(e)$ such that $att(e)[n] = ext_R(i)$.

The binary 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 $I \subseteq \mathbb{N}$.

Intuitively the predicate $WS(H, I)$ holds when the graph H is well-sourced and terminal, and I records the external nodes that are the sources of edges. $WS_0(H)$ records that the graph is well-sourced, by quantifying over I.

Example 6.13 (WS')**.** Consider the following graph R with terminal edge label E and non-terminal edge label X. Edges have subscripts on their bottom left corner to uniquely identify them as the edges e_1, e_2 and e_3.



Then $WS'(R, assign, I)$ holds, with $ass = \{e_3 \mapsto \{1, 3\}\}$ and $I = \{1, 3\}$. $WS'(R, assign, I)$ holds because every node is either the source of at most one terminal edge, or is attachment point 1 or 3 of the nonterminal node e_3. The $assign$ function records that these attachment points will reduce to the sources of terminal nodes. I records the sources of the external nodes of the graph, including those that will be generated from e_3.

Lemma 6.6. *Predicate WS_0 holds for a graph H if and only if H is well-sourced.*

Proof. The if direction follows directly from the definition of predicate WS'. 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.

Now we prove that if H satisfies WS_0, then H must be well-sourced. The condition on terminal edges ensures that the graph satisfies the restriction on E-labelled edge sources. The definition of a heap-graph (Definition 6.9) 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. $\square$

Lemma 6.7. *The predicate WS_0 is a compatible predicate.*

Proof. According to the Definition 6.14, to show that WS_0 is compatible 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 $e \in E_R^N$.

The 'if' direction is simple. 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 projected graph $H(e)$, so if we replace the non-terminals with the terminal graphs, $WS(H, I)$ must hold.

The 'only-if' is slightly more difficult. Assume that $WS(H, I)$ holds. Then there must exist an assignment $assign$ 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. $\square$

Proposition 6.8. *For any hyperedge-replacement grammar H we can construct a grammar $\lfloor H \rfloor$ that is a source-normalisation of H.*

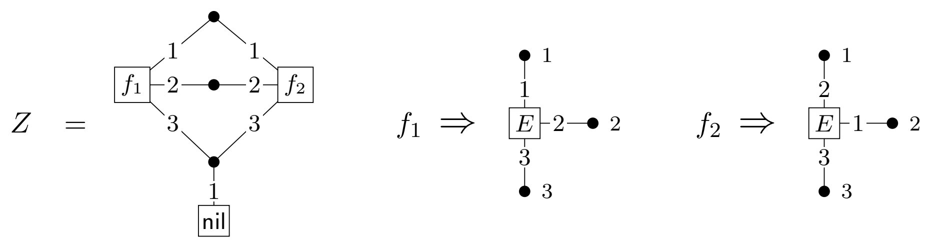
Proof. In [27] it is proved that given a grammar $H \in \mathcal{C}$ and a $\mathcal{C}$-compatible predicate $PROP_0$, we can construct a new grammar H' such that a graph g is in $\mathrm{L}(H')$ if and only if $g \in \mathrm{L}(H)$ and $PROP_0(g)$. We use this result to prove that we can define the source-normalisation operator.

Lemma 6.7 shows that the predicate WS_0 is a compatible predicate, and Lemma 6.6 shows that $WS_0(H)$ holds if and only if H is well-sourced. Therefore, by application of the language restriction result of [27] we can construct a grammar $\lfloor H \rfloor$ such that a graph g is a member of $L(\lfloor H \rfloor)$ if and only if $g \in \mathrm{L}(H)$ and g is well-sourced. $\square$

The source normalisation operator works by recording in the nonterminal symbols which external node (or nodes) will receive the source of a terminal edge.

Example 6.14 (source normalisation)**.** We apply the source normalisation operator to the non-heapgraph grammar shown in Fig. 6.6. The operator first modifies the nonterminal symbol f, giving cases f_1 and f_2. The symbol f_1 points the first attachment point of the terminal E to the first external node, while f_2 points it at the second external node.

The operator then factors the initial graph into two graphs by replacing the two instance of f with f_1 and f_2. The fact that the symbol name records the location of the source means that we can ensure statically that the constructed graphs result in well-sourced terminal graphs. The two resulting graphs are isomorphic, so the new grammar has only a single initial graph. The resulting source-normalised grammar is as follows.



This example is quite simple – all of the productions return terminal graphs. However, the same case-splitting and propagation approach can be applied to an arbitrarily complex grammar, resulting in a source normalisation of the grammar.

Chapter 7

Mapping between formulas and grammars

This chapter defines formally the mappings between formulas and grammars that prove the existence of a correspondence. We show that our mappings are semantics-preserving. The chapter is structured as follows. Section 7.1 describes intuitively the correspondence between formulas and grammars. Section 7.2 defines the mapping g from separation logic formulas to hyperedge replacement grammars. Then in Section 7.3 we prove that g is semantics-preserving. Section 7.4 defines the mapping s from hyperedge-replacement grammars to separation logic formulas. Finally, in Section 7.5 we prove that s is also semantics-preserving.

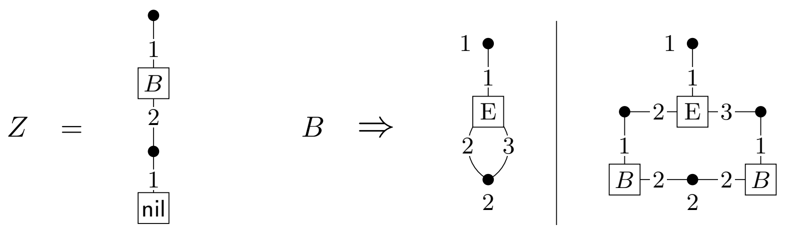
7.1 Intuitive relationship

A correspondence exists between separation logic formulas and hyperedge replacement grammars because (1) the recursive definitions commonly used in separation logic closely resemble hyperedge replacement productions, and (2) the separating property enforced by separating conjunction corresponds to the context-free replacement of non-terminal hyperedges. The following example illustrates this correspondence.

Example 7.1 (binary tree with shared nil-leaf)**.** The following formula defining the class of all binary trees with nil-labelled leaves was presented in Example 6.2.

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

This formula corresponds to the hyperedge replacement grammar $BT = \langle T, N, Z, P\rangle$. This defines the language of binary tree graphs with a shared nil-labelled leaf.[1] The sets of terminal and non-terminal edge labels are respectively $T = \{E, \mathsf{nil}\}$ and $N = \{B\}$. The initial graph Z and set of productions P are:



The upper node of the initial graph Z corresponds to the address x with which the predicate bt is called. The lower node of Z models the unique value nil by a single node with an attached nil-labelled edge. The individual cases of the production defined for label B in the grammar correspond to the two disjuncts defining the predicate bt. The first disjunct corresponds to a terminal branch, and the second a branch and a pair of child trees.

Intuitively the elements of hyperedge replacement grammars and separation logic formulas are related as follows:

- Productions over a single nonterminal symbol in the grammar correspond to the definition of a single recursively-defined predicate in separation logic.
- Terminal hyperedges in the grammar's initial graph and in the right-hand sides of productions correspond to separation logic's points-to assertion ($\mapsto$).
- The atomic value nil is modelled as a single node with an attached nil-labelled hyperedge of arity one.

[1]The nil-leaves are shared because our mapping from graphs to heaps models nil by a single node – see §6.3 for justification.

- Non-terminal hyperedges in the grammar correspond to instances of recursively-defined predicates in separation logic. The attachment nodes of the edges correspond to the arguments passed to the predicate.

We now prove that this correspondence exists by defining a pair of mappings and proving that the mappings are semantics-preserving.

7.2 Mapping from formulas to grammars

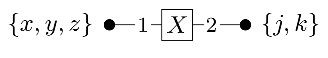
In this section we define the function $g : \mathcal{SLF} \to \mathcal{HGG}$ that maps from flat separation logic formulas to hyperedge-replacement grammars. This mapping implements formally the intuitive mapping between domains that we described in §7.1.

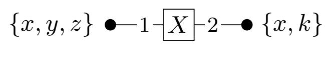
7.2.1 Basic notions used in the mapping

We first define some extra notions used by the definition. Graphs in the mapping g are constructed by 'gluing together' small graphs into larger graphs. These graphs are then used as either the right-hand sides of productions or as the members of the initial graph-set. To define g we therefore need a definition for this gluing process. In order to define gluing operations, we need to identify the nodes that should be merged. We do this using *tags* attached to the nodes of a graph.

Definition 7.1 (tagged graph)**.** Let F be a countably infinite set of tags. A *tagged graph* $T = \langle G, t \rangle$ consists of a hypergraph G and a partial tagging function $t \colon V_G \to Pow(F)$ that associates tags sets to nodes. The *tag-set* for a node v is $t(v)$. The function $\mathrm{tag}(T)$ gives the union of all tag-sets in T. A tagged graph is *well-tagged* if the tag-sets of the nodes are pairwise disjoint. The tagged graph $T \setminus x$ is the graph T with the tag x removed from all tag-sets.

Example 7.2 (tagged graph)**.** We show tags visually as label-sets attached to graph nodes.



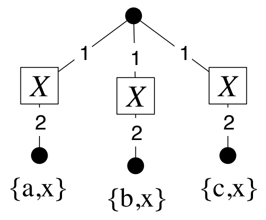


In the left two-node graph, the first attachment point of the X-labelled edge has the tag-set $\{x, y, z\}$, and the second has tag-set $\{j, k\}$. The left-hand graph is well tagged. The right-hand graph is not well-tagged, as both nodes are tagged with x.

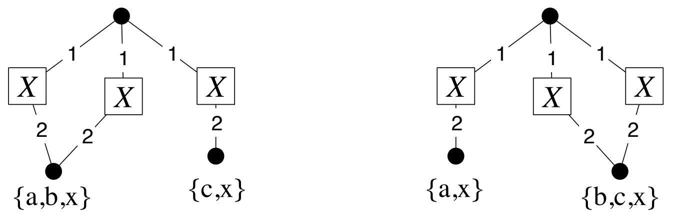
Tags are used by the *unify labels* and *graph join* operators to merge graphs.

Definition 7.2 (unify tags)**.** Let $T = \langle G, t\rangle$, be a tagged graph. A unification step on T 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 denoted by $\Downarrow T$, is the well-tagged graph constructed by the reflexive transitive closure of unification steps.

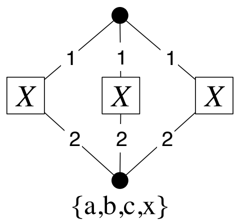
Example 7.3 (unify tags)**.** We begin by giving a tagged graph T that is not well-tagged.

$T =$ 

Applying a single tag unification to this graph gives two possible resulting graphs.



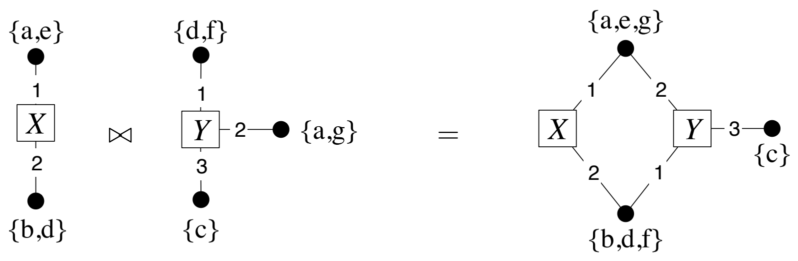
The tag unification $\Downarrow T$ is the graph given below. Here all nodes containing x in their label sets have been merged.

$\Downarrow T =$ 

At least one tag unification of a graph is guaranteed to exist because (1) each unification step decreases the size of the graph, and (2) a unification step can be applied to any non-well-tagged graph. The tag unification for a graph is unique because unification steps are confluent.

Definition 7.3 (join)**.** The join of two tagged graphs T_0 and T_1, denoted by $T_0 \bowtie T_1$, is constructed by combining T_0 and T_1 by disjoint union, and then constructing the tag unification of the resulting graph.

Example 7.4 (join)**.** The following diagram show the result of $\bowtie$ when applied to a pair of graphs with overlapping tag sets.



The tagging function for a well-tagged heap-graph is similar to a variable interpretation for a heap. We define α_t as a function mapping a heap h and variable interpretation i to a tagged heap graph $\langle G, t\rangle$.

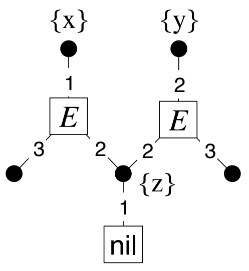
Definition 7.4 (α_t)**.** Let h be a heap and i a variable interpretation. Then $\alpha_t(h, i)$ is the tagged graph $\langle G, t\rangle$. Here G is $\alpha(h)$, and the tagging function t is defined so that for any node $v \in \alpha(h)$ and any variable $x \in \text{Var}$, $x \in t(v)$ if $i(x) = l$ and l is the unique heap location in h mapped to v by α.

The function α_t is bijective to well-tagged graphs, so it defines by implication α_t^{-1}, the inverse function from well-tagged graphs to a heap and variable interpretation.

Example 7.5 (α_t)**.** We apply α_t to the following heap and variable interpretation.

$$\begin{aligned} h &= \{l_1 \mapsto \langle l_3, \mathsf{nil}\rangle, l_2 \mapsto \langle l_4, \mathsf{nil}\rangle, l_3 \mapsto \boxslash, l_4 \mapsto \boxslash\} \\ i &= \{x \mapsto l_1, y \mapsto l_2, z \mapsto \mathsf{nil}\} \end{aligned}$$

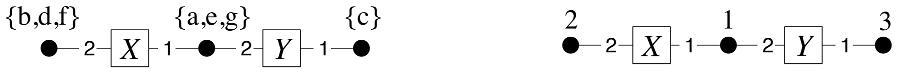
The resulting tagged heapgraph $\alpha_t(h, i)$ is as follows.



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

Definition 7.5 (expose)**.** Let $T = \langle H, t \rangle$ 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 tag(T). 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 tagged with x_i.

Example 7.6 (expose)**.** Left: a well-tagged graph T. Right: the untagged graph produced by $expose(T, [a, b, c])$.



7.2.2 Mapping from formulas to grammars

Function $g : \mathcal{SLF} \rightarrow \mathcal{HGG}$ maps a single flat separation-logic formula F from the restricted fragment we have defined, 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 are inferred from the labels used in the productions and the initial graphs. This suffices to define a complete hyperedge replacement grammar $H = \langle T, N, P, Z \rangle$. A corresponding heap-graph grammar can then be constructed by applying the source normalisation operator to this grammar (see §6.4). Figure 7.1 defines g recursively over the structure of a separation-logic formula.

g takes as its argument a single let-expression and constructs from it a grammar. Much of the work of the mapping is done by two subsidiary functions: h, which constructs a set of graphs from a formula, and r, which constructs productions.

The function h takes as its argument a let-free formula and constructs a set of tagged graphs. It does this by recursing over the structure of the formula. The terminal E-labelled hyperedges are constructed from points-to assertions, with attached nodes labelled according to the assertion's variable

$g[\![\mathsf{let}\ \Gamma\ \mathsf{in}\ F]\!] = \langle Z, P\rangle$
where

$H = h[\![F]\!] \ \bowtie\ \overset{\{\mathsf{nil}\}}{\bullet}\!-\!1\!-\!\boxed{\mathsf{nil}}$

$Z = \mathit{expose}(H, \{\mathsf{nil}\})$

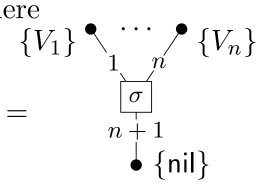
$P = r[\![\Gamma]\!]$

$g[\![F]\!] = \langle Z, \emptyset\rangle$ if F is let-free
where

$H = h[\![F]\!] \ \bowtie\ \overset{\{\mathsf{nil}\}}{\bullet}\!-\!1\!-\!\boxed{\mathsf{nil}}$

$Z = \mathit{expose}(H, \{\mathsf{nil}\})$

$h[\![\sigma(V_1, \ldots, V_n)]\!] = \Downarrow H$
where

$H =$ 

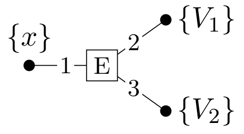
$h[\![\exists x.\, P]\!] =$
$\{H' \mid H \in h[\![P]\!] \wedge H' \cong H \setminus x\}$

$h[\![\mathsf{emp}]\!] = \langle\ \textit{empty graph}\ \rangle$

$h[\![P \vee Q]\!] = h[\![P]\!] \cup h[\![Q]\!]$

$h[\![P * Q]\!] = h[\![P]\!] \bowtie h[\![Q]\!]$

$h[\![x \mapsto V_1, V_2]\!] = \Downarrow H$
where

$H =$ 

$r[\![\Gamma_1, \Gamma_2]\!] = r[\![\Gamma_1]\!] \cup r[\![\Gamma_2]\!]$

$$\begin{aligned} r[\![\, \sigma(x_1, \ldots, x_n) = P\,]\!] = \{(\sigma, H') \mid\ & H \in h[\![P]\!] \wedge \\ & tag(H) = \{x_1, \ldots, x_n, \mathsf{nil}\} \wedge \\ & H' = \mathit{expose}(H, (x_1, \ldots, x_n, \mathsf{nil}))\,\} \end{aligned}$$

Figure 7.1: Mapping g from separation logic formula to hyperedge-replacement grammar.

arguments. Non-terminal hyperedges are constructed from calls to predicates. The attachment points of such a non-terminal edge correspond to the variable arguments, with the addition of a nil-tagged node. This attachment point to nil is needed to keep track of a common nil-node for the whole graph.

In both cases, for terminal and nonterminal hyperedges, after generating the edge the tag unification operator $\Downarrow$ is applied, merging all nodes that have the same label. This ensures that for any variable name x, only a single node exists with the tag x in its variable-set.

The disjunction operator $\vee$ merges by union the graph-sets constructed for each subexpression of the formula. Intuitively, a heap satisfies the disjunction if it satisfies either the left-hand subexpression or right-hand subexpression, so set-union models the intuitive semantics. The separating conjunction $*$ constructs a new set of graphs by applying the graph-join operator $\bowtie$ pairwise to the members of the two sets constructed from its subexpressions. Because the tag-set for newly constructed tagged graph corresponds to sets of variables, this merge fuses any nodes that are pointed to by the same variable, which results in a correctly-joined set of graphs.

The function r is called by g to handle recursive predicate definitions. The right-hand sides of the productions are constructed by applying the h function to the right-hand sides of the predicate definitions, and then applying the expose operator to attach the appropriate external nodes. Tags are removed from graphs by the *expose* operator.

The natural way to model a let formula is as a set of rules defined from predicate definitions applied to a set of graphs defined by the let body. We noted in Chapter 2, p. 31 that we have modified the standard definition of a hyperedge-replacement grammar (as given in for example [27]) to use an initial set of graphs, rather than the more conventional single initial graph. The reason for this redefinition was to simplify the definition of s.

As noted in Chapter 2, we could preserve the standard definition by introducing extra rules to allow a grammar with a single initial graph. We have decided however that it is simpler to alter the basic grammar definition.

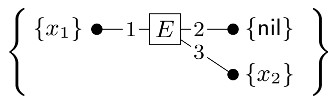
Example 7.7. We can demonstrate the application of g by taking the following small formula for a linked list:

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

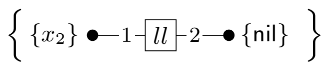
$$\mathsf{let}\ ll(x_1) = (x_1 \mapsto \mathsf{nil}, \mathsf{nil}) \vee (\exists x_2.\, (x_1 \mapsto \mathsf{nil}, x_2) * ll(x_2))\ \mathsf{in}\ \exists x.\, ll(x)$$

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.

$h[\![x_1 \mapsto \mathsf{nil}, x_2]\!] \quad =$

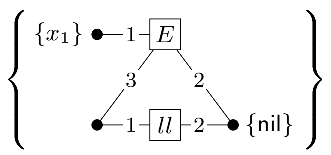


$h[\![ll(x_2)]\!] \quad =$



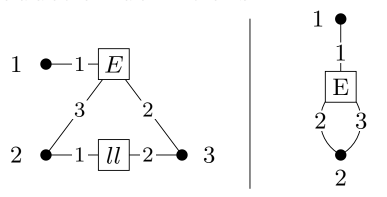
The fusion of these two graphs gives the following single graph. Note that the label x_2 has been removed as per the mapping of existentially quantified variables.

$h[\![\exists x_2.\, (x_1 \mapsto \mathsf{nil}, x_2) * ll(x_2)]\!] \quad =$



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

$ll \Rightarrow$



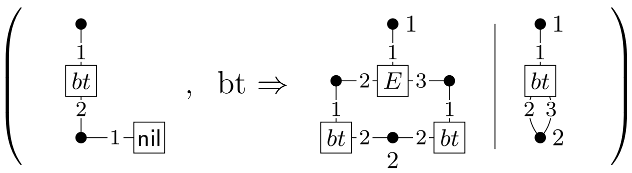
When combined with an initial graph constructed from $\exists x.\, ll(x)$, these productions give a grammar corresponding to the formula.

Example 7.8. Figure 7.2 and Figure 7.3 show a complete derivation of a graph grammar from the binary tree predicate given in Example 6.2. This derivation shows the intermediate subgraphs constructed by the function while building the grammar. Z_F is used to refer to sets of initial graphs for the constructed grammar, and P_Γ to refer to sets of constructed productions.

7.3 Proving the correctness of mapping g

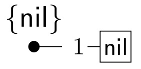
We now prove that the mapping g from formulas in $\mathcal{SLF}$ to heap-graph grammars is correct with respect to the semantics of separation logic and hyperedge-replacement, and of α, the mapping from heaps to heap-graphs.

$$g\left[\!\left[\begin{array}{l}\mathsf{let}\ bt(x_1) = (x_1 \mapsto \mathsf{nil}, \mathsf{nil}) \vee \\ \qquad (\exists x_2, x_3.\, (x_1 \mapsto x_2, x_3) * bt(x_2) * bt(x_3)) \\ \mathsf{in}\ \exists x.\, bt(x)\end{array}\right]\!\right] = \langle Z_F, P_\Gamma \rangle =$$

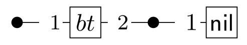


where:

$$Z_F = expose\left(g[\![\exists x.\, bt(x)]\!] \bowtie \right.$$



$$\left. , \{\mathsf{nil}\}\right) = \Big\{$$

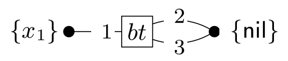


$$\Big\}$$

$$P_\Gamma = r\left[\!\left[\begin{array}{l} bt(x_1) = (x_1 \mapsto \mathsf{nil}, \mathsf{nil}) \vee \\ \quad (\exists x_2, x_3.\, (x_1 \mapsto x_2, x_3) \\ \qquad\qquad * bt(x_2) * bt(x_3)) \end{array}\right]\!\right] = \{(\sigma, g') \mid g \in \mathcal{G} \wedge lab_v(g) = \{x, \mathsf{nil}\} \ \wedge g' = \text{expose}(g, (x, \mathsf{nil}))\}$$

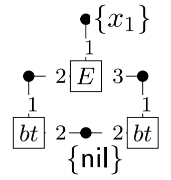
$$\mathcal{G} = g\left[\!\left[\begin{array}{l} (x_1 \mapsto \mathsf{nil}, \mathsf{nil}) \vee \\ \quad (\exists x_2, x_3.\, (x_1 \mapsto x_2, x_3) \\ \qquad\qquad * bt(x_2) * bt(x_3)) \end{array}\right]\!\right] = \mathcal{G}_1 \cup \mathcal{G}_2$$

$$\mathcal{G}_1 = g[\![(x_1 \mapsto \mathsf{nil}, \mathsf{nil})]\!] = \Big\{$$



$$\Big\}$$

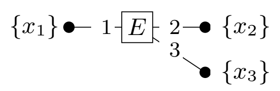
$$\mathcal{G}_2 = g\left[\!\left[\begin{array}{c} \exists x_2, x_3.\, (x_1 \mapsto x_2, x_3) \\ * bt(x_2) * bt(x_3) \end{array}\right]\!\right] = \mathcal{G}_3 \setminus \{x_2, x_3\} = \Bigg\{$$

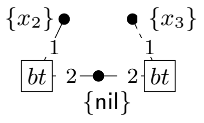


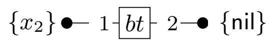
$$\Bigg\}$$

Figure 7.2: Transforming the binary tree predicate from Example 6.2 into the corresponding hyperedge replacement grammar.

$$\mathcal{G}_3 = g[\![(x_1 \mapsto x_2, x_3) * bt(x_2) * bt(x_3)]\!] = \mathcal{G}_4 \bowtie \mathcal{G}_5$$

$\mathcal{G}_4 = g[\![(x_1 \mapsto x_2, x_3)]\!] = \Big\{$  $\Big\}$

$\mathcal{G}_5 = g[\![bt(x_2) * bt(x_3)]\!] = \mathcal{G}_6 \bowtie \mathcal{G}_7 = \Big\{$  $\Big\}$

$\mathcal{G}_6 = g[\![bt(x_2)]\!] = \Big\{$  $\Big\}$

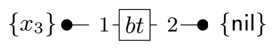
$\mathcal{G}_7 = g[\![bt(x_3)]\!] = \Big\{$  $\Big\}$

Figure 7.3: Transforming the binary tree predicate from Example 6.2 into the corresponding hyperedge replacement grammar (cont).

We first define our requirement for correctness. For the mapping to be correct, if we map a separation logic formula to its corresponding grammar using g and construct the language for the grammar, we get the same set of graphs as if we construct the satisfying set of heaps, and mapped the set of heaps to graphs using α. In other words, the mapping g must be semantics-preserving.

Formally we say that the mapping g between separation logic formulas and graph grammars is correct if for all formulas $F \in \mathcal{SLF}$ and heaps h, $h, i_0, \eta_0 \models F$ if and only if $\alpha(h) \in \mathrm{L}(g[\![F]\!])$. This correctness requirement can be visualised as the requirement that the following square commutes.

$$\begin{tikzcd}
\mathcal{HGG} \arrow[r, "\mathrm{L}(-)"] & Pow(\mathcal{HG}) \\
\mathcal{SLF} \arrow[u, "{g[\![-]\!]}"] \arrow[r, "\models"'] \arrow[ur, phantom, "="] & Pow(\mathcal{HE}) \arrow[u, "\alpha(-)"']
\end{tikzcd}$$

Our strategy for proving correctness is as follows:

1. We define the notion of a *graph environment* and define a correspondence between graph environments and predicate interpretations (Definition 7.7 and Definition 7.8).

2. We show that a let-free formula F evaluated in a given predicate interpretation corresponds to the set of graphs $h[\![F]\!]$ evaluated in the corresponding graph environment (Proposition 7.3 and Theorem 7.4).
3. We show that the predicate interpretation defined by a flat formula F corresponds to the graph environment defined by the productions in $g[\![F]\!]$ (Lemma 7.5).
4. We apply (2) and (3) to prove that g is correct (Theorem 7.6).

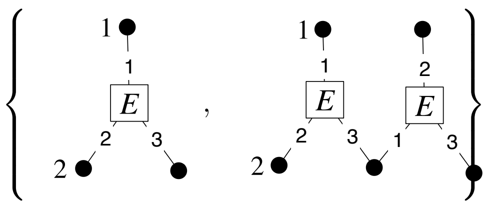
We have found in practice that plain sets productions (as defined in §2.3) are unwieldy to reason about, especially when proving results about fixed-points and let-formulas. Instead we define the notion of a graph environment that records the set of terminal graphs for each non-terminal edge. We also define a notion of evaluation in a graph environment, which substitutes all non-terminal edges with some corresponding terminal graph.

Definition 7.6 (graph environment, graph evaluation[2])**.** Let N be a set of non-terminal symbols. A graph environment $L : N \to Pow(\mathcal{H}_C)$ over N maps each non-terminal symbol to a (possibly infinite) set of terminal graphs with the same number of external nodes. A graph H is *evaluated* in a graph environment L, denoted $\kappa(L, H)$, by replacing all of the non-terminal edges in H labelled $\sigma \in \mathrm{dom}(L)$ with some terminal graph $r \in L(\sigma)$. This gives the set of graphs

$$\kappa(L,H) = \left\{H[repl : E_H^N \to \mathcal{H}_T] \mid repl(e) \in L(l_E(e)) \text{ for } e \in E_H^N\right\}.$$

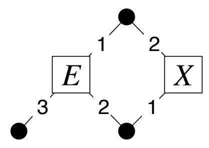
Example 7.9 (graph environment, graph evaluation)**.** The graph environment L maps the single nonterminal symbol X to two possible terminal heap-graphs.

$L(X) =$

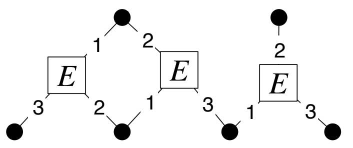


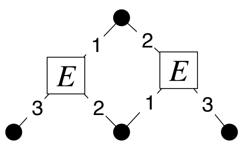
Let H be the following nonterminal graph.

[2]This definition of a graph environment is modified from the discussion of hyperedge replacement and fixed-points in [27].



Evaluating H in environment L (written $\kappa(L, H)$) results in the following pair of distinct terminal graphs.



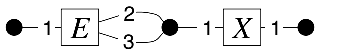
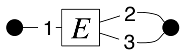


We also define an alternative formulation for the productions of a graph grammar, more suitable for use in fixed-point calculations. This definition is taken from [38].

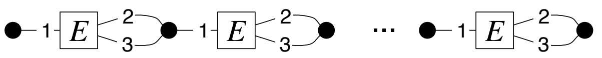
Definition 7.7 (equation system)**.** Let $H = \langle T, N, P, Z \rangle$ be a hyperedge replacement grammar. The *equation system* EQ_P associated with H is a function $EQ_P : N \to Pow(\mathcal{H}_{N \cup T})$, defined as $EQ_P(A) = \{R \mid (A, R) \in P\}$.

We say that a graph environment L is the fixed-point of an equation system if $L(A) = \kappa(L, EQ_P(A))$ for all symbols in $A \in \mathrm{dom}(EQ_P)$. If an environment L is such a least fixed-point for a set of productions P, we say that L is the environment *defined by* P.

Example 7.10 (equation system, least fixed-point)**.** The following equation system EQ assigns a pair of graphs to the nonterminal symbol X.

$EQ(X) =$ {  ,  }

The fixed-point L for this equation system maps X to the infinite family of graphs of the following form (as well as the small graphs of size 1, 2 etc.).



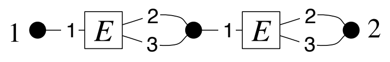
In [27], it is proved that the least fixed-point of the equation system associated with H is the language family defined by H. That is, suppose we have a grammar $H = \langle T, N, P, Z \rangle$. The language family defined for H is L_H, and the language of H can be constructed by applying the language family to the set of initial graphs, $\kappa(L_H, Z)$. Let EQ_P be the equation system defined by P, and L_P the least fixed-point of the equation system. Then it must be true that $L_P = L_H$, and consequently $\kappa(L_P, Z)$ also defines the language of H.

The consequence of this result is that we can use the fixed-point for a grammar's equation system to reason about the language defined by the grammar. The semantics of a separation logic let-statement is defined in terms of a least fixed-point, and we later show that formulas and grammars correspond by showing that the fixed-point of a let-statement corresponds to the fixed-point of a equation system.

Graph environments and predicate interpretations play the same role in their respective domains: that of defining for a 'symbol' (either a non-terminal or predicate) a class of structures for which the symbol can stand. Both also define attachment points that are used to replace the non-terminal in the structure with a corresponding terminal structure. It therefore makes sense to define a notion of correspondence between them that holds when the sets of structures for each symbol are the same, modulo α.

Definition 7.8 (correspondence)**.** Let graph H be a heap-graph with n external nodes. Let h be a heap and $\vec{x} = x_1, \ldots, x_n$ be a sequence of locations. We say that H *corresponds* to the pair $(\vec{x}, h)$ if applying the mapping function $\alpha(h)$ results in H and the heap-location x_i is mapped to the ith external node for $i = 1 \ldots n$. A predicate interpretation η *corresponds* to a graph environment L if: (1) $\text{dom}(\eta) = \text{dom}(L)$. (2) For pair $(\vec{x}, h) \in \eta(s)$ there exists a corresponding graph $H \in L(s)$. (3) For every graph $H \in L(s)$ there exists a corresponding pair $(\vec{x}, h) \in \eta(s)$.

Example 7.11 (correspondence)**.** The following graph environment L just maps the symbol X to a single terminal graph with two external nodes.

$L = \{X \mapsto \{$  $\}\}$

The following predicate interpretation η corresponds to L.

$$\eta = \{X \mapsto \{\langle [l_1, l_3], \{l_1 \mapsto \langle l_2, l_2 \rangle, l_2 \mapsto \langle l_3, l_3 \rangle\}\rangle\}\}$$

In this correspondence location l_1 corresponds to external node 1, and location l_3 corresponds to external node 2.

We now prove some subsidiary results needed for the main proof of correctness.

