

The University of Edinburgh



Department of Computer Science

**Generating Program Animators
from
Programming Language Semantics
by
Dave Berry**

CST-79-91

Thesis

(also published as ECS-LFCS-91-163)

**James Clerk Maxwell Building,
The King's Buildings,
Mayfield Road,
Edinburgh,
EH9 3JZ.**

June, 1991

Generating Program Animators
from Programming Language Semantics

Dave Berry

Ph. D.

University of Edinburgh

1991

(Graduation date July 1991)

Abstract

I present a theory of program animation based on formal semantics. This theory shows how an animator for a language can be generated from a formal specification of that language. Such an animator presents a model of evaluation that is formally correct with respect to the semantics. The theory also provides a framework for comparing definitions of animation operations.

The main part of the theory is the definition of an evaluation step. I compare two definitions. The first is based on the transitions used in the transitional style of structured operational semantics, and is motivated by the idea that these transitions represent an intuitive idea of a computation step. Unfortunately this definition produces unsatisfactory animations. However, it can be augmented to give one that better satisfies the needs of the user.

Both of these definitions are given in the relational style of structured operational semantics. The first definition is based on an equivalence between the relational and transitional styles; I give a definition of this equivalence. I also discuss the relation between the definition of a step and the choice of semantic formalism.

Views of a program in mid-evaluation can be defined by extending the specification of the programming language to include semantic display rules. Each semantic display rule specifies the display of one sub-phrase of the program in mid-evaluation. This approach is powerful enough to define a wide range of views. I also show how the definition of a step can be parameterised on a view.

More advanced operations can also be defined in terms of this theory. These operations and the views mentioned in the previous paragraph cover most of the

features found in existing animators. This indicates that the theory is powerful enough to specify useful systems. The main feature that is not yet provided is the ability to define views that are specific to a single program.

These ideas have been implemented in a system called The Animator Generator. Animators produced by The Animator Generator support multiple views and the advanced operations mentioned here. I give a brief description of this system. I finish by discussing how this work could be developed further.

Acknowledgements

I would like to thank Robin Milner, my supervisor, for his encouragement and advice.

I have been fortunate in being able to study at the Laboratory for Foundations of Computer Science (LFCS), which is a stimulating and congenial place to work. I would particularly like to thank Rod Burstall and Kevin Mitchell for several interesting discussions.

I would also like to thank Paul Brna of the Department of Artificial Intelligence for reading and commenting on the first part of this thesis.

Gilles Kahn and his group at INRIA made several helpful comments and made my visit to Sophia-Antipolis very enjoyable.

Alex Zbyslaw and Diana Bental proof-read several parts of this thesis. My thanks go to them. Any remaining errors are entirely my responsibility.

Diana also put up with me during the hard times, for which I am deeply grateful.

The first three years of my work on this thesis were funded by the Science and Engineering Research Council. I am also extremely grateful for the time that the LFCS has granted me to finish my thesis during the last three months. Finally, my thanks go to both these bodies for funding my visit to INRIA.

Declaration

All the work in this thesis is original.

A handwritten signature in cursive script that reads "Dave Berry". The letters are fluidly connected, with a prominent loop at the start of the first name and a similar flourish for the last name.

Dave Berry

July 1991

Table of Contents

1. Introduction	1
1.1 Program Animation	2
1.2 Generating Program Animators	5
1.3 A Theory of Program Animation	5
1.4 The Animator Generator	6
1.5 Structure of the Thesis	7
1.6 A Note on Terminology	9
1.7 Chapter Summary	10
2. Background: Animation	11
2.1 Task 1: Learning a New Language	12
2.2 Task 2: Understanding Someone Else's Programs	15
2.2.1 Method 1: Presenting the Structure of the Program	15
2.2.2 Method 2: Viewing the Data	16
2.2.3 Summary	18
2.3 Task 3: Debugging	19
2.3.1 Brna's Model Of Debugging	19

2.3.2	Views for Debugging	20
2.3.3	Animation Techniques for Debugging	22
2.3.4	Summary	24
2.4	Task 4: Learning to Program	24
2.4.1	Translating a Solution	25
2.4.2	Learning the Programming Language	26
2.4.3	Acquiring Program Schemas	26
2.4.4	Summary	26
2.5	Chapter Summary	27
3.	Background: Generating Animators	28
3.1	Task 1: Producing New Animators	29
3.2	Task 2: Designing New Languages	29
3.3	Task 3: Learning About Semantics	30
3.4	A Theory of Animation	32
3.4.1	Defining A Step	33
3.4.2	Defining A View	34
3.4.3	View-Specific Steps	35
3.4.4	Summary	36
3.5	Assessing the Theory	37
3.6	Previous Work	38
3.7	Chapter Summary	41

4. Structured Operational Semantics	43
4.1 Discussion	44
4.1.1 Operational Semantics	44
4.1.2 Semantic Rules	45
4.1.3 Transitional and Relational Semantics	47
4.1.4 Environments and Memories	50
4.1.5 Summary	51
4.2 Notation and Meta-notation	51
4.2.1 Abstract Syntax	52
4.2.2 Objects	53
4.2.3 Evaluation States	55
4.2.4 Semantic Rules	55
4.2.5 Rule Forms	57
4.2.6 Substitutions and Instances of Rules	58
4.2.7 Rule Sets	60
4.2.8 Side Conditions	62
4.2.9 Summary	63
4.3 Chapter Summary	63
5. Transitional Evaluations	65
5.1 Transitional Rules	67
5.1.1 Forms of Transitional Rules	67
5.1.2 Determinacy Restriction	70

5.1.3	Examples	71
5.1.4	Comparison with Plotkin's Rules	73
5.1.5	Comparison with Milner's Process Calculus	76
5.1.6	Summary	77
5.2	Transitional Evaluation Histories	78
5.2.1	Consistent Stacks	79
5.2.2	Consistent Sequences	80
5.2.3	Definition of Evaluation Histories	81
5.2.4	Initial and Complete Evaluations	81
5.2.5	Summary	83
5.3	Transitional Computation Steps	83
5.3.1	Definition	83
5.3.2	Example	83
5.4	Chapter Summary	86
6.	Relational Evaluations	87
6.1	Relational Rules	88
6.1.1	Form of Relational Rules	88
6.1.2	Evaluation Model	89
6.1.3	Determinacy Restriction	90
6.1.4	Examples	91
6.1.5	Comparison with the Definition of Standard ML	93
6.1.6	Summary	95

6.2	Relational Evaluation Histories	95
6.2.1	Definition	98
6.2.2	Initial and Complete Evaluations	100
6.2.3	Summary	102
6.3	Relational Computation Steps	102
6.3.1	Extension	102
6.3.2	Definition	103
6.3.3	Example	103
6.3.4	Congruency	104
6.3.5	Summary	108
6.4	Chapter Summary	108
7.	A Definition of Equivalence	109
7.1	Equivalent Syntaxes	110
7.2	Transitional Realisation of Relational Rule Sets	111
7.2.1	Discussion	111
7.2.2	Definition	113
7.2.3	Examples	115
7.2.4	Recursive Premises Revisited	117
7.2.5	Trivial Premises	118
7.2.6	Summary	119
7.3	Relational Realisation of Transitional Rule Sets	119
7.3.1	Partially Consistent Stacks	120

7.3.2	Sequences as Instances of Stacks	121
7.3.3	Transitional Trees	123
7.3.4	Definition of Relational Realisation	126
7.3.5	Summary	128
7.4	Equivalent Evaluations	129
7.4.1	Definition	130
7.4.2	Summary	135
7.5	Computation Steps Preserve Equivalence	136
7.5.1	The Proof of the Proposition	139
7.6	Chapter Summary	145
8.	Relational Animations	146
8.1	Computation Steps in Animations	147
8.2	Relational Animation Histories	151
8.2.1	Definition	153
8.2.2	Initial and Complete Animations	156
8.2.3	Summary	156
8.3	Relational Animation Steps	157
8.3.1	Extension	157
8.3.2	Definition	158
8.3.3	Example	158
8.3.4	Congruence	160
8.3.5	Summary	161

8.4	Choice of Formalism	161
8.5	Chapter Summary	163
9.	Views	165
9.1	Static Source Views	167
9.1.1	Lexical Proximity: Lexical Views	167
9.1.2	Lexical Proximity: Outline Views	169
9.1.3	Structural Proximity: Static Call Graphs	171
9.1.4	Referential Proximity: Static Slices	173
9.1.5	Summary	176
9.2	Static Environment Views	176
9.3	Dynamic Views	178
9.3.1	Lexical Proximity: Simple Lexical Views	179
9.3.2	Lexical Proximity: More Complicated Examples	181
9.3.3	Temporal Proximity: Traces	185
9.3.4	Referential Proximity: Dynamic Slices	185
9.3.5	Structural Proximity: Dynamic Call Graphs	186
9.3.6	Structural Proximity: Procedure Call Stacks	187
9.3.7	Hybrid Source Views	188
9.3.8	Summary	188
9.4	Dynamic Environment Views	189
9.5	Memory Views	190
9.6	Semantic Views	192

9.7	Abbreviations	193
9.8	Chapter Summary	196
10.	Operations	197
10.1	View-specific Steps	198
10.1.1	Display Attributes	199
10.1.2	View Sets	200
10.1.3	Definition	201
10.1.4	Highlighting Multiple Views	202
10.1.5	Discussion	203
10.1.6	Summary	204
10.2	Breaks, Marks and Interrupts	204
10.2.1	Break Points	205
10.2.2	Marks	206
10.2.3	Break Conditions	206
10.2.4	Interrupts	208
10.2.5	Break Levels	208
10.2.6	Summary	209
10.3	Other Advanced Operations	209
10.3.1	Phrase Evaluation	209
10.3.2	Reverse Evaluation	211
10.3.3	Remote Rule Sets	213
10.3.4	Changing an Evaluation	214