Lemma 7.1. *Let L be a graph environment and G_1, G_2 and G_3 be tagged graphs. Then $G_3 \in \kappa(L, G_1 \bowtie G_2)$ iff $G_3 \in \kappa(L, G_1) \bowtie \kappa(L, G_2)$.*

Proof. The tagged graphs G_1 and G_2 are joined by $\bowtie$ by merging nodes, meaning that the new graph $G_1 \bowtie G_2$ contains the same set of non-terminal edges as the two original graphs. If we consider an individual node or edge in G_1 or G_2 we can see that the same connections between nodes must be constructable by either merging first and then applying the environment, or the other way round. $\square$

Lemma 7.2 (decomposing heap-graphs)**.** *Let L be a graph environment, and let $\mathcal{G}_1$ and $\mathcal{G}_2$ be sets of tagged heap-graphs. Let h be a heap and i a variable interpretation. Heap-graph $\alpha_t(h, i)$ is a member of set $\kappa(L, \mathcal{G}_1) \bowtie \kappa(L, \mathcal{G}_2)$ if and only if there exist heaps h_0,h_1 and interpretations i_0,i_1 such that $h = h_0 \cdot h_1$, and $i = i_0 \cup i_1$, and $\alpha_t(h_0, i_0)$ is a member of $\kappa(L, \mathcal{G}_1)$ and $\alpha_t(h_1, i_1)$ is a member of $\kappa(L, \mathcal{G}_2)$.*

Proof. Assume that $\alpha_t(h, i)$ is a member of $\kappa(L, \mathcal{G}_1) \bowtie \kappa(L, \mathcal{G}_2)$. By the definition of $\bowtie$ we can pick graphs $g_0 \in \mathcal{G}_1$ and $g_1 \in G_2$ such that $\alpha_t(h, i) = \kappa(L, g_0) \bowtie \kappa(L, g_1)$. Each defined location in the heap must be mapped to a node in either $\kappa(L, g_0)$ or $\kappa(L, g_1)$, or both. We must be therefore able to construct heaps h_0,h_1 and interpretations i_0,i_1 by splitting h and i, such that such that $h = h_0 \cdot h_1$, and $i = i_0 \cup i_1$, and $\alpha_t(h_0, i_0)$ is a member of $\kappa(L, \mathcal{G}_1)$ and $\alpha_t(h_1, i_1)$ is a member of $\kappa(L, \mathcal{G}_2)$. The converse result follows immediately from the definition of $\bowtie$ and α. $\square$

We now state a correspondence result for the mapping h over let-free formulas. This result forms the basis of our correctness claim for let-free formulas.

Proposition 7.3 (κ and let-free formulas)**.** *Let formula $F \in \mathcal{SL}$ be a let-free separation-logic formula, and s a heap, i a variable interpretation and η a predicate interpretation. Let L be a graph environment corresponding to η. Then $s, i, \eta \models F$ if and only if $\alpha_t(s, i) \in \kappa(L, h[\![F]\!])$.*

Proof. The proof proceeds by structural induction on the structure of formula F. Note that the use of the mapping α_t restricts these results to the domain of heap-graphs. Later in the chapter we give a result for the language defined by a grammar using the normalisation operator.

emp

Formula emp is satisfied by the empty heap, and $h[\![\mathsf{emp}]\!]$ results in the empty graph. Therefore trivially $s, i, \eta \models \mathsf{emp}$ if and only if $\alpha_t(s,i) \in \kappa(L, h[\![\mathsf{emp}]\!])$.

$x \mapsto V_1, V_2$

We can see that $s, i, \eta \models x \mapsto V_1, V_2$ if and only if $\alpha_t(s,i) \in \kappa(L, h[\![x \mapsto V_1, V_2]\!])$. After applying α_t to s and i, we must have a graph with a single terminal hyperedge and three nodes. The label unification $\Downarrow$ ensures that vertices which are assigned the same logical variable are correctly merged. Simple enumeration of all possible instances then shows that all cases are correct.

$\sigma(V_1, \ldots, V_n)$

If: Assume $s, i, \eta \models \sigma(V_1, \ldots, V_n)$. Therefore it must be true that the pair $(([\![V_1]\!]s, \ldots, [\![V_n]\!]s), h)$ is in $\eta(\sigma)$. Then in the heap s and interpretation i, V_1 must point to the first value in $[\![V_1]\!]i$, and so on for $V_2 \ldots V_n$. By the definition of h the singleton set $h[\![\sigma(V_1, \ldots, V_n)]\!]$ must consist of a graph with only a single σ-labelled hyperedge. Let L be a graph environment that corresponds to η. Then by the correspondence of η and L, it must be true that $\alpha_t(s,i) \in \kappa(L, h[\![\sigma(V_1, \ldots, V_n)]\!])$.

Only if: Assume $\alpha_t(s,i) \in \kappa(L, h[\![\sigma(V_1, \ldots, V_n)]\!])$. By the same argument used in the 'if' case, it must be true that $s, i, \eta \models \sigma(V_1, \ldots, V_n)$.

$P \vee Q$ Consequence of the semantic equivalence between disjunction and union.

$P * Q$

If: Assume $s, i, \eta \models P * Q$. Therefore there must exist interpretations i_0, i_1 and heaps s_0, s_1 such that $s_0 \cdot s_1 = s$ and $s_0, i_0, \eta \models P$ and $s_1, i_1, \eta \models Q$. By the inductive assumption $\alpha_t(s_0, i_0) \in \kappa(L, h[\![P]\!])$. The same holds for $h[\![Q]\!]$. Applying $h[\![P * Q]\!]$ gives $h[\![P]\!] \bowtie h[\![Q]\!]$. From Lemma 7.2 we know that if $\alpha_t(s_0, i_0) \in \kappa(L, h[\![P]\!])$ and $\alpha_t(s_1, i_1) \in \kappa(L, h[\![Q]\!])$ and $s_0 \cdot s_1$ is defined, then $\alpha_t(s_0 \cdot s_1, i_0 \cup i_1) \in \kappa(L, h[\![P]\!]) \bowtie \kappa(L, h[\![Q]\!])$. By Lemma 7.1, $\alpha_t(s_0 \cdot s_1, i_0 \cup i_1) \in \kappa(L, h[\![P]\!] \bowtie h[\![Q]\!])$.

Only if: Assume $\alpha_t(s,i) \in \kappa(L, h[\![P]\!] \bowtie h[\![Q]\!])$ and $\alpha_t(s,i)$ is a heap-graph. From Lemma 7.1 and Lemma 7.2 we conclude there exist variable interpretations i_0, i_1 and heaps s_0, s_1 such that $\alpha_t(s,i) = \alpha_t(s_0 \cdot s_1, i_0 \cup i_1)$ and

$\alpha_t(s_0, i_0) \in \kappa(L, h\llbracket P \rrbracket)$ and $\alpha_t(s_1, i_1) \in \kappa(L, h\llbracket Q \rrbracket)$. By the inductive assumption, we have $s_0, i_0, \eta \models P$ and similarly $s_1, i_1, \eta \models Q$. We therefore conclude that $s_0 \cdot s_1, i_0 \cup i_1, \eta \models P * Q$.

$\exists x. P$ Consequence of the semantic correspondence between variables and tags in α_t. □

The normalisation operator $\lfloor - \rfloor$ is applied to the result of g to ensure that the class of graphs only includes heap-graphs. The result of Prop. 7.3 applies only to g applied in the presence of the evaluation function κ, but it can be used to prove the correctness of g over let-free formulas.

Theorem 7.4 (correctness of g over let-free formulas)**.** *Let formula $F \in \mathcal{SL}$ be a let-free separation logic formula, and let h be a heap. Then $h, i_0, \eta_0 \models F$ if and only if $\alpha(h) \in \mathrm{L}(\lfloor g\llbracket F \rrbracket \rfloor)$. Also for graph H with label alphabet $\{E, \mathsf{nil}\}$, if $H \in \mathrm{L}(\lfloor g\llbracket F \rrbracket \rfloor)$ then H is a heap-graph.*

Proof. All graphs in the language $L(\lfloor g\llbracket F \rrbracket \rfloor)$ must be heap-graphs by the definition of the normalisation operator. Let $\langle Z, P \rangle = \lfloor g\llbracket F \rrbracket \rfloor$, and let L be the graph environment defined by P. By the definition of a graph environment, we know that $\mathrm{L}(\lfloor g\llbracket F \rrbracket \rfloor) = \kappa(L, Z)$. As F is let-free, by the definition of g it must be true that $P = \emptyset$. Therefore L must be the empty graph environment, where for all non-terminal symbols σ, $L(\sigma) = \emptyset$. This means that L corresponds to the empty predicate interpretation η_0. Therefore, by the result of Proposition 7.3 and the correctness of normalisation, $h, i, \eta_0 \models F$ if and only if $\alpha_t(h, i) \in \kappa(L, Z)$, which holds if and only if $\alpha_t(h, i) \in \mathrm{L}(\lfloor g\llbracket F \rrbracket \rfloor)$. □

We can extend this result further to prove the correctness of g for formulas of the form $\mathsf{let}\,\Gamma\,\mathsf{in}\,P$, where both Γ and P are let-free. We begin by proving a lemma showing that the predicate interpretation defined by a let corresponds to the environment defined by the corresponding set of productions constructed by g.

Recall that the semantics of let given in §6.1.3 makes use of the following if and only if expression:

$$h, i, \eta \models \mathsf{let}\,\Gamma\,\mathsf{in}\,F \quad \text{iff} \quad h, i, \eta\,[\beta \mapsto k(\beta)]_{\beta \in \mathrm{dom}(\Gamma)} \models F$$
$$\text{where } k = \text{fix } f_{\eta,\Gamma}$$

As discussed previously, the new predicate interpretation $\eta[\beta \mapsto k(\beta)]$ is defined over the symbols in $\mathrm{dom}(\Gamma)$ by the predicate interpretation k, which is the least-fixed-point of the following function.

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

We use this definition to state our lemma about the correspondence of graph environments and predicate interpretations.

Lemma 7.5 (environmental correspondence for let). *Let* let Γ in F *be a flat formula. The least fixed-point of $f_{\eta_0,\Gamma}$ corresponds to the graph environment L defined by the set of rules $r[\![\Gamma]\!]$.*

Proof. Let k be the least fixed-point of $f_{\eta_0,\Gamma}$. That is, k the least predicate interpretation such that $k = f_{\eta_0,\Gamma}(k, \sigma)$ for all symbols $\sigma \in \mathrm{dom}(\Gamma)$. As stated earlier in the section, a graph environment L is defined by a set of rules P if L is the least environment satisfying $L(\sigma) = \kappa(L, EQ_\Gamma(\sigma))$ for any symbol $\sigma \in \mathrm{dom}(EQ_\Gamma)$, where EQ_Γ is the equation system defined by P.

Let k_i be a predicate interpretation, and L_i be a corresponding graph environment. Now let $k_j(\sigma) = f_{\eta_0,\Gamma}(k_i, \sigma)$, and let $L_j(\sigma) = \kappa(L_i, EQ_\Gamma(\sigma))$ for all $\sigma \in \mathrm{dom}(\Gamma)$. To show our result we need to show that k_j corresponds to L_j.

Let us look at the definitions of these two functions for a particular symbol σ. First consider k_j, which is defined by instantiating the definition of the fixed-point function. In the following equation, the expression $\eta\left[\beta \mapsto k_i(\beta)\right]_{\beta\in\mathrm{dom}(\Gamma)}$ is replaced by k_i because the surrounding predicate interpretation is η_0.

$$\begin{aligned} k_j(\sigma) \quad = \quad & \text{bind } (\sigma(\vec{x}) = Q) \text{ to } \Gamma(\sigma) \\ & \text{in } \{\vec{v}, h) \mid (h, [\vec{x} \mapsto \vec{v}], k_i \models Q\} \end{aligned}$$

The graph environment L_j is defined by the following application of $\kappa(L_i, -)$, taken from the definition of a graph environment.

$$L_j(\sigma) = \kappa(L, EQ_\Gamma(\sigma))$$

From the result of Proposition 7.3, we know that a heap s, variable interpretation i and predicate interpretation η satisfy the formula Q if and only if the graph $\alpha_t(s,i)$ is a member of the class of graphs $\kappa(L, h[\![Q]\!])$, where L is a graph environment corresponding to η.

We now have to show that a pair $(\vec{v}, h)$ is a result of the function if and only if the corresponding graph G is a member of the set of graphs constructed from $g[\![Q]\!]$. A pair $(\vec{v}, h)$ is a member of the set if $([\vec{x} \mapsto \vec{v}], h), k_i \models Q$. By Prop 7.3, this set of pairs corresponds to the set of graphs g' such that there exists a $g \in \kappa(L_i, Q)$ where $var(g) = \vec{x}, \mathsf{nil}$ and g is exposed to g' with $\{\vec{x}, \mathsf{nil}\}$. This set of graphs is equal to the set of graphs defined by applying $\kappa(L, -)$ to the right-hand sides in $EQ_\Gamma(\sigma)$. Consequently the set of pairs in $k_j(\sigma)$ corresponds to the set of graphs in $L_j(\sigma)$.

As a result, any predicate interpretation k that is a fixed point of the function $f_{\eta_0,\Gamma}$ must correspond to an environment L that is a fixed-point of κ applied to the equations in EQ_Γ. The correspondence between predicate interpretation and graph environments is monotone (that is, if $k_1 \leq k_2$ and k_1 corresponds to L_1 and k_2 corresponds to L_2 then $L_1 \leq L_2$, and vice versa), so the least fixed-point k_0 corresponds to the least fixed-point L_0. This proves our result. $\square$

We now use this lemma to prove that g is correct when applied to a let-formula with let-free sub-arguments.

Theorem 7.6. *Let formula $F' = \mathsf{let}\,\Gamma\,\mathsf{in}\,F$ be a flat separation logic formula. 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. A heap h, variable interpretation i and predicate interpretation η satisfies the formula if the heap satisfies formula F' in predicate interpretation η_k, formed by redefining $\eta(\sigma)$ to $k(\sigma)$ for all $\sigma \in \mathrm{dom}(k)$. We have shown in lemma 7.5 that the k defined by the forcing relation from Γ in environment η_0 must correspond to the graph environment L defined by the set of rules $r[\![\Gamma]\!]$. In the case of formula F' we have that $\eta = \eta_0$, so $\eta_k = k$.

Because F is let-free, we know that in the pair $\langle Z_F, P_\Gamma \rangle = g[\![F']\!]$, $P_\Gamma = \emptyset$. Because F is let-free, we know that in the pair $\langle Z', P_F \rangle = g[\![F]\!]$, $P_F = \emptyset$ and $Z' = Z_F$. So a graph is in $\mathrm{L}(g[\![F]\!])$ if and only if it is in $\kappa(L, Z_F)$.

We have shown in Proposition 7.3 that for any let-free formula K and pair $\langle Z, P \rangle = g[\![K]\!]$, and corresponding predicate interpretation η and graph environment L, $h, i, \eta \models K$ if and only if $\alpha_t(h,i) \in \kappa(L, Z)$.

Applying these results gives us the following logical expression that holds for any heap h.

$$\begin{array}{llll} h, i_0, \eta_0 \models F' & \iff & h, i_0, k \models P & \textit{(semantics of } \mathsf{let}\textit{)} \\ & \iff & \alpha(h) \in \kappa(L, Z_{F'}) & \textit{(equivalence of } k \textit{ and } L\textit{)} \\ & \iff & \alpha(h) \in \mathrm{L}(g[\![F']\!]) & \textit{(}P \textit{ is } \mathsf{let}\textit{-free)} \end{array}$$

This completes the proof. □

We have only proved this result for the class of flat formulas in our fragment of separation logic. However, we have shown in §6.2 that any formula in our fragment can be transformed into a corresponding flat formula, so by composing the flattening function *flat* with g we can construct the mapping $g \circ \mathit{flat}$ from any formula in the fragment to a semantically equivalent graph grammar.

7.4 Mapping from grammars to formulas

We now define the function $s : \mathcal{HRH} \to \mathcal{SLF}$, 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 7.4.

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.

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.

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

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.

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

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

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

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

$$\begin{array}{l} s_q[\![EQ_P]\!] \stackrel{\text{def}}{=} \ \ \sigma(x_1, \ldots, x_n, \mathsf{nil}) = s_G[\![EQ_P(\sigma)]\!],\ s_q[\![EQ_P \setminus \sigma]\!] \\ \quad \text{where } n = \text{ari}(\sigma) \end{array}$$

Figure 7.4: Mapping from heap-graph grammars to separation logic formulas.

The function s_G maps a set of tagged heap-graphs to a disjunction of formulas, each of which correspond 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 graph edge. The conjunction for a graph is wrapped in existential quantifications that quantify out variables corresponding to internal nodes.

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

The function s_q takes the equation system EQ_P corresponding to a set of productions P (see Def. 7.7). It 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 nonterminal symbol, something that is easier to specify in an equation system.

s_q constructs a single recursive definition for each nonterminal symbol in P. The set of right-hand side graphs for a single symbol are mapped to a right-hand side formula, using the function s_G for sets of graphs. The arguments to the predicate are given $x_1, \dots, x_n$, which by the definition of the tagging function form the free variables of the right-hand side formula.

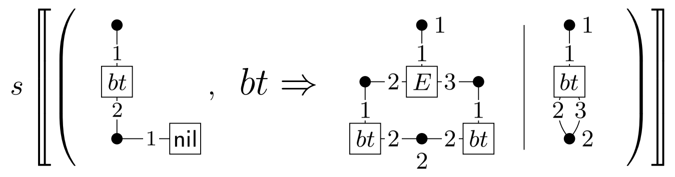
Figure 7.5 shows the derivation of a separation-logic formula from the binary tree predicate given in Example 7.1. We omit the tagging function, as the mappings from nodes to variables should be obvious. As with Figure 7.2, to fit the example on the page, arrows are used to indicate the values of some intermediate variables.

This example shows the derivation of individual elements in the formula. s_g constructs the let-statement body $\exists x.\, bt(x, \mathsf{nil})$ from the single initial graph. For the right-hand side of the definition of bt, s_q constructs the two disjuncts $\exists y_1, y_2.\, x_1 \mapsto y_1, y_2 * bt(y_1, x_2) * bt(y_2, x_2)$ and $(x_1 \mapsto x_2, x_2)$ from the two production right-hand sides. The derivation shows these elements being glued together to form the final formula, which is identical to the one in example 6.2 (modulo bound variable renaming).

7.5 Proving the correctness of mapping s

In this section we prove that the mapping s, as defined in the previous section, is correct. For s to be correct, it must be true that for all hyperedge-replacement grammars $H = \langle T, N, P, Z \rangle$ and heap-graph G, $G \in \mathrm{L}(H)$ if and only if $\alpha^{-1}(G), i_0, \eta_0 \models s[\![\langle Z, P \rangle]\!]$. This can be visualised as a the requirement that the following square commutes:

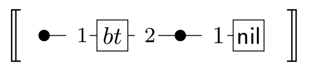
$$\begin{tikzcd}
\mathcal{HGG} \arrow[r, "\mathrm{L}(-)"] \arrow[d, "s[\![-]\!]"'] \arrow[dr, phantom, "="] & Pow(\mathcal{HG}) \arrow[d, "\alpha^{-1}(-)"] \\
\mathcal{SL} \arrow[r, "\models"'] & Pow(\mathcal{HE})
\end{tikzcd}$$

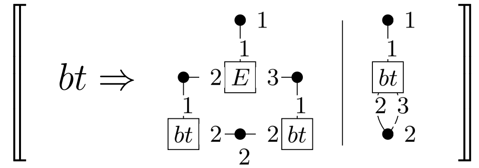
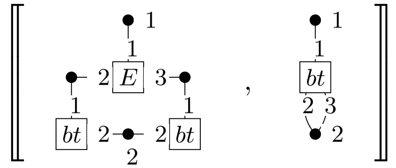


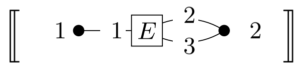
$= \mathsf{let}\ \Gamma\ \mathsf{in}\ F =$

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

where:

$F = s_g$  $= \exists x.\, bt(x, \mathsf{nil})$

$\Gamma = s_q$  $= bt(x_1, x_2) = s_G$  $= F_1 \lor F_2$

$F_1 = s_g$  $= (x_1 \mapsto x_2, x_2)$

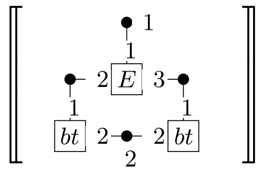
$F_2 = s_g$  $= \exists y_1, y_2.\, x_1 \mapsto y_1, y_2 * bt(y_1, x_2) * bt(y_2, x_2)$

Figure 7.5: Transforming the binary tree grammar from Example 7.1 into a corresponding separation logic formula.

We prove our correctness result incrementally beginning with single graphs, moving to sets of graphs, and then finally proving the result for full grammars. Because the non-terminal symbols and predicates must be modelled in the proof of correctness, our results are proved with the addition of graph environments and corresponding predicate interpretations (see Definition 7.8).

We first prove that s is correct for single graphs (Lemma 7.7), then sets of graphs (Lemma 7.8). We then apply this result to prove that the let-formula defined by s from a grammar H defines a predicate interpretation corresponding to the graph environment defined by H (Lemma 7.9). Finally we apply the previous results to prove the overall correctness of g (Theorem 7.10).

In the following we use the function α_t^{-1} (Def. 6.10) to map from a tagged heap-graph g to a pair (h, i) consisting of a heap h and a variable interpretation i. To shorten proofs involving α_t^{-1}, we sometimes write $(h, i), \eta \models F$ to stand for $h, i, \eta \models F$.

Lemma 7.7. *Let G and H be tagged heap-graphs with n external nodes. Let L be a graph environment and η a predicate interpretation corresponding to L. Then $H \in \kappa(L, G)$ if and only if $h, i', \eta \models s_g[\![G]\!]$, where $(h, i) = \alpha_t^{-1}(H)$ and i' is i restricted to the domain $\{x_1, \ldots, x_n\}$.*

Proof. We first show for graphs G and H where all nodes are external that $H \in \kappa(L, G)$ if and only if $\alpha_t^{-1}(H), \eta \models s_g[\![G]\!]$. The proof works by induction on the number of hyperedges in the graph G.

In the base case there is only a single hyperedge in the graph. There are two possible cases for a heap-graph G_1 containing a single hyperedge: either the hyperedge can be an E-labelled terminal hyperedge or a nonterminal hyperedge with some label σ.

If the graph G_1 contains a terminal edge, then $\kappa(L, -)$ performs no substitution and so $H \in \kappa(L, G)$ if $H = G$. The resulting expression $s_g[\![G_1]\!]$ consists of $x_1 \mapsto x_2, x_3$. By the semantics of α_t^{-1}, $\alpha_t^{-1}(H), \eta \models s_g[\![G]\!]$.

If the graph consists of a non-terminal edge labelled σ, then the resulting expression $s_g[\![G_1]\!]$ will consist of some predicate $\sigma(x_1, \ldots, x_n, \mathsf{nil})$. By assumption, the environments η and L correspond. Therefore $\kappa(L, -)$ must replace the σ-labelled hyperedge with a heap subgraph corresponding to one of the heaps in $\eta(\sigma)$. As a result, the heap $\alpha_t^{-1}(H), \eta \models s_g[\![G]\!]$.

Now consider an arbitrary heap-graph G_n with n hyperedges. G_n can be decomposed into two smaller tagged graphs G' and G'' such that $G_n = G' \bowtie G''$. By the inductive hypothesis, $H' \in \kappa(L, G')$ iff $\alpha_t^{-1}(H), \eta \models s_g[\![G']\!]$, and the same for G''.

We have shown in Lemma 7.1 that κ distributes over $\bowtie$, so $\kappa(L, G_n) = \kappa(L, G' \bowtie G'') = \kappa(L, G') \bowtie \kappa(L, G'')$. Because the two graphs are fully-labelled, H' and H'' must result in pairs of heaps and variable interpretations $\alpha_t^{-1}(H')$ and $\alpha_t^{-1}(H'')$ where the heaps have disjoint domains. separating conjunction is both associative and commutative so the formula $s_g[\![G]\!]$ can be rewritten in the form $s_g[\![G']\!] * s_g[\![G'']\!]$. By the semantics of conjunction, a state satisfies $P*Q$ if it can be decomposed into two disjoint states satisfying P and Q. By the inductive assumption, this holds, so $\alpha_t^{-1}(H) \in s_g[\![G]\!]$.

We now want to extend this result to graphs with internal nodes. We extend it by incrementally removing external nodes. Proof proceeds by induction over the number of internal nodes in the graph.

By the semantics of existential quantification, removing a node from the sequence of external nodes results in the existential quantification of the corresponding variable name. The order of node removal is immaterial because existential quantification is commutative. Any graph can be constructed by removing external nodes, so this completes the proof. □

This lemma for single graphs is now extended to sets of graphs, which are used to form the initial element of the grammar and form the basis of production definitions.

Lemma 7.8. *Let $\mathcal{G}$ be a set of heap-graphs and H a single heap-graph. Let L be a graph environment and η a corresponding separation logic environment. Then $H \in \kappa(L, \mathcal{G})$ if and only if $\alpha_t^{-1}(H), \eta \models s[\![\mathcal{G}]\!]$.*

Proof. First assume that $H \in \kappa(L, \mathcal{G})$. There must be some graph $G' \in \mathcal{G}$ such that $H \in \kappa(L, G')$. By application of Lemma 7.7, $\alpha_t^{-1}(H) \models s[\![G']\!]$. As $s[\![G']\!]$ must be an argument of the disjunction $s[\![\mathcal{G}]\!]$ we conclude $\alpha_t^{-1}(H) \models s[\![\mathcal{G}]\!]$.

Now assume that $h, i, \eta \models s[\![\mathcal{G}]\!]$ (as α_t is bijective, we can use a pair h and i in place of the corresponding heap-graph). Formula $s[\![G]\!]$ is a disjunction consisting of arguments $s[\![G'']\!]$ constructed from graphs $G'' \in \mathcal{G}$. Therefore h, i, η must satisfy at least one such argument, so we assume that $h, i, \eta \models$

$s[\![G'']\!]$. By application of Lemma 7.7, $\alpha_t(h, i) \in G''$, and $G'' \in \mathcal{G}$, which proves our result. □

As with the proof of correctness for g, we need to prove that the graph environment defined by a rule set is the same as the one defined by the separation-logic semantics for let.

Lemma 7.9. *Let $H = \langle T, N, P, Z\rangle$ be a heap-graph grammar, and let $s[\![\langle Z, P\rangle]\!] =$ let Γ in F be the resulting separation-logic formula. Then the graph environment defined by P corresponds to the environment k defined by the semantics of satisfaction when evaluating the formula in environment η_0.*

Proof. This proof uses much the same approach as the proof of Lemma 7.5 and so we give only sketch rather than the full details of the proof. We observe that both environments are defined as the fixed-point of a function over environments, and show that, if we begin with a corresponding pair of environments, the two functions always construct a pair of environments. This suffices to show that the two least fixed-points are the same, which proves our result.

To prove that the two functions over environments produce corresponding environments when passed corresponding environments, we use the result of Lemma 7.8. As in the earlier proof, we show that the correctness result for graph sets carries through to a correspondence between resulting functions. □

We now apply Lemma 7.7, Lemma 7.8 and Lemma 7.9 to prove the correctness of the mapping s from grammars to formulas.

Theorem 7.10. *Let $H = \langle T, N, P, Z\rangle$ be a heap-graph grammar. Let $G \in \mathcal{H}_C$ be a graph. Then $G \in \mathrm{L}(H)$ if and only if $\alpha^{-1}(G), \eta_0 \models s[\![\langle Z, P\rangle]\!]$.*

Proof. By the semantics, $\alpha_t^{-1}(G), \eta_0 \models s[\![\langle Z, P\rangle]\!]$ if and only if $\alpha_t^{-1}(G), k \models s_H[\![Z]\!]$, where k is defined by a fixed-point function based on η_0 and $s_q[\![P]\!]$. We have shown in Lemma 7.9 that this environment k must correspond to the graph environment defined by P.

A graph G is a member of $\langle Z, P\rangle$ if and only if it is a member of $\kappa(L, Z)$, where L is the graph environment defined by P. We concluded in Lemma 7.8 that, given corresponding environments k and L, then $G \in \kappa(L, Z)$ if and only if $\alpha_t^{-1}(G) \models s_H[\![P]\!]$. We have shown that the environments

k and L must correspond, so we conclude that $G \in \mathrm{L}(H)$ if and only if $\alpha_t^{-1}(G), \eta_\emptyset \models s[\![\langle Z, P\rangle]\!]$. $\square$

Note that Theorem 7.10 proves a full correctness result for s, rather than correctness over a restricted domain of flattened formulas as with our results for g. This difference is due to the fact that there is no nesting of recursive definitions in the definition of hyperedge replacement grammars, which makes a full proof considerably simpler than for separation-logic formulas.

Chapter 8

Consequences and limitations

In this chapter we examine some of the consequences of our translation from separation logic formulas into hyperedge replacement grammars, and vice versa. We examine both the theoretical and practical reasons that our correspondence is interesting. We also look at the limitations of our approach, some of which can be overcome by further work, and some of which are theoretical limits that cannot be overcome without changing our basic approach.

The chapter is structured as follows. In section 8.1 we examine the separation-logic semantics of some of the constructs not present in our fragment, and prove that formulas featuring these constructs *cannot* be modelled in general by a hyperedge-replacement grammar. In §8.2 we consider an extension of the heap-model permitting tuples, rather than pairs. In §8.3 we describe the consequences of the correspondence for both separation logic and hyperedge replacement. Finally, in §8.4 we consider some of the related work on separation logic, grammars, and expressiveness.

8.1 Inexpressible separation logic operators

The fragment of separation logic we have chosen has been restricted so that it is expressible by hyperedge replacement grammars. We claim in §6.1 that the omitted constructs are fundamentally non-context-free. In this section we justify this claim.

$h, i, \eta \models \texttt{true}$	holds always	
$h, i, \eta \models P \wedge Q$	iff	$h, i, \eta \models P$ and $h, i, \eta \models Q$
$h, i, \eta \models \neg P$	iff	$h, i, \eta \not\models P$
$h, i, \eta \models P \mathrel{-\!\!*} Q$	iff	$\forall h'.$ if $h' \# h$ and $h, i, \eta \models P$ then $(h' \cdot h), i, \eta \models Q$

Figure 8.1: Definition of satisfaction for separation logic operators not present in our fragment.

To do this, we extend our semantics for satisfaction to include separation logic operators not present in our initial fragment. The extended semantics is defined in Figure 8.1. As with our basic semantics for satisfaction, this definition is based heavily on [74, 52].

8.1.1 Conjunction

Conjunction is omitted from our fragment because it corresponds to language intersection, and the class of HR-definable languages of heap-graphs is not closed under intersection. A conjunction $P \wedge Q$ is satisfied by any heap h satisfying both P and Q. Suppose that P and Q correspond in our mappings to grammars $g[\![P]\!]$ and $g[\![Q]\!]$. Then the grammar $g[\![P \wedge Q]\!]$ should construct the set $\mathrm{L}(g[\![P]\!]) \cap \mathrm{L}(g[\![Q]\!])$.

It is shown in [39] that the class of hyperedge-replacement languages is not closed under intersection. The proof of this works by defining the following pair of string languages:

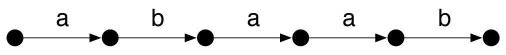
$$L_1: \quad \{a^{n_1}b^{n_1}a^{n_2}b^{n_2}\dots b^{n_{k-1}}a^{n_k}b^{n_k} \mid k \geq 1 \wedge n_1 \dots n_k \geq 1\}$$

$$L_2: \quad \{a^{n_1}b^{n_2}a^{n_2}b^{n_3}\dots b^{n_k}a^{n_k}b^{n_{k+1}} \mid k \geq 1 \wedge n_1 \dots n_{k+1} \geq 1\}$$

These definitions use the normal notation for strings, so the string $a^n b^m$ consists of n 'a' characters followed by m 'b' characters. L_1 and L_2 are used to define languages of string-graphs.

Definition 8.1 (string-graph)**.** The graph $w^\bullet$ is the string-graph corresponding to string w. A string-graph for string $w = c_1 c_2 \dots c_n$ consists of $n+1$ unlabelled nodes $v_1, \dots, v_{n+1}$ and n labelled edges such that edge e_i has source v_i and target v_{i+1} and label c_i. The graph language $L^\bullet$ is the language of string-graphs $\{w^\bullet \mid w \in L\}$.

Example 8.1 (string-graph)**.** For the string 'abaab', the corresponding string-graph $(abaab)^\bullet$ is as follows.



L_1 and L_2 are used to define the HR languages $L_1^\bullet$ and $L_2^\bullet$. The intersection between $L_1^\bullet$ and $L_2^\bullet$ gives the following language:

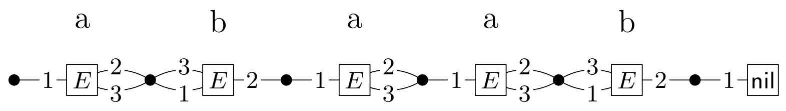
$$L_1^\bullet \cap L_2^\bullet = \{((a^n b^n)^k)^\bullet \mid k \geq 1 \wedge n \geq 1\}$$

In [27] it is shown by application of the pumping lemma for hyperedge replacement languages that $L_1^\bullet \cap L_2^\bullet$ is HR-inexpressible. However, this proof applies to the class of unrestricted HR languages. To show that intersection is not possible in the restricted domain of heap-graph languages we must alter the proof.

First we define a new encoding. Rather than using edge-labels to record character names, we use configurations of edges. Our string encoding is defined over characters 'a' and 'b' only.

Definition 8.2 (string heap-graph)**.** The graph w° is the string heap-graph corresponding to string w. A string heap-graph for string $w = c_1 c_2 \ldots c_n$ consists of $n+1$ unlabelled nodes $v_1 \ldots v_{n+1}$ and n E-labelled edges $e_1 \ldots e_n$. If character c_n is an 'a', then the edge E has attachment points $[v_n, v_{n+1}, v_{n+1}]$. If c_n is a 'b' then the edge has attachment points $[v_n, v_{n+1}, v_n]$. The graph language L° is the language $\{w^\circ \mid w \in L\}$.

Example 8.2 (string heap-graph)**.** For the string 'abaab', the corresponding string heap-graph $(\text{abaab})^\circ$ is as follows.



Let H_1 and H_2 be HR grammars defining L_1° and L_2° respectively. Applying our translation function s allows us to construct formulas $P_1 = s[\![H_1]\!]$ and $P_2 = s[\![H_2]\!]$.

Now suppose we include conjunction in our fragment of separation logic. The set of heaps satisfying the formula $P_1 \wedge P_2$ would correspond to the class of heap graphs $\{((a^n b^n)^k)^\circ \mid k \geq 1 \wedge n \geq 1\}$. But by the same argument used in the original proof, this language is not expressible by any HR grammar. Therefore grammar $g[\![P_1 \wedge P_2]\!]$ cannot be constructed, and so conjunction cannot be modelled using hyperedge replacement.

8.1.2 Negation

Negation is prohibited in our fragment because it corresponds to language complement, and the class of HR languages of heap-graphs is not closed under complement. We use this fact to prove that complement is inexpressible. We write L^C to denote the complement of a language L.

Proposition 8.1. *The class of of HR heap-graph languages is not closed under complement.*

Proof. We have shown in the previous section that the class $\mathcal{L}_{HR}$ of HR-expressible heap-graph languages is not closed under intersection. It is known that the class $\mathcal{L}_{HR}$ is closed under union. The union of two grammars can be constructed by simply merging their respective sets of initial graphs and sets of productions by disjoint union.

Suppose now that $\mathcal{L}_{HR}$ is closed under complement. For languages A and B, by De Morgan's laws, $(A^C \cup B^C)^C = A \cap B$, so if $\mathcal{L}_{HR}$ is closed under complement, it implies that it is closed under intersection, which is false. Therefore, the class is not closed under complement. □

This result means that we cannot express separation-logic negation in a hyperedge-replacement grammar. By Proposition 8.1 there must exist a heap-graph grammar G such that the complement of its language is not HR-expressible. We construct the corresponding separation-logic formula $s[\![G]\!]$. If we apply the negation operator, to give formula $\neg s[\![G]\!]$, then by the semantics of negation, the grammar $g[\![\neg s[\![G]\!]]\!]$ must generate the complement of $L(G)$. But $\mathrm{L}(G)^C$ is HR-inexpressible, so no grammar $g[\![\neg s[\![G]\!]]\!]$ can be defined.

8.1.3 The elementary formula true

In [38], Chapter IV, 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. Heap-graphs cannot contain parallel edges, so the class of all loop-free heap-graphs is such a language of simple graphs. Consequently the set of loop-free heap graphs cannot be expressed by any hyperedge-replacement grammar.

The formula **true** is satisfied by any heap h. Suppose that we can construct a corresponding grammar $G = g[\![\mathbf{true}]\!]$. This grammar would have to construct the language of all possible heap-graphs. The loop-free property is a compatible property in the sense of [27]. This is because the loop-free property for a graph can be composed from the pairwise reachability of the external nodes of the graph's constituent subgraphs. Consequently we can construct grammar G' that defines the language of loop-free heap-graphs. No such grammar exists. Therefore no such grammar $g[\![\mathbf{true}]\!]$ can be defined.

8.1.4 Separating implication

The separating implication operator $-\!\!*$ is the separating equivalent of normal implication. A heap satisfies formula $P -\!\!* Q$ if joining it with a heap satisfying P results in a heap satisfying Q.

Conjecture 8.2. *For any DPO graph grammar G defining a language of heap-graphs, there exists a corresponding separation-logic formula F in $SL+ \{-\!\!*\}$ such that $g \in \mathrm{L}(G)$ iff $\alpha(g) \models F$.*

This conjecture rests on the observation that the 'holes' constructed by separating implication can be seen as corresponding to the context matched by the left-hand of a DPO rewrite rule.

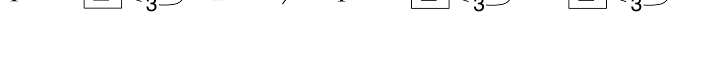
Suppose that we have a context-sensitive rule $\langle L \leftarrow K \rightarrow R \rangle$. Let L, R and K be heap-graphs. We have shown in §7.4 that we can construct a formula from any terminal heap-graph by applying the function s_g. Let F_L be a formula corresponding to the left-hand side and F_R a formula corresponding to the right-hand side. We use the interface graph K to ensure that nodes shared between the left and right-hand sides share free variable names in F_L and F_R.

Suppose we have a formula F_H describing the starting heap-graph H. Now we write the following formula representing the rule.

$$(F_L -\!\!* F_H) * F_R$$

This formula is satisfied by a heap satisfied by F_H which has had a subheap satisfying F_L removed and a subheap satisfying F_R added.

Example 8.3 (simulating DPO rules)**.** Suppose we have the following DPO rule.



Then the corresponding formula is as follows, given a formula F_H that is a description of the starting graph H.

$$\exists x, y. \, (((x \mapsto y, y) \mathrel{-\!\!*} F_H) * \exists z. \, (x \mapsto z, z * z \mapsto y, y))$$

The shared variables x and y take the place of the interface nodes between the left and right-hand graphs.

If true, this conjecture would also suffice to show that separating implication is not expressible by any hyperedge-replacement grammar. This is because languages are known to exist (such as the language of balanced binary trees or the set of all (hyper)graphs) that can be expressed by a double-pushout-based grammar but that are not HR-expressible.

8.2 Extending the heap model

To simplify both the translations and our proofs of correctness, in previous chapters we have used a pair-based heap model (see §6.1). At the cost of more complexity, we can extend this model to a more expressive heap model where locations can map to tuples of arbitrary size. In this section we sketch the extended model and describe the extension it permits to our fragment of separation logic and notion of a heap-graph grammar.

To control the size of the tuples in the heap, locations in the heap must be typed. This extension requires that we define syntactically new points-to assertions for each location type. We also must extend the set of labels permitted in a heap-graph to record the new types.

We assume a finite set Typ of *types*. Then we define a tuple-heap as follows:

Definition 8.3 (tuple-heap)**.** A *tuple-heap* $j = \langle h, t \rangle$ consists of a heap function $h \colon \mathrm{Loc} \to \mathrm{Elem}^* \cup \{\boxslash\}$, a partial function with finite domain mapping each location to a tuple of elements or $\boxslash$, and a typing function $t \colon \mathrm{Loc} \to \mathrm{Typ}$, a partial function mapping locations to types.

Definition 8.4 (tuple-type schema)**.** A tuple-type schema $s \colon \mathrm{Typ} \to \mathbb{Z}$ is a total function mapping types to integers. We say that a tuple-heap $\langle h, t \rangle$ is *well-formed* with respect to s if $\mathrm{dom}(s) = \mathrm{ran}(t)$ and for every location $l \in \mathrm{dom}(h)$, $s(t(l)) = |h(l)|$. In a well-formed tuple-heap, the type-schema gives each type T an integer size n, and every tuple stored at a location with type T is of size n.

Example 8.4 (tuple-heap, tuple-type schema)**.** Suppose we have set of types $\{X, Y, Z\}$ and a tuple-type schema s defined as follows.

$$s = \{X \mapsto 3, Y \mapsto 1, Z \mapsto 2\}$$

Of the following tuple-heaps, h_1 is well-formed, but h_2 is not well-formed because location l_2 has an arity of 3 but a type of Y, where $s(Y) = 1$.

$$\begin{aligned} h_1 = &\quad \langle \{l_1 \mapsto \langle l_2, \mathsf{nil}\rangle, l_2 \mapsto \langle l_1\rangle\}, \{l_1 \mapsto Z, l_2 \mapsto Y\}\rangle \\ h_2 = &\quad \langle \{l_1 \mapsto \langle l_2, \mathsf{nil}\rangle, l_2 \mapsto \langle l_1, \mathsf{nil}, l_2\rangle\}, \{l_1 \mapsto Z, l_2 \mapsto Y\}\rangle \end{aligned}$$

For a given type-schema s we can define a fragment of separation logic specifying well-formed tuple-heaps. For each type $T \in \mathrm{Typ}$ we define a distinct points-to assertion $x \overset{T}{\mapsto} y_1, \ldots, y_n$, where $n = s(T)$. Such a points-to assertions states that the tuple associated with location x is $\langle y_1, \ldots, y_n\rangle$.

We also define a notion of a heap-graph over s. This is defined in largely the same way as a normal heap-graph (see Definition 6.9). The main alteration is that the set of labels for such a graph is $\mathrm{Typ} \cup \{\mathsf{nil}\}$, and the arity of a terminal symbol $T \in \mathrm{Typ}$ is defined as $\mathrm{ari}(T) = s(T) + 1$. The arity of nil remains 1. A heap-graph grammar over s is defined as a grammar producing heap-graphs over s.

Once we have defined such an extension of the semantics, we can redefine our mappings between domains. Our intuitive correspondence is as before, but points-to assertions of type T now correspond to terminal edges with label T. Other elements of the mapping are unchanged. This is because the other constructs of separation logic and hyperedge replacement interact in the same way with the tuple-model as with the cons-model.

The tuple-model of separation logic includes the model given in the previous chapter as a special case. Such a model of separation logic corresponds more closely to C-like pointer structures, where each location corresponding to a structure with a fixed number of fields. The corresponding heap-graphs also correspond more closely to a general notion of hyperedge replacement, with the restriction on terminal labels removed.

It will be the subject of future work to modify the functions and proofs given in the previous chapter to match with this extended model. The proofs of correctness for such a mapping will be more complex, as the notion of a type introduces more cases to the mapping, but we do not expect that they will be conceptually more difficult.

8.3 Consequences of the correspondence

In this section we describe some of the consequences of the correspondence between our separation-logic fragment and heap-graph grammars. We look both at the theoretical consequences of the correspondence, in providing properties that can be imported between domains, and the practical usefulness of our fragment of separation logic.

First, let us define the meaning of the correspondence we have defined between these two domains. 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.

Theorem 8.3 (equivalent expressive power)**.** *Any class of structures satisfying a formula in $\mathcal{SL}$ can be generated by some heap-graph grammar, modulo α, and* vice versa*.*

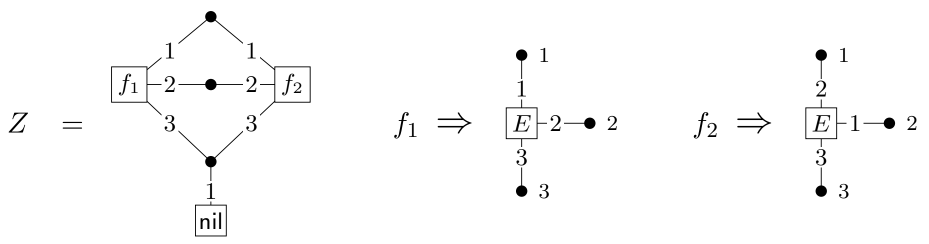
Proof. Function s is defined as a total function. The composition *flat* $\circ\, g$ of the flattening formula *flat* and g is total over formulas in $\mathcal{SL}$. The result follows as a consequence of the correctness of flattening (Corollary 6.5) and of the correctness of g and s (Theorem 7.6 and Theorem 7.10). $\square$

This proves that s is a semantic inverse to g. For heap h and heap-graph g, $h, i_0, \eta_0 \models s[\![g[\![F]\!]]\!] \iff h, i_0, \eta_0 \models F$, and $g \in \mathrm{L}(g[\![s[\![G]\!]]\!]) \iff g \in \mathrm{L}(G)$. In other words, s and g define a correspondence between the domains of separation logic and hyperedge replacement. This is the major result of our work on separation logic and hyperedge replacement.

However, s is *not* syntactically the inverse of g, even if we discount variable naming and trivial formula reordering. This is because the source normalisation operator (see §6.4) removes information when normalising to a heap-graph grammar. To see this, consider the example given in §6.4. We have a formula

$$\mathsf{let}\ f(z_1, z_2) = ((z_1 \mapsto z_2, \mathsf{nil}) \vee (z_2 \mapsto z_1, \mathsf{nil}))\ \mathsf{in}\ \exists x, y.\, f(x, y) * f(x, y).$$

Applying g to formula results in a non-heap-graph grammar, which we then normalise to the following grammar.



But this is the same grammar that results from applying g to the following formula.

$$\begin{array}{l}\mathsf{let}\ f_1(z_1,z_2) = (z_1 \mapsto z_2, \mathsf{nil}),\ f_2(z_1,z_2) = (z_2 \mapsto z_1, \mathsf{nil}) \\ \mathsf{in}\ \exists x, y.\, f_1(x,y) * f_2(x,y).\end{array}$$

Consequently g is non-bijective, so s cannot be its inverse.

However, this problem does not occur if we consider only formulas in the range of s. This is because formulas constructed by s are already source-normalised. We call this class of formulas $\mathcal{SLF}'$. Aside from source normalisation and variable renaming, corresponding formulas and grammars are structurally very similar. Consequently we conjecture that $g\colon \mathcal{SLF}' \to \mathcal{HGG}$, the function g restricted to the range of s, *is* the inverse of s.

8.3.1 Making use of theoretical results

An immediate consequence of this result is that some of the theoretical results that have been proved for hyperedge-replacement grammars hold for formulas in the fragment.

The most obvious of these are the results about inexpressible languages. 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.

Examples of known results include the facts that both the languages of grid graphs, balanced binary trees are inexpressible by any hyperedge replacement grammar (consequences of the pumping lemma and linear-growth theorem of [38] respectively) as is the language of all heap-graphs (consequence of the clique-size limit discussed in §8.1.3).

The most flexible result for proving inexpressibility results is the pumping lemma for hyperedge-replacement languages (already mentioned in §8.1.1).

The interested reader is directed to [38] for an extended discussion of the inexpressibility results that follow from the pumping lemma.

We have already made use of several expressibility results in defining our mappings. For example, the class of hyperedge replacement expressible languages is closed under intersection with so-called compatible properties. We used the fact that heap-graph conformance is a compatible property in §7.2 to show that we can define a heap-graph normalisation operator. This result holds for any formula in our fragment. So for any formula in $\mathcal{SL}$, a new formula can be constructed expressing the intersection between the formula and a compatible property.

8.3.2 Relationship with symbolic heaps

Our approach is related to other work in the composition of our fragment $\mathcal{SL}$. This fragment might appear at first to be quite restrictive. However, it is actually quite close to the symbolic heaps that form the basis of the Space Invader tool [5, 79] (see §11.3 for more on Space Invader). Loosely speaking, symbolic heaps are a fragment of separation logic that has been developed for symbolic execution.

The exact fragment of separation logic used varies with the paper cited. However, a symbolic heap $\Pi \mid \Sigma$ as defined in [20] consists of a finite set Π of equalities and a finite set Σ of heap predicates. The equalities $E = F$ are between expressions E and F, which are variables x or primed variables x' or nil. 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.

Π is referred to as the *pure part* of the heap, and Σ is referred to as the *spatial part* of the 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

$$(P_1 \wedge P_2 \wedge \ldots \wedge P_n) \wedge (Q_1 * Q_2 * \ldots * Q_m)$$

Later papers extend the symbolic heap notion to include more expressive pure formulas [15], and more sophisticated recursive predicates, such as a ‘list of lists’ predicate [5].

Our fragment includes a general notion of recursive predicate, while symbolic heaps use specific recursive 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.

While the various versions of the symbolic heap fragment are quite close to our fragment, they all include non-spatial assertions that we do not include in our fragment. All of the symbolic heap fragments include assertions that describe the arithmetic relationships between locations. In models that include pointer arithmetic these includes less-than and greater-than assertions. Under our model, without pointer arithmetic, such formulas are meaningless. However, we could meaningfully extend our fragment to include pure equality and inequality assertions.

We believe, but have not yet shown, that equality and inequality assertions can be modelled using hyperedge-replacement grammars. In [28], grammars are examined that extend the semantics of hyperedge replacement to permit repetition in a rule's sequence of external nodes. Operationally, such rules can 'fuse' the external nodes of a nonterminal hyperedge. These node-fusing rules seem like a natural approach to modelling equality and inequality statements in separation logic.

It is proved in [28] that for any grammar defined using rules with repetition, a corresponding repetition-free grammar exists that defines the same language. For this reason, we conjecture that conventional HR rules as defined in Chapter 2 should be expressive enough to model equality and inequality.

The predicate junk is equivalent to true, and so by the result given in §8.1.3 cannot be expressed by any hyperedge replacement grammar. However, as the name suggests, 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.

The result of the similarity between our fragment and the symbolic heaps used in other work on separation logic is to suggest that our fragment, while restricted, is still suitable for practical use in the description of program states. This fits with our intuition about hyperedge replacement grammars; they are general enough to describe useful classes of structures.

8.4 Other related work

Our work is relatively novel. The idea of relating separation logic to context-free graph grammars has not appeared in other work that we have discovered. The work of Lee *et al.* [52] is the closest we have found. In this, separation logic is used to give a semantics to grammars, which is then used as the abstract domain in an automatic shape analysis [1].

A semantics is given to a grammar by mapping it to a separation logic formula. However, the mapping is only one-way, meaning no correspondence result can be derived (this is not the aim of the paper). In addition, the grammars used in [52] are severely restricted compared to our heap-graph grammars. While the initial elements of the grammar can be graphs, productions over nonterminals can only construct single binary branches.

Some other work has been done on the expressibility 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 [56]. However, in [16] it is shown that the correspondence does not hold for separation logic formulas that include logical parameters standing for formulas, and 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 to these results, because the fragment we use 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 we map to a class of grammars with many well-understood properties. The inclusion of a general notion of recursion also allows us to cope with any recursively-defined predicate, rather than just lists.

[1]It is interesting that the work on symbolic heaps is descended from this work. As mentioned in the previous section, symbolic heaps exhibit many grammar-like properties.

Part IV

A language for shape safety

Chapter 9

CGRS: A language for shape safety

Pointers in imperative programming languages are indispensable for the efficient implementation of many algorithms at both applications and systems level. It is however notoriously difficult to program using pointers without introducing subtle errors. This is because the type systems of current programming languages are in general too weak to detect ill-shaped pointer structures. At best, a language may ensure that pointer structures are locally safe, ensuring pointers refer to allocated heap-structures of the correct type. This kind of correctness prevents the most obvious pointer errors, such as dereferencing errors and bounds errors. Java programs are pointer-safe in this sense; C programs are not.

However, such local typing places no restriction on the overall shape of the pointer structure. For a pointer program to behave as expected, without error, it must preserve certain properties of its pointer structures. For example, a binary search tree insertion must preserve membership of the class of binary search trees. We call these large-scale properties *shapes*, and we describe programs that preserve required shape properties as *shape safe*. There exists no mechanism in current widely-used programming languages for ensuring that programs are shape safe.

The Safe Pointers by Graph Transformation (SPGT) project [4, 3] is a recent approach to ensuring the safety of pointer manipulations. Under this approach, shapes are specified by graph reduction, and rewrites are modelled as graph transformation rules. The SPGT project has developed a checking algorithm that allows the checking of graph transformation rules to determine whether they are shape safe.

This chapter describes a new language that applies the SPGT shape specification approach to the C programming language. Our new language extends C with constructs that are the analogues of graph-transformation rules and of SPGT-style shape specifications. This allows a direct application of the SPGT shape-checking methods to the checking of the program for shape safety. We have called our extended programming language CGRS.

The chapter is structured as follows. Section 9.1 summarises how shapes are defined by graph reduction under the GRS approach and sketches the checking algorithm for shape-preservation. Section 9.2 describes the new constructs in CGRS that allow programmers to write shape specifications and operations on shapes. Section 9.3 describes an extended example of the use of a CGRS program for inserting values into an AVL tree. Finally Section 9.4 considers the size of programs written in CGRS, both before and after compilation.

9.1 Safe pointers by graph transformation

Under the Safe Pointers by Graph Transformation approach, the properties of pointer structures are specified as *shape types.* A shape type consists of a class of graph structures with common properties defined by a so-called *graph reduction specification* (GRS). The kinds of nodes that can be used in a GRS are defined by a signature that gives a set of labels for nodes and outgoing edges corresponding to these labels.

Definition 9.1 (GRS typing)**.** Each node models a tagged record of pointers where the node label, drawn from the node-label alphabet $\mathcal{L}_V$, is the tag. Each edge leaving a node corresponds to a pointer field where the edge label, drawn from the edge-label alphabet $\mathcal{L}_E$, is the name of the pointer field. We use a function type: $\mathcal{L}_V \to \wp(\mathcal{L}_E)$ to associate with each record tag its set of field names: if node v is labelled l and has an outgoing edge e, then the label of e must be in type(l) and no other edge leaving v must have this label.

We can encode the type requirement in our notion of a graph signature (see Definition 2.13). The signature $\Sigma = \langle \mathrm{L}, in, out \rangle$ corresponding to a type function type is defined so that $\mathrm{L} = \langle \mathcal{L}_V, \mathcal{L}_E \rangle$, $in(l, l') = \perp$ for all labels l, l', and $out(l, l') = 1$ if $l' \in \mathrm{type}(l)$ and $out(l, l') = 0$ otherwise.

To operate on these Σ-graphs, we use the double-pushout approach rewrite rules defined in §2.2. Σ-graphs in rules need not be Σ-total, they can contain nodes with an incomplete set of outgoing edges or unlabelled nodes with no outgoing edges.

We refer to [4] for conditions on unlabelled nodes and outgoing edges in rules that ensure that rule applications preserve both Σ-graphs and Σ-total graphs. For a rule $\langle L \leftarrow K \rightarrow R \rangle$ we require the following.

- Unlabelled nodes in L are preserved and remain unlabelled with the same outlabels.
- Labelled nodes in L which are preserved with the same label have the same outlabels in L and R.
- Relabelled nodes have a complete set of outlabels in L and R. Nodes may not be labelled in L and unlabelled in R, or vice versa.
- Deleted nodes have a complete set of outlabels.
- Allocated nodes are labelled and have a complete set of outlabels.

See [4] for the formal definitions of these conditions. Rules satisfying these conditions are called Σ-total rules.

The members of a GRS shape are specified by graph reduction. Each GRS has a single accepting graph and a set of reduction rules. Any graph that can be reduced to the accepting graph by some sequence of rule applications is a shape-type member.

Definition 9.2 (graph reduction specification)**.** A *graph reduction specification* $\mathcal{S} = \langle \Sigma, \mathcal{R}, Acc \rangle$ consists of a signature Σ, a set of Σ-total rules $\mathcal{R}$ and a Σ-total $\mathcal{R}$-irreducible *accepting graph Acc*. It defines the graph language $\mathrm{L}(\mathcal{S}) = \{G \mid G \Rightarrow^*_{\mathcal{R}} Acc\}$.

Definition 9.3 (polynomially terminating GRS)**.** A GRS $\mathcal{S}$ is *polynomially terminating* if there is a polynomial p such that for every reduction $G_0 \Rightarrow_{\mathcal{R}} \cdots \Rightarrow_{\mathcal{R}} G_n$ on Σ-total graphs, $n \leq p(|V_G| + |E_G|)$. It is *closed* if for all $G \in \mathrm{L}(\mathcal{S})$, $G \Rightarrow_{\mathcal{R}} H$ implies $H \in \mathrm{L}(\mathcal{S})$. A *polynomial* graph reduction specification, PGRS for short, is a polynomially terminating and closed GRS.

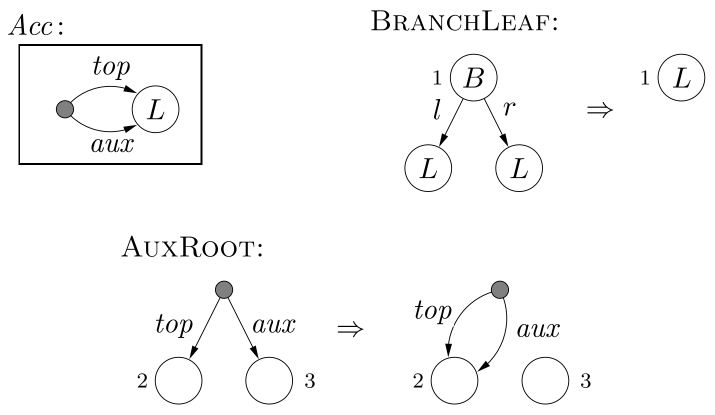


Figure 9.1: Rooted GRS defining the language of binary trees.

Example 9.1 (graph reduction specification)**.** Figure 9.1 shows a GRS for full binary trees with an extra root node labelled with R (shown in the picture as a grey-shaded node) and an auxiliary pointer from the root node to any node in the tree. Its signature is $\mathcal{L}_V = \{R, L, B\}$, $\mathcal{L}_E = \{top, aux, l, r\}$, $\text{type}(R) = \{top, aux\}$, $\text{type}(B) = \{l, r\}$ and $\text{type}(L) = \emptyset$. Tree nodes are either L-labelled leaves or B-labelled branch nodes with outgoing pointers l and r, and there is a unique R-labelled node with pointers *top* and *aux* that point to the root of the tree and to an arbitrary tree node, respectively. The accepting graph, *Acc*, is the smallest graph of this kind.

Fig. 9.1 gives two reduction rules for the GRS. The AUXROOT rule redirects the auxiliary pointer to the top of the tree (regardless of the labels of nodes 2 and 3), while the BRANCHLEAF rule deletes two leaves with the same parent and relabels that parent as a leaf. Any full binary tree with an auxiliary pointer can be reduced to *Acc* by the repeated application of these two rules, but no other graph can be reduced to *Acc*.

To see that the rules in the GRS cannot reduce ill-shaped graphs to *Acc*, consider their inverses (which are obtained by swapping left- and right-hand sides). The inverse of AUXROOT can only move the *aux* pointer to an arbitrary location, while the inverse of BRANCHLEAF can only create more leaf nodes. Consequently both inverse rules preserve full binary trees with an auxiliary pointer, which implies that the specified shape cannot contain other graphs.

The GRS is polynomially terminating – actually linearly terminating – because for every step $G \Rightarrow H$ on Σ-graphs, the number of nodes without

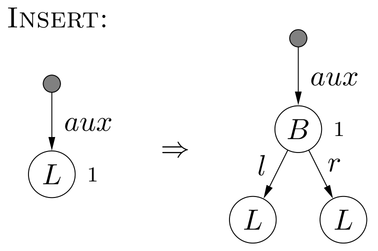


Figure 9.2: Graph transformation rule INSERT that inserts a new branch into a binary tree at the position of the *aux* pointer.

outgoing parallel edges is reduced. The GRS is also *non-overlapping*, meaning that for each pair of steps $H_1 \Leftarrow G \Rightarrow H_2$ on Σ-graphs, either $H_1 \cong H_2$ or there is a Σ-graph M such that $H_1 \Rightarrow M \Leftarrow H_2$. This property implies closedness and hence the GRS is a PGRS.

Just as pointer structures are modelled by GRSs, operations on those structures are modelled by graph transformation rules. Figure 9.2 shows an operation on the shape of Figure 9.1 that replaces a leaf destination of the auxiliary pointer with a branch node and two new leaves. This rewrite rule is an example of a *shape safe* rule: when applied to a graph that is a binary tree with an auxiliary pointer, it is guaranteed to produce a graph in the same shape.

Unrestricted GRSs are universally powerful in that they can define every recursively enumerable shape, but their membership problem is undecidable in general.

To check that pointer rewrites conform to the restriction of a GRS, they must be modelled by graph-transformation rules (which need not obey the restrictions of PGRSs).

Definition 9.4 (shape-safe rule)**.** A graph-transformation rule r is *shape-safe* with respect to a shape $\mathrm{L}(\mathcal{S})$ if for all G in $\mathrm{L}(\mathcal{S})$, $G \Rightarrow_r H$ implies $H \in \mathrm{L}(\mathcal{S})$.[1]

The static checking algorithm SHAPE-CHECK developed in the SPGT project is described in [3]. Briefly, given a graph-transformation rule r and

[1]For simplicity, we assume that rules have the same input and output shape. The shape-checking method of [3] can also handle shape-changing rules.

a GRS $\mathcal{S}$, the algorithm constructs two *abstract reduction graphs* (ARGs) that represent all contexts of r's left- and right-hand side in members of L($\mathcal{S}$). The rule is safe if the right-hand ARG includes the left-hand ARG as a subgraph. Some ARGs are infinite and hence their construction does not terminate, but in many practical cases the algorithm produces finite ARGs representing all left- and right-hand contexts so that inclusion can be checked. The general shape-safety problem is undecidable even for context-free shapes [32] and hence every checking method is necessarily incomplete.

The incompleteness which must exist in any shape-checking algorithm manifests itself in two ways in Shape-Check: first, the construction of either of the ARGs may not terminate, and second, even given termination a shape-safe rule may fail the inclusion test. In practice, this algorithm can check insertions and deletions in over context-free shapes such as cyclic list, linked list and binary search trees. However, it often nonterminates when applied to non-context-free shapes such as balanced trees. Intuitively, such shapes require an arbitrarily large amount of context, causing nontermination of the ARG construction algorithm. See [3] for more discussion.

9.2 CGRS: a language for safe pointers

CGRS is an extension to C that implements the GRS approach to shape specification and shape checking. The main ideas (adopted from [31]) are that certain pointer structures used in the program have a declared shape that specifies the possible form of the pointer structure, and that such pointer structures are only manipulated by transformers that correspond to graph transformation rules.

9.2.1 Conforming to the C model

The main aim for CGRS is to enable the shape-checking of C programs by introducing constructs corresponding to graph transformation rules. However, the graph transformation model of computation differs considerably from the C model.

Rules under the DPO approach can apply at any location in a graph where their left-hand sides match. As there may be more than one such location, they are also non-deterministic. C pointer assignments in contrast are deterministic and local, in that they can be applied exclusively at locations held by pointers.

The non-deterministic matching of graph transformation rules creates another mismatch between the two models. To find a match for its left-hand side, a rule must search the graph, an operation that requires polynomial time in the worst case for a rule of fixed size. C pointer assignments in contrast are guaranteed to terminate in constant time.

As a result, an individual graph transformation rule is considerably more powerful than an individual C pointer assignment. To apply the GRS approach to C, the graph-transformation idiom must be fitted more closely with the conventions of C.

In CGRS we use a restricted form of graph transformation more compatible with the expectations of C programmers. The pointer structures and transformer functions used in CGRS are restricted by requiring that they are *rooted*. Individual rewrite rules are less powerful than general rules, but they are still sufficiently powerful to replace C pointer manipulations.

Definition 9.5 (C-like rooted graph)**.** We describe a graph as a *C-like rooted graph* if: (1) There exists a *root label* ϱ that is the label for exactly one node in the graph. (2) Distinct edges with the same source-node have distinct edge labels.

C pointer structures naturally correspond to such rooted graphs. Such structures are accessed through a finite group of distinctly-labelled stack variables, corresponding to roots. The fields of a C `struct` are distinct, conforming to the restriction on out-edges in rooted graphs.

Definition 9.6 (C-like rooted rule)**.** A graph transformation rule is a *C-like rooted rules* if (1) it has a root label ϱ such that the left-hand side contains exactly one ϱ-labelled node, and (2) every left-hand side node is reachable from some root. Transformer functions correspond to C-like rooted rules.

Example 9.2 (rooted graphs and rules)**.** The GRS shown in Fig. 9.1 defines a class of C-like rooted graphs. The rule in Fig. 9.2 is a C-like rooted rule.

Chapter 3 discusses rooted rules in detail, as part of a more general theory of fast graph transformation. The requirement for a unique root node-type and the requirement for left-hand side reachability ensures that C-like rooted graphs and rules conform to our Condition R3. For this reason, C-like rooted rules are deterministic and rule application requires only constant time.

The failure behavior of graph transformation also differs from the standard C behavior. In the general definition of a graph-transformation rule, the result of a rule is undefined if no match can be constructed. This is also the case in programming languages based on graph transformation (e.g. the GP language [65]). This fits badly with the C approach, where a function only ever has an undefined value if an error has occurred. This behaviour is also quite impractical, in that the failure to find a match can be a useful result for a transformer function. For example, a match failure can be used to check whether the root node points to a node of a particular type.

As a result, CGRS transformer functions return a boolean value recording whether rule application succeeded. This allows us to use transformers as conditions as well as a rewriting system. Transformers are passed pointers to shape-structures and modify these structures in-place. In-place rewriting is safe because a transformer can only fail during the matching portion of the execution, before any rewriting has occurred. If the matching process for a transformer function fails, the input graph is left unmodified. Once the function terminates, a boolean value is returned that indicates whether the application succeeded.

The handling of dangling edges in graph transformation differs greatly from C pointer rewriting. In a graph transformation rule, the dangling condition ensures that dangling edges cannot be created. Every edge has as its target a valid node. In C pointer rewriting, no such restriction exists. Any value can be written into a pointer field, and consequently pointers do not necessarily point to valid values in the heap. In this case, the graph transformation behavior is safer than the standard C behavior. Avoiding dangling edges is an important objective of shape safety. In CGRS we maintain information in the data structure so that the dangling condition can be checked for pointer structures. Consequently transformer function cannot create dangling edges.

Aside: Injective and non-injective matching

Given that we are modelling pointer rewrites, the choice of a formulation for context-sensitive graph transformation that uses injective matching may seem slightly surprising. Conventional pointer rewrites are non-injective, in that rewrites can 'match' the same locations in a pointer structure with several different names in a single rewrite. For example consider the following pointer accessing rewrite.

```
v->a->a = w->a
```

The first and second instance of `a` referenced from `v` can refer to the same location, if the value in `v->a` points back to itself. In addition, `v` and `w` may point to the same location. Both possibilities correspond to non-injective matching.