10.3.5 Summary	217
10.4 Chapter Summary	217
11.An Example Language: Prolog	219
11.1 Review	219
11.2 Discussion	221
11.3 Semantics	223
11.4 An Example Animation	227
11.5 Display Rules	234
11.6 Chapter Summary	238
12.The Animator Generator	239
12.1 Review of the Theory	239
12.2 Designing the Implementation	241
12.3 LSL	244
12.3.1 Semantic Rules	244
12.3.2 Views	247
12.3.3 Summary	249
12.4 Using an Animator	249
12.4.1 Getting Started	250
12.4.2 Command Buttons	251
12.4.3 Menu Buttons	251
12.4.4 Advanced Features	254
12.4.5 Summary	255

12.5 Implementation	255
12.5.1 The Generator	255
12.5.2 Operations	257
12.5.3 Views	261
12.5.4 Summary	263
12.6 Shortcomings	263
12.6.1 Operations	263
12.6.2 Views	264
12.6.3 The User Interface	265
12.6.4 Summary	265
12.7 Chapter Summary	266
13.Future Work	267
13.1 Other Languages	267
13.2 Semantics	269
13.2.1 Removing Recursive Premises	269
13.2.2 Adding Predicate Premises	270
13.2.3 Non-determinism	270
13.2.4 Using Variables to Represent Identifiers	272
13.2.5 Unification	272
13.2.6 Other Formalisms	273
13.2.7 Summary	273
13.3 Views	274

13.3.1 Graphics	274
13.3.2 Pattern Matching	274
13.3.3 New Display Attributes	275
13.3.4 Abbreviations	277
13.3.5 Language Independence	277
13.3.6 Program-Specific Views	278
13.3.7 Summary	279
13.4 Operations	279
13.4.1 Defining an Animation Step	279
13.4.2 Changing the Current Environment	282
13.4.3 Input and Output	283
13.4.4 Bug Location Algorithms	283
13.4.5 Symbolic Evaluation	284
13.4.6 Summary	284
13.5 Implementation	285
13.5.1 Including Static Semantics	285
13.5.2 Concrete Syntax	286
13.5.3 Efficiency	286
13.5.4 Programming Environments	287
13.5.5 Summary	288
13.6 Chapter Summary	289

14.Review and Conclusion	290
14.1 Animation	290
14.2 Generating Animators	291
14.3 Defining a Step	292
14.4 Specifying Views	293
14.5 Advanced Operations	294
14.6 Implementation	295
14.7 Assessing the Theory	295
14.8 Conclusion	297
 A. Equivalent Animations	 298
A.1 Transitional Animation Histories	299
A.1.1 Auxiliary Definitions	301
A.1.2 Definition of Animation Histories	302
A.1.3 Initial and Complete Animations	303
A.1.4 Transitional Trees Revisited	304
A.1.5 Summary	304
A.2 Transitional Animation Steps	305
A.2.1 Definition	305
A.2.2 Example	305
A.2.3 Summary	310
A.3 Equivalent Animations	310
A.3.1 Definition	311

A.3.2 Summary	313
A.4 Step Functions Preserve Equivalence	313
A.5 Appendix Summary	315
B. A Guide to the Notation	316
B.1 Terms and Metavariables	316
B.2 Substitution	317
B.3 Sequents and Rules	317
B.4 Stacks and Sequences	319
B.5 Relational Evaluations and Animations	320
B.6 Equivalence	321
B.7 Animation Operations	322
B.8 LSL Display Expressions	323
C. The Specification of Proc	324
C.1 The Header	324
C.2 The LOOKUP_ENV Rule Set	325
C.3 The LOOKUP_MEMORY Rule Set	327
C.4 The PROC Rule Set	328
C.5 Constructor Display Rules	336
C.6 Semantic Display Rules	338
Bibliography	344
Index of Lemmas and Propositions	356

Chapter 1

Introduction

For almost as long as there have been computers for people to program, people have produced tools to show themselves what their programs actually do. These systems have helped people to debug their programs and to understand what other people's programs do. Similar systems have been used to teach people new programming languages and the basic concepts of programming.

The first debugging tools gave a murky view of a program, usually by letting people print the contents of specific memory locations (in hexadecimal). Nowadays most debuggers for imperative languages display the values of variables instead of memory locations, and can display them in a variety of formats. Modern debuggers also support a variety of stepping commands, breakpoints and other facilities. Debuggers for logic languages and functional languages provide similar (and often more advanced) facilities.

A related trend has been the development of animated programs for teaching. Several early projects used films; nowadays the animations are usually produced on computers. Again the sophistication of views and commands have grown.

These trends have converged to the point where they can be considered to be different aspects of the subject of **program animation**. Research institutes are producing debuggers that present views as complex as most of the teaching

materials, and teaching systems are providing a range of commands that match the most sophisticated debuggers.

In this thesis I marry this subject with the field of language independent programming environments. Several people have produced structure editor generators and compiler generators; I extend this approach to **animator generators**. An animator generator provides similar advantages to other sorts of generators, including reuse of code and the quick development of systems that share a common user interface across a range of languages.

The development of an animator generator poses interesting semantic questions. The most important of these is how define an evaluation step. Nearly as important are the questions of how to specify the display of a program in mid-evaluation and how to define more advanced operations.

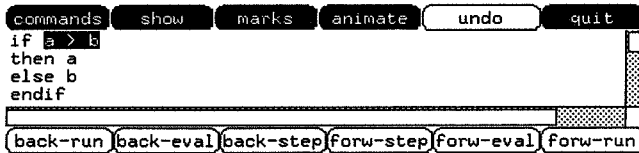
I consider the generation of animators for imperative, eager applicative and logic languages, but not for non-deterministic or concurrent languages.

1.1 Program Animation

A **program animator** evaluates a program one step at a time. Each step changes the display. Thus the evaluation of the program is shown as an animated sequence of displays, just as a film consists of a series of frames. Compared to a film, a program animator has the advantage that the user has more control over the order and speed of the displays.

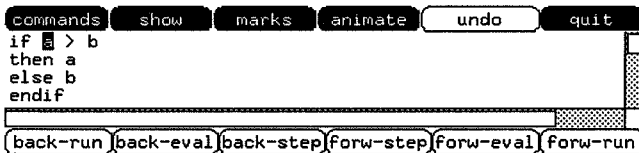
The representation of the program source code and the current environment and memory is called a **view** of the program. An animator may show several views, each presenting a different aspect of the structure of source or of the current state.

For example, the following sequence of displays animates the evaluation of part of a program in a simple imperative language called Proc. Each display shows a section of the program, highlighting the phrase to be evaluated next or the phrase just evaluated in reverse video. As each phrase is evaluated it is replaced by its value.

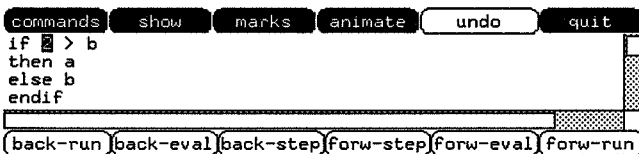


The first display shows the program to be animated. A full animation would first step through the evaluation of the condition ' $a > b$ ' and then through the evaluation of the appropriate branch. This sequence of displays shows the first part of this process.

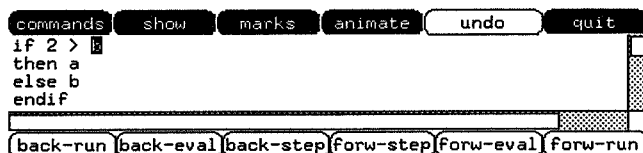
The animation starts by highlighting the first sub-phrase of the program to be evaluated:



The next display shows the result of the evaluation step. The highlighted occurrence of ' a ' is replaced by the value bound to ' a ' in the current environment. Other occurrences of ' a ' would be evaluated independently when required:



The next display shows the next phrase to be evaluated, without actually evaluating it:



The next display shows the result of the evaluation step:

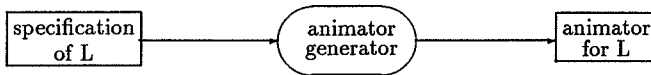


The animation of the remaining evaluation would first show the comparison produce the result 'false' and then step through the evaluation of the second branch of the conditional expression.

When people use a system, whether it be a cash point or a programming environment, they have a mental model of how the system works and how to interact with it. That model might be inaccurate or incomplete or both. A good program animator lets users compare their model of the program to its actual behaviour. This helps them perform several important tasks. Chapter 2 gives a more detailed discussion of how animators can support people performing four specific tasks.

1.2 Generating Program Animators

An **animator generator** takes a formal semantic specification of a programming language L and produces a program animator for L :



This process ensures that the model of evaluation presented by the animator is formally correct. I conjecture that if the user interface and the clarity of presentation are adequate, then an animator generator should produce optimal animators with a minimum of effort. Much work is needed to test this conjecture; this thesis is only a beginning.

I conjecture that animator generators will help people in a range of activities. Chapter 3 discusses the role of animator generators in more detail.

Existing language independent programming environments could be extended to include an animator generator as well as compiler generators and structure editor generators. This would increase the flexibility of the environments that they produce.

1.3 A Theory of Program Animation

Previous work on language independent systems has focussed on generating structure editors or compilers. Structure editor generators are based on compile-time information, ignoring dynamic semantics. Compiler generators use compile-time information too, but they also use dynamic semantics to relate input to output. An animator generator requires even more; it needs a definition

of an evaluation step, and a definition of how to display the program in mid-evaluation.

To formalise these definitions we need a theory of animation. This theory must define a semantic representation of a program in mid-evaluation, and define a step and views in terms of that representation. Such a theory also provides a framework for comparing different definitions of steps and views, and for examining different animation operations.

Chapter 3 discusses the role of a theory of animation in more detail. The bulk of the thesis develops such a theory, based on structured operational semantics. This development includes a comparison of different definitions of a step. It also includes a definition of an equivalence between two styles of structured operational semantics.

1.4 The Animator Generator

A theory of program animation needs a demonstration of its practicality. One such demonstration for the above theory is the fact that it can describe all the operations and most of the program independent views mentioned in the review of useful features in Chapter 2. Another is the implementation of an animator generator, which is described in Chapter 12.

This system is called The Animator Generator (capitalised to distinguish the name from the general idea of an animator generator). This system shows that the ideas presented in the thesis can be made into a practical tool. It allows a range of views to be defined for each language, supports several kinds of step, lets users step backwards through an evaluation, and provides some control over what is displayed. It is not production quality software, but does indicate that a production quality system would be a practical project.

The input to The Animator Generator is a specification written in a language called LSL (**L**anguage **S**pecification **L**anguage). LSL is based on structured operational semantics, extended with the technique for specifying views introduced in Chapter 10.

1.5 Structure of the Thesis

The rest of the thesis is organised as follows:

Chapter 2 explores the idea of animation in more detail. It shows how animators can support several important tasks, and introduces some features of existing animators that support particular tasks. These features are used to test that the theory developed in the thesis is sufficiently powerful to describe useful tools.

Chapter 3 discusses the roles that an animator generator could fill, and sets out the problems that need to be solved in order to build one. It also discusses the difference between this work and existing work.

Chapter 4 introduces two forms of structured operational semantics, called relational semantics and transitional semantics. These forms are used to develop the theory of animation in the following chapters. Chapter 4 also specifies the notation used throughout the thesis.

Chapters 5 to 8 are concerned with the definition of an evaluation step. The first approach produces a definition based on the transitions used in transitional semantics, which give a definition of a computation step. A computation step is the smallest amount of computation that can be performed by a program. This is shown to be inadequate for animation, and so Chapter 8 refines it to give a definition of an animation step.

The division of this presentation into two stages achieves two goals. First, it demonstrates and contrasts the methodological differences underlying the two definitions. Second, it simplifies the explanation of the final definition by refining it from a simpler one. As a bonus, it develops an equivalence between the two forms of structured operational semantics which has some interest in its own right.

Chapters 5 and 7 are concerned with transitional semantics. Since the main theory is expressed in relational semantics, readers who are primarily interested in the practical side of the work may skip these chapters.

Chapter 9 describes several possible views of a program in some detail, and shows how a specification written in relational semantics can be extended to specify these views. It also discusses how multiple views can be defined. Most of the program independent views mentioned in Chapter 2 can be specified using this approach. (The specification of program-specific views remains an area for further research.)

Chapter 10 describes several advanced animation operations in some detail and shows how they can be defined in terms of the theory developed in Chapters 5 to 8. All of the operations mentioned in Chapter 2 can be defined in this way.

Chapter 11 sketches how pure Prolog could be specified in the theory developed in Chapters 4 to 10. It shows how the semantics of pure Prolog can be specified in relational semantics, and then shows that the animation of a simple program produces the same steps as an existing Prolog animator. It finishes with a discussion of how the view used by the existing animator could be specified in terms of the theory.

Chapter 12 gives an overview of The Animator Generator: how to use it, how to write specifications for it, how it is implemented, and what its shortcomings are. The input language for The Animator Generator is called LSL. Full details

of how to use The Animator Generator and how to write specifications in LSL can be found in the forthcoming user manual [Ber].

Chapter 13 discusses possible extensions to both the theory and the implementation. These extensions include integrating the animator generator with a compiler generator and extending the system to support non-deterministic systems.

Chapter 14 reviews the thesis and restates the main results.

The appendices give a definition of an animation step in transitional semantics, a guide to the notation and a specification of Proc in LSL. Finally there is a bibliography, an index of lemmas and propositions, and a general index.

1.6 A Note on Terminology

When discussing generators, one has to avoid confusing the rôles involved in creating and using a system. I use **implementer** to mean someone who creates an animator generator, **language definer** to mean someone who specifies a language and creates an animator from that specification, and **user** to mean someone who uses the generated animator. Even if the same person does all three tasks, it is useful to keep the rôles separate.

Also, I use **program** to mean a program being animated, **tool** to mean a software tool that analyses a program, and **system** to mean a suite of tools.

When new terms are introduced in the text, they are written in **bold font**, as above. All definitions of terms are referenced by the index. The index references all occurrences of all but the most common concepts. Appendix B is a short reference guide to the notation used throughout the thesis.

1.7 Chapter Summary

The main contributions of this thesis are the definition of a semantic framework for defining animation operations, the definition of an evaluation step in terms of this framework, the comparison of different definitions of a step, a notation for specifying views of a program in mid-evaluation, the design of LSL, and the implementation of The Animator Generator. Other contributions include a classification of views of programs and a definition of equivalence between two styles of structured operational semantics.

Chapter 2

Background: Animation

This chapter explores the idea of program animation in more detail. It describes a range of tasks and shows how animators can support these tasks. For each task, it uses results from cognitive psychology to provide a framework that describes how people perform the task, and shows how the tasks can be supported by some of the features provided by existing animators.

The tasks considered are:

1. Learning a new language.
2. Understanding someone else's programs.
3. Debugging programs.
4. Learning to program.

The animators mentioned in the discussion of each task have been used successfully to support that task. The features that they provide form a checklist of views and operations that a theory of program animation should be able to describe. We will see in Chapters 9 and 10, which describe these features in more detail, that the theory developed in the rest of this thesis can indeed describe

these features. This indicates that the theory is adequate to describe real systems.

The role of an animator generator, and the problems that we must solve to create a theory of program animation, will be discussed in the next chapter.

2.1 Task 1: Learning a New Language

Cognitive psychology tells us that we learn a programming language by building a mental model of the entities that the language acts on and how it affects them [BOM81]. We derive these models from the way that programs behave, from the tools that we use, and from the documentation and training that we are given. The aim of the implementers and documentation writers should be to present the correct conceptual model of the system in such a way that the users will build the same mental model [Nor82].

A useful way of presenting the correct model, in any field, is to use an **advance organiser** [May81]. This is a model, such as a summary or diagram, that novices may use to structure the information they are given. The utility of advance organisers for novices, especially weaker novices, is well documented [May81,KB84,PF88].

In the case of teaching a programming language, Mayer [May81] suggests that a **notional machine** can be used as an advance organiser. A notional machine is an explicit model of the entities acted on by the language. (It is “notional” because these objects need not directly correspond to the actual hardware running the program). Mayer shows that people tend to understand a simple programming language better if they are given a diagram of an appropriate notional machine before a description of the language.

The design of a notional machine depends on the language that it is intended to illustrate. Du Boulay, O'Shea and Monk [BOM81] describe three notional machines. The first teaches a database query language by presenting a simple database. The second teaches assembly language by presenting a simple computer. The third teaches a high-level language by presenting a simple set of commands for turtle graphics; these commands can be combined using imperative concepts such as sequences, loops and procedures. One could also illustrate imperative languages by a set of variables, procedural languages by the procedure call stack, and Prolog by its search tree. These views (and others) could be combined as appropriate.

The descriptions of notional machines in the references cited above are limited in that they only present a static picture, whereas programs are inherently dynamic. Animating these machines makes them more than just advance organisers; it allows people to see what the description of the language means in detail, applied to actual examples. People can use the animator to check their mental models against the actual behaviour of a program.

The snapshots of programs in mid-evaluation that have been used as illustrations in books and lectures [CM84,WH84] attempt to animate notional machines. However, space constraints usually limit the number of steps that can be shown, and users can't choose which parts of the program to animate.

Early attempts to overcome these limitations focused on the use of film. Examples include Baecker's work producing films for teaching micro-PL/1 and Logo [Bae75], and Gross's films for teaching Fortran [Gro75]. (Baecker's paper contains a good bibliography of earlier work.) These films could show more steps than books, but they were still limited to particular programs.

The limitation to particular programs was removed by implementing notional machines on computers. This allowed users to animate any program in the language. For example, Leap [Lea84] and Pangratz [Pan81] describe animated

notional machines that teach the principles of assembly languages by showing the effects that programs have on a simple computer. These animators can be used with any programs and can be controlled by the user. Thus they are significantly more flexible than the pre-programmed systems.

Anecdotal evidence suggests that all these systems have helped people learn a new programming language.

Experience with these systems suggests that an animated notional machine should display both the current state of the machine and also the program with the current phrase highlighted. This enables users to see what effect the highlighted part of the program has on the state of the machine.

The main effect of some constructs is to return a value rather than to cause a change in the state of an underlying machine. This can be displayed by replacing each phrase of a program with its value when that phrase is evaluated [Lie84]. If a language forbids side-effects, this is all that needs to be displayed.