However, our objective in CGRS is to improve pointer program safety, rather than replicate all the features of conventional pointer rewriting. It is known that injective matching is more general than non-injective matching, in that non-injective rules can be simulated by injective rules [40]. The ability to distinguish between injective and non-injective matches gives a substantially increased expressive power rewrite rules, and so we have chosen to use it in our graph transformation formulation and so to apply it to pointer rewriting.

In addition, using injective matching simplifies the overall presentation of the thesis. Injective matching provides the most general basis for Part II, where we generalise to rewriting arbitrary graphs, rather than just pointer structures. Using non-injective matching would unnecessarily restrict the scope of this work. The alternative of using different but similar approaches in different parts of the thesis seems confusing for the reader.

9.2.2 Signatures and shapes

Transformer functions operate on shape structures that are guaranteed to have the form specified by the shape declaration. Shape structures are composed of nodes, each of which has a type and an associated fixed set of fields. In normal C these types and fields are defined by declaring `struct` types, while in CGRS they are defined by a signature. The abstract syntax of CGRS signature declarations and shape declarations is given in Figure 9.3. Here struct-decl-cont stands for any normal C type declaration.

A signature defines a set of node-types. However, node-types are not manipulated directly in a CGRS program – instead they are referenced through shape declarations (see page 186).

Signatures consist of a `signature` block, containing node definitions. Nodes are defined with a `node` block, which strongly resembles a C `struct` declaration. The 'fields' of the declaration consist of edge names labelled sig-def ::= `signature` *sigid*{ [node-def;]$^+$ }

node-def ::= `nodetype` *ntid* { [node-cont;]$^+$ } |
`root nodetype` *ntid* { [node-cont;]$^+$ }
`nt nodetype` *ntid* { [node-cont;]$^+$ }

node-cont ::= `edge` *ed* [, *ed*]* | struct-decl-cont

shape-def ::= `shape` *shid sigid* {
`accept` { [node-dec;]$^+$ [*nid.ed* `=>` *nid*;]* }
`rules` { [*rdid*;]* } }

node-dec ::= *ntid nid*

Figure 9.3: Syntax for signatures and shapes.

with the `edge` keyword, and non-edge fields. Unlike normal C pointers, edge-fields are defined without stating the type of the objects they are pointing to – edges in CGRS structures can point to any type of node permitted by the signature.

The following code declares a node-type for use in a binary tree signature.

```
nodetype branchnode {
  edge l, r;
  int val;
}
```

Node declarations can include any type of data field (including pointer data). To ensure that pointer rewrites are safely encapsulated, transformer functions cannot do anything other than read or write from these fields. In the abstract domain of graphs, the data values stored in nodes are abstracted away, leaving only node and edge labels.

Some classes of graphs require nonterminal nodes for a GRS to successfully specify their properties (for example, the language of complete binary trees, see Theorem 5 in [4]). To model this in CGRS, node types in a signature can be labelled with the keyword `nt`. Nodes that are so declared cannot be used in a transformer.

To fit with our requirement for rootedness structures in CGRS must include distinguished root nodes (see §9.2.1). These are distinctly labelled and can appear at most once in the pointer structure. In the signature exactly one node-type must be declared with the `root` keyword.

A *shape declaration* in CGRS is the analog of a graph reduction specification. It defines a class of shape structures, and defines a type for the shape that can be passed to transformers for manipulation. On an abstract level, a shape declaration defines a class of conforming shape structures. The content of this class is defined by mapping shapes to graph reduction systems and shape structures to graphs (this mapping is defined in §10.1).

A shape declaration consists of a block prefaced with a `shape` keyword, a shape name, and a signature name. In the shape block, there are two blocks: an `accept` block and a `rules` block. The accept block uses the same syntax as a transformer definition to specify the accepting graph, while the `rules` block consists of a list of names of reduction rules.

In addition to defining a class of conforming structures, the shape declaration has two roles. First, it defines a C type for pointers to shapes, and these pointers can be passed around in the same way as pointers to normal heap data-structures.

Second, the shape declaration implicitly defines a constructor for the shape. For shape safety to hold, it must be true that all of the shape structures constructed during program execution are members of the class of structures defined by the shape. To ensure this is true for when the shape is first created, we automatically generate a shape-safe constructor function. We then assume that all shape structures are created by applying shape-preserving transformers to shape-safe structures, starting with the initial graph.

The constructor for a shape S is called `newgraph_S`, so for our binary tree example it will be called `newgraph_tree`. A constructor takes no arguments and returns a pointer of type S, the top-level shape type defined by the shape declaration. The structure resulting from the constructor corresponds to the accepting graph for the shape. (See §10.2.2 for a definition of the constructor function generated for a shape).

Example 9.3 (signature, shape declaration)**.** In the case of our running example, the tree insertion code operates over structures composed of root nodes, branch nodes with two outgoing edges, and leaf nodes with no outgoing edges. We call such a signature with binary branches and leaf nodes '`bin`'. The left side of Figure 9.4 shows the CGRS signature declaration for `bin`.

The node types `treeroot`, `branchnode` and `leafnode` correspond to the

```
signature bin {
  root nodetype treeroot {
    edge top, aux;
  }
  nodetype branchnode {
    edge l, r;
    int val;
  }
  nodetype leafnode {}
}
```

```
shape tree bin {
  accept {
    treeroot rt;
    leafnode leaf;
    rt.top => leaf;
    rt.aux => leaf;
  }
  rules {
    branchleaf;
    auxroot;
  }
}
```

Figure 9.4: Left: Signature for a structure composed of branches and leaves. Right: shape declaration corresponding to the GRS declaration in Figure 9.1.

root, B-labelled branches and L-labelled leaves of the binary tree GRS. The root node-type is called `treeroot`.

The signature of a shape is defined separately from the shape itself so that it can be reused in several shapes. For example, the `bin` signature can be used to build structures other than plain binary trees. Clearly, all graph languages that are subsets of the set of binary trees (such as the language of *balanced* binary trees) use the same signature.

We want our transformers to operate over rooted binary trees as defined by the GRS shown in Fig. 9.1. The corresponding declaration for a CGRS shape `tree` is shown on the right-hand side of Figure 9.4. The accepting graph for the `tree` shape consists of a `rt`-type node and a `leafnode`-type node. We specify the signature as `bin` by giving its name as an argument.

The `rules` block of a signature refers by name to a set of *reducers*, which correspond to the reduction rules of the GRS. Reducers are declared separately from the shape declaration using the `reducer` keyword. See the Section 9.2.3 for a description of the syntax of reducers.

9.2.3 Transformer functions

Transformer functions (or just *transformers*) are the construct in CGRS used to define pointer manipulations. They are intended to correspond syntactically and semantically to graph transformation rules. CGRS uses a textual syntax that correspond closely to the graphical syntax for DPO

```
transformer   ::=  transformer trid sigid
                            ( shid id [, tid *id]+ ) {
                     left ( [nid]+ )
                       { [node-dec;]+ [nid.ed => nid;]* }
                     right( [nid]+ )
                       { [node-dec;]* [right-graph;]* }
                   }

reducer-def   ::=  reducer rdid sigid {
                     left ( [nid]+ )
                       { [node-dec;]+ [nid.ed => nid;]* }
                     right( [nid]+ )
                       { [node-dec;]* [nid.ed => nid;]* }
                   }

node-dec      ::=  ntid nid

right-graph   ::=  nid.id => nid | nid.id = nid.id |
                   nid.id = id | id = nid.id
```

Figure 9.5: Syntax for transformers and reducers.

rules.

Transformer functions are declared in CGRS in much the same way as normal C functions. Declarations are added to the top level of the C program (or to another file, with the use of a linker). The resulting transformer function can then be called in the same manner as a conventional function. Transformer function declarations are translated into boolean-typed C function declarations with the same name, so in most ways they can be treated as if they were normal function declarations. The abstract syntax of a transformer declaration is given in Figure 9.5.

The first declared argument to a transformer function must be a shape-typed pointer that gives the transformer its shape, and so also its signature. Following this argument, there can be any number of additional arguments of pointer type. Values are passed to the transformer function by reference to ensure that they can be modified in-place.

The constituent nodes of the left- and right-hand sides of a transformer are declared in a list for each side of the transformer. Unlike graph-transformation rules, nodes in transformer functions are named, and the sharing of names defines the interface nodes between the sides of the transformer function. The nodes preserved by the transformer rewrite are those appearing in both the left- and right-hand node lists. Nodes present on one or the other side only are either allocated or deleted.

The permissible types in these node-type declarations are defined by the signature used by the transformer (see §9.2.2 above).

Nodes are assigned a node-type (or *tag* in the terminology of §9.1) using a syntax similar to C variable declarations. For example, the following code declares a branch-node for

```
branchnode n1, n2;
```

The edges between declared nodes are defined using an arrow syntax similar to the field access syntax for normal C structures. The following code matches an `l`-labelled edge from node `n1` to `n2`.

```
n1.l => n2;
```

To conform to our requirement for rootedness (see §9.2.1), there must be exactly one node on both the left- and right-hand side with a root type. Furthermore, all nodes in the left-hand side must be reachable by traversing edges from the root-typed node. This ensures that the corresponding graph-transformation rules are C-like rooted rules (see Def. 9.6).

To ensure that all nodes in the resulting shape structure have the correct complement of edges, nodes that are created or retyped must have all the edges for their node-type declared on the right-hand side. Nodes that are retyped must also have all of their associated edges matched on the left-hand side. Without this condition, a transformer could require assignment of two values to the same field. This condition on transformers corresponds to the restriction of DPO rules to Σ-total rules.

The left-hand side of the transformer is matched against the shape structure to identify the section of the transformer for rewriting. Matching operates by locating a portion of the shape structure that is isomorphic to the left-hand side, starting at the uniquely-labelled root. The match must satisfy the dangling condition, meaning that applying the rule will not create dangling edges. The transformer returns the boolean value `FALSE` if matching fails.

If matching of the left-hand side succeeds, the transformer applies the rewrites expressed by the right-hand side. A transformer function performs five kinds of rewrites on a shape structure:

1. *Node creation* occurs when a node is named in the right-hand side list but not the left-hand side.
2. *Node deletion* occurs when a node is named in the left-hand side list but not the right-hand side. The matched node is removed from the shape structure (and from memory entirely).
3. All edges in the left-hand side are *deleted*.
4. All edges in the right-hand side are *created*.
5. *Node relabelling* occurs when the same node is declared in the right-hand side with a different label than on the left-hand side.

As well as edges and nodes, the right-hand side of a transformer function can also include value assignments and retrievals. A value may be written from a variable passed by reference into the value fields of a node, or vice-versa, out from a value field into a variable. Any C type can be used as the value field for a node-type, so any kind of value can be stored by a transformer function in this way.

Values can also be rearranged inside the shape structure by assigning from one node's field to another. In transformers with several such rearranging assignments, it might appear that there is a danger of non-determinism resulting from the undefined order of assignment. For example, the meaning of the following pair of assignments seems to depend on the order in which assignment takes place:

```
n1.a = n2.a;
n2.a = n1.a;
```

Our solution is simple: in any assignment $a = b$, the value assigned to a is the value of b before execution of the transformer. In other words, values are frozen on entry to the transformer, and are only altered on exit from the transformer. In the case of the code fragment immediately above, the values in `n1.a` and `n2.a` are swapped. This has the advantage of avoiding the need for intermediate variables.

```
transformer tree_goleft bin
            ( tree *t ) {
  left (rt, n1, n2) {
    treeroot rt;
    branchnode n1, n2;
    rt.aux => n1;
    n1.l => n2;
  }
  right (rt, n1, n2) {
    rt.aux => n2;
    n1.l => n2;
  }
}
```

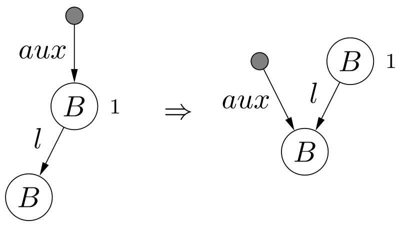


Figure 9.6: Textual syntax for transformer function for moving a root pointer down a left-hand branch, and corresponding graph-transformation rule.

Example 9.4 (transformer function)**.** Figure 9.6 shows the declaration for the transformer function `tree_goleft`, used in the search tree insertion code to move the auxiliary pointer down the tree. As with a graph transformation rule, a transformer function declaration consists of a left- and right-hand side, identified by the `left` and `right` keywords.

The right-hand side of Figure 9.6 shows the corresponding graph transformation rule to this transformer function. We define formally the mapping from transformers to graph transformation rules in §10.1.

A transformer declaration such as the one given in Example 9.4 results in a transformer function that can be used in much the same way as an ordinary C function. A transformer function must take as its first argument a pointer to a shape structure. It can then have an arbitrary number of additional arguments consisting of pointers to variables which can be read from or written to.

Example 9.5 (transformer function program)**.** We will illustrate CGRS with the simple example of insertion into a binary search tree. This code fragment inserts a value `i` into a binary search tree pointed to by the pointer variable `b`. The pointer `b` into the binary search tree has the shape `tree`, corresponding to rooted binary tree GRS given in Figure 9.1. The pointer manipulations required for performing the insertion have been encapsulated into transformer functions.

```
insert( tree *b, int i ) {
    int t;
    tree_reset(b);
    while ( tree_getval(b, &t) ) {
      if ( t == i ) return b;
      else if ( t > i ) tree_goleft(b);
      else tree_goright(b);
    }
    tree_insert(b, &i);
  }
```

The insertion into the tree works as follows: First the transformer `tree_reset` moves the auxiliary pointer to the root of the tree. Then the tree is traversed by repeatedly comparing the integer values in branch nodes with the integer `i` and following either the left or the right pointer, using the transformers `tree_goleft` and `tree_goright` (as defined in Example 9.4). Leaves don't hold values, so if the search ends at a leaf (identified because the value recovery transformer function `tree_getval` fails) then `tree_insert` inserts a branch node, and the value is written into the appropriate field.

In addition to the transformer functions in a CGRS, there can be a number of *reducers*, which correspond to the reduction rules of the GRS. Reducers are declared using the `reducer` keyword. They have a similar syntax to transformer functions, including left and right-hand side declarations. They differ in that reducers have no arguments and cannot be called as functions in the resulting program. Their semantics is purely abstract.

Example 9.6 (reducer)**.** The left side of Figure 9.7 shows `branchleaf`, a reducer for the binary tree example, and the right side shows the corresponding reduction rule BranchLeaf from the binary tree GRS in example 9.1.

9.3 Example: tree insertion and rebalancing

In this section we give a more complicated example of the use of CGRS: rebalancing operations. This example illustrates the fact that complex graph rewrites can be specified in CGRS in a way which closely conforms to their original specifications.

An *AVL tree* is a balanced search tree that allows inexpensive rebalancing after an insertion or deletion [50, §6.2.3].

```
reducer branchleaf bin {
  left (br, l1, l2) {
    branchnode br;
    leafnode l1, l2;
    br.left => l1;
    br.right => l2;
  }
  right (br) {
    leaf br;
  }
}
```

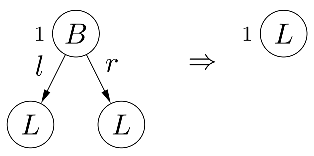


Figure 9.7: Reducer for the binary tree shape shown in Figure 9.4 and corresponding reduction rule from the GRS given in Figure 9.1.

Definition 9.7 (AVL tree)**.** A binary tree is an AVL tree if the heights of the two child subtrees of any node differ by at most one. The *balance factor* of a node is the height of its right subtree minus the height of its left subtree. A node with balance factor 1, 0, or -1 is considered balanced, while nodes with any other balance factor are considered unbalanced and require a *rebalancing* of the tree. To allow fast rebalancing, AVL tree structures record the balance factor of each node in the node itself.

Lookup, insertion and deletion all require time $O(\log\ n)$ for an AVL tree, so AVL trees are often used as a data-structure for the efficient recording of sorted data.

In [4] it is shown that the class of AVL trees can be specified using a GRS. A corresponding AVL tree shape can be constructed in CGRS by simply implementing this example as a shape. The CGRS syntax is close enough to graph transformation that this example can be translated without much difficulty.

However, the GRS is quite complex, and we require in addition to the AVL shape a stack to record back-pointers which makes the GRS larger still. Our objective in this section is illustrate CGRS as a programming language, rather than to just convert existing GRSs into CGRS syntax. To simplify the example, and focus on the rebalancing operations, we here define a weaker shape that includes AVL trees as a sub-class of its language. The shape we define is the class of *unbalanced trees with AVL-like labels.*

Tree-nodes in our shape can be labelled with $+$, $-$ or $\bullet$. Tree-nodes have two outgoing edges, labelled l and r. Our AVL trees also have an attached

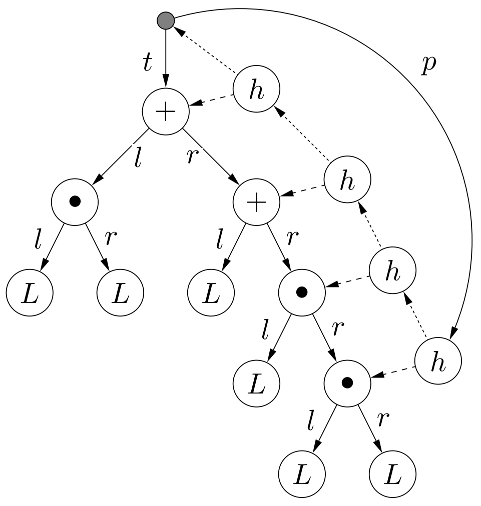


Figure 9.8: Tree with stack and AVL labels.

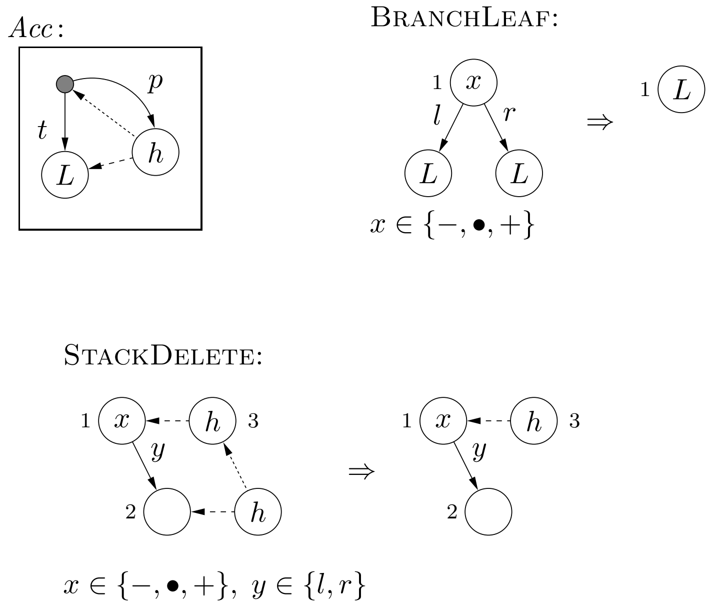
stack permitting traversal up the tree, where stack nodes are labelled h. Stack nodes have two outgoing edges. These are labelled b and c, but are shown in diagrams as finely-dotted and coarsely-dotted edges respectively. Figure 9.8 shows an (unbalanced) member of the class.

To specify a GRS for trees with AVL labels we modify the binary tree GRS given in Fig. 9.1 to specify a language of arbitrarily unbalanced AVL trees with an attached stack. Figure 9.9 shows the reduction rules and accepting graph of the GRS for the GRS. The accepting graph consists of a single leaf-node, a stack node, and a root node. The reduction rules consist of BranchLeaf, which replaces tree branches with leaves, and Stack-Delete, which removes stack nodes.

The class of validly-labelled balanced AVL trees is a subclass of the language specified by the GRS. In a valid AVL tree, nodes that are balanced are marked with the $\bullet$ symbol; nodes have a balance factor of -1 have the symbol $-$; nodes with balance factor 1 are marked with $+$. Figure 9.10 shows a balanced AVL tree.

Figure 9.11 shows the CGRS shape `avl` which corresponds to this GRS. The signature `avltree` defines the required node types, and the reducers `branchleaf` and `stackdelete` correspond directly to the reduction rules in the GRS.

In defining `avl` we make use of a slightly extended syntax for nodes and labels. The existence of several kinds of distinct branch node means that



Acc:
BranchLeaf:
StackDelete:
$x \in \{-, \bullet, +\}$
$x \in \{-, \bullet, +\},\ y \in \{l, r\}$

Figure 9.9: GRS for unbalanced trees with AVL labels.

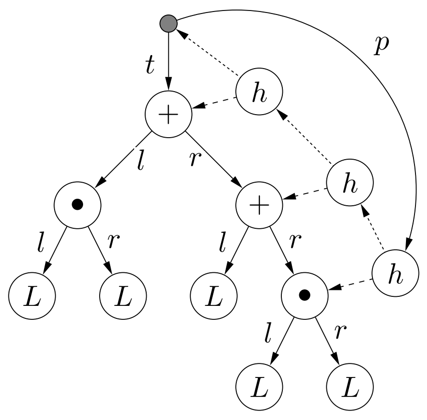


Figure 9.10: Balanced AVL tree with stack.

```
signature avltree {
  root nodetype avlroot
    { edge t, p; }
  nodetype lbr
    { edge l, r; }
  nodetype rbr
    { edge l, r; }
  nodetype bbr
    { edge l, r; }
  nodetype stack
    { edge c, b; }
  nodetype leaf {}
}

shape avl avltree {
  accept {
    avlroot rt;
    leaf lf;
    stack st;
    rt.t => lf;
    rt.p => st;
    st.c => lf;
    st.b => rt;
  }
  rules {
    branchleaf;
    stackdelete;
  }
}
```

```
reducer stackdelete avl {
  left ( br, n1, s1, s2 ) {
    [lbr,rbr,bbr] br;
    stack s1,s2;
    br.[x] => n1;
    s1.c => br;
    s2.c => n1;
    s2.b => s1;
  }
  right ( br, n1, s1 ) {
    br.[x] => n1;
    s1.c => br;
  }
}

reducer branchleaf avl {
  left (br, l1, l2) {
    [lbr,rbr,bbr] br;
    leaf l1, l2;
    br.l => l1;
    br.r => l2;
  }
  right (br) {
    leafnode br;
  }
}
```

Figure 9.11: Signature and shape declarations for the `avl` shape.

our original syntax would require separate reducer definitions covering each case. To avoid this, we define a syntax for matching sets of node labels, and for matching variable names to edge labels.

In `branchleaf` and `stackdelete` we write the following to match any node `br` with label `rbr`, `lbr` or `bbr`:

```
[rbr, lbr, bbr] br;
```

We also extend the syntax of edge matching to include variables. To associate the label `x` with any edge from node `br` to `n1` we write the following:

```
br.[x] => n1;
```

These edge variables can then be use on the right-hand side, allowing us to redirect an edge with any label permitted by the signature.

This minor extension does not require that we alter our semantics, as reducers written using the new syntax can always be expanded to finite sets of reducers written in our original syntax.

We now look at the CGRS code that rebalances an unbalanced AVL tree following an insertion. We will assume that the first stage of insertion is implemented using a modified version of the tree insertion code from §9.2. This code constructs the stack as it traverses down the tree.

The rebalancing code then steps up the stack, incrementally updating the balance factors recorded in the node, until either the height of the tree is unaltered by the update, or a single a single rotation can be applied. Once the tree has been rebalanced, the rest of the stack is removed. The following code implements this process using calls to transformer functions.

```
for ( ; ; ) {
  if ( rup(tree) || lup(tree) ) ;
  else if ( relim(tree) || rsingle(tree) ||
          rdouble(tree) || lelim(tree) ||
          lsingle(tree) || ldouble(tree) )
    break;
}
```

The code takes advantage of the lazy evaluation of C boolean operators. Because a transformer returns a boolean value recording whether the transformer succeeded or failed, at most a single transformer will succeed out of any of the transformers given in the disjunction, meaning that only a single rotation or tree-climbing operation will be applied at each iteration of the loop.

The `lup` and `rup` operations update the balance factor of the current node and pop a single element off the stack. The `rup` transformer is shown in Figure 9.12, along with the corresponding graph transformation rule.

In Figure 9.13 we show the transformer function `rdouble` that implements a right-hand double rotation. This rotation rearranges three of the tree's nodes, while preserving the descendant subtrees and parent node. To do this, the transformer function has to include the surrounding nodes in the left-hand side, with the result that this transformer is larger than the abstract specification of a rotation.

`rdouble` also makes use of our extended syntax for variable matching. By covering more cases this substantially reduces the number of reducers

```
transformer rup
           avltree (avl *x){
 left (rt,h1,h2,n1,n2) {
  avlroot rt;
  bbr n1;
  stack h1, h2;
  rt.p => h2;
  h2.b => h1;
  h2.c => n2;
  h1.c => n1;
  n1.r => n2;
 }
 right (rt,h1,n1,n2) {
  rbr n1;
  rt.p => h1;
  h1.c => n1;
  n1.r => n2;
 }
}
```

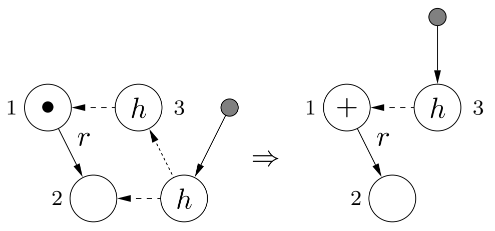


Figure 9.12: Transformer `rup` and corresponding graph transformation rule.

that must be defined. At most a single instance of the labelling can match any single target shape structure, so this extension does not introduce any unwanted nondeterminism into transformers.

As with reducers, this extension does not require a change to the semantics. For any transformer function written using this new syntax, we can construct a set of corresponding transformers written in our core syntax by instantiating the node and variable names with all possibilities satisfying the signature. This set of transformers can then be inserted into a C disjunction in the same way as in the code above.

The tree rotation shown in Figure 9.13 illustrates an advantage of CGRS as a language for specifying complex graph algorithms. Because its syntax is close to graph transformation rules, very large rewrites can be defined in a way that is close to their abstract specification. The CGRS function `rdouble` corresponds syntactically to the underlying AVL tree rotation. In contrast, the corresponding C code would be quite different from the specification of the rotation, making it more difficult to specify and debug.

The remaining right-hand rotations `relim` and `rsingle` are shown in Fig. 9.14 and Fig. 9.15 respectively. We also require the left-hand mirrors of the rotations, `lup`, `ldouble`, `lelim` and `lsingle`. These have been omitted as they are very similar to the transformers that have already been defined.

```
transformer rdouble
           avltree (avl *x){
 left (rt,h0,h1,h2,
     n1,n2,n3,n4,n5) {
  avlroot rt;
  stack h0, h1, h2;
  [rbr, lbr, bbr] n0;
  rbr n1;
  lbr n2;
  [rbr, lbr, bbr] n3;
  rt.p => h2;
  h2.c => n2;
  h2.p => h1;
  h1.c => n1;
  h1.p => h0;
  h0.c => n0;
  n0.[v] => n1;
  n1.r => n2;
  n2.l => n3;
  n3.l => n4;
  n3.r => n5;
 }
 right (rt,h0,h1,n1,
      n2,n3,n4,n5) {
  bbr n1,n2;
  rt.p => h1;
  h1.c => n3;
  h1.p => h0;
  h0.c => n0;
  n0.[v] => n3;
  n3.l => n1;
  n3.r => n2;
  n1.r => n4;
  n2.l => n5;
 }
}
```

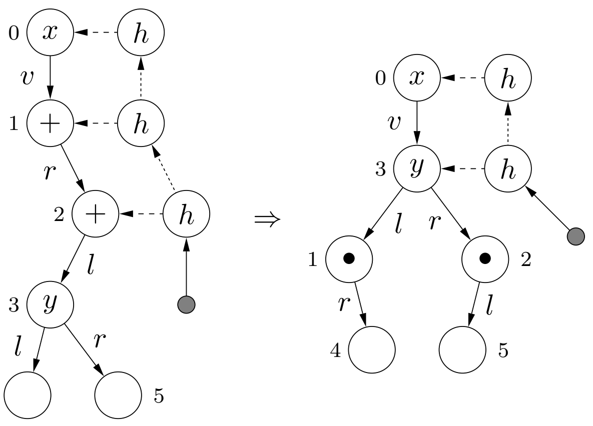


Figure 9.13: Transformer `rdouble` and corresponding graph transformation rule.

```
transformer relim
           avltree (avl *x){
 left (rt,h1,h2,n1,n2) {
  avlroot rt;
  lbr n1;
  stack h1, h2;
  rt.p => h2;
  h2.b => h1;
  h2.c => n2;
  h1.c => n1;
  n1.r => n2;
 }
 right (rt,h1,n1,n2) {
  bbr n1;
  rt.p => h1;
  h1.c => n1;
  n1.r => n2;
 }
}
```

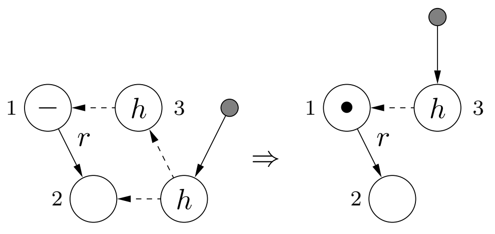


Figure 9.14: Transformer `relim` and corresponding graph transformation rule.

9.4 Code size in CGRS

The large AVL tree example given in §9.3 illustrates one of the problems with CGRS as a language – the specification of rewrites are typically quite large. To compare this to standard C, let us consider the transformer `rup` given in Fig. 9.12.

We can implement this in plain C by the following function. This checks that the balance factor of the tree-node is correct, then pops the stack. It returns `NULL` if matching fails. We assume the presence of a separately-allocated stack. With a tree including back-pointers, the call to `free` could be removed.

```
stack *rup_c (stack *h2) {
  node *n1;
  stack *h1;
  h1 = h2->back;
  n1 = h1->node;
  if (n1.balance = 0) {
    n1.balance = 1;
    free(h2);
    return h1;
```

```
transformer rsingle
           avltree (avl *x){
 left (rt,n0,n1,n2,n3,
         h0, h1, h2) {
  avlroot rt;
  [lbr,bbr,rbr] n0;
  rbr n1, n2;
  rt.p => h2;
  h2.c => n2;
  h2.p => h1;
  h1.c => n1;
  h1.p => h0;
  h0.c => n0;
  n0.[v] => n1;
  n1.r => n2;
  n2.r => n3;
}
 right (rt,n0,n1,n2,
         n3,h0, h1) {
  bbr n1,n2;
  rt.p => h1;
  h1.c => n2;
  h1.p => h0;
  h0.c => n0;
  n0.[v] => n2;
  n2.r => n1;
  n1.r => n3;
 }
}
```

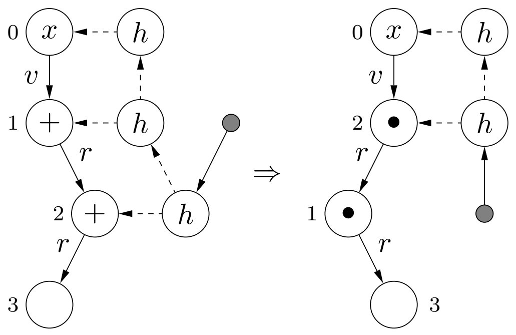


Figure 9.15: Transformer `rsingle` and corresponding graph transformation rule.

```
      } else
        return NULL;
    }
```

Clearly the code is shorter in C – 12 lines, compared to 19 for the CGRS version. The difference in size comes from two sources. First, CGRS requires that removed nodes have all edges specified in their left-hand sides, which means that the node `n2` has to be explicitly referenced. Second, CGRS reproduces several pieces of information, such as the shared graph between the left and right-hand sides.

This increase in code-length seems to be typical when comparing CGRS transformers with similar C functions. Let us consider a second example – the transformer `tree_goleft` defined in Example 9.4. The following code implements an equivalent operation in C.

```
node *tree_goleft (node *h2) {
  if (node->type = branchnode)
    return node->left;
  else
    return NULL;
}
```

This function is only 6 lines, compared to 12 for `tree_goleft`. If we assume that the input to the function is a branch-node, we can reduce this even further, down to a single-line function.

Care must be taken when comparing C with CGRS, because the matching process for a CGRS transformer is semantically more restrictive than pointer rewriting. A C pointer rewrite performs no checks about the structure being rewritten before rewriting, while a CGRS transformer checks a match for injectivity and conformance to the required pattern of labels before performing an update.

To see this, consider the rewrite `rdouble` given in Fig. 9.13. Any corresponding code written in C must check that the tree matches the whole of the left-hand side of the corresponding rule. While the transformer function is quite large, the corresponding C code must also be large.

Another source of code-size increase in CGRS is the fact that a large number of similar rules are often required. For example, to encode a full rebalancing function for AVL trees, the function would require both `rup` and `lup` functions, going up right branches and left branches. The same holds for `rdouble` and `ldouble`. It is currently difficult to express similar but distinct rewrites in a compact style in CGRS. This is caused by the limited amount of parametricity currently available in CGRS. C function in contrast can express several different cases internally in a function.

Signature declarations are however typically quite similar in size to the corresponding structure declarations in a C program. A shape declaration is obviously not present in a C program, but a shape declarations avoids the need for an initialisation function for a structure.

We hope that these problems can be can be ameliorated in future work. One possibility is that CGRS programs could be written graphically, in a notation matching the graph transformation syntax. This would remove the problem of the verbose syntax by hiding it entirely. Further development of the syntax may also give ways to parametrise more successfully, to reduce the number of explicit cases.