Much psychological theory on how people learn is concerned with how people use existing knowledge to form analogies to help understand new knowledge (see [DM83] for an example). This work suggests that if someone is learning a new language that is similar to one they already know, it may be helpful to present a notional machine that is similar to one for the known language. Users can then transfer their knowledge about the old machine to the new one, so that they only have to learn the new features. Conversely, if they are learning a different style of language (such as a logic language when they are used to imperative languages), presenting a different notional machine may help them deal with the differences.

To summarise, this section shows that animators can help people to learn a new programming language. It also shows that the notional machine presented by the animator should be chosen carefully to suit the language. Such a presentation of a notional machine is a view of the program that is being

animated. Chapter 9 will show how several views can be defined in terms of structured operational semantics.

2.2 Task 2: Understanding Someone Else's Programs

By “understanding a program”, I mean the task of working out what that program does and how it is structured. There are two ways that an animator can help users in this task. The first is by presenting views of the program that show the structure of the program. The second is by presenting views of the entities that the program manipulates. The first approach uses views that are independent of the program, whereas the second requires views that are particular to the program or application area. (In the rest of the thesis I concentrate on the more general first approach.)

2.2.1 Method 1: Presenting the Structure of the Program

Vessey [Ves85] has analysed the way that programmers debug other people's programs. She found that experienced programmers usually form a mental model of the overall structure of the program before looking for the cause of the bug, and always form such a model before making any changes. This is supported by Ward's anecdotal evidence [War86, pp. 41-2].

Wiedenbeck [Wie86] shows that when experienced programmers read a program they look for key features that indicate the structure of the program. For example, a function called `sort` is an indication that something is being sorted, and a for loop is an indication that an array is being traversed. This supports Vessey's observation; the behaviour of the programmers suggests that

they are building a mental model of the structure of the program. This suggests that users may benefit from a system that presents the structure of a program.

There are several ways that the structure of a program can be presented. Tischler, Schaufler and Payne [TSP83] show that experienced users benefit from seeing multiple views of a program when debugging. I believe that the same advantage applies when people are trying to understand someone else's code.

By analogy with the argument for notional machines, it seems likely that an animated structural view would be more useful than one that isn't animated. Indeed, some aspects of a program's structure grow as the program is executed. One example of this is a procedure call graph of an imperative or functional program, or the search tree of a Prolog program. Several Prolog animators display a view of the search tree [DC86,EB86].

A particularly useful structural view of an imperative program is a *slice*. A slice is a mini-program consisting of those lines of the main program which affect the value of a given variable. It is useful because there is evidence that people often mentally construct slices when debugging [Wei82]. It seems likely that they also construct slices when trying to understand someone else's code. Indeed, the existing literature often fails to distinguish the two tasks.

Chapter 9 describes several structural views in more detail. It also shows how these views can be defined in terms of the theory that I develop in Chapters 6 to A.

2.2.2 Method 2: Viewing the Data

To view the entities that a program manipulates, we first have to decide what these entities are. At the lowest level that we are concerned with, they are entities of the programming language, but when writing programs we prefer to use higher levels of abstraction.

Brooks [Bro83] models a program as a series of levels of abstraction. The lowest level is the domain of the programming language. The highest level is a representation of the problem domain. This could be a direct representation such as a simulation of each object in the problem domain by an entity in the program, or it could be an indirect representation such as an equational description. According to Brooks, the programmer's task is to construct these levels and to define maps between them so that each level is defined in terms of the level below.

We have seen that an animator can illustrate the lowest level of this model. Brooks's theory suggests that an animator would be more useful if it could also illustrate some of the other levels.

This can be done by defining intuitive views of the data. For example, the Balsa system has been used to animate a wide range of programs [BS81,BS85]. Balsa runs on a network of workstations, each with a high resolution bitmap screen. It shows graphical versions of data structures, updating them as the program is evaluated. The high resolution screens allow quite complex examples, including finding the transitive closure of a graph and updating a binary tree. For very large data structures the whole data structure can be shown with little detail and a small window can be used to see some sections in more detail. The system has been successful and popular with the students.

Balsa has the disadvantage that the animations must be pre-programmed. Ideally we would like a system that could animate an existing program. For example, Graphtrace [GKS83] lets users display Pascal representations of graphs in graphical form. PV [BCH⁺85] lets users attach a range of pictures to the variables of an imperative program. Garden [Rei87] lets the programmer design a pictorial display for each type, which can be used either for program output or for animation.

The systems mentioned in the previous paragraphs work with imperative

programs. By contrast, logic and functional languages don't update fixed variables, but pass values around, so the distinction between program and data is blurred. Therefore to show the data pictorially there has to be a way of selecting parameters to display. For example, the Dewlap system [DC86] displays the search tree of a Prolog program and allows users to select all occurrences of a value. This could be extended to display that value pictorially.

Another approach is to display an argument of a function every time that function is called. This is a simple extension to tracing mechanisms found in systems such as CAML [WmAL⁺87].

2.2.3 Summary

The evidence presented here, and our own intuition, suggests that animating a program can help people understand some aspects of the program. It also shows the importance of presenting different views of a program and its data.

Structural views help people form a mental model of the program's structure. Views of the data at different levels of abstraction show how the program represents the entities of the problem domain.

Chapter 9 will describe several of these views in more detail. It will also show how a specification of a language in structural operational semantics can be extended to include definitions of these views.

The technique defined in Chapter 9 can only be used to define views that apply to any program written in the language. Specifying data views that are specific to a particular program or problem domain remains a topic for further research. I include the above discussion to show the current limits of my theory.

2.3 Task 3: Debugging

Debugging is the task of finding and correcting the cause of an error in a program. I assume that the user already has a mental model of the structure of the program before starting the debugging task proper. We can assume that this is the case when people are debugging their own programs, and the psychological research cited in the previous section shows that experienced programmers approach the task of debugging someone else's code by first building a model of that code.

Debugging techniques are part of the folklore of computer science. They are rarely analysed or taught formally; they develop more by trial and error and spread by word of mouth or informal articles and books such as [Cec84, War86, Tra79, Lau79]. However, some cognitive psychologists have conducted experiments to find out how people debug programs, and recently some computer scientists have begun to develop the area formally.

This section first looks at a framework for discussing the debugging task, and then considers animation techniques for performing the task.

2.3.1 Brna's Model Of Debugging

Brna et. al. use a psychological model of the debugging process to guide their work on programming environments for Prolog. They present the following classification of bug descriptions [Brn88]:

Symptom: At this level bugs are described in terms of the visible effects of the program such as error messages, unexpected results or (non-)termination problems.

Program (mis)behaviour: This level describes bugs in terms of what the program does. Examples include infinite loops and unwanted successes of subgoals.

Program code error: This level describes bugs in terms of errors in code. Brna gives two classification schemes at this level. The first is syntactic, including bugs such as missing predicates or clauses in a wrong order. The second is schematic, comparing written code to a set of program schemas, such as Prolog's 'failure driven loop'. This catches bugs such as a missing side-effect in a loop (which often results in an infinite loop).

Underlying (mis)conception: This level describes bugs in terms of the user's model of the program, such as thinking that if a Prolog subgoal fails then the goal fails, or using a Prolog clause with the wrong mode.

Brna claims that these levels of description reflect the process that people use to write a program (read them bottom to top), and also the process they use to debug a program (read them top to bottom) [BPB87].

Brna suggests that animators are useful for examining program misbehaviour with the aim of locating the program code errors that cause the misbehaviour. This is the essential task of debugging: to find the source of the observed erroneous behaviour. This applies both to debugging one's own programs and to debugging someone else's programs. An animator can provide operations that support this task.

2.3.2 Views for Debugging

Ward [War86] discusses the ways in which a program code error may be related to the corresponding program misbehaviour. He bases his discussion on the idea of **proximity**, which describes the ways in which two pieces of code may be considered to be adjacent:

Lexical proximity: Two pieces of code are lexically adjacent if one occurs immediately after the other in the source of the program. The detected effect of a syntactic error is usually lexically adjacent to its cause.

Temporal proximity: Two pieces of code are temporally adjacent if one is executed immediately after the other. The effect of a bug in the control flow of a program is usually temporally adjacent to its cause.

Referential proximity: Two pieces of code are referentially adjacent if they reference the same memory location and no references to that location occur between the times when they reference it. The effect of a bug in the data-flow of a program is usually referentially adjacent to its cause.

From these definitions, Ward suggests his first principle of debugging. Using Brna's terminology, this says:

"Every program misbehaviour is related to its corresponding code error either lexically, temporally or referentially."

Chapter 9 will classify several views in terms of proximity. If users have a range of views that correspond to different types of proximity, they can choose one that is best suited to examining their current hypothesis.

Unfortunately, Ward's classification isn't complete. It doesn't deal with type mismatches between different modules. We can extend his classification with a notion of **static structural proximity** to allow for this. This kind of proximity also applies to the structure of a program shown by a flow chart or parse tree.

Ward's classification also excludes code linked by the choice of arguments to a function. Such pieces aren't linked referentially, since arguments aren't stored in the memory; neither are they linked temporally, since an argument may be passed through several functions before it is used. This suggests a notion of

dynamic structural proximity. This is particularly useful when debugging functional or logic programming languages. For example, the Dewlap system [DC86] can highlight all occurrences of a value in a Prolog search tree.

Another important aspect of program structure is that of **granularity**. For example, an Ada program can be considered as a number of modules, with the contents of the modules ignored, or as a collection of procedures, or as statements, or as expressions. For example, a flow chart is a view of a program at the statement level, a parse tree is a view at the expression level, and a procedure call graph is a view at the procedure level.

Granularity is also useful for classifying bugs. For example, a mismatched module interface is a bug at the module level, while an assignment to the wrong identifier is a bug at the statement level. This suggests that an animator should provide views and operations at different levels of granularity.