Chapter 10

Semantics of CGRS and shape safety

In this chapter we give a concrete and abstract semantics to CGRS programs. The chapter is structured as follows: Section 10.1 describes how we extract graph reduction specifications and graph transformation rules from the constructs of CGRS. Section 10.2 defines the operational semantics of CGRS constructs by mapping them to fragments of standard C. Section 10.3 defines an operational semantics for a small fragment of C called μC, used in our proof of correctness. In Section 10.5 we use this semantics to give a proof of the correctness of the two mappings described in Sections 10.1 and 10.2. In Section 10.6 we use the results derived in previous sections to define the shape-safety guarantees given by CGRS. While we have defined an operational semantics for CGRS, the language has not been implemented. In Section 10.7 we discuss some of the possibilities for an implementation of CGRS.

10.1 Extraction of GRSs and rules from CGRS

Our main aim in designing CGRS is to enable static checking for shape-safety, using the checking algorithm described in [3]. To use this algorithm, we must extract from the constructs of the CGRS program corresponding GRSs and graph transformation rules. In this section we define the extraction function $\mathcal{G}[\![-]\!]$ that maps from the domain of CGRS programs to the domain of graph transformation.

$\mathcal{G}[\![-]\!]$ operates over the domain of CGRS declarations – signature declarations, shape declarations and transformer function declarations. Signature declarations are mapped to graph signatures, shape declarations to graph reduction systems and transformer function declarations to graph transformation rules. CGRS constructs have been designed to be syntactically extremely close to graph transformation rules, graph signatures, and GRSs, so $\mathcal{G}$ can be defined quite informally without the risk of ambiguity.

To reason about CGRS, we abstractly represent CGRS shape declarations.

Definition 10.1 (abstract declaration)**.** In order to refer unambiguously to elements of a concrete CGRS declaration, we break them down into an *abstract declaration.* In an abstract declaration $\sigma = \langle \sigma_g, \sigma_h, \sigma_r \rangle$, σ_g is the shape declaration body, σ_h the associated signature, and σ_r is the set of reducers.

We also assume several fixed values as part of the abstract declaration. Let G be the name of the shape and S the name of the signature. Let $N_1, \dots, N_m$ be the names of the node-types defined in σ_g. For each node N_i, let E_{N_i} be the set of edges assigned for node N_i and V_{N_i} the set of value fields. Let R be the name of the unique root node type, and V_R the set of edge fields for R.

Example 10.1 (abstract declaration)**.** For the concrete shape declaration shown in Fig. 9.4, we can construct the abstract declaration $\sigma = \langle \sigma_g, \sigma_h, \sigma_r \rangle$. Here σ_g refers to the concrete definition of the shape `tree` given in Figure 9.4, σ_h refers to the signature `bin` given in the same figure, and σ_r refers to the definitions of `branchleaf` and `auxroot` given in Fig. 9.7.

Of the associated given values, G is `tree` and S is `bin`. The set $N_1, \dots, N_m$ of node names is $\{\texttt{treeroot}, \texttt{branchnode}, \texttt{leafnode}\}$. R is `treeroot`. The set of edges can be determined by examining the definitions in `bin`, so for `branchnode` $E_{\texttt{branchnode}}$ is $\{\texttt{l}, \texttt{r}\}$.

Definition 10.2 (corresponding graph signature)**.** Given an abstract declaration σ we define a corresponding graph signature $\Sigma_\sigma = \langle \mathcal{C}_\sigma, in_\sigma, out_\sigma \rangle$, with set of vertex labels $\mathcal{C}_V$ defined as the set of nodetype labels defined in σ, and set of graph edge labels $\mathcal{C}_E$ defined as the set of all CGRS `edge`-field labels used in any of the node-types. The outgoing edge function for signature Σ_σ is defined for node label N_i and edge label l so that, $out_\sigma(N_i, l) = 1$

if $l \in E_{N_i}$, and $out_\sigma(N_i, l) = 0$ otherwise. The incoming edge function is defined so that $in_\sigma(N_i, l) = \bot$ for all N_i and l.

The graph signature Σ_σ abstracts away from values held in the nodes. This is because our shape checking approach verifies only the shape properties of structures – values are ignored. For this reason, non-edge fields in a CGRS node-type declaration have no counterparts in the resulting graph signatures.

Example 10.2 (corresponding graph signature)**.** Given the abstract declaration σ defined in Example 10.1, we can define a corresponding graph signature $\Sigma_\sigma = \langle\langle \mathcal{C}_{\sigma,V}, \mathcal{C}_{\sigma,E}\rangle, in_\sigma, out_\sigma\rangle$. Here $\mathcal{C}_{\sigma,E} = \{\texttt{top}, \texttt{aux}, \texttt{l}, \texttt{r}\}$ and $\mathcal{C}_{\sigma,V} = \{\texttt{branchnode}, \texttt{leafnode}, \texttt{treeroot}\}$. The inbound edge function $in(-,-)$ is $\bot$ for all values. The outbound edge function $out(l_v, l_e)$ is 1 if

$$\begin{aligned}(l_v, l_e) \in \{\ &(\texttt{branchnode}, \texttt{l}), (\texttt{branchnode}, \texttt{r}),\\ &(\texttt{treeroot}, \texttt{top}), (\texttt{treeroot}, \texttt{aux})\}\end{aligned}$$

and 0 otherwise.

Graph transformation rules are constructed by $\mathcal{G}$ from the reducers in σ_r. The left- and right-hand side graphs of the rule are first constructed from the `left` and `right` blocks of the transformer by mapping each node-name to a node in the corresponding graph. The interface graph for the rule is then constructed from the nodes named in both of the transformer's node lists.

Example 10.3 (rule abstraction)**.** Figure 9.6 in §9.2 shows the graph transformation rule produced by applying $\mathcal{G}$ to the transformer `tree_goleft`. The 'edges' and 'nodes' of the transformer function are abstracted to corresponding graph edges and graph nodes in the rule, and the interface is constructed from the shared node names. (We use the letter B to stand for `branchnode` and L for `leafnode`).

Because $\mathcal{G}$ abstracts away value fields, the graph transformation rules it constructs only model structural rewrites and ignore value field rewrites. This means that the graph transformation rules produced by $\mathcal{G}$ abstractly model the concrete transformer functions.

```
transformer bin tree_insert
    ( tree *t, int *inval ) {
  left (rt, n1) {
    treeroot rt;
    leafnode n1;
    rt.aux => n1;
  }
  right (rt, n1, l1, l2) {
    branchnode n1;
    leafnode l1, l2;
    rt.aux => n1;
    n1.l => l1;
    n1.r => l2;
    n1.val = *inval;
  }
}
```

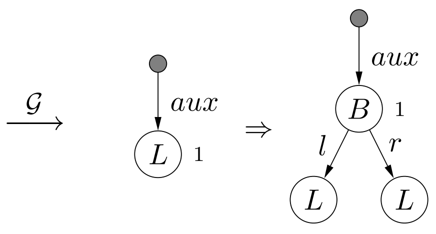


Figure 10.1: CGRS transformer function `tree_insert` and corresponding graph transformation rule produced by applying $\mathcal{G}$.

For example, Figure 10.1 shows the binary tree insertion transformer `tree_insert` and the graph transformation rule constructed from it by $\mathcal{G}$ (this rule was originally shown at the start of Chapter 9 in Figure 9.2). In this rule, $\mathcal{G}$ 'forgets' the integer value field `val` in node `n1`.

Shape declarations are mapped by $\mathcal{G}$ to graph reduction specifications. The accepting graph for the new GRS is constructed from the `accept` block in the same way as the left- and right-hand blocks of a transformer function. Reducers are syntactically almost identical to transformer functions, and they are mapped to graph transformation rules in the exactly the way we have described above.

10.2 Translating CGRS to C

In this section we define an operational semantics for CGRS constructs by translating them to C code. Only the extra constructs defined in CGRS (transformers, signatures, and shape declarations) result in modified C code, while the pure C portions of a CGRS program are left unmodified by the translation. After CGRS constructs have been translated, the resulting C code can be compiled and executed in the normal way. As a result, programmers can mostly treat CGRS constructs as if they were normal C functions and types.

The function $\mathcal{C}[\![-]\!]$ takes as its input CGRS constructs and produces blocks of C code. The translation of shapes, signatures and transformer functions performed by $\mathcal{C}$ is discussed below, but note that reducers are not executable and so are not translated to C code.

The translation function $\mathcal{C}[\![-]\!]$ implements the CGRS constructs in μC, a fragment of C. This fragment, which is defined in Section 10.3, includes restricted conditional statements and assignment to heap structures, but omits almost all control flow, including function calls. The aim of translating CGRS to a restricted language such as μC is to allow a simple proof of correctness. μC has an operational semantics defined in Section 10.3, which we use in Section 10.5 to prove the correctness of the function $\mathcal{G}$ with respect to $\mathcal{C}$.

Figure 10.2.1 shows the definition of $\mathcal{C}[\![-]\!]$ for signatures, Figure 10.2.2 shows it for shape declarations and 10.2.3 shows it for transformers. The definitions are annotated with comments identifying different parts of the definitions written in the C99 '`//`' style [45].

To allow substitution, the code includes logical variables. Given code fragments `C` and `C'`, $\mathtt{C}[x \setminus \mathtt{C}']$ stands for `C` with variable x replaced by `C'`. The notation $[\mathtt{C}]_{x \in vals}$ stands for the concatenation of $\mathtt{C}[x \setminus \mathtt{C}_1]$, $\mathtt{C}[x \setminus \mathtt{C}_2] \ldots$, where $vals = \{\mathtt{C}_1, \mathtt{C}_2, \ldots\}$. If $vals$ is an sequence of values rather than a set, then the code fragments are concatenated in sequence-order. Otherwise, the ordering is arbitrary.

10.2.1 Signature translation

Figure 10.2 shows the definition of the translation function $\mathcal{C}[\![-]\!]$ over CGRS signature declaration. We use the `bin` signature and `tree` shape given in Figure 9.1 as the running example in this section and the next. The complete code produced by $\mathcal{C}[\![-]\!]$ when applied to `bin` and `tree` is given in Figure 10.3.

From a signatures declaration in CGRS, $\mathcal{C}[\![-]\!]$ constructs a set of C `struct` types that form the basic building blocks of shape structures. These data-structures are of two kinds. First, $\mathcal{C}[\![-]\!]$ constructs a set of types corresponding directly to individual node-types. Second, $\mathcal{C}[\![-]\!]$ constructs a pair of 'wrapper' data-types that can be used polymorphically to stand for any node type. These wrapper types record extra node data and simplify node

$$\mathcal{C}\left\llbracket \begin{array}{l}\texttt{signature}\ \mathrm{S}\\ \quad \{\ \mathrm{N}_1\texttt{;}\ldots\mathrm{N}_n\texttt{;}\ \}\end{array}\right\rrbracket \quad = \quad \begin{array}{l}
\text{// Node structures}\\
[\ \mathcal{N}\llbracket\ \mathrm{N}\ \rrbracket\ ]_{\mathrm{N}\in\{\mathrm{N}_1\ldots\mathrm{N}_n\}}\\
\\
\text{// Wrapper structure}\\
\texttt{typedef struct }\mathrm{S}\texttt{\_struct}\ \{\\
\quad\texttt{int type;}\\
\quad\texttt{int indeg;}\\
\quad\mathrm{S}\texttt{\_union *node;}\\
\}\ \mathrm{S}\texttt{\_struct;}\\
\\
\text{// Wrapper union}\\
\texttt{typedef union }\mathrm{S}\texttt{\_union}\ \{\\
\qquad[\ \texttt{struct}\ n\ n;\ ]_{n\in Nodes}\\
\}\ \mathrm{S}\texttt{\_union;}\\
\\
\text{where:}\\
Nodes = \text{node-types defined in } \{\mathrm{N}_1\ldots\mathrm{N}_n\}
\end{array}$$

$$\begin{aligned}
\mathcal{N}\llbracket\ \texttt{nodetype}\ \mathrm{N}\ \{\ \mathrm{F}\ \}\ \rrbracket \quad &= \quad \texttt{struct}\ \mathrm{N}\ \{\ \mathcal{T}\llbracket\ \mathrm{F}\ \rrbracket\ \}\\
\mathcal{T}\llbracket\ \texttt{edge}\ \mathrm{E}_1\ldots\mathrm{E}_n\ \rrbracket \quad &= \quad [\ \mathrm{S}\texttt{\_struct *}e;\ ]_{e\in\{\mathrm{E}_1\ldots\mathrm{E}_n\}}
\end{aligned}$$

Figure 10.2: Translation from CGRS signatures and data types to C type declarations.

retyping.

The data-types defined from a signature declaration are held in a *container structure*. Container structures are then passed to transformer functions to manipulate the shape structure. Container structures are defined by $\mathcal{C}\llbracket - \rrbracket$ from shape declarations (see §10.2.2 below).

Individual declarations for terminal node types are translated into C `struct` declarations. Edge fields are translated into pointers of type `*S_struct`, where 'S' is the name of the signature. This `S_struct` type is the wrapper type used to stand for any node-type in signature S, so edge fields in a node type can point to any type of node in the signature. Non-edge fields in a node-type declaration are included directly as fields in the resulting `struct`.

In the signature `bin` for our running binary tree example, branch nodes are declared as the node-type `branchnode`. The declaration is translated by $\mathcal{C}\llbracket - \rrbracket$ into the `struct` declaration given below.

```
struct treeroot {
  struct bin_struct *top;
  struct bin_struct *aux;
};

struct branchnode {
  struct bin_struct *l;
  struct bin_struct *r;
  int val;
};

struct leafnode { };


typedef union bin_union {
  struct treeroot treeroot;
  struct branchnode branchnode;
  struct leafnode leafnode;
} bin_union;

typedef struct bin_struct {
  int type;
  int indeg;
  bin_union *node;
} bin_struct;


typedef struct tree {
  bin_struct *root;
} tree;
```

```
tree *newgraph_tree () {
  tree *new;
  bin_struct *rt_s;
  bin_struct *leaf_s;
  bin_union *rt_u;
  bin_union *leaf_u;

  rt_s =
     malloc(sizeof(bin_struct));
  rt_u =
     malloc(sizeof(bin_union));
  rt_s->node = rt_u;
  rt_s->type = TREEROOT;
  rt_s->indeg = 0;

  leaf_s =
    malloc(sizeof(bin_struct));
  leaf_u =
    malloc(sizeof(bin_union));
  leaf_s->node = leaf_u;
  leaf_s->type = LEAFNODE;
  leaf_s->indeg = 0;

  rt_u->treeroot.top = leaf_s;
  rt_s->indeg = rt_s->indeg + 1;
  rt_u->treeroot.aux = leaf_s;
  rt_s->indeg = rt_s->indeg + 1;

  new = malloc(sizeof( tree ));
  new->root = rt_s;
  return new;
}
```

Figure 10.3: Source code produced from the shape declarations `bin` and `tree`.

```
node branchnode {
  edge l, r;
  int val;
}
```

$\xrightarrow{\mathcal{C}}$

```
struct branchnode {
  struct bin_struct *l;
  struct bin_struct *r;
  int val;
};
```

For each signature declaration, the translation function $\mathcal{C}[\![-]\!]$ also defines a pair of wrapper data-types S_`struct` and S_`union` that together form a generic node-type for nodes in the S signature. In other words, these data-types can simulate any of the node-type `struct`s we have defined, such as `branchnode`. They also provide a common interface to auxiliary data stored in the structure.

The wrapper structures are used in pairs. An instance of the S_`struct` datatype records auxiliary data and points to an instance of the S_`union` datatype, which records the outgoing edges for the node. Such an instance in the heap is called a *node-pair*.

Auxiliary data is required in shape structures because simple pointer structures composed of `struct`-declared data-structures and pointers do not include enough information to support our general model of graph transformation. The wrapper structures record two pieces of auxiliary data for each node-pair: (1) the node-type currently simulated, and (2) the number of pointers in the rest of the shape structure that point to the current node-pair.

Access to the type of a node is necessary to allow *un*typed edges. In simple C pointer-structures, the type of a pointer specifies the type of its target in the heap. This is necessary because the type of a heap element is not recorded at run-time. Unlike C pointers, edges in GRS graphs do not record the type of the object pointed to – an edge-field can point to a node of any type defined in the signature. To implement this in C, edge-fields are all typed `*`S_`struct` for signature S, and individual node types are simulated by the wrapper structure.

The type simulated is recorded in the `type` field. For simplicity, the types of nodes are recorded as integer type-codes rather than as strings. The translation assumes a globally-consistent injective function $\mathit{typenum}(-)$ from node-type names to integers.

When deleting nodes, we must prevent dangling pointers. In the graph-transformation framework this is enforced by the *dangling condition*, which prohibits node deletion that result in dangling edges (see §2.2). To check the dangling condition given a matching morphism, it suffices to know the number of incoming and outgoing edges to the matched node. To implement this in C, it is sufficient to know the number of incoming pointers from other structures in the shape structure, as outgoing pointers are limited by the number of fields.

The use of wrapper structures enables a third feature of graph transformation that is not present in simple C pointer structures. In C it is not in general possible to alter the type of a heap object in-place. This is because different heap data-structures are of different sizes, and so retyping objects may violate memory integrity. In graph transformation rules however, we want to be able to relabel nodes in-place, which corresponds to node retyping in the C domain. This is implemented by wrapping all of the node-types declared by a signature into a single union.

Note that the auxiliary data stored in the wrapper structures must be initialised when the node is created, so for safety reasons the creation of nodes should only occur through the application of transformer functions.

The wrapper structure S_`struct` records the type-code of the node in the integer field `type` and the number of incoming edges in field `indeg`. The `node` field points to the wrapper union type S_`union`. For the binary tree signature `bin`, the following wrapper structure `bin_struct` is constructed.

```
typedef struct bin_struct {
  int type;
  int indeg;
  bin_union *node;
} bin_struct;
```

The wrapper S_`union` is constructed so both the name and type of each field corresponds to a declared node-type, as defined above on page 210. As a result, such a union can simulate any of the node types in the signature. For the binary tree example, there are three node types, `treeroot`, `branchnode` and `leafnode`, which results in the following union declaration.

```
typedef union bin_union {
  struct treeroot treeroot;
  struct branchnode branchnode;
  struct leafnode leafnode;
} bin_union;
```

The elements of this union are accessed through the standard C syntax for unions. So to access the `val` field of a `branchnode` structure through a variable `x` pointing to a `bin_union`-typed heap object, the syntax will be `x->branchnode.val`.

10.2.2 Shape translation

Figure 10.4 shows the definition of the translation function $\mathcal{C}[\![-]\!]$ over CGRS shape declarations.

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\texttt{shape}\ \mathrm{S}\ \mathrm{C}\ \{\\ \quad\texttt{accept}\\ \quad\quad\{\ \mathrm{A}_1;\ldots\mathrm{A}_n;\ \}\\ \quad\texttt{rules}\\ \quad\quad\{\ \mathrm{P}_1;\ldots\mathrm{P}_m;\ \}\\ \}\end{array}\right]\!\!\right] =$$

```
// Container type for shape structures
typedef struct S {
  C_struct *root;
} S;

// Shape structure constructor
S *newgraph_S () {
  S *new;
  [ C_struct *v_s ; ]_{v∈Nodes}
  [ C_union *v_u ; ]_{v∈Nodes}
  [ M⟦ T V ⟧ ]_{(T V)∈{A_1...A_n}}
  [ I⟦ s.e=>i ⟧ ]_{(s.e=>i)∈{A_1...A_n}}
  new = malloc(sizeof( S ));
  S->root = R_s;
  return new;
}
```

where:
R is the root-typed node name
$Nodes$ is the set of node names in $\{\mathrm{A}_1 \ldots \mathrm{A}_n\}$

$$\mathcal{M}[\![\ \mathrm{T}\ \mathrm{V}\ ]\!] =$$

```
V_s = malloc(sizeof(T_struct));
V_u = malloc(sizeof(T_union));
V_s->node = V_u;
V_s->type = typenum(T);
V_s->indegree = 0;
```

$$\mathcal{I}[\![\ \mathrm{V.E}\ \texttt{=>}\ \mathrm{T}\ ]\!] =$$

```
V_u->t.E = T_s;
T_s->indeg = T_s->indeg + 1;
```

where t is the node-type of source node V

Figure 10.4: Translation to C for CGRS shapes.

A CGRS shape declaration is mapped by $\mathcal{C}[\![-]\!]$ to a definition of a top-level container type for shape structures and a constructor function for the shape. The container type is the main handle for passing shape structures to transformer functions. The constructor function initialises a structure that corresponds to the shape's accepting graph.

The container type has the same name as the shape declaration, so in our running example of a binary tree signature, the container type `tree` is declared. The only field in the container type is the `root` field, pointing to the node-type corresponding to the graph root. Note that this node must be of the signature's root type. In the binary tree example, the following shape type is constructed.

```
typedef struct tree {
  bin_struct  *root;
} tree;
```

The advantage of defining a container for a shape-structure is that the C type-system can be used to keep track of the shape structure's type.

From a shape declaration S, the function $\mathcal{C}[\![-]\!]$ defines the constructor function `newgraph_S`. For example, for the tree shape `tree` $\mathcal{C}[\![-]\!]$ builds a constructor function called `newgraph_tree`. The constructor function for a shape constructs the shape-structure corresponding to the accepting graph. This requires the construction of three kinds of heap element: (1) the top-level shape container type, (2) instances of the appropriate nodes-types defined from the signature and (3) edges to give the correct pointers between the new nodes.

The equivalent in the heap of a graph node is a node-pair, consisting of a node structure and a node union. For each node in the accepting graph, the constructor allocates a node-pair using `malloc`, then points the `node` field of the structure at the union and initialises the fields of the structure. For example, for the leaf-node `leaf` in the binary tree example, this results in the following code (here `LEAFNODE` stands for the integer *typenum*(`leafnode`)):

```
leaf_s = malloc(sizeof(bin_struct));
leaf_u = malloc(sizeof(bin_union));
leaf_s->node = leaf_u;
leaf_s->type = LEAFNODE ;
leaf_s->indegree = 0;
```

The edges of the graph are instantiated and the `indegree` fields of the node-pairs updated. Finally the constructor allocates the shape container type using `malloc`, and the `root` field of the container is pointed at the newly-allocated root-typed node pair. The location of the new top-level structure is returned by the constructor, so that it can be recorded in a variable by the programmer.

10.2.3 Transformer translation

We now define the transformation function $\mathcal{C}[\![-]\!]$ from CGRS transformer functions to C. The formal definition of this function is given in Figure 10.5. We use the `tree_goleft` transformer function given in Figure 9.6 as the running example in this section. The complete code produced by $\mathcal{C}[\![-]\!]$ is given in Figure 10.6.

From a transformer function declaration, $\mathcal{C}[\![-]\!]$ constructs a C function with the same name. The function takes as its first argument a shape container structure holding the shape structure to be manipulated.

Transformer functions resemble graph transformation rules quite closely. As with the a graph-transformation rule, there are three phases in a transformer application:

1. Variables are constructed corresponding to a matching morphism between the transformer's left-hand side and the nodes in the shape structure.
2. The dangling condition is checked for the constructed variables.
3. The image of the left-hand side is transformed into the image of the right-hand side by creating and deleting nodes and modifying the content of preserved nodes.

The code in Figure 10.5 includes comments identifying these three phases.

The function $\mathcal{C}$ assumes an *edge enumeration* $e_1 \ldots e_n$ of the edges in the transformer's left-hand side. This enumeration must be ordered so that for all $1 \leq i \leq n$, either the source of e_i is the root node, or $\exists j.\, 1 \leq j < i \wedge s(e_i) = t(e_j)$. Any edge enumeration satisfying the property can be chosen. Due to the reachability and rootedness side-conditions on transformer functions (given in section 9.2.3) at least one such edge enumeration must exist.

(See Chapter 3 and especially §3.2 for more on graph transformation algorithms which make use of edge enumerations, and on conditions ensuring the existence of edge enumerations.)

The translation to C also assumes the existence of two typing functions t_l and t_r. These take a node name and return the node-type of a node in, the left- and right-hand sides of the transformer function respectively, as given by the node-type declarations.

$\mathcal{C}$ ⟦
```
transformer
    F C (S *G, A) {
  left(N_l)
      { L_1;... L_n; }
  right(N_r)
      { R_1;... R_m; }
}
```
⟧ =

```
bool F (S *G, A) {
  // MATCHING
  [ C_struct *v_n = NULL; ]_{v ∈ N_l ∪ N_r}
  [ C_union *v_u = NULL; ]_{v ∈ N_l ∪ N_r}
  K_s = G->root;
  K_u = K_s->node;
  [ 𝓛⟦L⟧ ]_{(s.e=>i) ∈ {L_1...L_n}}
  [ if ( x_s == y_s )
        return FALSE; ]_{(x,y) ∈ Pairs}
  // DANGLING CONDITION
  [ if ( d_s->indeg != C_L(d) )
            return FALSE; ]_{d ∈ N_l/N_r}
  // REWRITING
  [ free( *d_u ); free( *d_s ); ]_{d ∈ N_l/N_r}
  [ p->type = typecode(t_r(p)); ]_{p ∈ Retype}
  [ 𝓜⟦t_r(a), a⟧ ]_{a ∈ N_r/N_l}
  [ 𝓡⟦R⟧ ]_{R ∈ {R_1...R_m}}
  [ n->indeg
      = n->indeg + (C_R(n) − C_L(n)); ]_{n ∈ N_r}
  return TRUE;
}
```

where:
K is the root-typed node name
$C_L(i) = |\{(s,e) \mid (s.e\texttt{=>}i) \in \{\mathrm{L}_1 \ldots \mathrm{L}_n\}\}|$
$C_R(i) = |\{(s,e) \mid (s.e\texttt{=>}i) \in \{\mathrm{R}_1 \ldots \mathrm{R}_m\}\}|$
$\mathit{Retype} = \{p \in \mathrm{N}_l \cap N_r \mid t_l(p) \neq t_r(p)\}$
$\mathit{Pairs} = \{(x,y) \in N_l \times N_l \mid x \neq y\}$

$\mathcal{L}$⟦ S.E => T ⟧ =
```
T_s = S_u->t_l(S).E;
if ( T_s->type != t_l(T) )
  return FALSE;
else if (T_u == NULL)
  T_u = T_s->node ;
else if (  T_u != T_s->node
)
  return FALSE;
```

$\mathcal{R}$⟦ S.E => T ⟧ = `S_u->t_r(S).E = T_s;`

$\mathcal{R}$⟦ S.V = X ⟧ = `S_u->t_r(S).V = X;`
$\mathcal{R}$⟦ X = S.V ⟧ = `X = S_u->t_r(S).V;`

Figure 10.5: Translation from CGRS to C for transformers.

```
int tree_goleft( struct tree *t ) {
  bin_struct *rt_s = NULL;
  bin_struct *n1_s = NULL;
  bin_struct *n2_s = NULL;
  bin_union *rt_u = NULL;
  bin_union *n1_u = NULL;
  bin_union *n2_u = NULL;

  rt_s = t->root;
  rt_u = rt_s->node;

  n1_s = rt_u->treeroot.aux;
  if ( n1_s->type != BRANCHNODE )
    return FALSE;
  else if (n1_s == NULL )
    n1_u = n1_s->node;
  else if (n1_u != n1_s->node )
    return FALSE;

  n2_s = n1_u->branchnode.l;
  if ( n2_s->type != BRANCHNODE )
    return FALSE;
  else if (n2_s == NULL )
    n2_u = n2_s->node;
  else if (n2_u != n2_s->node )
    return FALSE;

  if (rt_s == n1_s) return FALSE;
  if (rt_s == n2_s) return FALSE;
  if (n1_s == n2_s) return FALSE;

  (rt_u->treeroot).aux = n2_s;
  (n1_u->branchnode).l = n2_s;

  return TRUE;
}
```

Figure 10.6: Source code produced from the transformer function `tree_goleft`.

The function constructed by $\mathcal{C}[\![-]\!]$ simulates matching of the left-hand side using a set of *matching variables*. For each named node `n` on the left-hand side, variables `n_s` and `n_u` are declared. The aim of the matching code is to assign to each pair of variables the location of a node-pair such that all of the left-hand side variables together correspond to a matching morphism for the left-hand side.

The function first assigns `NULL` to all of the matching variables. It then begins matching by assigning to the root node variable pair the location pointed to by the `root` field of the shape structure. If the structure is a valid shape structure corresponding to a rooted graph, then the root of the structure must exist. When the root node-pair has been attached to the corresponding variable, the non-root matching process is generated by the $\mathcal{L}$ constructor function

Matching proceeds by following edges outgoing from previously matched nodes, starting with the root. Edges are matched in the order that they appear in the edge enumeration. By the construction of the edge enumeration no node can occur as the source of an edge before it occurs as a target of some edge. As all nodes in the left-hand side of a transformer are reachable from the root, each node will eventually be matched by this process, unless matching fails.

In normal graph transformation, this kind of incremental graph matching may require backtracking. However, matching for a transformer function is deterministic for the reasons outlined in §9.2.1, so if matching fails at any edge, then no matching morphism exists and the whole process fails.

Matching of an edge starts by selecting the already-assigned variable for the source node's node-union. It then assigns to a the target variable the location held in the outgoing edge field for the source node-union. There are then three possible cases:

1. The target structure may be of the wrong node-type. This is checked by comparing the expected type-code (retrieved using the *typenum* function) to the `type` field of the target. If the type-code is not correct, the match fails.

2. The matching union variable may be `NULL`. This shows that this left-hand side node has not been previously matched. In this case, the matching variables are assigned the value of target.

3. The matching union variable may have been assigned previously during the matching process. This occurs when two edges in the left-hand side point to the same node. Both edges will be followed when matching, but only the first results in an assignment. The matching process checks by comparison that the current edge points to the same node-pair as the stored value held in the variable. Otherwise the match fails.

The edge `n1.l => n2` in `tree_goleft` results in the following code. This code first assigns the value held in `n1_u->branchnode.l` to the variable `n2_s`. It then checks the three possible cases using an `if` statement, and responds appropriately depending on the value held in the matching variable.

```
n2_s = n1_u->branchnode.l;
if ( n2_s->type != BRANCHNODE)
  return FALSE;
else if ( n2_u == NULL )
  n2_u = n2_s->node;
else if ( n2_u != n2_s->node )
  return FALSE;
```

Once all nodes of the left-hand side of a transformer have been matched to corresponding matching variables, the function checks by comparison that the values in the matching variables are pairwise-distinct. This corresponds to the requirement that a matching morphism must be *injective*.

The dangling condition is then checked for any nodes that are removed by the rule. As discussed Section 10.2.1, given a matching morphism h the dangling condition can be checked for a left-hand side node `n` by comparing the number of incident edges to the `n` in the left-hand side graph to the number of edges incident to the host-graph node $h_v(\mathtt{n})$. The same method is used to checking the dangling condition for a set of matching variables.

The code used to check the dangling condition for a node `n1` is as follows, if the number of incoming edges according to the left-hand side is 1. (This example is not taken from `tree_goleft` because no nodes are deleted in `tree_goleft`.)

```
if (n1_s->indeg != 1) return FALSE;
```

Once a set of matching variables satisfying the dangling condition have been constructed, the node-pairs identified by the matching variables are modified to match the right-hand side of the transformer function. Five kinds of update are performed: (1) node deletion, (2) node retyping, (3) node creation, (4) edge construction, and (5) updating auxiliary data. The order is important: nodes must be constructed before their edges can be constructed.

Node deletion uses the `free` operator in C. Nodes can be safely deleted because the dangling condition ensures that no dangling edges will be created in the shape-structure.

Node-pairs are retyped (the equivalent of relabelling nodes in a graph transformation rule) by updating the `type` field of their node structure. The side-conditions on transformer functions described in Section 9.2.3 ensure that when a node is retyped, all the edge-fields required by the signature will also be added.

New nodes are constructed using the C allocation function `malloc`, and their auxiliary data is then updated. The code for node allocation and initialisation used here is the same as that used in the definition of a constructor function in §10.2.2. Both use the code-generation function $\mathcal{M}[\![-]\!]$.

Finally, all edges in the right-hand side are created (or possibly recreated if they already existed on the left-hand side) by assigning new values to the pointer fields in the right-hand side node-pairs. The `indeg` fields of all right-hand side nodes are updated based on edges added and deleted by the transformer. In the `tree_goleft` example, the `n2` node has its incoming degree increased by one, to reflect the new incoming root pointer.

```
n2_s.indeg = n2_s.indeg + 1;
```

The return value of the transformer depends on the success or otherwise of the matching process. If matching fails or the dangling condition is not satisfied, the transformer returns `FALSE` and the shape structure passed to the function is left unmodified. Otherwise the shape structure is updated and the function returns `TRUE` to indicate that it has succeeded.

10.3 Syntax and semantics of μC

We want to show that the extraction function $\mathcal{G}$ produces rules that correctly model the operational semantics of transformer functions produced by $\mathcal{C}$. To prove this correctness property, we need a formal operational semantics for

prog	::=	statement*
statement	::=	`if ( `condition` ) `operation` else `operation |
		operation
operation	::=	location `=` term`;` | *var*`.`*field* `=` val`;` |
		`return` val `;` | `free( * `*var*` );`
condition	::=	location `==` val | location `!=` *var* |
		val `!=` int
term	::=	location | `malloc(sizeof( `*cons*` ))`
location	::=	*var* | *var*`->`*field* | *var*`->`*field*`.`*field*
val	::=	*var*`.`*field* `+` *int* | *int*

Figure 10.7: Abstract syntax of μC.

transformer functions. The operational semantics of a CGRS transformer function is defined by mapping it to a corresponding C function. Unfortunately, C's standard operational semantics is given informally. There is no widely-accepted formally-defined operational semantics for C.[1]

Our solution is to define a micro-semantics for C, based on [48]. For reasons of simplicity this semantics covers only a very small fragment of C that we call *μC*. We define an operational semantics for enough of C to implement transformer functions, while avoiding the complexity of a complete C semantics. Any full semantics of C should conform to our semantics over the domain of μC. Consequently any correctness results derived from the semantics for μC will carry over to the full semantics for C.

The concrete syntax of μC is given in Figure 10.7. The sets of μC identifiers *var*, *const* and *field* are defined as members of the language of strings Σ^+.

The control-flow operators available in C are mostly omitted from μC. The fragment includes an `if-else` construct, but omits loops, case statements and procedure calls. Programs in μC are assumed to model blocks of C that are inside function definitions (or inside the standard `main()` function). For this reason, μC programs can return values. The fragment includes a `return` keyword that terminates execution and gives the whole program a return value.

The only kind of primitive value that μC permits is the integer type. As

[1]Full formal semantics have been proposed, such as [37] and [61]. These semantics for C are quite complex, and so too difficult to use in our proof of correctness.

with all the restrictions on μC, this is intended to reduce the complexity of the semantics. The proof results given in Section 10.5 should still hold even if the semantics was extended to permit more primitive types.

μC's memory model consists of a set of variables, and a heap. The heap consists of locations of two kinds: structs and unions. Structs are defined by a type-name and a set of field names. Unions associate fields with structure types. Unlike general C, unions in μC are assumed to only have struct-typed fields. Heap locations are integers, so integer values can be used as references to heap locations.

μC restricts C's syntax for accessing heap data so that it is possible to distinguish syntactically between dereferencing a field of a struct not held in a union and dereferencing one held in a union. The first case is written *var*`->`*field*, while the second is written *var*`->`*field*`.`*field*. In full C, these cases are distinguished semantically, at run time. That is, the same syntax can have a different effect depending on the type of the pointer. However, the heap-access cases permitted in μC should have the same effect in correctly-typed C.

μC omits variable and type declarations. The semantics assumes a fixed set of defined types and defined variables. This reduces the complexity of the semantics. C's variable and type declaration mechanism is quite simple however, so μC could easily be extended to cover such declarations.

We assume that we have a pre-processor that can replace predefined strings with other strings. Such macro values are written in all capitals, following the normal C convention. We assume that we have defined the standard values `TRUE` as 1 and `FALSE` as 0.

The μC semantics is defined in the structural operational style [60]. This style simplifies some of the proofs in Section 10.5 by exposing the operational details of execution.

The semantics of μC constructs manipulate *states*, which are defined over a *type schema*. A type schema defines the field-names for μC structs, and associates union fields with struct types. Type schemas are defined using the following sets of primitive elements: A set *Var* of variable names. A set *Uni* of union type names. A set *Str* of struct names. A set $Loc \subseteq Int$ of locations. A set of field names *Fld* used to index the fields of unions and structures. The set of types *Type* is defined as $Str \cup Uni$.

Definition 10.3 (μC type schema)**.** A *type schema* $\tau = \langle \tau_u, \tau_s \rangle$ is composed of two functions. The partial function $\tau_u : \mathit{Uni} \times \mathit{Fld} \rightharpoonup \mathit{Str}$ associates with a pair (u, f) of union and field name a corresponding struct name. The total function $\tau_s : \mathit{Str} \to \mathcal{P}(\mathit{Fld})$ associates with a struct name a set of field names.

Given a type schema τ we can define a μC state. Each state records the contents of both program variables and the heap, and also keeps track of running values used during execution of a program.

Definition 10.4 (μC state)**.** A μC *state* $s = \langle ret_s, res_s, var_s, typ_s, fld_s\rangle$ consists of the following: $ret_s : \mathit{Bool}$, a value recording whether the program should return; $res_s : \mathit{Int}$, a value used to record the result of evaluating a construct; $var_s : \mathit{Var} \rightharpoonup \mathit{Int}$, a partial function mapping variable names to values; $typ_s : \mathit{Loc} \rightharpoonup \mathit{Type}$, a partial function recording the type of memory locations, assuming they are defined; $fld_s : \mathit{Str} \times \mathit{Fld} \times \mathit{Loc} \rightharpoonup \mathit{Int}$, a partial function recording the fields for the structure held at a particular location.

Given a state s, we write $s[r\colon n]$ to replace element r of the state with n. If r is a function, we write $s[r\colon v \mapsto n]$ to make $r(v) = n$ in the state.

A type schema restricts the permitted states. States that conform to a type schema are said to be *well-typed*.

Definition 10.5 (well-typed state)**.** Let $\tau = \langle \tau_u, \tau_s\rangle$ be a μC type schema. A memory state s is *well-typed* with respect to τ if for all locations $l \in \mathit{Loc}$, either

1. $typ_s(l) \in \mathit{Str}$ and $fld_s(t, f, l)$ is defined for all $f \in \tau_s(typ_s(l))$ and undefined otherwise, or
2. $typ_s(l) \in \mathit{Uni}$ and for some $n \in \mathit{Fld}$, $fld_s(t, f, l)$ is defined for $t = \tau_u(typ_s(l), n)$ and all $f \in \tau_s(\tau_u(typ_s(l), n))$, and is undefined otherwise, or
3. $typ_s(l) = \bot$ and $fld_s(t, f, l) = \bot$ for all t, f.

Example 10.4 (μC type schema, state, well-typed state)**.** Let S_1 and S_2 be struct names, F_1, F_2, F_3 and F_4 field names, and U_1 a union name. Now we define a type schema τ as follows.

$$\tau = \langle \{\langle U_1, F_1\rangle \mapsto S_1, \langle U_1, F_2\rangle \mapsto S_2\}, \{S_1 \mapsto \{F_4\}, S_2 \mapsto \{F_3\}\}\rangle$$

The following μC state s is well-typed with respect to τ. Note that fld_s assigns a structure value to l_3, even though it is in fact a union. S_3 is the type that l_3 is currently simulating.

$$\begin{aligned} ret_s = {} & \mathbf{ff} \qquad res_s = 0 \qquad var_s = \{x \mapsto l_1, y \mapsto 1, z \mapsto l_3\} \\ typ_s = {} & \{l_1 \mapsto S_1, l_2 \mapsto S_2, l_3 \mapsto U_1\} \\ fld_s = {} & \{\langle S_1, l_1, F_4\rangle \mapsto l_3, \langle S_2, l_2, F_3\rangle \mapsto 0, \langle S_2, l_3, F_3\rangle \mapsto l_1\} \end{aligned}$$

If we replace fld_s with the following alternatives, the state is no longer well-typed.

$$\begin{aligned} fld_s = \{ & \langle S_1, l_1, F_4\rangle \mapsto l_3, \langle S_1, l_1, F_2\rangle \mapsto 1, \\ & \langle S_2, l_2, F_3\rangle \mapsto 0, \langle S_2, l_3, F_3\rangle \mapsto l_1\} \end{aligned}$$

This definition violates τ by assigning a value to field F_4 of location l_1, which is typed S_1. This field does not appear in the type schema for the struct S_1. The following state is also not well-typed.

$$\begin{aligned} fld_s = \{ & \langle S_1, l_1, F_4\rangle \mapsto l_3, \langle S_2, l_2, F_3\rangle \mapsto 0, \\ & \langle S_2, l_3, F_3\rangle \mapsto l_1, \langle S_1, l_3, F_4\rangle \mapsto l_2\} \end{aligned}$$

This definition violates τ by giving location l_3 an inconsistent value, by giving it a value as a field of S_2 and of S_1. Unions must only have a single assigned type, even though they can simulate different types at different times.

The rules of the μC semantics are defined in Figure 10.8, Figure 10.9 and Figure 10.10. The semantics operates on pairs of values $\langle c, s\rangle$, where c is a program and s a state. Except where explicitly stated, reading an undefined value from any element in the state results in an undefined value for the rule application.

Figure 10.8 shows the rules of the semantics that deal with control flow constructs. The rules handling composition and if-else statements are standard for an SOS semantics, and are based on the rules of the while-language in [60]. We assume a simple boolean function $\mathcal{B}$ that can test equality and inequality for integers and locations.

Most of the control flow rules are standard, apart from the **ret** rule. Programs in μC can use `return` to break program execution at any time. Rather than explicitly jump to the end of computation, `return` writes **`tt`** into the boolean state-value *ret*. If $ret_s = $ **`tt`**, then the rule **comp** immediately applies, which ends the program.

$$[\mathbf{comp_1}] \qquad \frac{\langle c, s\rangle \Rightarrow \langle c', s'\rangle}{\langle c\ cs, s\rangle \Rightarrow \langle c'\ cs, s'\rangle} \quad \text{if } ret_s = \mathbf{ff}$$

$$[\mathbf{comp_2}] \qquad \frac{\langle c, s\rangle \Rightarrow s'}{\langle c\ cs, s\rangle \Rightarrow \langle cs, s'\rangle} \quad \text{if } ret_s = \mathbf{ff}$$

$$[\mathbf{comp_r}] \qquad \langle c\ cs, s\rangle \Rightarrow s \quad \text{if } ret_s = \mathbf{tt}$$

$$[\mathbf{if}_T] \qquad \langle \mathtt{if}(\,t\,)\, c_1\, \mathtt{else}\; c_2;, s\rangle \Rightarrow \langle c_1, s\rangle \quad \text{if } \mathcal{B}(t, s) = \mathbf{tt}$$

$$[\mathbf{if}_F] \qquad \langle \mathtt{if}(\,t\,)\, c_1\, \mathtt{else}\; c_2;, s\rangle \Rightarrow \langle c_2, s\rangle \quad \text{if } \mathcal{B}(t, s) = \mathbf{ff}$$

$$[\mathbf{ret}] \qquad \frac{\langle t, s\rangle \Rightarrow s'}{\langle \mathtt{return}\; t, s\rangle \Rightarrow s'\,[\,ret_{s'} \colon \mathbf{tt}\,]}$$

Figure 10.8: SOS semantics of μC control flow.

Example 10.5 (μC control flow)**.** Suppose we have a μC state s, typed according to the type schema given in Example 10.4. The state is defined as follows.

$$\begin{aligned} ret_s = {} & \mathbf{ff} \qquad res_s = 0 \qquad var_s = \{x \mapsto l_1, y \mapsto 1, z \mapsto l_3\} \\ typ_s = {} & \{l_1 \mapsto S_1, l_2 \mapsto S_2, l_3 \mapsto U_1\} \\ fld_s = {} & \{\langle S_1, l_1, F_4\rangle \mapsto l_3, \langle S_2, l_2, F_3\rangle \mapsto 0, \langle S_2, l_3, F_3\rangle \mapsto l_1\} \end{aligned}$$

Now we apply the semantics to the following program.

```
if ( x = 1 ) return 1; else return 2;
return 3;
```

The resulting derivation is as follows.

$$\begin{aligned} & \langle \mathtt{if(x\ =\ 1)\ return\ 1;\ else\ return\ 2;\ return\ 3;}, s\rangle \\ [\mathbf{comp_1}], [\mathbf{if}_T] \Rightarrow \quad & \langle \mathtt{return\ 1;\ return\ 3;}, s\rangle \\ [\mathbf{ret}] \Rightarrow \quad & \langle \mathtt{return\ 3;}, s[ret \colon \mathbf{tt}][res \colon 1]\rangle \\ [\mathbf{comp}_r] \Rightarrow \quad & s[ret \colon \mathbf{tt}][res \colon 1] \end{aligned}$$

The `if`-statement is replaced by its first argument, as $x = 1$ holds. Then the statement `return 1` is evaluated using the [**ret**] rule. This immediately ends the program with the return value 1, skipping the remaining `return` statements.

$$[\mathbf{ref}_v] \quad \langle v, s\rangle \Rightarrow s\,[\,res : var_s(v)\,]$$

$$[\mathbf{ref}_s] \quad \langle v\texttt{->}f, s\rangle \Rightarrow s\,[\,res : fld_s(typ_s(var_s(v)), f, var_s(v))\,]$$

$$[\mathbf{ref}_u] \quad \langle v\texttt{->}u\,.\,f, s\rangle \Rightarrow s\,[\,res_s\!: fld_s(\tau_u(typ_s(var_s(v)), u), f, var_s(v))\,]$$

$$[\mathbf{assn}_v] \quad \frac{\langle t, s\rangle \Rightarrow s'}{\langle v\texttt{=}t;, s\rangle \Rightarrow s'\,[\,var_{s'}\!: v \mapsto res_{s'}\,]}$$

$$[\mathbf{assn}_s] \quad \frac{\langle t, s\rangle \Rightarrow s'}{\langle v\texttt{->}f\texttt{=}t;, s\rangle \Rightarrow s'\,[\,fld_{s'}\!: (typ_{s'}(v), f, var_{s'}(v)) \mapsto res_{s'}\,]}$$

$$[\mathbf{assn}_u] \quad \frac{\langle t, s\rangle \Rightarrow s'}{\begin{array}{l}\langle v\texttt{->}u.f\texttt{=}t;, s\rangle \Rightarrow s'\ [\,fld_{s'}\!: (k, g, l) \mapsto \bot\,]_{g\in Fld,\, k\in Str/\{h\}} \\ \qquad\qquad\qquad\qquad\quad [\,fld_{s'}\!: (h, f, l) \mapsto res_{s'}\,]\end{array}}$$
$$\text{where } \begin{array}{l} h = \tau_u(typ_{s'}(var_{s'}(v)), u) \\ l = var_{s'}(v)\end{array}$$

Figure 10.9: SOS semantics of μC assignment and dereferencing.

Figure 10.9 shows the rules of the semantics dealing with assignment and dereferencing. Rather than splitting cases semantically, μC distinguishes syntactically between different kinds of dereferencing of heap locations. The three different syntactic cases are (1) accessing a variable value, (2) accessing a struct field referred to by a variable, and (3) accessing a field for a structure inside a union, referred to by a variable. The semantics provides a [**ref**] rule for each of these cases.

Variable access is handled by simple dereferencing of the *var* state-function. The other two cases are slightly more complicated. Access to a struct field uses the *typ* functions to retrieve the type of the target memory location and *fld* to retrieve the contents of the field. Access to a struct in a union must refer to the type-schema function τ_u to associate a union field with a particular structure type. Dereferencing of the structure then works in much the same way as the struct case.

The semantics also defines an assignment rule for each of these three cases. Three distinct assignment rules are required, rather than a single rule exploiting the [**ref**] rules, because of assignment to unions. Unions in μC consist of fields associated with a particular structure type. The programmer can write to any of the fields of any of the structures inside a union, but writing to the field of one structure deletes the fields of any other structures inside the union. To implement this, the rule [$\mathbf{assn}_u$] for assignment to a structure inside a union makes other structures undefined (written $\perp$).

Example 10.6 (μC assignment)**.** Let s be the same state considered in Example 10.5. Evaluating a variable assignment gives a simple derivation.

$$\langle \texttt{x = y}, s\rangle \quad [\mathbf{ref}_v], [\mathbf{assn}_v] \Rightarrow \quad s[var_s\colon x \mapsto 1]$$

Assigning to the field of a structure gives a slightly more complex result. Here the variable x holds a reference to location l_1, which is of structure type S_1.

$$\langle \texttt{x->}F_4\texttt{ = 0}, s\rangle \quad [\mathbf{assn}_v] \Rightarrow \quad s[fld_s\colon (S_1, l_1, F_4) \mapsto 0]$$

Here the variable z holds a reference to location l_3, which is of union type U_1.

$$\begin{array}{l}\langle \texttt{z->}F_2\texttt{.}F_3\texttt{ = 1}, s\rangle \\ [\mathbf{assn}_v] \Rightarrow \quad s[fld_s\colon (S_1, l_3, F_4) \mapsto \perp, (S_2, l_3, F_3) \mapsto 1]\end{array}$$

In this derivation the union type is currently simulating a structure of type S_1, but we assign to it instead through field F_2, which is of type S_2. This overwrites $fld_s(S_1, l_3, F_4)$ with $\perp$, as we have assigned to the union through a different field, forcing it to alter the structure it is presently simulating. We write (S_2, l_3, F_3) to 1, and so simulate an S_2 structure.

The rules of the semantics dealing with the allocation and deallocation of heap resources are shown in Figure 10.10. These rules are quite simple because μC's syntax allows only very restricted forms of memory handling. Allocation of heap objects in μC can take place only through the application of `malloc` and `sizeof` to an identifier in *Str* or *Uni*. This means, for example, that the allocation of memory objects of arbitrary size cannot occur. The rule [**alloc**] for allocation fetches an unused memory location n, and writes the type of the constructor into the *typ* function. It then returns the freshly-allocated memory location n.

$$\textbf{[alloc]} \qquad \langle \texttt{malloc(sizeof(}c\texttt{))}, s\rangle \Rightarrow s\,[\,typ_s\colon n \mapsto c\,]\,[\,res_s\colon n\,] \quad \text{if } typ_s(n) \mapsto \bot$$

$$\textbf{[free]} \qquad \langle \texttt{free(*}v\texttt{)}, s\rangle \Rightarrow s\,[\,typ_s\colon var_s(v) \mapsto \bot\,]\,[\,fld_s\colon (var_s(v), t, f) \mapsto \bot\,]_{t\in Type, f\in Fld}$$

Figure 10.10: SOS semantics of μC memory handling.

Similarly, deallocation is severely restricted by the syntax of μC. Deallocation can only take place by applying the `free` keyword to dereferenced variables. The deallocation rule **[free]** simply writes $\bot$ into the state functions *typ* and *fld*.

Example 10.7 (μC memory management)**.** Let s be the same state considered in Example 10.5. Evaluating `malloc(sizeof(`S_2`))` assigns the type S_2 to an arbitrary untyped location and writes it into the *res* value.

$$\begin{aligned}&\langle \texttt{y = malloc(sizeof(}S_2\texttt{))}, s\rangle\\ &\quad \textbf{[alloc]}, \textbf{[assn}_v\textbf{]} \Rightarrow \quad s[var_s\colon y \mapsto l_4][typ_s\colon S_2][res_s\colon l_4]\end{aligned}$$

Evaluating `free x` overwrites the memory location l_1 with $\bot$, and overwrites the associated entry in *fld* with $\bot$.

$$\langle \texttt{y = free(*x)}, s\rangle \quad \textbf{[free]} \Rightarrow \quad s[typ_s\colon x \mapsto \bot][fld_s\colon (S_1, l_1, F_4) \mapsto \bot]$$

10.4 Translating from memory states to graphs

We have defined the semantics of transformer functions in both the operational C domain and the abstract graph transformation domain. However, the two semantics are defined over different domains. The μC implementation operates over μC states, while graph transformation rules operate over graphs. In this section we define a translation from states to graphs. We first define formally what is meant by a 'shape structure' in a state, and then define a function β that translates from a shape structure to a corresponding graph[2]

[2]The function β is so-named due to the similarity between it and the function α from separation logic heaps to graphs defined in Chapter 6, Def. 6.10. Both function map from classes of pointer-structures to classes of graphs, although their details are quite different.

We begin by defining precisely the states and graphs that we are interested in. Shapes in CGRS do not correspond to unrestricted pointer structures; Rather they define classes of *shape structures*. These are portions of the heap, reachable from a single container structure, that have a particular form corresponding to a rooted graph.

Assumption 10.1. In the following section, let σ be an abstract declaration (Def. 10.1). Let S be the name of the declared shape, and let G be the name of the signature. Let $N_1, \ldots, N_m$ be the names of the node-types defined in σ_g. For each node N_i, let E_{N_i} be the set of edges assigned for node N_i and V_{N_i} the set of value fields. Let R be the name of the root node type, and V_R the set of edge fields for R.

Definition 10.6 (σ-schema)**.** A type schema τ is a *σ-schema* if: (1) $\mathrm{dom}(\tau_s) = \{N_1, ., N_m\} \cup \{G,\ S\_\texttt{struct}\}$. (2) For each struct $N_i \in \{N_1, \ldots \mathrm{N}_m\}$, $\tau_s(N_i) = L_{N_i} \cup V_{N_i}$. (3) For each struct $N_i \in \{N_1, \ldots \mathrm{N}_m\}$, $\tau_u(S\_\texttt{union}, N_i) = N_i$. (4) $\tau_s(G) = \{\ \texttt{root}\}$. (5) $\tau_s(S\_\texttt{struct}) = \{\texttt{indeg}, \texttt{type}\}, \texttt{node}\}$.

Example 10.8 (σ-schema)**.** Let σ be the abstract declaration we defined in Example 10.1. This gives struct names `tree`, `bin_struct`, `treeroot`, `branchnode` and `leafnode`, the single union name `bin_union`, and field names `root`, `top`, `aux`, `l`, `r`, `treeroot`, `branchnode`, `leafnode`, `indeg`, `type`, `node` and `val`.

We define the corresponding σ-schema $\tau_\sigma = \langle \tau_{(\sigma,u)}, \tau_{(\sigma,s)} \rangle$ as follows.

$$\begin{aligned}
\tau_{(\sigma,u)} = \{ & \langle \texttt{bin\_union}, \texttt{branchnode} \rangle \mapsto \texttt{branchnode}, \\
& \langle \texttt{bin\_union}, \texttt{leafnode} \rangle \mapsto \texttt{leafnode}, \\
& \langle \texttt{bin\_union}, \texttt{treeroot} \rangle \mapsto \texttt{treeroot}, \} \\
\tau_{(\sigma,s)} = \{ & \texttt{tree} \mapsto \{\texttt{root}\}, \\
& \texttt{bin\_struct} \mapsto \{\texttt{indeg}, \texttt{type}, \texttt{node}\}, \\
& \texttt{treeroot} \mapsto \{\texttt{top}, \texttt{aux}\}, \\
& \texttt{branchnode} \mapsto \{\texttt{l}, \texttt{r}, \texttt{val}\} \\
& \texttt{leafnode} \mapsto \{\ \}\ \}
\end{aligned}$$

Definition 10.7 (σ node-pair)**.** Let s_σ be a well-typed state with respect to a σ-schema τ_σ. We call any pair of heap-locations consisting of an $S\_\texttt{struct}$-typed struct and an $S\_\texttt{union}$-typed union a σ *node-pair* if (1) the `node` field of the structure points to the union, and (2) the type of the struct recorded in the union corresponds to the type-code recorded in `type`-field of the S`_struct`-typed struct.

A node-pair is a *total node-pair* if all of the edge-fields of the struct inside the union are defined and point to node-pairs. The set of *child node-pairs* from a particular node-pair is the set of node-pairs pointed to by the edge-fields of the node-pair. The *reachable structure* from a node-pair is the transitive closure of the child relation on the node-pair.

Example 10.9 (σ node-pair)**.** The following definitions of typ and fld for some state s conform to the σ-schema τ_σ defined in Example 10.8.

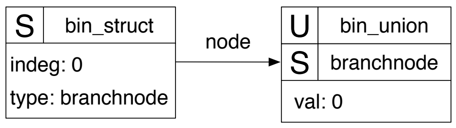
$$typ = \{\ l_1 \mapsto \texttt{bin\_struct}, l_2 \mapsto \texttt{bin\_union}\}$$

$$\begin{aligned} fld = \{\ & \langle \texttt{bin\_struct}, l_1, \texttt{indeg}\rangle \mapsto 0, \langle \texttt{bin\_struct}, l_1, \texttt{node}\rangle \mapsto l_2, \\ & \langle \texttt{bin\_struct}, l_1, \texttt{type}\rangle \mapsto \mathit{typecode}(\texttt{branchnode}), \\ & \langle \texttt{branchnode}, l_2, \texttt{l}\rangle \mapsto l_4, \\ & \langle \texttt{branchnode}, l_2, \texttt{r}\rangle \mapsto l_5, \langle \texttt{branchnode}, l_2, \texttt{val}\rangle \mapsto 0\ \} \end{aligned}$$

The locations l_1 and l_2 in the state contain a node-pair encoding a `branchnode`. Other locations are undefined.

Definition 10.8 (σ-structure)**.** A heap location in s_σ is a *σ-structure* if (1) the location holds a struct of type G, (2) the reachable structure for this struct's `root` pointer consists entirely of total σ node-pairs, and (3) the `indeg` field for any node-pair in the reachable structure is equal to the number of edge-fields in reachable node-pairs that point to this node-pair.

Example 10.10 (σ-structure)**.** We represent unions and structs graphically as follows. This example represents a `branchnode` node-pair.



Here structs are represented by S-labelled boxes, and unions with U-labelled boxes with S-annotations showing which struct the union is simulating. Non-edge fields are shown explicitly. Edge fields are shown as edges between nodes, as with `node` shown above.

Using this notation, Figure 10.11 shows a larger example of a structure at location `k`. This structure is defined using the σ-schema defined in Example 10.8. In this structure, the `indeg` fields of the nodes are correct and all

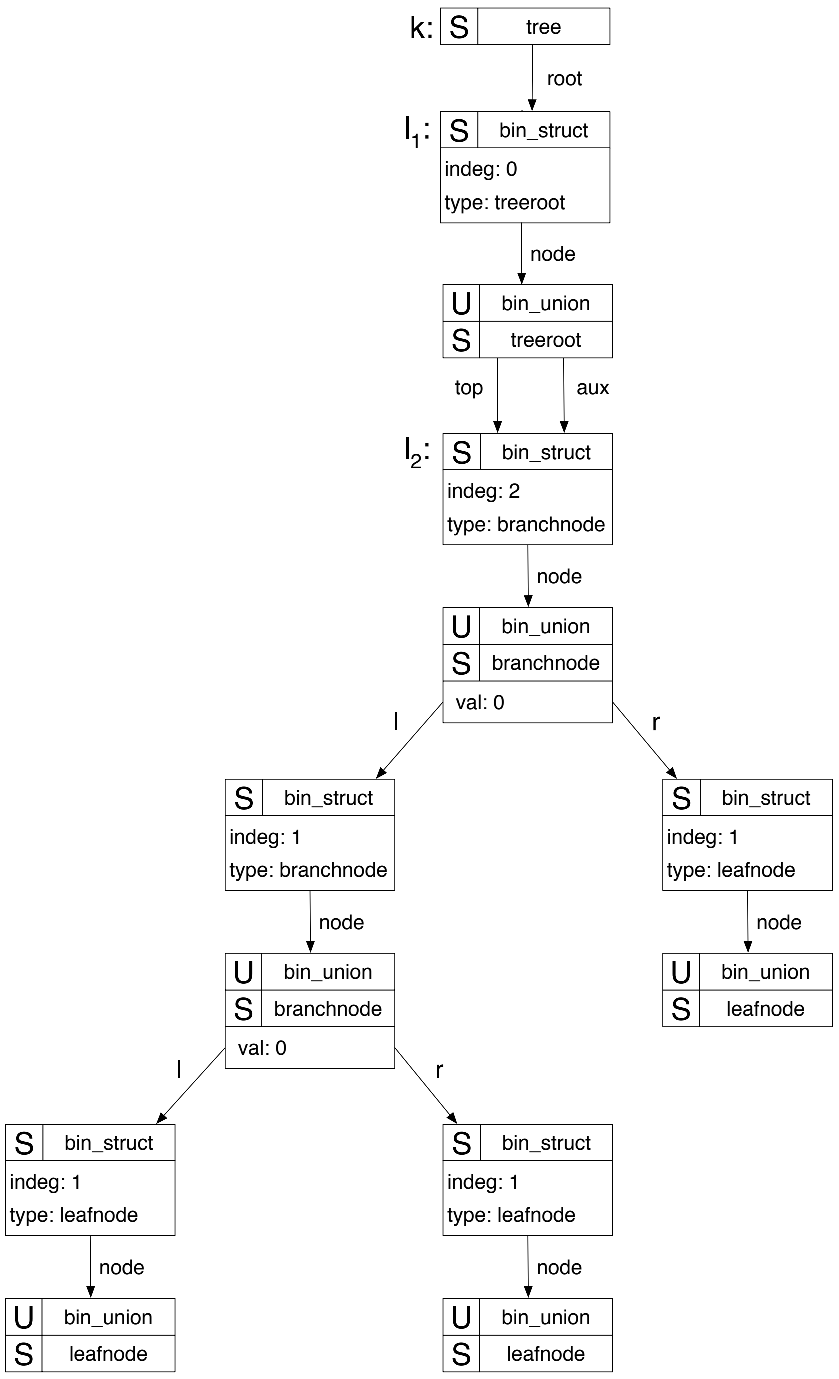


Figure 10.11: Large σ-structure conforming to the σ-schema τ_σ defined in Example 10.8.

reachable locations are node-pairs, which means this structure is a valid σ-structure.

In Section 10.1 we defined the graph signature Σ_σ resulting from a CGRS shape declaration σ. A graph is called a *σ-graph* if (1) it is a Σ_σ-graph and (2) there exists exactly one node with root label R.

We have now defined for a shape declaration σ both a class of σ-structures that exist in C states, and a class of corresponding σ-graphs. These σ-structures form the domain for transformer functions. Our mapping between the domains of states and graphs therefore operates only on these two heavily-restricted domains.

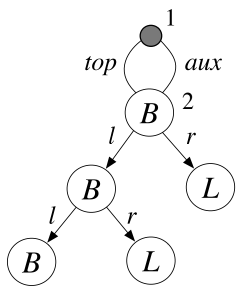
The relationship between the two is defined by the mapping function β_σ, derived from the CGRS signature declaration σ. This function takes as its input a state s over σ-schema τ_σ and a σ-structure l in s, and constructs a corresponding σ-graph $\beta_\sigma(s,l)$. Intuitively, the β_σ function maps heap locations to graph nodes, node-pairs to nodes, and edge fields to edges.

Definition 10.9 (β_σ)**.** Let s be a state over the σ-schema τ_σ. Let $l \in \mathit{Loc}_s$ be the heap-location of some σ-structure, and let $r \in \mathit{Loc}_s$ be the location pointed to by the struct's `root` field. Then graph $\beta_\sigma(s,l)$ is constructed as follows:

1. Construct a node v for every heap location n holding node-pairs in the reachable structure from the `root` field of r. v is labelled with the type recorded in the `type` field of the node-pair. This is determined using the inverse of the type-code function $\mathit{typenum}(-)$ described in §10.2.1. In other words the label function $l_V(b(n))$ has the value $\mathit{typenum}^{-1}(fld(n, S_s, \texttt{type}))$.

2. Let v_1,v_2 be nodes corresponding to locations n_1 and n_2. If the field f of the node-pair at n_1 is defined and points to n_2, then an edge e labelled f exists from v_1 to v_2.

Example 10.11 (β_σ)**.** Applying β_σ to location `k` in the σ-structure described in Example 10.10 gives the following Σ_σ graph.



As usual `branchnode` and `leafnode` are abbreviated to B and L, and the root node is shown as a small grey node. The nodes tagged with 1 and 2 correspond to nodes l_1 and l_2 in the σ-structure.

10.5 Correctness of translations

In this section we prove two main correspondence results for $\mathcal{C}[\![-]\!]$ and $\mathcal{G}[\![-]\!]$. First in §10.5.1 we show that the constructor functions $\mathcal{C}[\![-]\!]$ produced from a shape declaration σ constructs a shape-structure that corresponds to the accepting graph defined in σ. Then in §10.5.2 we show that the graph transformation rule constructed by applying $\mathcal{G}[\![-]\!]$ to a transformer declaration correctly models the transformer implementation produced by $\mathcal{G}[\![-]\!]$.

10.5.1 Correctness of constructor functions

The correctness requirement for constructors is as follows. Let s be a state and σ a shape declaration with name S. Let s' be the result of applying the constructor function in state s, and let l be the return value of the function in s'. For the constructor to be correct, it must be true that the graph $\beta_\sigma(s', l)$ is isomorphic to Acc_σ, the accepting graph defined from σ, and that no other shape structures are altered or created by the constructor function.

Theorem 10.1. *Let σ be a CGRS shape declaration. The corresponding constructor* `newgraph_`*S defined by $\mathcal{C}[\![\sigma]\!]$ always returns a shape-structure l in state s such that $\beta_\sigma(s, l)$ is isomorphic to Acc_σ.*

Proof. Let σ be a CGRS shape declaration with shape name S in program P, and let `newgraph_`S be the constructor defined from σ in the resulting C program $C[\![P]\!]$. Let Acc_σ be the accepting graph constructed from σ by $\mathcal{G}[\![\sigma]\!]$. Let s be a μC state. Let s' be the state resulting when `newgraph_`S is executed in state s, and l the return value of the function. Let $G_\beta = \beta_\sigma(s', l)$.

The graph G_β is constructed from all the node-pairs reachable from the root location. So to show that G_β is isomorphic to Acc_σ we show that for every node a node-pair is constructed, and show that the constructed nodes constitute the reachable structure from the root.

Suppose in the `accept` block there are defined node names V, W, ..., Z. Then in the resulting constructor, $\mathcal{C}[\![-]\!]$ declares corresponding variables V_s, W_s, ..., Z_s and V_u, W_u, ..., Z_u. Each declared node V has the following associated code, constructed by the semantic function $\mathcal{M}[\![-]\!]$.

```
V_s = malloc(sizeof(S_struct));
V_u = malloc(sizeof(S_union));
V_s->node = V_u;
V_s->type = typenum(T);
V_s->indegree = 0;
```

By appeal to the [$\mathbf{assn}_v$] and [**alloc**] rules, the two `malloc` assignments results in a node-pair. So, at the end of these assignments, for each declared node V there exists a variable V_s pointing to the struct of a node-pair and variable V_u pointing to the union. By appeal to the [$\mathbf{assn}_v$] rule, we now can see that the type field of each node-pair corresponds to the type of the node in Acc_σ.