Together, proximity and granularity provide a useful classification of bugs and views. Chapter 9 classifies every view of the a program's source in these terms. An animator that provides views that correspond to different proximities and granularities enables users to select the most convenient view for examining their hypotheses about the cause of the program misbehaviour.

2.3.3 Animation Techniques for Debugging

Gould [Gou75] and Vessey [Ves85] describe a model of how people debug programs, based on their behavioural experiments. They suggest that people first select a tactic, such as tracing the evaluation of the program or comparing the input of the program with its output. They then search for clues using this tactic. If they find any clues, they use them to generate hypotheses about the cause of the error, select a new tactic based on these hypotheses, and begin a new search for more clues. This continues until they find the bug.

It follows that people want to examine certain aspects of the program structure or certain parts of the evaluation to confirm or refute their hypotheses. They obviously want to do this quickly and accurately. People have used debuggers extensively for this purpose.

Debuggers let people step through an evaluation and examine the current state at each step. In fact they provide all the functionality of an animator except the display – and some modern debuggers provide that as well [Plu88, Lie84, Hue84, EB86, DC86, Moh88, Raj86].

One useful feature of debugging-oriented animators is the ability to set break points on selected parts of the source code. The user can then run the evaluation until the selected part of the code is reached. This lets users concentrate on parts of the program that they consider suspect, ignoring the parts of the program that they aren't interested in. A similar feature lets the user run the evaluation until a certain variable is modified. This is akin to animating a slice, as described (and shown to be useful) in Section 2.2.1.

Animators can also provide features that help people to locate the parts of their code that contain bugs. Users can then examine these pieces of code in more detail.

Lieberman [Lie84] suggests the following divide-and-conquer method for locating bugs:

1. Select a major part of the evaluation under consideration.
2. Evaluate that part completely.
3. If this produces the correct interim result, the error must be in another part of the program.
4. If not, undo the evaluation and repeat the procedure with a sub-part of the erroneous part.

5. Repeat this procedure until the bug is found.

For example, if the program is the application of a function to two arguments, the two arguments can be evaluated and checked. If these sub-evaluations are correct, then the bug must lie in the final function application. If the bug is in one of the arguments, the evaluation of that argument can be undone and examined more closely.

Lieberman designed his ZSTEP system to support this method by providing an operation to evaluate the current phrase completely and also the ability to run the evaluation backwards.

2.3.4 Summary

Brna's model provides a framework in which to discuss debugging. In his terms the central task that an animator can support is to locate the code error that causes the the program to misbehave. The existence of the systems described here indicates that animation can indeed help people to debug programs.

An animator will be particularly useful for debugging if it provides the features described here. Chapter 10 will discuss these features in more detail and will show that they can be defined in terms of the theory developed in Chapters 6 to A. This indicates that the theory is sufficiently powerful to be useful.

2.4 Task 4: Learning to Program

Learning to program is difficult. Many novices make mistakes that seem trivial to experienced programmers. By analysing the task of programming we can see how to provide suitable teaching tools. Taylor and Du Boulay [TdB86] identify six sub-tasks that people face when learning to program (Brooks [Bro83] gives a similar analysis):

1. **Orientation.** Identifying problems that can be solved by programming.
2. **Interpreting Problem Descriptions.** Getting a sufficiently precise understanding of a problem to be able to determine what might count as a solution.
3. **Translating a solution.** Expressing a solution in a programming language. There is good evidence that novices have problems both in creating solutions and in translating their solutions into a programming language [SEBG82,SBE83,Kah82].
4. **Learning the programming language.** Bayman and Meyer [BM83] show that the syntax and semantics of each construct can cause problems for novices, although Spohrer and Soloway [SS86] suggest that only 1 in 10 of novices' problems fall into this category.
5. **Acquiring Programming Schemas.** There is considerable evidence from Cognitive Psychology that experienced programmers build programs from schemas of typical program actions. Allwood gives a good review of this literature [All86].
6. **Pragmatics.** Learning how to specify, develop, test and debug a program using whatever tools are available.

Animators can help the third, fourth and fifth of these sub-tasks.

2.4.1 Translating a Solution

An animator can show how a program implements (or fails to implement) the intended solution of a problem. This provides useful feedback for the third sub-task, although it doesn't help the student to express a solution in the language in the first place.

Also, animators can help students debug their programs, as shown in the previous section. This also helps them to learn how their solution needs to be improved.

2.4.2 Learning the Programming Language

This sub-task has already been discussed in Section 2.1.

2.4.3 Acquiring Program Schemas

Program schemas (also known as cliches) are patterns of code that are used in many places. Some schemas are general, such as using a loop to traverse an array. Others are particular algorithms, such as the mergesort algorithm.

Animation can help novices learn particular algorithms. For example, the BALSA system described above (in Section 2.2.2) has been used to teach people about sorting, trees, and many other aspects of basic computer science [BS81, BS85].

I haven't found any references to people using animators to teach general schemas. Possibly this is because most animators can only show concrete cases, and general schemas are more abstract than this. Animators are less useful for teaching people about general schemas. However, they can show how these schemas are used in particular cases. So they can help people learn how a general schema works, even though they can't show how generally it can be applied.

2.4.4 Summary

An animator can help novices with some of the sub-tasks of learning to program. A lot of anecdotal evidence exists to suggest that this help is significant [Bae75,

Gro75,Pan81,Lea84,MS76,BBA76,Hir84,Raj86]. It would be useful if someone were to perform a controlled experiment to test this hypothesis.

2.5 Chapter Summary

Animators can help people learn a new language, understand someone else's program, debug a program and learn to program. They do this by letting users compare their mental models of how a program should behave with the behaviour of a correct model.

The discussion of these tasks has mentioned several features that animators can provide to support these tasks. I will use these features as a base for judging the theory that I develop in the rest of this thesis.

The discussion has shown that there are many views that an animator may display. These include representations of the hardware, representations of the data, views of the structure of the program, and the source of the program itself. In Chapter 9 I will show how many views of a program can be specified as part of the specification of a programming language.

Views of data are usually specific to particular programs or application areas. In this thesis I ignore most data views; I consider only those that can be described in a program-independent way. Specifying more sophisticated data views remains a topic for further research.

The discussion has also shown that there are many operations that an animator may provide to support these tasks, beyond the basic notion of stepping through the program. These include breakpoints, full evaluation of the current phrase, and reverse evaluation. In Chapter 10 I will show that the theory developed in Chapters 6 to A is powerful enough to define these operations independently of a particular programming language.

Chapter 3

Background: Generating Animators

The first part of this chapter discusses the roles that an animator generator could fill. Since few people have used an animator generator, this discussion relies heavily on analogy with similar tools such as parser generators, compiler generators and structure editor generators.

The tasks that an animator generator could support are:

1. Producing new animators.
2. Designing new languages.
3. Learning about semantics.

The second part of this chapter sets out the requirements that must be satisfied by a theory of program animation if it is to be used to construct an animator generator. The construction of an animator generator and the ability to describe the animation features mentioned in Chapter 2 are the tests that I use to judge the success of my theory.

3.1 Task 1: Producing New Animators

There exists a body of literature on parser generators, compiler generators and structure editor generators. Most of this work has the aim of simplifying the task of producing tools for new languages or new host computers. Parser generators have shown themselves to be very useful in this regard; the most popular system being YACC [Joh78].

I have the same aim in my investigation of animator generators. In this sense, my theory is an extension of previous work (although I use a different semantic formalism from most previous work). An extension of the work in this thesis would be to implement an animator generator that produced compiled code instead of an interpreter. There is some work being done in this vein at the University of Edinburgh [dS].

3.2 Task 2: Designing New Languages

Generators also have the advantage that they force the implementer to write a formal specification of the relevant part of the language. This specification can then be checked for consistency. Thus the language designer can detect flaws in a language at an early stage of its design.

For example, YACC takes an attributed grammar as its input. As part of the parser generation process it reports any ambiguities in the grammar (with respect to the theory of LALR(1) parsing with extensions for handling precedence).

As another example, a formal specification of the semantics of a language may be used with a theorem prover to prove useful properties about a language,

such as the soundness of a type system. Some work in this area has been done by the Centaur project [mDD⁺85].

Centaur generates a structure editor and associated tools from a specification in a language called Typol [Des84], which is based on natural semantics. The associated tools may include type-checkers, interpreters or compilers. These compilers must be generated from a Typol specification of the translation between the source and target languages. The theorem proving work is based on showing that these translations are sound and complete with respect to the semantics of the source and target languages [Des86].

A related idea is to implement operational semantics as a theory in a meta-logic such as the Extended Calculus of Constructions [Luo89]. Some preliminary work is being done on this topic at the University of Edinburgh [Mit, Dam].

I also conjecture that an animator generator will be useful to language designers because they will be able to see how their languages behave on concrete examples, for little effort. Furthermore, if they use an established system that will have a high degree of confidence that the generated tool will correctly implement their specification. These arguments also apply to a lesser extent to compiler generators.

3.3 Task 3: Learning About Semantics

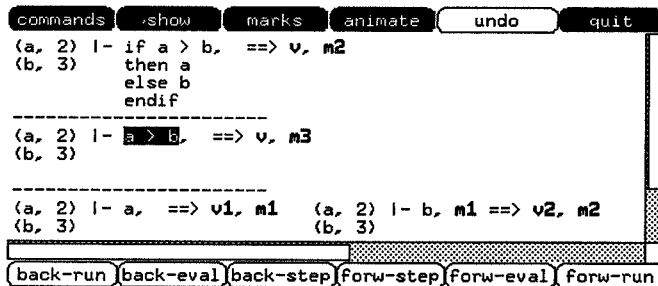
Animator generators can help people learn about language semantics. They can do this in two ways.

The first way is for the students to write semantics for a programming language. These semantics can then be used to generate an animator. The students benefit from seeing whether their semantics specify the desired

behaviour. In part this seems to be because the generated animators link the semantic notation to a concrete result; some students seem to prefer to deal with concrete examples of abstract notions. The animator also provides feedback on the correctness of the specification, which can be used to debug it, much as an animator can be used to debug a program.

The second way is for the animator to show the evaluation of a program in semantic terms. Since a specification of a language can be thought of as a program in the specification language, an animated semantic view can help people learn semantics in the same way that an ordinary view helps people learn the programming language.

For example, an animator generated for the language Proc by The Animator Generator can display the following view of the program animated in Chapter 1 (it may not make much sense if you aren't familiar with structured operational semantics):



This shows the inference tree of an if expression. The tree is displayed with its root at the top, so the topmost sequent is the conclusion of the rule at the root of the tree. The sequent below that is the first premise of that rule. Since this is also the conclusion of the corresponding rule, it is displayed with the premises of that rule immediately below it. These premises include semantic variables that haven't been instantiated, which are displayed in bold font.

3.4 A Theory of Animation

The previous sections in this chapter provide the motivation for producing an animator generator. In this section I look at what this involves. I conclude that the development of an animator generator should be based on a formal semantic theory of animation. The rest of the thesis develops such a theory based on structured operational semantics.

An animator generator needs a specification of a language to produce an animator for that language. Therefore we must specify the language in which this specification will be written.

To clarify our terminology, I will refer to the language being specified as the **object language** and the language in which it is specified as the **specification language**.

Between them, the specification of a language and the animator generator itself must contain all the information required by an animator. This information must be enough for the animator to parse programs, to evaluate them one step at a time, and to display them after each step.

The information required to parse a program must clearly be part of the specification, because the animator generator can't know the syntax of the object language in advance. Similarly, the semantics of the language must also be part of the specification. However, the definition of a step and the definition of how programs are displayed could conceivably either be part of the specification or be programmed into the animator generator.

3.4.1 Defining A Step

Consider the definition of a step first. An animator generator could incorporate a definition of an evaluation step that was parameterised on the semantics of the object language. Alternatively, the specification of each language could include the information explicitly, in addition to the semantics of the language.

Whichever choice is made, a step must be defined in terms of the semantic representation of a program in mid-evaluation. Either this definition will be part of the specification of the language, or it will be the specification of an algorithm in the animator generator.

Which option is selected may be affected by the choice of semantic formalism, and vice versa, because different formalisms may be better suited to implicit or explicit definitions of step. In this thesis, I use structured operational semantics, because the original presentation of this formalism [Plo81] aimed to encode the idea of a computation step, and this suggests that structured operational semantics would be a suitable formalism to use in an animator generator. It also provides concise specifications that can be reasoned about easily. Structured operational semantics is presented in detail in Chapter 4

The encoding of a step in Plotkin's presentation suggests that the definition of a step can be defined independently of a specification of a particular language, and thus that the animator generator can incorporate the corresponding information. It also suggests that the definition of an animation step should be the computation step encoded in the semantics. However, later chapters of this thesis will show that this is not in fact an appropriate definition of an evaluation step.

A specification written in structured operational semantics consists of a set of semantic rules. Each rule defines part of the evaluation of a single phrase of the language in terms of zero or more sub-evaluations. Each rule can be read as an inference rule or as an operational rule. The semantic representation of a

complete evaluation is a tree of instances of semantic rules. The sub-nodes of each node represent the evaluation of the sub-phrases of the phrase described by the node itself. This tree is called a **inference tree** because it is a tree that shows how the conclusion of each node is inferred from the premises. of the rule.

So although the rules in this formalism define a step, the effect of a step must be described by a supporting theory. The existing theory doesn't do this; no-one has defined a semantic representation of a program in mid-evaluation. I define a partial evaluation to be an incomplete inference tree, in which some rules contain uninstantiated variables. A step can than be defined as a map from one partial inference tree to another. The process of evaluation then consists of repeatedly applying a step to gradually construct the final inference tree.

3.4.2 Defining A View

Now consider the definition of how to display a program in mid-evaluation. The choice of where to make the information reside is similar to that for the definition of a step. An animator generator could conceivably analyse the parsing information to determine how to display the programs in the object language. In this case it could have several built-in views that would be available in every animator that it generated. Alternatively, the specification of the language could include a specification of each view.

There are several ways that views could be specified in a specification language. Each view could be specified in terms of the abstract syntax of the object language, with the animator generator containing enough information to highlight the current phrase. Alternatively, the view could be specified in terms of the semantic representation of a program in mid-evaluation. The animator generator could still highlight the current phrase automatically, or this could be specified explicitly as well.

The second approach would allow a greater range of views. Views that could be described this way and that couldn't be described in terms of the abstract syntax include call graphs that expand dynamically as the program is evaluated and views that update each phrase with its result when that result has been evaluated. Chapter 9 discusses these views in more detail.

Given the choice of structured operational semantics, and the theory developed to support the definition of a step, it seems plausible to define views in terms of a partial inference tree. This gives greater flexibility than defining views in terms of the abstract syntax. An alternative would be to provide built-in views, but in this case I would still need to define a view in terms of a program in mid-evaluation. As with the definition of a step, either this definition will become part of the specification of the language, or it will be the specification of an algorithm in the animator generator. In the case of a step the formalism itself suggests a canonical definition. There is no such suggestion of a set of canonical views, so I have opted for the more general approach for defining them. I discuss the success of these decisions in Chapter 14 (the conclusion).

3.4.3 View-Specific Steps

A fundamental rule of animation is that each step should change the representation of the program in such a way that the display of the representation, as specified by the current view, also changes. This is so that the user gets some feedback for each command. This means that the definition of a step and the definition of how to display a program are intrinsically related. A view may present only part of a program. For example, a call graph only shows the procedure names. Thus a step for a call graph will usually evaluate more of the program than a step that shows the full source code.

This relationship must be reflected in the theory. One way to do this is to parameterise the definition of a step on the definition of a view, so that a step

extends the inference tree until the next phrase to be evaluated is one of those displayed to the user. An alternative approach is to make the definition of a step part of the definition of the corresponding view. A third approach is to repeatedly apply a fundamental definition of step until it changes the display generated by the view. This choice partly depends on the above choices of where to put the specification of steps and views. Since I have chosen to incorporate the definition of a step into the animator generator itself, I have to select the first of these options.

Therefore the development of the theory in the rest of the thesis first defines a step for a view that shows the whole program, with the assumption that no other view will show more information than this. Then it shows how other views can be defined in the same framework. Finally, it gives a definition of a step that is parameterised on the definition of a view.

3.4.4 Summary

The information supplied by the specification of a language plus the information included in an animator generator must define an animation step and the required views of a program in mid-evaluation. This information may be divided between the specification and the animator generator in various ways. The division may be influenced by the choice of semantic formalism.

A step must be defined in terms of a semantic representation of a program in mid-evaluation. For structured operational semantics, the obvious representation is a partial inference tree. With this representation the process of evaluation repeatedly applies the step function to a inference tree, beginning with the empty tree and ending with a completely instantiated tree.

A view may be defined either in terms of an abstract syntax tree or in terms of a semantic representation of a program in mid-evaluation. The latter approach allows a wider range of views to be specified. For structured

operational semantics, this approach means defining a view in terms of a partial inference tree.

The definition of a step depends on the view to which it is applied. Either the definition of a view must include the definition of a step, or the definition of a step must be parameterised on the view, or there must be a fundamental definition of a step which is iterated until the change in the semantic representation of the program is shown in the display.

The theory that I develop in the rest of the thesis defines a step independently of a particular language. It requires views to be defined as part of the language specification. The definition of a step is parameterised on the current view.

3.5 Assessing the Theory

A theory of animation can (and should) be judged in several ways. We can ask how general it is: how many operations it describes, how many views can be specified in it, and how well it applies to different languages. We can also ask whether it provides a useful framework for comparing different ideas of animation, and whether it gives any insight into the process of animation itself.

The operations and views mentioned in chapter 2 can be used as a benchmark for a theory of animation. It should be possible to describe as many features as possible in terms of the theory. Chapters 9 and 10 discuss this aspect of my theory.

The range of languages that a theory supports can be judged by using the theory to specify and build animators for different languages. I designed my theory with imperative and eager functional languages in mind, and have tested it on Proc. It needs to be tested on real languages, and on languages from other

paradigms. It may be that other paradigms will require different features, but the fact that the features examined so far can be expressed in the theory suggests that the new features may also be expressible. In Chapter 11 I discuss a possible approach to specifying Prolog using my theory.

Although some forms of structured operational semantics have been used to define concurrent languages, such as CCS [Mil89], the present theory does not support concurrency or non-determinism. This is partly because the theory is based on the relational style of structured operational semantics, which does not support concurrency well. It is also the result of a wish to keep the model of evaluation reasonably simple for an initial exploration of a theory of animation. Chapter 13 discusses some extensions to the theory that would admit non-determinism.

In Chapter 8 I use the theory to compare different notions of step. This shows that the theory provides a framework that enables some aspects of animation to be compared. However, the framework isn't as elegant as one might wish. It remains to be seen whether someone can produce a more elegant theory.