We now show the edges of the accepting graph exist. Each edge statement results in the following code for node called V:

```
V_u->t_r(V).E = T_s;
T_s->indeg = T_s->indeg + 1;
```

In the resulting state, by appeal to the [$\mathbf{ref}_v$] and [$\mathbf{assn}_u$] rules, the E field of node-pair V must point to node-pair T. As each edge statement also results in an edge in graph Acc_σ between nodes V and T, the bijection b satisfies the third condition, in that for each field in the corresponding graph there will exist an edge between the two corresponding nodes with label E.

No matter what the structure of the `accept` block, for a shape called S, the constructor function ends execution with the following code.

```
new = malloc(sizeof( S ));
S->root = R_s;
return new;
```

$$\begin{tikzcd}
G \arrow[r, "\mathcal{G}[\![r]\!]"] & G' & \Sigma_\sigma\text{-total graphs} \\
s, l \arrow[u, "\beta_\sigma"] \arrow[r, "\mathcal{C}[\![r]\!]"] \arrow[ur, phantom, "="] & s', l \arrow[u, "\beta_\sigma"'] & \mu\text{C state}
\end{tikzcd}$$

Figure 10.12: Correctness requirement for $\mathcal{C}$ with respect to $\mathcal{G}$ and β_σ.

By the definition of the [$\mathbf{assn}_v$] and [**alloc**] rules, in the resulting state the variable `new` points to a fresh S-typed memory location. The constructor assigns the `root` field the location of the root-typed node-pair. Therefore $\beta_\sigma(s', l)$ is isomorphic to Acc_σ. $\square$

10.5.2 Correctness of transformer functions

The correctness requirement for transformers is as follows. Let r be a transformer function. Then $\mathcal{C}[\![r]\!]$ is the fragment of μC code that defines by implementation the transformer function's operational semantics. $\mathcal{G}[\![r]\!]$ is the corresponding graph transformation rule extracted from the transformer function. We write application of the rule r to graph H as $r(H)$, and application of transformer function t to shape-structure l in state s as $t(s, l)$.

The mapping function $\mathcal{C}[\![-]\!]$ is correct with respect to $\mathcal{G}[\![-]\!]$ and β_σ if for function r, state s and σ-structure l in s, the application of the code fragment $\mathcal{C}[\![r]\!]$ to state s has the same structural effect, modulo $\beta_\sigma(-,-)$, as applying the corresponding graph transformation rule to graph $\beta_\sigma(s, l)$. Formally, we require that $\mathcal{G}[\![r]\!](\beta_\sigma(s, l)) = \beta_\sigma(\mathcal{C}[\![r]\!](s, l), l)$. Equivalently, the diagram in Figure 10.12 must commute.

We prove this correctness result in three stages. The first part of the proof shows that the 'matching' portion of $\mathcal{C}[\![r]\!]$ constructs a set of *matching variables* that correspond to the matching morphism constructed by the left-hand side of the graph transformation rule $\mathcal{G}[\![r]\!]$. The second part shows that code checking the dangling condition fails if and only if the dangling condition does not hold. The third part proves that the 'rewriting' portion of $\mathcal{C}[\![r]\!]$ performs the same structural graph rewrite as rule $\mathcal{G}[\![r]\!]$. We ignore rewrites applied to value fields in the transformer as they have no effect on the correctness of the extraction function.

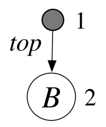
The first stage of the proof consists of proving that the portion of the code in Figure 10.5 labelled 'MATCHING' constructs matching variables that correspond to a matching morphism. This requires a notion of correspondence between a variable set and a morphism. Intuitively, a set of variables correspond to a morphism if each variable corresponds to a node in the morphism domain, and the value held in the variable points to a the associated location in the range.

Definition 10.10 (morphism correspondence)**.** Let s be a state and l the location of a σ-structure in s. Let $G = \beta_\sigma(s, l)$ be the resulting Σ_σ-graph, and $b : Loc_s \to V_G$ the bijection between locations and graph nodes that defines $\beta_\sigma(s, l)$. Let $v_1, \ldots, v_n$ be a set of variables in s. Let $r = \langle L \leftarrow K \to R\rangle$ be a graph transformation rule with n nodes on its left-hand side. Let $t\colon Var \to V_L$ be a partial bijection associating variables with left-hand side nodes. Let $h\colon L \to G$ be an injective morphism between L and G. Then we say that variables $v_1, \ldots, v_n$ *correspond to* morphism h in state s, if for all i such that $1 \leq i \leq n$, $b(var_s(V_i)) = h_V(t(V_i))$.

Parallel edges are forbidden by the definition of a Σ_σ-graph, as outgoing edges must be distinctly labelled. As a result, a set of variables in state s that correspond to a morphism $h : L \to G$ suffices to uniquely define the morphism.

Example 10.12 (morphism correspondence)**.** Let s be a state containing the σ-structure given in Example 10.10. This structure contains locations l_1 and l_2. Applying β_σ to this σ-structure gives the graph G shown in Example 10.11, with nodes v_1 and v_2 the nodes with tags 1 and 2.

We also have the following left-hand side graph L. Nodes v'_1 and v'_2 are respectively the nodes with tags 1 and 2.



Suppose we have variables `x1` and `x2`, and mapping function $t = \{\texttt{x1} \to v_1, \texttt{x2} \to v_2\}$. Let $h\colon L \to G$ be a partial morphism defined as $h_V = \{v'_1 \mapsto v_1, v'_2 \mapsto v_2\}$ and $h_E = \emptyset$. If $s(\texttt{x1}) = \mathsf{l}_1$ and $s(\texttt{x2}) = \mathsf{l}_2$, then `x1`, `x2` correspond to h in the state s.

Proposition 10.2. *Let T be a transformer function. If there exists a matching morphism between the $\mathcal{G}[\![T]\!]$ and $\beta_\sigma(s,l)$ then applying the matching code from $\mathcal{C}[\![T]\!]$ to a σ-structure l in s results in a set of matching variables corresponding to the morphism. Otherwise the code exists with return-value* `FALSE`.

Proof. Let $N_1, \ldots, N_m$ be the names of the nodes declared in T. Each node n declared in the left-hand side of transformer function r results in (1) a pair of variables n`_u` and n`_s` declared in function $\mathcal{C}[\![T]\!]$, and (2) a node in the left-hand side of rule $\mathcal{G}[\![T]\!]$. This relationship implicitly defines a bijection b between variables and left-hand side nodes.

Let s be a state and l the location of a σ-structure in s. Let graph G_β be the graph constructed by $\beta_\sigma(s,l)$. Let L be the left-hand side graph of the rule $\mathcal{C}[\![T]\!]$.

We prove that the correspondence holds by induction on the size of the partially-constructed matching morphism h_i. In the base case only a single root node exists. Let R be the name of the root node, and G the shape-name passed to the transformer. The matching code begins with the following assignment.

```
R_s = G->root;
R_u = R_s->node;
```

By the definition of a σ-structure, the `root` field must point to a root-typed node-pair. By appeal to the [assn] rules, after this code has executed the single variable R`_s` points to the root-typed node-pair. This variable alone must therefore correspond to a partial morphism h_0 that has a singleton domain consisting of the root-labelled left-hand side node r, with $h_0(r)$ pointing to the root-labelled node in the graph G_β.

For each declared node-name in the left-hand side, the function $\mathcal{C}[\![-]\!]$ constructs code to incrementally extend the matching morphism by an edge-field at a time. Let $e_1, \ldots, e_n$ be the sequence of left-hand side edge declarations defined by the edge enumeration.

Let us assume that edges $e_1, \ldots, e_i$ have been matched. Let $V_1, \ldots, V_m$ be the source and target nodes of these edge declarations. Assume variables V_1`_s`$, \ldots, V_m$`_s` correspond to partial morphism h_i that has as its domain the nodes of the left-hand side graph related to $V_1, \ldots, V_m$ by bijection b.

Let `S.E => T` be the next edge declaration e_{i+1} in the enumeration. This results in the following code-fragment.

```
T_s = S_u->$t_l(S)$.E;
if ( T_s->type != $typenum(t_l(T))$ )
  return FALSE;
else if (T_u == NULL)
  T_u = T_s->node ;
else if ( T_u != T_s->node )
  return FALSE;
```

Let $V_1, \ldots, V_p$ be the sources and targets of the edge declarations $e_1, \ldots, e_i$. We now show that at the end of this code fragment either (1) variables V_1`_s`$, \ldots, V_p$`_s` correspond to partial morphism h_{i+1} between the corresponding nodes and edges of the left-hand side graph and the target graph, or (2) no such extended partial morphism exists and the code returns `FALSE`.

Because graph G_β is a Σ_σ-graph, there must exist exactly one edge with source $h_i(b(S))$ and label E. There are therefore two possible failure-cases in extending the domain of h_i by the edge corresponding to edge-declaration e_{i+1}. (1) The target node of the edge in G_β is of the wrong type. (2) The target of the edge is already in the domain of h_0, but the edge in G_β points to the wrong node. Otherwise the match must succeed. We show that matching fails and the code returns `FALSE` if cases (1) or (2) hold, and succeeds otherwise.

In case (1), the code fails if the type field of the target node-pair is of the wrong type. By the definition of the $typenum(-)$ function, this shows that extending h_i along this edge reaches a wrongly-labelled node in graph G_β.

In case (2) both the edge source and target have been matched in h_i. The code will fail if the edge-field of the node-pair in S`_u` does not point to the node-pair in T`_s`. This occurs exactly if no edge exists between the two nodes in the resulting graph G_β. By appeal to the fact that the source of the new edge has already been matched, the code will return `FALSE` if no field exists labelled E between the heap locations held by S`_s` and T`_s`.

Otherwise the code writes the value into the target variable for the node-declaration. By the definition of β_σ, there must exist a corresponding node in graph G_β and edge labelled E between the source and target, so the new variable set V_1`_s`$, \ldots, V_p$`_s` corresponds to morphism h_{i+1}.

This suffices to show that at the end of the matching code we have a total morphism $h\colon L \to G$. The final section of the matching code ensures that the morphism is injective, by comparing the values held in all of the variables. The result is that the code fails if any of the variables point to the same node-pair, which occurs if and only if the corresponding morphism is non-injective. $\square$

The second portion of the code in Figure 10.5 (marked by the label DANGLING CONDITION) checks that the dangling condition is respected by the constructed morphism.

Proposition 10.3 (checking the dangling condition)**.** *Let s be a state resulting from the matching code in $\mathcal{C}[\![T]\!]$ with σ-structure at location l, and let $v_1, \ldots, v_n$ be variables in s corresponding to morphism h for graph $\beta_\sigma(s, l)$. Then $\mathcal{C}[\![T]\!]$ returns* `FALSE` *in s if and only if $\mathcal{G}[\![T]\!]$ violates the dangling condition in $\beta_\sigma(s, l)$ with matching morphism h.*

Proof. Let s be a σ-state and v a σ-structure. By the definition of a σ-structure, the `indeg` field of a node-pair in the σ-structure is equal to the in-degree of the corresponding node in the graph $\beta_\sigma(s, v)$.

The following code is generated for node V declared in the left-hand side graph if the corresponding node $n \in L$ in rule $r = \langle L \leftarrow K \rightarrow R\rangle$ is (1) deleted by the rule, and (2) has in-degree c in L.

```
if ( V_s->indeg != c ) return FALSE;
```

This code fragment will return `FALSE` if the in-degree of the node $h^{-1}(n)$ in graph $\beta_\sigma(s, v)$ differs in degree from the node n in L. This holds if and only if the dangling condition fails for the corresponding node. As the code is generated for all deleted nodes, the result is that the dangling condition is checked for the whole rule. $\square$

We can now prove that the extraction function $\mathcal{G}$ produces from a transformer function T a graph transformation rule that corresponds to the operational semantics of the function defined by the implementation function $\mathcal{C}$.

Theorem 10.4 (correctness of translation)**.** *Let s be a σ-state that includes a σ-structure l. Let T be transformer function. Applying $\mathcal{C}[\![T]\!]$ to l in s using* *the semantics given in Section 10.3 results in state s' such that $\beta_\sigma(s, l) \Rightarrow_{\mathcal{G}[\![r]\!]} \beta_\sigma(s', l)$, or will return false if no such derivation exists.*

Proof. Let G_β be the graph $\beta_\sigma(s, l)$. We have shown in Proposition 10.2 and Proposition 10.3 that the values written into variables by the matching and dangling portions of $\mathcal{C}$ correspond to an injective matching morphism $h : L \to G_\beta$ that satisfies the dangling condition. To show that the code $\mathcal{C}[\![T]\!]$ has the same structural effect as $\mathcal{G}[\![T]\!]$ therefore requires only to prove the section of code labelled 'Rewriting' performs rewrites on the state that correspond to the graph rewrites performed by graph transformation rule $\mathcal{G}[\![T]\!]$.

The rewriting code of $\mathcal{C}$ first generates code to delete nodes. The code consists of a pair of calls to the `free` function. By the [**free**] rule, in the resulting state the corresponding node-pair will be deleted, with the result that the same nodes will be deleted in the graph. Note that incoming edges are deleted by the edge rewriting code, meaning that reachability is preserved.

The code then retypes node-pairs by writing into the `type` field. This has the result that the node labels for the corresponding nodes in $\mathcal{G}_\beta$ are modified to conform to the new type. The rewrite rule has the same effect when relabelling nodes.

The code then constructs new nodes. This code is generated using the function $\mathcal{M}[\![-]\!]$ used in defining the constructor function, so by the same argument used in Theorem 10.1 this code successfully generates a node-pair of the required type with undefined outgoing edges.

The rewriting code of $\mathcal{C}$ then reassigns all of the fields present in the right-hand side of the transformer function. In the domain of graph transformation, if an edge f exists between nodes l_1 and l_2 in the left-hand side, then in the resulting graph G' there must be an edge between the corresponding nodes matched by the morphism. The code generated is of the form:

S_u->$t_r(S)$.E = T_s;

By Proposition 10.2, the source and target variables correspond to the source and target in the graph G_β. By the rules of the μC semantics, the above code will result in an edge-field between node-pairs matched by the morphism, as required by the left-hand side. Therefore, in the corresponding graph, there will exist edge between the corresponding nodes matched by the morphism.

Finally, the code updates the `indeg` fields of all of the right-hand side nodes to reflect the updated indegree counts. By the definition of $\mathcal{C}[\![-]\!]$ the indegree is altered by the number of edges deleted and generated by the graph update, so the σ-graph property is maintained. This auxiliary data is not present in the graph G_β.

We have shown that the code will exit with `FALSE` if it fails to construct a variable set corresponding to a matching morphism. By the structure of the constructed code, it is not possible for it to exit while executing the rewriting portion of the function until the final `return TRUE` statement. This completes the proof. □

10.6 Shape safety guarantees in CGRS

Let us assume that $\overset{C}{\Rightarrow}$ is a semantics of full C, and let $\overset{\mu}{\Rightarrow}$ be the μC semantics defined in 10.3. Given a C memory state s, we extract a μC state s_μ by preserving any structs and unions that include only integer and pointer fields, and turning all other memory locations to undefined values. We assume that the semantics of μC corresponds to full C, in the sense that for any valid piece of μC syntax P, $\langle P, s\rangle \overset{C}{\Rightarrow} s'$ if and only if $\langle P, s_\mu\rangle \overset{\mu}{\Rightarrow} s'_\mu$.

Definition 10.11 (shape safe location)**.** Let σ be a shape declaration in P. Let s be a state in full C and let s_μ be the corresponding μC state. Let l be a location in s that is the location of a σ-structure in s_μ. Then we say that *l is shape-safe in s* if the graph $\beta_\sigma(s_\mu, l)$ is a member of $\mathrm{L}(\mathcal{G}[\![\sigma]\!])$, the language of graphs defined by the graph reduction system $\mathcal{G}[\![\sigma]\!]$.

Definition 10.12 (shape safety preservation)**.** Let P be a C program, and let σ be a shape declaration, and let l be a location in s that is shape-safe with respect to σ. We say that P *preserves shape safety* if every state s' such that a derivation $\langle P, s\rangle \overset{C}{\Rightarrow}^* s'$ exists is shape-safe with respect the σ.

In this section we assume a sound shape-checking algorithm that determines whether a graph-transformation rule is shape-preserving with respect to a GRS. That is, given a GRS G and a graph transformation rule r, if the algorithm returns **tt**, then $\mathrm{L}(G)$ is closed under r. The problem of shape checking is known to be undecidable even for context-free shapes – see [31] for a reduction of the inclusion problem for context-free graph languages to shape safety checking. Consequently no general sound and complete shape-checking algorithm exists.

An example of a shape-checking algorithm is Shape-Check, described in [3]. See 9.1 for a of this algorithm.Given a shape-checking algorithm, we can make the following guarantees for CGRS programs.

Theorem 10.5. *Shape structures constructed by a CGRS constructor are shape-safe by construction.*

Proof. Let σ be a shape declaration and let P be the constructor defined by $C[\![\sigma]\!]$. As a consequence of Theorem 10.1, applying P in any state results in a state s and location l such that $\beta_\sigma(s_\mu, l)$ is isomorphic to the accepting graph for $\mathcal{G}[\![\sigma]\!]$. As a consequence, l is shape-safe in s with respect to σ. □

Theorem 10.6. *Given a sound shape-checking algorithm, there exists a sound procedure for checking statically whether the transformer functions in a CGRS program preserve shape-safety. No general procedure that is sound and complete exists however.*

Proof. Let σ be a shape declaration, and let l be a location in s that is shape-safe with respect to σ. Let T be a transformer function. As a consequence of Theorem 10.4, l is shape-safe in the state s' resulting from applying $\mathcal{C}[\![T]\!]$ to l if and only if $\mathcal{G}[\![T]\!](\beta_\sigma(s, l)) \in \mathrm{L}(\mathcal{G}[\![\sigma]\!])$.

As a consequence of this, if the shape checking algorithm verifies that the rule $\mathcal{C}[\![T]\!]$ preserves membership of the language $\mathrm{L}(\mathcal{G}[\![\sigma]\!])$, then function $\mathcal{C}[\![T]\!]$ is guaranteed to preserve shape-safety for shape σ. Conversely, if the algorithm shows that $\mathcal{C}[\![T]\!]$ is not shape preserving, then there exists a state s and location l that is shape-safe with respect to σ such that l is not shape-safe in the state s' resulting from applying $\mathcal{C}[\![T]\!]$ to l.

As a result of the undecidability of shape-checking, no sound and complete procedure exists for checking statically whether transformer functions preserve shape-safety. □

Pointer manipulations that are not modelled as transformer functions cannot be checked by the algorithm, and can potentially violate the shape membership guarantees. CGRS can therefore only make guarantees of shape safety in programs that manipulate shape structures only by transformer functions.

10.7 Implementing and optimising CGRS

We have defined the concrete semantics of CGRS by a mapping function to C. However, our focus in this work has been abstract correctness rather than concrete implementation, and we have not implemented CGRS in practice. It should be the subject of future work to develop an implementation, consisting of a compiler and suite of small test programs.

An implementation will require of a parser for CGRS constructing a parse tree for signatures, shapes and transformers. From this parse tree the implementation will then construct C code conforming to the definition of a transformer function, and a textual representation of the corresponding graph transformation rules. Haskell may be a suitable language for this implementation.

The functions produced by the mapping to C are considerably larger than the corresponding C code. In the case of `tree_goleft` defined in Example 9.4, the resulting code is 35 lines long, with most of the body of the code consisting of matching code and checks for injectivity. The code is given in §10.2.3. This compares to 12 lines for the transformer declaration, and 6 lines for the corresponding C code given in §9.4.

This level of code length increase also seems quite typical for CGRS transformer functions. For the rebalancing transformer `rup` from §9.3, the constructed C function is 49 lines long. The transformer is 19 lines long, and the C code given in §9.4 is 12 lines.

We hope that future work will reduce the size of code produced from CGRS, by cutting down on redundant checking. It may be possible to use some of the information in the shape declaration to remove redundant checks, by eliminating checks that cannot fail in shape members. It may also be that some redundant checks are already eliminated by the compiler.

A consequence of the lack of an implementation is that CGRS has only been applied to a small number of toy problems. It should be the subject of future work to apply CGRS to a wider range of example problems, and refine the language and verification approach based on this practical experience. Testing the language in this way will also allow us to compare the speed of CGRS programs to corresponding C programs.

Chapter 11

Other approaches to shape safety

In this chapter we compare CGRS to other approaches that have been developed for ensuring the shape safety of C-like programs. We also consider approaches to specifying classes of graphs based on graph transformation.

11.1 Shape types and structured gamma

The approach used in CGRS is most similar to Fradet's and Le Métayer's work on shape specification using structured gamma [31]. Structured gamma programs are based on a 'chemical reaction' metaphor. Operations consist of *reactions*, which act non-deterministically on a *structured multiset* of tuples. These tuples consist of a type field that restricts the tuple's arity, and a fixed number of address fields. These multisets can be seen as representing hypergraphs, with the addresses as vertices and the tuples as labeled edges. Structured gamma reactions are know to be equivalent to context-free hyperedge replacement rules as defined in §2.3.

The structured gamma approach forms the basis of Fradet and Le Métayer shape-checking approach. They model pointer structures as structured multisets, with the properties of structures defined by sets of reactions that generate sets of multisets. Pointer rewrites are modelled as pattern-matching rewrites.

This approach is implemented as an extension to C called *Shape-C*, which takes a similar approach to implementing the structured gamma approach as we have in CGRS for the SPGT project. Shape-C introduces a special syntax for defining both pointer structure shape and pointer rewrites. Like CGRS, it is implemented by a mapping to C.

There are several differences between our approach and that of Shape-C. Most fundamentally, Shape-C is restricted to shapes specified by context-free graph grammars. The graph reduction specifications incorporated in CGRS – even when restricted to polynomial GRSs – allow programmers to specify non-context-free data structures such as grids and various forms of balanced trees.

In addition, our approach to transformer functions in the design of CGRS is quite different. While Shape-C defines a simple syntax for rewrites that appear inline in programs, in CGRS transformers are defined as functions. This has the advantage of encapsulating large rewrites away from the main body of the program.

In Part II of this thesis, we give conditions under which GRSs have an efficient run-time membership test. We have shown that several of the classes of graphs defined by graph reduction systems in [4] permit linear-time membership checking (for example: balanced binary trees, §4.2.1; grid graphs 4.2.2; cyclic lists, Example 4.2). Fradet and Le Métayer, in contrast, do not consider the efficiency of run-time shape checking. Their reaction-based approach is equivalent to context-free graph grammars, which in general are known to have an NP-complete membership problem.

11.2 Specifying structures using logic

Several approaches to specifying the properties of pointer structures are based on program logics. An example of this is the work on *graph types* [49]. These are spanning trees with additional pointers defined by path expressions; they form the basis of *pointer assertion logic* [58], a monadic second-order logic for expressing properties of pointer structures in program annotations. In addition to a logic, this approach defines a C-like language for declaring the properties of pointer structures.

Graph types are declared in much the same way as C structure declarations. Nodes in a graph type can have two kinds of outgoing edges: *data* fields and *pointer* fields. A data field can only point to a node that is not pointed to by any other data field of any other node. In this way, the data

```
type Node = {
  data next : Node;
  pointer prev : Node[ this^Node.next={prev} ];
}
```

Figure 11.1: Graph type for a doubly-linked list node.

fields of a pointer structure form a spanning tree.

Pointer fields, in contrast, can point to any node in the structure. However, pointer fields are annotated with *routing expressions*, written in pointer assertion logic, which express a set of acceptable target nodes. Pointer fields are used to implement the non-tree properties of a graph structure.

For example, Figure 11.1 shows the graph type for a doubly-linked list. The tree-shaped backbone of the list is formed by the data field `next`. The `prev` pointer field is marked with the following routing expression:

```
this^Node.next={prev}
```

This states that the set of nodes that point to the current node `this` through the data field `next` must consist of the node pointed to by `prev`.

For verification, routing expressions are mapped to a fragment of monadic second-order logic: the *weak monadic second-order theory of 2 successors*. Hoare triples based on this fragment are decidable over loop-free code using the MONA tool, described in [44]. Loops must however be explicitly annotated with invariants in the normal Hoare logic manner.

This approach requires programmers to use quite a sophisticated logic, but the formalism is still too weak to express some important properties. Møller and Schwartzbach give a list of shapes that have been verified using this tool, including threaded trees and non-balanced red-black search trees. However, balanced structures such as balanced binary trees and red-black trees are outside the expressive power of pointer assertion logic. In contrast the GRS approach allows us to specify balanced structures and other non-context-free shapes.

Another logic that has been the basis of a large amount of work is *separation logic* [46, 69]. This is a logic for shared mutable data-structures that adds a so-called *separating conjunction* to the normal operators of first-order logic. Separating conjunction divides the heap into disjoint regions for which different logical formulae hold, making it possible to reason locally. The syntax and semantics of separation logic are described in detail in §6.1 , so here we will describe only the approaches to verification that have been developed based on it.

The main use for separation logic is as the basis of an extension of Hoare logic that allows local reasoning about programs. Separation logic can reason about quite complex pointer-manipulating algorithms in a way that would be difficult in normal Hoare logic. The earliest work on verification based on separation logic consists of by-hand proofs of small programs, with explicit annotations of Hoare triples. The Schorr-Waite graph marking algorithm was verified in [78], a copying garbage collector in [9].

Such by-hand proofs suffer from the same problems of complexity as Hoare logic. Even small programs can require very large proofs, which can take a considerable amount of time to write. The Schorr-Waite algorithm requires a twenty-eight page paper to detail the proof, even though the program itself is only thirty lines long.

More recent work on separation logic has resulted in the SmallFoot tool [7], which automatically verifies invariants written in a decidable fragment of separation logic. The decision procedure used by SmallFoot makes use of symbolic execution. Recursive predicates other than simple lists must be added by hand to the tool's logic, and the inference rules in the tool must be updated to reflect the semantics of the new predicate.

In comparison with CGRS, Smallfoot has the advantage that it works over plain C, rather than an augmented extension of C. Smallfoot's fragment of separation logic however is less expressive than CGRS's reduction systems, in that it operates over a fragment of separation logic similar to the symbolic heaps used for the Space Invader tool (see below, p. 251.

In addition, even with automatic checking, the correct invariant for verifying a program can be difficult to discover. Verification also depends strongly on the selection of recursive predicates. Smallfoot copes well with invariants defined using list segments, but for more complex structures such as trees, which CGRS handles quite naturally, the proof system must be extended by hand to reason about them successfully. The construction of a correct set of inference rules for a given predicate can present a considerable challenge.

Other logics have been used to specify the properties of pointer structures. Alias Logic [14] abstracts away from the presence of a garbage collector to reason purely about aliasing. Navigation Temporal Logic [19] uses temporal reasoning to define all of the possible evolutions of the heap. However, both of these logics are presented in largely theoretical terms, rather than as part of a system for verifying pointer programs.

11.3 Shape analysis

Other approaches to shape safety are based on the automatic inference of the shape properties of pointer structures. *Shape Analysis* is the major approach in analysis-based shape safety checking. A shape analysis constructs by symbolic execution a over-approximated representation of all possible pointer structures at each point in a program. These structures can then be analysed for required shape properties.

The initial work on shape analysis uses three-valued logic as an abstract domain [54]. Concrete structures in this work are represented by logical structures in two-valued boolean logic with transitive closure. A two-valued structures are defined from a universe of elements and a set of predicates that give an interpretation over these elements. Core predicates record the pointers between elements, while instrumentation predicates record larger-scale properties, such as reachability.

Three-valued logic extends two-valued boolean logic with a new 'indefinite' value. A three-valued structure *embeds* a two-value structure if there exists a mapping from the nodes of the two-value structure such that predicates either maintain their value or have value unknown. Intuitively, embedding 'blurs' the embedded two-valued structure by mapping from definite to indefinite values. A single three-valued structure represents the (possibly infinite) class of two-valued structures embedded in it.

Shape analysis as presented in [54] constructs for each program point a set of three-valued structures. These structures embed all the possible two-valued structures that can occur at the program point. The set of structures is constructed by symbolic execution over the set of structures. First the structures are focussed, breaking them into cases with definite values for a so-called focus formula. Then the structures are updated according to a program semantics. Finally the resulting set of formulas is coerced to remove contradictory structures.

Three-valued shape analysis has been implemented in a tool called TVLA

[55, 53]. A considerable amount of effort has been focussed on improving the speed of TVLA's analysis. A recent paper claimed a 50-fold increase in speed over earlier versions of the tool [12]. Other work has developed interprocedural version of shape analysis for improved modularity [70, 47].

Shape analysis differs from CGRS in its general approach to verification. CGRS defines a new class of program construct which are more suitable for shape checking, and annotates the program with shape invariants consisting of GRSs. Shape analysis in contrast keeps the original syntax of C and generates invariants by symbolic execution of the program. Shape analysis can simulate the invariant-checking style by annotating invariants and checking that they are a fixed-point of the analysis. The converse does not hold however: CGRS has no mechanism for automatically generating shapes.

CGRS shapes must be annotated on the program, and the shape of a structure in CGRS is invariant throughout the program. (The SPGT approach can in general deal with shape-changing graph transformations [3], but CGRS currently does not implement this feature). Shape analysis in contrast can handle invariants that alter throughout the program.

Shape analysis has the advantage of a large amount of automation. However, the precision of the structures generated by analysis varies considerably. The choice of instrumentation predicate is extremely important in determining how quickly the algorithm will derive a three-value structure, and how precise the resulting structures will be. Consequently, while analysis is entirely automatic once the correct predicates have been chosen, selecting and correctly defining these predicates may require considerable human intervention. In contrast in CGRS annotations are of a precision determined by the user.

The approach to semantics differs between CGRS and shape analysis. CGRS has a single semantics for transformers, while the semantics of constructs in shape analysis depends on the abstract domain chosen. The two can be seen as similar however, in that transformers must be defined using the node and edge labels defined by the signature.

The abstract domain of shape analysis is restricted to predicates in three-valued logic with reachability. This logic is sufficient to express structures such as trees. However, it gives no means to express large-scale properties such as tree balance, so we conjecture that it is less expressive than CGRS's shapes based on graph transformation. Shape analysis is guaranteed to terminate however, as the domain of structures is finite. This is not true for the CGRS shape-checking algorithm, which may diverge.

Separation logic has also been used as an abstract domain for shape analysis. The Space Invader tool replaces three-valued logic with *symbolic heaps*. These are formulas defined in a decidable fragment of separation logic. As with three-valued logic, the analysis generates for each program point an abstract representation of possible heaps. Termination is ensured by a normalisation process based on entailment which replaces large formulas with smaller formulas.

Just as the success of shape analysis using three-valued logic is sensitive to the choice of instrumentation predicate, the success of analysis in Space Invader is sensitive to the choice of abstract predicates and of normalisation procedure used. Consequently the Space Invader tool has been targeted closely on the domain of device drivers.

This has the advantage that drivers are written to a standard template and the data structures they use are normally of a particular form. This has allowed the tool to be very successful, in terms both of locating bugs and of the size of programs analysed. Space invader claims verification of shape-safety for 10,000 line programs [79], and location of substantial bugs in real-world device drivers [5].

Symbolic heaps are closely related to the fragment of separation logic that we use in Part III. See §8.3.2 for a discussion of this correspondence between our fragment and symbolic heaps. As grammars based on hyperedge replacement are known to be formally less expressive than grammars based on double-pushout rewriting, this suggests that Space Invader's abstract domain is formally less expressive than the shapes used by CGRS.

A major advantage of both separation logic and three-valued logic shape analysis is that they have been applied to a large number of real-world problems, some of which are of considerable size [12, 79]. This has meant that TVLA and Space Invader tools have been fine-tuned in the face of experience, and that their tools have been developed to perform well when applied to large blocks of code. This contrasts our own approach, CGRS, which has only been applied to a small number of toy problems.

Our objective in CGRS has been to design a sound foundation language on which further work can be based. Early shape analysis work, such as [54], similarly dealt only with simple example problems. It will be the subject of future work to apply CGRS to more substantial case-studies, and so to fine-tune it based on practical experience. See §10.7 for a discussion of this.

11.4 Specifying structures by graph transformation

We have described above the major approaches to verifying pointer structures in C-like languages using invariants. In this section we describe several approaches that use graph transformation rules to analyse and verify of the properties of classes of graphs.

High-level conditions [2, 41, 2] are an approach to defining the properties of graphs based entirely on graphs and morphisms. Graph properties are specified by *conditions*, which require the existence of certain combinations of morphisms. Conditions can be negated and nested, giving a highly expressive system for specifying properties. These conditions are used in two ways: to control the application of rules, and to specify invariants.

In [41] it is shown that conditions can be used in practice to specify the pre and post-conditions of a graph program, in the sense of [42]. It is shown that the weakest precondition of a graph program can be constructed. In [2] a generalisation is presented that defines so-called program conditions that relate objects between the start and end of a program.

High-level conditions differ from CGRS's graph reduction systems in that high-level conditions are a purely static, existential constraint. GRSs are defined algorithmically, by reduction. A consequence of this is the weakest preconditions for a high-level condition can be easily defined. In contrast, the problem of shape-checking for GRSs is known to be undecidable. This also suggests that GRSs are formally more expressive than high-level conditions.

The more algorithmic approach of CGRS has the advantage that every GRS has an associated membership test. We demonstrate in Chapter 4 that certain GRSs can be tested for membership in linear time using graph reduction. In addition, high-level conditions seem difficult to understand without a deep understanding of graph transformation. Graph reduction seems to have a stronger intuitive meaning as a means of specifying classes of graphs.

Another approach to specifying and verifying the properties of graph classes is the abstract graph transformation work of [68, 13]. In this work classes of graphs are specified by a *shape*, consisting of a graph and a node and edge multiplicity function. A concrete graph G is represented by a shape S if there exists a shaping morphism $s\colon G \to S$ respecting the multiplicity functions. A shape S is abstractly represented by a more abstract shape S' if there is an abstraction morphism $s'\colon S \to S'$.