Finally, I believe that my theory does give some insight into the process of animation. Even with the limited work in this thesis, it has shown that certain ideas of step are untenable, and has shown the intrinsic link between the definition of views and steps.

Chapter 14 (the conclusion) discusses these aspects of my theory.

3.6 Previous Work

I know of two other systems that produce animators from specifications: Centaur and PSG. Neither of them bases their definition of a step on a formal theory of animation, and neither of them support multiple views. Although they provide

facilities to examine values in the current environment and store, the only view they display is the source code of the program.

Centaur [mDD⁺85] produces a structure editor and interpreter from a language specification written in a specification language called Typol [Des84]. Typol is an implementation of natural semantics, another formalism based on inference rules. Natural semantics differs from structured operational semantics in that its rules don't necessarily have an operational reading.

Evaluation is implemented in Centaur by translating the semantic rules to Prolog. A step is defined informally, and is geared more to proof than program evaluation. For example, it is common for a construct to be specified by several inference rules. When a Typol specification of a construct is translated to Prolog, the backtracking evaluation strategy of the Prolog system may be visible to the user, even though the rules will often have an intuitive reading that factors them in such a way that backtracking is unnecessary.

The display of the source code of the program is defined in terms of the abstract syntax. The current phrase may be highlighted. This must be specified using action routines. Action routines are operations that can be added to typol rules. They don't affect the semantic definition of the language, but perform a side-effect such as highlighting a phrase or printing a message. A phrase is referenced by a path to the phrase from the root of the abstract syntax tree; a special variable always references the current phrase to make highlighting the current phrase relatively straightforward.

PSG [BS86b] produces a context sensitive structure editor [BS86a] and an interpreter with debugging facilities [BSM87] from a language specification based on denotational semantics. The interpreter is produced by compiling the denotational semantics into a functional language, which is then executed by a language independent interpreter. Prior to the evaluation of any phrase, the interpreter calls a debugger coroutine. This updates the display, checks for break

points, and so forth. A step is implemented by stopping before the evaluation of every phrase.

This definition of a step is imprecise. The cited papers don't define the evaluation process clearly, and don't explain what is meant by "stopping before the evaluation of a phrase".

The display of the source code is defined in terms of the abstract syntax, and the current phrase is highlighted automatically. More advanced operations and the display of the store and environment are defined in the language specification. They are grouped into trace functions, display functions, interrupt functions, continue functions, entry functions and result functions. These may be attached to nodes in the abstract syntax tree.

Trace functions are invoked when the debugger is in trace mode (which the user may select from a standard menu). Display functions display their arguments, which are typically values in the current environment or store, without affecting control flow. Interrupt and continue functions only affect control flow, and are used to implement advanced operations. Entry functions and result functions are used to implement advanced operations. Entry functions are executed when the interpreter begins evaluation of a node, and result functions are executed when the interpreter leaves the corresponding node.

These features allow the language definer to specify facilities such as single stepping, stepping across procedure calls, tracing statements or procedures, the setting of break points and the watching of variables. This approach is probably flexible enough to specify most operations found in existing debuggers.

Both Centaur and PSG lack a formal base for their animation features. This shows in their imprecise definitions of a step. It also shows in their ad-hoc facilities for displaying the program, especially the environment and store. It also restricts their usefulness for exploring and comparing different animation features.

3.7 Chapter Summary

Animator generators will help people to produce animators quickly, to design new languages and to learn about semantics. They help the latter two tasks by providing immediate feedback on a specification. Indeed, the implementation described in Chapter 12 has already been used in the University of Edinburgh's M.Sc. course in Information Technology to help people learn about structured operational semantics. Animator generators may also stimulate, and complement, the development of systems that support reasoning about semantic specifications.

The information required to produce an animator for a language may either be included in the specification of the language or be built-in to the animator generator itself. This information must include the definition of a step and the definitions of views. The decision of whether this information should be part of the specification or built-in to the generator can be influenced by the choice of semantic formalism, and the choice for steps and the choice for views are intrinsically related.

Some theory is needed to support the design of an animator generator. Steps must be defined in terms of a semantic representation of a program in mid-evaluation. Views may be also defined like this; alternatively a smaller range of views may be defined in terms of an abstract syntax. A theory can be judged on several forms of generality and on how useful it is as a framework for reasoning about animation.

I have chosen to specify languages using structured operational semantics. This formalism is reasonably concise and amenable to reasoning, and has a construct (the inference tree) which is an obvious candidate for representing a program in mid-evaluation. Also, Plotkin's original presentation was designed to

incorporate computation steps explicitly, and this suggests that a step can be defined independently of a particular language. The following chapters introduce the formalism in detail and examine the nature of a step.

Chapter 4

Structured Operational Semantics

As the previous chapter mentioned, structured operational semantics is the formalism used throughout the thesis. This chapter describes structured operational semantics in detail.

The first part of the chapter discusses the general approach of structured operational semantics, in particular the use of semantic rules. It explains how semantic rules can be read in two ways (inferential and operational) and how these two readings have different uses.

There are two styles of structured operational semantics, which I call transitional semantics and relational semantics. These styles are introduced and compared.

The rest of the chapter specifies the notation used throughout the thesis for both relational and transitional semantics. Since the aim of the thesis is to specify languages for an animator generator, this notation is more precise than most previous uses of structured operational semantics. For example, most authors have been happy to specify side-conditions for some rules, and to write these side-conditions in ordinary mathematical notation. My notation specifies how side-conditions can be written in the semantic formalism itself, giving a complete semantic specification language. The notation is based on existing

work, to ensure that specifications written in this notation are still suitable for other tasks, such as theorem proving.

4.1 Discussion

A formal semantics of a programming language gives a precise specification of what that language “means”; that is, what result will be generated by evaluating it. This specification is independent of particular implementations of the language. It enables properties of the language to be formally stated and proven. Also, various programming tools can be generated from it.

4.1.1 Operational Semantics

An **operational semantics** defines the meaning of a language in terms of the operations of an abstract machine. Thus it describes how a program is evaluated (on an ideal machine). This contrasts with other forms of semantics, which define the meaning of a program either as a mathematical entity or as a set of inference rules without concern for the evaluation process.

This use of an abstract machine suggests that an operational semantic formalism would be a good base for a theory of animation. The idea of the state of an abstract machine being changed by a sequence of steps parallels the idea of animation as a sequence of displays. If we have a map from the state of the machine to a display, such that each step changes the state in a way that is shown in the display, then we have an animation of the evaluation. This map is exactly what we mean by a view of the program.

This is not to say that a theory of animation could not be based on another style of semantics. The point is that the operational approach might be more amenable for an initial investigation.

The abstract machine of the operational semanticist is similar to the notional machines used to teach programming that were discussed in Chapter 2. The difference is one of abstraction; teachers and students want intuitive designs that illustrate the main points of a design, while operational semanticists want precise and detailed definitions. Thus a notional machine is a simple view of a semantic abstract machine.

4.1.2 Semantic Rules

Structured operational semantics [Plo81] is a particular style of operational semantics that combines the operational approach described above with the use of **inference rules**. This makes it easy to reason about language specifications while retaining the operational flavour. Structured operational semantics has been used to define real languages [MTH90], and gives reasonably concise and readable definitions.

In this formalism, a language is defined by a set of semantic rules. These rules may be read in either of two ways. The inferential reading says that each rule has one proposition (the **conclusion**) which holds if certain other propositions (the **premises**) also hold. The operational reading says that each rule specifies that the evaluation of the phrase on the left hand side of the conclusion comprises a sequence of sub-evaluations that are specified by the premises.

For example, the inferential reading of the following rule says that if we can evaluate (p_1, m) to (nil, m') and (p_2, m') to (nil, m'') in the environment e , then we can evaluate $(p_1; p_2, m)$ to (nil, m'') in e :

$$\frac{e \vdash p_1, m \Rightarrow \text{nil}, m' \quad e \vdash p_2 \Rightarrow m' \text{nil} m''}{e \vdash p_1; p_2, m \Rightarrow \text{nil}, m''}$$

In this reading the rule is considered to be an inference rule. The reading says nothing about the order in which the truth of the premises should be established. It also treats each proposition as a relation, without considering

some parts of it as inputs and others as outputs. The only order that this reading imposes on the rule is that the truth of the premises is necessary to establish the truth of the conclusion. This is illustrated by the pair of vertical arrows in the following diagram:

$$\left| \frac{e \vdash p_1, m \Rightarrow \text{nil}, m' \quad e \vdash p_2, m' \Rightarrow \text{nil}, m''}{e \vdash p_1; p_2, m \Rightarrow \text{nil}, m''} \right|$$

The operational reading of this rule says that to evaluate $(p_1; p_2, m)$ in the environment e , we first evaluate (p_1, m) , which must produce (nil, m') , and then evaluate (p_2, m') , which must produce (nil, m'') . Then the result of the whole evaluation is also (nil, m'') . In other words we begin with the left hand side of the conclusion, loop around the premise, and finish with the right hand side of the conclusion. This can be illustrated as follows:

$$\left(\frac{e \vdash p_1, m \Rightarrow \text{nil}, m' \quad e \vdash p_2, m' \Rightarrow \text{nil}, m''}{e \vdash p_1; p_2, m \Rightarrow \text{nil}, m''} \right)$$

In the operational reading, each proposition is treated as a function that takes the environment (on the left of the turnstile) and the left hand side of the arrow and returns the right hand side of the arrow. Furthermore, the premises are evaluated in order. Often this order can be determined from the occurrences of variables; since propositions are read as functions, a variable that appears on the left hand side of an arrow must be instantiated before that proposition is evaluated. In this thesis I assume that premises are written such that they can be evaluated in order from left to right, and I define a restriction on where variables may appear such that all variables on the left hand side of a proposition will be instantiated before that proposition is evaluated.