It is shown in this work that graph transformation rules over graphs can be lifted to shapes. Consequently, for a given set of rules and initial shape a transition system between shapes can be constructed reflecting all the behaviors of the system of rules. In later work [13] a modal logic is presented that is preserved and reflected by shaping and abstraction, making it suitable for reasoning about abstract graph transformation.

This work on abstract graph transformation be seen as a high-level generalisation of shape analysis (see §11.3). In both approaches, a finite representation is defined and transition system induced from some program (in this case a set of graph transformation rules). However, unlike other shape-analysis work, abstract graph transformation has not been applied to real-world examples.

Abstract graph transformation differs from CGRS in much the same ways as shape analysis. Invariants in abstract graph transformation are generated by analysis, rather than checked. Invariant checking can be simulated however by supplying an invariant and checking whether it is a fixed-point of the analysis. As with shape analysis, a major disadvantage is that precision depends strongly on the choice of finite domain.

Unlike shapes in CGRS, abstract graphs do not come with a membership test. Checking membership seems to require the construction of a matching morphism between a concrete graph and abstract graph, which may be expensive. In [13] it is suggested that the symbolic execution in abstract graph transformation also suffers from problems with performance.

Abstract graph transformation are formally less expressive than graph reduction systems. Abstract graphs as presented in [68] correspond to a fragment of first-order logic, means that unaugmented abstract graphs are weaker than even hyperedge replacement as a mechanism for specifying properties. Like shape analysis, to express interesting large-scale properties such as reachability and cyclicity, extra annotations must be added to the abstract graph. In addition, the analysis presented in [13] cannot in general check whether annotated properties such as cyclicity are preserved.

Unlike CGRS, neither high-level conditions nor abstract graph transformation has been applied to pointer programming. However, both approaches could in principle be applied to the problem of pointer verification. To do this would require either (1) an extraction function from a pointer language to a graph transformation system, or (2) the addition of constructs to the pointer language corresponding to graph transformation rules.

In fact, both abstract graph transformation and high-level conditions are to some degree orthogonal to the results we present for CGRS. We have defined a semantics for graph-transformation constructs in a C-like language, and an extraction function from graph transformation rules (see §10.1). The rules produced by extraction could be checked by a system based on abstract graph transformation or high-level conditions, rather than one based on graph reduction. See 12.2.10 for a brief discussion of this idea.

Part V

Conclusion

Chapter 12

Conclusions and further work

This thesis describes three pieces of research concerned with the use of graph transformation rules for specifying and manipulating pointer structures. We have described syntactic conditions ensuring efficient derivation for graph transformation rules and applied them to the problem of graph recognition. We have defined a correspondence between hyperedge replacement grammars and separation logic, an alternative approach to pointer safety. Finally, we have defined the language CGRS that implements shape checking using graph transformation rules. In this chapter we summarise and evaluate our major contributions (§12.1) and suggest areas for further work (§12.2).

12.1 Thesis summary

12.1.1 Fast graph transformation

Our first objective was to show that graph transformation rules can be made an efficient mechanism for checking the properties of graph structures. To achieve this aim, Part II describes defined a general framework for improving the worst-case time complexity for individual rules and sequences of rules. We then use this framework to achieve our objective, by defining two kinds of fast reduction system with a linear-time membership checking algorithm.

Our approach is based on restricting rules and graphs using simple syntactic conditions. We have defined conditions on rules that ensure an improved worst-case execution time. We define two kinds of condition: conditions requiring uniquely-labelled root nodes (Conditions 1, 2, and 3), and conditions requiring roots (Conditions R1, R2, and R3).

We have shown that Conditions 1, 2, and 3 (§3.2) ensure linear-time application of unrooted rules (Proposition 3.8 and Proposition 3.10). We have shown that Conditions R1, R2, and R3 (§3.3) ensure constant-time application of rooted rules (Proposition 3.15 and Proposition 3.17). In addition, we have shown that Condition 1 and conditions R1, R2 and R3 ensures linear-time termination for multi-step derivations (Theorem 3.23, Proposition 3.18).

We have applied these conditions to the recognition of graph languages by reduction to give two new classes of fast recognition systems - linear LGRSs based on unrooted rules, and linear RGRSs based on rooted rules. We have shown by example that these reduction systems can be used to define a number of interesting non-context-free languages (§4.2). We have shown that both LGRSs and RGRSs have a linear-time membership test (Theorem 4.4, Theorem 4.21). However, LGRSs and RGRSs are of incomparable expressive power (Proposition 4.23 and Proposition 4.24).

The work presented in this part of the thesis proves formally that rules under our syntactic conditions have an improved worst-case time complexity over the general case. In this sense, our rules are 'fast'. However, we have not shown that our approach is faster than other approaches when it is applied to any particular problem domain. An improved worst-case application time does not imply that our approach will improve the application time for a particular domain.

There are three reasons for this. For small rules, the cost of application may be dominated by fixed costs, crowding out the benefits of our approach. Also for particular domains, heuristic approaches may achieve better results than our approach in most cases. Finally, we have focused purely on worst-case results rather than on algorithm optimisations. Other systems may perform better than ours due to better optimisation.

To show that our approach improves application times in practice (and so to more fully justify the description of our approach as 'fast graph transformation') we require empirical evidence. To do this, we also require an implementation. The work on fast graph transformation presented in this thesis exists entirely abstractly, while testing it will require the concrete execution of real graph-transformation systems. We hope in future work to develop and test more concrete instantiations of fast graph transformation (see §12.2.1).

12.1.2 Hyperedge Replacement and separation logic

Our second objective was to relate approaches to shape safety based on graph transformation to other shape safety approaches. Part III examines the relationship between separation logic and heap-graph grammars based on hyperedge replacement, and shows that our fragment of separation logic is of corresponding expressiveness to heap-graph grammars. The correspondence we have discovered fulfils our objective by illuminating the properties of both hyperedge replacement and separation logic.

We have defined a fragment $\mathcal{SL}$ of separation logic and shown that that formulas in this fragment are of equivalent power to hyperedge replacement grammars. To do this, we have defined a translation function s from grammars to formulas, and a translation function g from formulas to grammars. We have also defined a bijective function α that maps from hyperedge replacement states to heap graphs.

We have proved our translation functions s and g correct with respect to α. That is $g \circ \alpha = \alpha \circ g$ (Theorem 7.6), and $s \circ \alpha^{-1} = \alpha^{-1} \circ s$ (Theorem 7.10). Consequently, the fragment $\mathcal{SL}$ is of equivalent expressive power to the class of heap-graph grammars, modulo α (Theorem 8.3)

Our fragment of separation logic inherits the inexpressibility properties of hyperedge replacement, and we have described several languages that consequently cannot be defined in our fragment. We have shown that hyperedge replacement cannot be used to model full separation logic by showing that the standard first-order logic constructs $\wedge$, $\neg$ and **true** cannot be modelled by a heap-graph grammars (§8.1). However, we have also shown that the symbolic heaps in common use in the separation logic world are close to our fragment, which suggests that our fragment is of practical utility, rather than just a curiosity.

12.1.3 A language for shape safety

The third objective of this thesis was to show that graph transformation rules can be used to verify the safety of pointer structures in a C-like pointer language. Part IV of this thesis describes our language CGRS that adds graph transformation constructs to C. CGRS fulfils our objective by merging the quite different idioms of graph transformation and C into a single language. Constructs in CGRS can be modelled as graph transformation rules, permitting shape-safety checking using the SPGT approach [4, 3].

CGRS extends C with transformer functions resembling graph transformation rules, and shape-specifications resembling graph reduction systems. We have defined a syntax for CGRS and given both small and large examples of CGRS constructs. We have shown by example that large pointer rewrites can be clearly expressed using CGRS transformers (§9.3).

We have defined both a concrete and abstract semantics for CGRS constructs. The abstract semantics is defined by the function $\mathcal{G}$, which extracts graph transformation rules and GRSs from CGRS constructs. The concrete semantics is given by the function $\mathcal{C}$, which maps constructs to blocks of C code. This concrete code defined functions corresponding to transformer declarations, and also defines constructor functions for shapes.

We have shown that the CGRS concrete semantics given by $\mathcal{C}$ is correct with respect to the abstract semantics given by $\mathcal{G}$ and the abstraction function β_σ (Theorem 10.4 and Theorem 10.1). This means that, for a given transformer function F and shape structure s, it is always true that $(\beta_\sigma \circ \mathcal{C}[\![F]\!])(s) = (\mathcal{G}[\![F]\!] \circ \beta_\sigma)(s)$.

We have define a notion of shape safety with respect to the shapes defined by a CGRS program (Section 10.6). Any shape structure constructed by a CGRS constructor function is shape-safe by construction (Theorem 10.5). Finally, we have shown that transformers can be checked statically to determine whether they preserve shape-safety using the checking algorithm described in [3] (Theorem 10.6)

While we have defined a mapping to C programs, and so abstractly defined an implementation, CGRS has not been concretely implemented. Examples given in this thesis have been hand-compiled to C using the translation given in Chapter 10. The resulting C programs have been compiled using a C compiler, but no large-scale testing with real-world examples has been undertaken.

We hope that CGRS forms a suitable basis for future research on shape safety, and hope in future work to develop a practical and efficient implementation (see §10.7 for a discussion of this).

12.2 Further work

12.2.1 Experimental results for fast graph transformation

As discussed at the end of §12.1.1, we require experimental results to show that our syntactic conditions on rules result in an improvement in application times in practice. However, we expect that the results of these experiments will depend strongly on (1) the choice of domain, and (2) the preexisting optimisations in the system. Experiments will therefore require careful choice of a methodology in order to make them fair and representative.

As a starting point, we propose to take an existing system such as the YAM compiler for the GP system [57] and adding support for fast graph transformation. We may have to extend the language with annotations denoting the root node, signature and so on. We can then compare the system using our approach with the system without it. Applying our work to a graph transformation benchmark, such as [76], may be a good way to produce results that can be compared with other systems.

Experimental work of this kind will also illuminate possible ways of optimising our approach. In this thesis we have prioritised clarity of explanation above optimisation. For example, in §3.4 we said that the given version of the algorithm Multistep-Apply is less efficient than an alternative (more complex) version of the algorithm. Experimenting with fast graph transformation will give us an opportunity to optimise more fully, on the basis of the sound theoretical results described in this thesis.

12.2.2 Relationship with special reduction systems

Section 5.3 examines in some detail the relationship between our approach to fast graph transformation and special reduction systems [11, 1]. However, several questions remain unanswered. In particular, we have not determined whether SRSs that include non-terminals are less powerful than either LGRSs or RGRSs. We have conjectured that this is the case for both LGRSs and RGRSs (end of §5.3). Proving or disproving this conjecture and further investigating the relationship between fast recognition systems and other approaches should be the subject of further work.

12.2.3 Practical application of the correspondence

At present the correspondence result between separation logic and hyperedge replacement proved at the end of Part III are of mostly theoretical interest. However, one aim in developing this correspondence is to apply results from the graph transformation domain to the work on separation logic (and *vice versa*). In future work we should try to develop more practical applications of our correspondence. We have shown that our fragment of separation logic is closely related to the symbolic heaps used in recent shape-checking work (see §8.3.2). The shape-checking tools based on symbolic heaps are therefore an obvious target for this work.

12.2.4 Inference between graph grammars

A large part of the recent success of separation logic for shape-checking has been that it is natural to talk about inference between logical formulas. This means rewriting of formulas can be reasoned about naturally. Graph grammars in contrast have largely been studied as fixed objects, without any notion of properties under mutation. It would be interesting to try to develop some notion of inference for graph grammars. Such work could begin by examining the proof systems developed for symbolic heaps [8] and apply them to graph grammars using our mapping to hyperedge replacement.

12.2.5 Separation logic and context-sensitive graph transformation

We have conjectured in §8.1.4 that a fragment of separation logic that includes both let-statements and the separating conjunction $-\!\!*$ could simulate grammars based on context-sensitive graph transformation. This raises several interesting questions. To begin with, can we simulate full DPO grammars using this approach, or only somewhere between DPO and hyperedge-replacement grammars? Also, can we construct a mapping from separation logic formulas with separating implication to the class of DPO grammars over heap-graphs? Solving this problem would provide interesting insights into full separation logic, as double-pushout graph rewriting is very well understood formally.

12.2.6 Implementation and optimisation of CGRS

CGRS currently lacks an implementation. See §10.7 for a discussion of the possibilities for implementing and optimising CGRS in future work.

12.2.7 Reducing the distance from CGRS to C

In designing CGRS we have tried to conform to the C programming model (see §9.2.1). However, both the structure and semantics of transformer functions still differs substantially from normal C programs. To program in CGRS requires an understanding of context-sensitive graph transformation rules. This substantially increases the cost to programmers of using the language. It should be the subject of further work to make GRS-based checking more acceptable to C programmers, by matching more closely their expectations. The approach of assigning a graph-transformation semantics to C-like constructs (prototyped in the Pasta language [72]) may be a productive direction to pursue.

12.2.8 Improving the modularity of shape declarations

CGRS lacks modularity. Shape declarations have to be included in full in every program, and there is no mechanism for combining shapes. In future work we should make CGRS more modular. The shape of a data structure should be abstracted from the type of data stored in it. This would allow reuse of shapes between programs, and so would allow the development of shape-safe libraries. Hiding transformer declarations in libraries would also make CGRS more acceptable to programmers unfamiliar with graph transformation.

A more complex form of modularity would be definition of shapes by composition of existing shapes. Currently such composed shapes (e.g. lists of trees) have to be constructed by hand. However, in principle such languages could be defined as the composition of existing graph transformation systems. To compose languages in this way would require us to develop a more composable notion of a graph reduction system. Shape composition may also provide a more compositional approach to shape verification, if each composed shape can be verified separately.

12.2.9 Applying fast graph transformation to CGRS

At the moment, shape-safety checking in CGRS is performed statically. As we have shown, in general it is expensive to check the properties specified by a GRS by reduction. However, linear LGRSs and linear RGRSs have a linear-time membership checking algorithm. This could be used to provide an efficient run-time shape safety checker for CGRS. This would run periodically to test whether shape safety properties have been preserved. Such a system would be natural complement to the existing static checking algorithm. When developing a program, run-time checking could be used for debugging. Once the program has been developed, the static checking algorithm (or other proof techniques, see §11.4) could then be used to verify that the program is guaranteed shape-safe.

12.2.10 Other shape-checking approaches

CGRS adds two new notions to C: transformer functions for defining rewrites, and shape specifications for defining graph properties. These two are somewhat orthogonal: transformers could be used by a program without shape restrictions. It would be interesting to apply a different approach to shape specification to CGRS. In §11.4 we discussed two other approaches for defining the properties of graph transformation rules: high-level conditions and abstract graph transformation. To apply these approaches to CGRS, the syntax of shapes would need to be changed to reflect the different annotation methods. The semantics of shape structures might also have to change. We expect however, that a large amount of our work could be reused CGRS and the new languages based on these approaches.

Appendix A

Balanced binary trees are not MS-expressible

The proof given in this appendix show that the language of balanced binary trees, as defined in Chapter 4, cannot be defined by any formula in monadic second-order logic, as defined in [17]. This proof is based on Courcelle's proof that reachability is not MS-expressible (Proposition 5.2.9 of [17]).

Enough background on monadic second-order logic is given here so that the proof should be understandable on its own, but giving complete definition for every construct would require an unreasonably large amount of space. Consequently definitions in this appendix are somewhat terser than elsewhere in the thesis, and the reader unfamiliar with monadic second-order logic may find it useful to refer to [17].

Formulas in monadic second-order logic define the properties of relational structures. A structure $S = \langle D_S, (R_S)_{R \in \mathcal{R}} \rangle$ defined over a set of relation symbols $\mathcal{R}$ consists of a domain D_S and a relation R_S over D_S for each $R \in \mathcal{R}$. In the results below, if u is a string, then we write $\parallel u \parallel$ to stand for the string structure. The relation symbols are suc_S, defining the sequence of positions, and an i_{yS} defining the character held at each position in the string. More formally,

$$\begin{array}{rcl} D_S & = & \{1, \ldots, n\} \text{ if } u \text{ has length } n; \\ suc_S & = & \{(1,2), (2,3), \ldots, (n-1, n)\}; \\ i \in lab_{yS} & & \text{iff } y \text{ is the } i\text{th letter of } u. \end{array}$$

We first prove the result that balanced binary trees are not MS_1-definable. MS_1 is the class of monadic second-order logic formulas without an incidence predicate, that is, a predicate expressing the fact that a particular edge is incident to a particular node. The proof depends on the following result, which Courcelle attributes to Büchi and Elgot [17].

Theorem A.1. *If $L \subseteq \{a, b\}^*$ is the set of words u such that $\| u \| \models \varrho$ where ϱ is a closed MS_1-formula, then L is a regular language. (Büchi & Elgot)*

Proposition A.2. *No formula exists in MS_1 that defines the class of balanced binary trees.*

Proof. Assume we have an MS_1-formula β that is satisfied by a graph iff it is a balanced binary tree.

Consider the language of strings of the form $\mathrm{a}^n\mathrm{c}\mathrm{b}^m$. We associate each string $w_{n,m} = \mathrm{a}^n\mathrm{b}^m$ with a graph $B_{n,m}$ with vertices $\{-n, \ldots, -1, 0, 1, \ldots, m\}$. The left-hand 'a' characters associate with vertices $\{-n, \ldots, -1\}$, and the right-hand 'b' characters associate with vertices $\{0, 1, \ldots, m\}$. There is an edge in $B_{n,m}$ from vertex 0 to vertices 1 and -1, and a vertex from each vertex greater than zero to each succeeding vertex, and from each vertex less than zero to the preceding vertex. A graph $B_{n,m}$ is a balanced binary tree iff $m = n + 1$.

The formula η that encodes in terms of a string structure the edge relation for a balanced binary tree:

$$\begin{aligned}\eta(x_1, x_2) \quad = \quad & ((lab_a(x_1) \wedge lab_a(x_2) \wedge suc(x_2, x_1)) \\ & \vee (lab_b(x_1) \wedge lab_b(x_2) \wedge suc(x_1, x_2)) \\ & \vee (lab_b(x_1) \wedge lab_a(x_2) \wedge suc(x_2, x_1))\end{aligned}$$

The formula η encodes the edge relation for the graph $B_{n,m}$ in terms of a string $w_{n,m}$. We can therefore construct the formula $\beta[\eta/edg]$ that is satisfied if and only if a string has the form $\mathrm{a}^n\mathrm{b}^{n+1}$.

$$B_{n,m} \models \beta \text{ iff } m = n + 1 \text{ iff } \| w_{n,m} \| \models \beta[\eta/edg]$$

But this results in a contradiction. The language of strings $w_{n,(n+1)}$ that satisfying $\beta[\eta/edg]$ is non-regular, contradicting Theorem A.1. Our initial assumption of the existence of β must therefore be false. □

Remark A.1. The above proof is for balanced binary trees constructed from unlabelled nodes and edges. However, we can extend the same proof so as to prove the result for balanced binary tree graphs with labelled nodes and edges. We simply replace the η formula with separate formulas for distinct edge labels, and add node-labelling functions.

We have proved that no formula for balanced binary trees exists in MS_1, the class of MSOL formulas without an incidence predicate. Courcelle shows in [17] that MS_2, the class of monadic second-order logic formulas with such an incidence predicate, is formally more expressive than MS_1. However, he also gives the following result, which allows us to apply Proposition A.2 to MS_2 formulas.

Lemma A.3 (MS_1 and MS_2 expressiveness)**.** *Let C be a class of: (1) planar directed simple graphs, or (2) directed simple graphs of degree at most k, or (3) finite directed simple graphs of tree-width at most k. A property of graphs in C is MS_2-expressible iff it is MS_1-expressible (Courcelle, [17, p.338]).*

Proposition A.4. *No formula exists in MS_2 that defines the class of balanced binary trees.*

Proof. The class of balanced binary trees satisfies all three cases for the applicability of lemma A.3, so as a consequence of Proposition A.2 balanced binary trees are also not MS_2-expressible. $\square$

Bibliography

[1] S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese. An algebraic theory of graph reduction. *Journal of the ACM*, 40(5):1134–1164, 1993. (Cited on page 19, 58, 59, 99, 102, 103, 104, 105, 260)

[2] K. Azab and A. Habel. High-level programs and program conditions. In H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, editors, *ICGT*, volume 5214 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2008. (Cited on page 252)

[3] A. Bakewell, D. Plump, and C. Runciman. Checking the shape safety of pointer manipulations. In *7th International Seminar on Relational Methods in Computer Science, Revised Selected Papers*, volume 3051 of *Lecture Notes in Computer Science*, pages 48–61. Springer, 2004. (Cited on page 17, 18, 177, 181, 182, 205, 243, 250, 259)

[4] A. Bakewell, D. Plump, and C. Runciman. Specifying pointer structures by graph reduction. In *Int. Workshop Applications of Graph Transformations With Industrial Relevance (AGTIVE 2003), Revised Selected and Invited Papers*, volume 3062 of *Lecture Notes in Computer Science*, pages 30–44, 2004. (Cited on page 17, 18, 66, 68, 77, 84, 93, 94, 97, 104, 177, 179, 186, 194, 246, 259)

[5] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *International Conference on Computer-Aided Verification*, pages 178–192, 2007. (Cited on page 173, 251)

[6] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and*

Systems, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68, 2005. (Cited on page 111)

[7] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005. (Cited on page 248)

[8] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *In Proceedings of FMCO'05*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005. (Cited on page 261)

[9] L. Birkedal, N. Torp-Smith, and J. Reynolds. Local reasoning about a copying garbage collector. In *Proceedings of the 31st ACM symposium on Principles of Programming Languages*, pages 1–58, 2004. (Cited on page 248)

[10] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993. (Cited on page 103)

[11] H. L. Bodlaender and B. van Antwerpen-de Fluiter. Reduction algorithms for graphs of small treewidth. *Inf. Comput.*, 167(2):86–119, 2001. (Cited on page 59, 99, 101, 102, 103, 104, 260)

[12] I. Bogudlov, T. Lev-Ami, T. W. Reps, and M. Sagiv. Revamping tvla: Making parametric shape analysis competitive. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 221–225. Springer, 2007. (Cited on page 250, 251)

[13] I. B. Boneva, A. Rensink, M. E. Kurban, and J. Bauer. Graph abstraction and abstract graph transformation. Technical Report TR-CTIT-07-50, University of Twente, Enschede, July 2007. (Cited on page 252, 253)

[14] M. Bozga, R. Iosif, and Y. Lakhnech. Storeless semantics and alias logic. In *PEPM*, pages 55–65. ACM, 2003. (Cited on page 248)

[15] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic.

In *13th International Symposium on Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 182–203, 2006. (Cited on page 173)

[16] C. Calcagno, P. Gardner, and 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*, pages 123–134, New York, NY, USA, 2007. ACM. (Cited on page 175)

[17] B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 313–400. World Scientific, 1997. (Cited on page 103, 264, 265, 266)

[18] R. Diestel. *Graph theory.* Springer, 2000. (Cited on page 104)

[19] D. Distefano, J.-P. Katoen, and A. Rensink. Who is pointing when to whom? In K. Lodaya and M. Mahajan, editors, *FSTTCS*, volume 3328 of *Lecture Notes in Computer Science*, pages 250–262. Springer, 2004. (Cited on page 249)

[20] D. Distefano, P. W. O'Hearn, and 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*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006. (Cited on page 17, 111, 173)

[21] M. Dodds. From separation logic to hyperedge replacement and back (extended abstract). In *Proc. International Conference on Graph Transformation (ICGT 2008)*, volume 5214/2008 of *Lecture Notes in Computer Science*, pages 484–486. Springer, 2008. (Cited on page 22)

[22] M. Dodds and D. Plump. Extending C for checking shape safety. In *Proc. Graph Transformation for Verification and Concurrency (GT-VC 2005)*, volume 154(2) of *Electronic Notes in Theoretical Computer Science*, pages 95–112. Elsevier, 2006. (Cited on page 22)

[23] M. Dodds and D. Plump. Graph transformation in constant time. In *Proc. International Conference on Graph Transformation (ICGT*

2006), volume 4178 of *Lecture Notes in Computer Science*, pages 367–382. Springer, 2006. (Cited on page 22)

[24] M. Dodds and D. Plump. From separation logic to hyperedge replacement and back. In *Proc. Doctoral Symposium at the International Conference on Graph Transformation*, volume 16 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, 2008. (Cited on page 22)

[25] H. Dörr. Bypass strong V-structures and find an isomorphic labelled subgraph in linear time. In W. Mayr, Ernst, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 305–318, 1995. (Cited on page 18, 100)

[26] H. Dörr. *Efficient Graph Rewriting and its Implementation*, volume 922 of *Lecture Notes in Computer Science*. Springer, 1995. (Cited on page 18, 68, 100, 101)

[27] F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Volume I: Foundations*, chapter 2, pages 95–162. World Scientific, 1997. (Cited on page 29, 31, 66, 76, 131, 135, 144, 148, 149, 166, 168)

[28] J. Engelfriet. Context-free graph grammars. In *Handbook of formal languages, vol. 3: beyond words*, pages 125–213. Springer, New York, NY, USA, 1997. (Cited on page 174)

[29] J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Volume I: Foundations*, chapter 1, pages 1–94. World Scientific, 1997. (Cited on page 76)

[30] C. Forgy et al. RETE: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982. (Cited on page 102)

[31] P. Fradet and D. L. Métayer. Shape types. In *Proceedings of the 1997 ACM Symposium on Principles of Programming Languages*, pages 27–39. ACM Press, 1997. (Cited on page 20, 182, 242, 245)

[32] P. Fradet and D. L. Métayer. Structured gamma. *Science of Computer Programming*, 31(2-3):263–289, 1998. (Cited on page 182)

[33] J. J. Fu. Linear matching-time algorithm for the directed graph isomorphism problem. In *Proceedings of the 6th International Symposium on Algorithms*, volume 1004 of *Lecture Notes in Computer Science*, pages 409–417. Springer, 1995. (Cited on page 100)

[34] J. J. Fu. Pattern matching in directed graphs. In *Proc. Combinatorial Pattern Matching*, volume 937 of *Lecture Notes in Computer Science*, pages 64–77. Springer, 1995. (Cited on page 100)

[35] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, January 1979. (Cited on page 39, 73)

[36] R. Geis, G. V. Batz, D. Grund, S. Hack, and A. M. Szalkowski. Grgen: A fast spo-based graph rewriting tool. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Proc. International Conference on Graph Transformation (ICGT 2006)*, Lecture Notes in Computer Science, pages 383 – 397. Springer, 2006. (Cited on page 99, 100)

[37] Y. Gurevich and J. K. Huggins. The semantics of the c programming language. In *Selected Papers from CSL '92*, volume 702 of *Lecture Notes in Computer Science*, pages 274–308. Springer, 1993. (Cited on page 222)

[38] A. Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer, 1992. (Cited on page 29, 31, 132, 149, 167, 172, 173)

[39] A. Habel and H.-J. Kreowski. Filtering hyperedge-replacement through compatible properties. In M. Nagl, editor, *WG*, volume 411 of *Lecture Notes in Computer Science*, pages 107–120. Springer, 1989. (Cited on page 115, 165)

[40] A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical. Structures in Comp. Sci.*, 11(5):637–688, 2001. (Cited on page 185)

[41] A. Habel and K.-H. Pennemann. Satisfiability of high-level conditions. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 430–444. Springer, 2006. (Cited on page 252)

[42] A. Habel and D. Plump. Computational completeness of programming languages based on graph transformation. In F. Honsell and M. Miculan, editors, *FoSSaCS*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2001. (Cited on page 252)

[43] A. Habel and D. Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2002. (Cited on page 26, 27)

[44] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. MONA: Monadic second-order logic in practice. In E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110. Springer, 1995. (Cited on page 247)

[45] International Organization for Standardization. ISO C standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft. (Cited on page 209)

[46] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 2001 ACM Symposium on Principles of Programming Languages*, volume 36(3) of *ACM SIGPLAN Notices*. ACM, March 2001. (Cited on page 111, 247)

[47] B. Jeannet, A. Loginov, T. W. Reps, and S. Sagiv. A relational approach to interprocedural shape analysis. In R. Giacobazzi, editor, *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 246–264. Springer, 2004. (Cited on page 250)

[48] B. W. Kernighan and D. M. Ritchie. *C Programming Language (2nd Edition)*. Prentice Hall, 1988. (Cited on page 222)

[49] N. Klarlund and M. I. Schwartzbach. Graph types. In *Proceedings of the 1993 ACM Symposium on Principles of Programming Languages*, pages 196–205. ACM, 1993. (Cited on page 246)

[50] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973. (Cited on page 193)

[51] K.-J. Lange and E. Welzl. String grammars with disconnecting. In *Fundamentals of Computation Theory*, volume 199 of *Lecture Notes in Computer Science*, pages 249–256. Springer, 1985. (Cited on page 76)

[52] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *Proceedings of the 14th European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 124–140. Springer, April 2005. (Cited on page 111, 113, 117, 118, 119, 120, 121, 122, 165, 175)

[53] T. Lev-Ami, R. Manevich, and S. Sagiv. Tvla: A system for generating abstract interpreters. In R. Jacquart, editor, *IFIP Congress Topical Sessions*, pages 367–376. Kluwer, 2004. (Cited on page 250)

[54] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 280–301. Springer, 2000. (Cited on page 249, 251)

[55] T. Lev-Ami and S. Sagiv. Tvla: A system for implementing static analyses. In J. Palsberg, editor, *SAS*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer, 2000. (Cited on page 250)

[56] 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. (Cited on page 175)

[57] G. Manning and D. Plump. The York abstract machine. In R. Bruni and D. Várro, editors, *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2006), Vienna, Austria, April 1–2, 2006*, volume 211 of *Electronic Notes in Theoretical Computer Science*, pages 231–240. Elsevier, 2006. (Cited on page 37, 99, 260)

[58] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the 2001 ACM Conference on Programming Language*

Design and Implementation, volume 36(5) of *SIGPLAN Notices*. ACM, 2001. (Cited on page 246)

[59] J. Müller and R. Geis. Speeding up graph transformation through automatic concatenation of rewrite rules. Technical report, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, 2007. (Cited on page 99, 100)

[60] H. R. Neilson and F. Nielson. *Semantics with Applications: An Appetizer*. Wiley Professional Computing, 2005. (Cited on page 223, 225)

[61] M. Norrish. An abstract dynamic semantics for C. Technical report, Computer Laboratory, University of Cambridge, Sept. 27 1997. (Cited on page 222)

[62] Object Management Group. Unified Modeling Language (UML) Specification, version 2.0, 2009. (Cited on page 33)

[63] D. Plump. Hypergraph rewriting: critical pairs and undecidability of confluence. In M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors, *Term graph rewriting: theory and practice*, pages 201–213. Wiley, Chichester, UK, UK, 1993. (Cited on page 72, 80, 87)

[64] D. Plump. Confluence of graph transformation revisited. In A. Middeldorp, V. van Oostrom, F. van Raamsdonk, and R. C. de Vrijer, editors, *Processes, Terms and Cycles*, volume 3838 of *Lecture Notes in Computer Science*, pages 280–308. Springer, 2005. (Cited on page 72)

[65] D. Plump and S. Steinert. Towards graph programs for graph algorithms. In *Proc. International Conference on Graph Transformation (ICGT 2004)*, volume 3256 of *Lecture Notes in Computer Science*, pages 128–143. Springer, 2004. (Cited on page 37, 184)

[66] A. Rensink. GROOVE: A graph transformation tool set for the simulation and analysis of graph grammars. Available at `http://www.cs.utwente.nl/~groove`, 2003. (Cited on page 37)

[67] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of

Lecture Notes in Computer Science, pages 479–485. Springer-Verlag, 2004. (Cited on page 37)

[68] A. Rensink and D. Distefano. Abstract graph transformation. *Electr. Notes Theor. Comput. Sci.*, 157(1):39–59, 2006. (Cited on page 33, 252, 253)

[69] 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. (Cited on page 14, 17, 111, 112, 113, 114, 247)

[70] N. Rinetzky and S. Sagiv. Interprocedural shape analysis for recursive programs. In R. Wilhelm, editor, *CC*, volume 2027 of *Lecture Notes in Computer Science*, pages 133–149. Springer, 2001. (Cited on page 250)

[71] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation.* World Scientific, 1996. (Cited on page 15)

[72] C. Runciman. Pasta shell: revision and first implementation. SPGT project memo, Department of Computer Science, University of York, 2002. (Cited on page 262)

[73] AGG development team. *The* AGG *1.4.0 Development Environment: The User Manual.* Technische Universität Berlin. `http://tfs.cs.tu-berlin.de/agg`. (Cited on page 33)

[74] É.-J. Sims. Extending separation logic with fixpoints and postponed substitution. *Theoretical Computer Science*, 351(2):258–275, 2006. (Cited on page 111, 113, 117, 118, 165)

[75] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955. (Cited on page 120)

[76] G. Varró, A. Schürr, and D. Varró. Benchmarking for graph transformation. In *Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 79–88, Dallas, Texas, USA, September 2005. (Cited on page 260)

[77] G. Varró and D. Varró. Graph Transformation with Incremental Updates. *Electronic Notes in Theoretical Computer Science*, 109:71–83, 2004. (Cited on page 102, 103)

[78] H. Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *Proceedings of the 1st Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, Jan 2001. Informal proceedings. (Cited on page 248)

[79] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *International Conference on Computer-Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008. (Cited on page 173, 251)