Sometimes we want to allow sub-phrases to be evaluated in any order. For example, many languages allow the arguments for arithmetic operations to be

evaluated in any order, so that the compiler may use a wider range of optimisations. I don't allow this in my theory of animation, but in Chapter 13 I discuss some extensions that would permit it.

The dual reading is what makes structured operational semantics useful. The inferential reading allows us to write compositional specifications; definitions in which the behaviour of a construct is described in terms of its parts. A compositional specification is easier to reason about than a non-compositional approach such as Landin's SECD machine [Lan64] or Milner's SMC machine [Mil76]. At the same time the operational reading gives an intuitive flavour to a specification, and suggests an approach to generating animators from a specification.

The dual reading of semantic rules is similar to the dual reading of horn clauses in Prolog. A Prolog program can be read as a specification in a restricted version of first order predicate logic, but it can also be given an operational reading. In the inferential reading the horn clauses are acting as inference rules. In the operational reading they specify the order in which goals should be evaluated. This analogy should not be stretched too far; the operational reading of semantic rules does not involve backtracking and two-way unification, whereas these concepts are implicit in the computational model of Prolog.

4.1.3 Transitional and Relational Semantics

There are two styles of structured operational semantics. The one shown above uses propositions that are relations between a phrase of the language and the result of evaluating that phrase. I call this style **relational semantics**.

The other style is the style that Plotkin used in his original development of structured operational semantics. It uses propositions that are transitions from one machine state to another. The intuition behind this style is that each transition corresponds to a computation step. I call this style **transitional**

semantics. I use double arrows ($\Rightarrow$) in relational semantic rules and single arrows ($\rightarrow$) in transitional semantic rules.

Relational semantics usually gives more concise specifications than transitional semantics. For example, the relational rule shown in the previous section describes the sequencing operator. Transitional semantics needs two rules to describe this operator:

$$\frac{e \vdash p_1, m \rightarrow p'_1, m'}{e \vdash p_1; p_2, m \rightarrow p'_1; p_2, m'}$$

$$\frac{}{e \vdash \text{nil}; p_2, m' \rightarrow p_2, m''}$$

The two readings for semantic rules apply to transitional semantics as well as relational semantics. The inferential reading of the first of the above rules says that if the first step of the evaluation of p_1 produces p'_1 , then the first step of the evaluation of $p_1; p_2$ produces $p'_1; p_2$. This is illustrated by the following diagram:

$$\left| \frac{e \vdash p_1, m \rightarrow p'_1, m'}{e \vdash p_1; p_2, m \rightarrow p'_1; p_2, m'} \right|$$

The operational reading says that to evaluate $p_1; p_2$ we perform one step of evaluation on p_1 to get p'_1 , and get $p'_1; p_2$ as a result. The following picture illustrates the operational reading:

$$\boxed{\frac{e \vdash p_1, m \rightarrow p'_1, m'}{e \vdash p_1; p_2, m \rightarrow p'_1; p_2, m'}}$$

Relational semantics has some advantages over transitional semantics. As noted above, it gives more concise definitions. It is also easier to reason about.

Transitional semantics often requires a proof by induction on a sequence of transitions, with another proof by induction on the height of a transition within the first proof. Relational semantics typically requires just one proof by induction on the depth of an evaluation tree. Relational semantics has also been used to describe a real language. As discussed in Section 6.1.5, the dynamic semantics of The Definition of Standard ML are defined in this formalism.

However, transitional semantics is based on the intuition that each transition corresponds to an intuitive definition of a computation step. Plotkin was explicit about this in his original formulation of transitional semantics. He was attempting to avoid the faults he saw in existing operational semantic definitions such as the SMC machine [Mil76], as shown by the following quote: [Plo81, page 29]:

Many of the transitions [of the SMC machine] are of little intuitive importance, contradicting our idea of the right choice of the “size” of the transitions. Further the definition of the transitions is not syntax-directed [...]. Finally but really the most important, the SMC machine is not a formalization of intuitive operational ideas but is rather, fairly clearly, correct *given* these intuitive ideas. [Italics in the original].

Unfortunately for our purposes, relational semantics doesn’t preserve the property of transitional semantics that propositions correspond to an intuitive definition of a computation step. So either we use transitional semantics, which is less satisfactory in other ways, or we recover this definition of a computation step. I take the second approach.

Therefore in the following chapters I begin the development of a theory of animation by defining a computation step for relational semantics that is equivalent to the transitions in transitional semantics. I prove that a transitional semantics for a language and an equivalent relational semantics for the language

generate the same sequence of steps for any program. This gives a theory that combines the advantages of the two styles of structured operational semantics.

4.1.4 Environments and Memories

Each proposition may be parameterised (as here) by an environment (e) and memories (m, m'). Both store auxiliary information which may affect the computation and be affected by it in turn, but which aren't part of the program text.

Memory terms are intended to represent a single memory which can be updated as the program is evaluated. For example, in imperative languages the memory is typically a set of bindings of values to locations. The rules are designed so that if the contents of a location are updated then the previous contents are lost. Environment terms are intended to represent environments that can be extended in such a way that the unextended environment can still be accessed.

The environment in which a phrase is evaluated is written to the left of the turnstile, to indicate that it can be accessed both before and after the evaluation. The initial memory is written on the left side of the arrow, and the final memory on the right side of the arrow, to indicate that those instances of the memory are only available at those points in the evaluation.

For example, the following rule describes the declaration of a local scope and the evaluation of an expression in that scope. At the end of the local scope the original environment is still available, but the original memory is lost:

$$\frac{e \vdash p_1, m \Rightarrow e', m' \quad e' \vdash p_2, m' \Rightarrow e', m''}{e \vdash \text{local}(p_1, p_2), m \Rightarrow v, m''}$$

This rule also shows how new environments are defined. A proposition can return an environment as its result. So in this case p_1 would be a declaration

that evaluated to the environment e' . This environment can then be used on the left of the turnstile of later premises.

4.1.5 Summary

Structured operational semantics combines two approaches to defining a language. The inferential approach facilitates reasoning about the definition. The operational approach, based on an underlying abstract machine, focuses on the process of evaluation.

There are two styles of structured operational semantics. In Plotkin's original presentation, which I call *transitional semantics*, each proposition is a transition. Each transition corresponds to an intuitive notion of a computation step. In the other style, which I call *relational semantics*, each proposition is a relation of a phrase and its value. Relational semantics is easier to reason about and produces more concise definitions, but doesn't have an obvious definition of a computation step.

In the development of my theory of animation I define a computation step for relational semantics that is equivalent to that of transitional semantics. This gives a theory that combines the features of the two semantics.

4.2 Notation and Meta-notation

This section defines the notation used throughout the thesis. Most of the notation applies to both relational and transitional semantics. The small amount that only applies to transitional semantics can be ignored by readers who are primarily interested in the practical side of the thesis. A summary of all notation is given in appendix B.

4.2.1 Abstract Syntax

An **abstract syntax** describes a program as a tree, instead of linear text. This allows the meaning of constructs to be defined without worrying about how they're written. Operator precedence, grouping and similar concerns can be ignored by the semantics.

Abstract syntax is usually specified by a set of **constructor** functions. Each constructor has a fixed arity (the number and type of its arguments). An abstract syntax **term** is a variable or a constructor of arity i applied to i other terms. Hence a term is a tree structure representing the program, with matters such as operator precedence and bracketing factored out. A **closed term** is a term containing no variables.

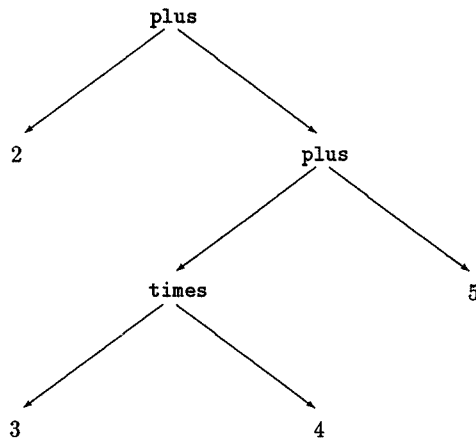
For example, if we have the constructors **plus** and **times**, both of arity 2, then the abstract syntax of:

`2 + 3 * 4 + 5`

could be (given a suitable representation of integers):

`plus (2, plus (times (3, 4), 5))`

This describes the following tree:



4.2.2 Objects

The objects that make up an evaluation are closed terms of an abstract syntax. Each object must have one of the following types described:

Values: Values include denotable values (those that can be bound to an identifier), expressible values (those that can be produced as the result of an evaluation), storable values (those that can be stored in a memory) and addresses (which index a memory). The different uses for values aren't distinguished.

Phrases: These are phrases of the language being specified. Examples include identifiers, arithmetic expressions and while loops. A phrase may also be a value, in which case it is called a **value phrase**. Integers are value phrases in most programming languages. If a phrase is not a value, it may not be made a value by replacing sub-phrases with values.

Environments: Environments are auxiliary objects that can affect an evaluation, and can be extended in ways that maintain access to the original object. A common example is a map from identifiers to values. An environment can also be a value, in which case it is called a **value environment**. For example, the value of a declaration is usually an environment.

Memories: Memories are auxiliary objects that can affect an evaluation, and that can be updated without preserving the original object. The forms of rules considered in this thesis force memories to be single-threaded; in other words these memories can be implemented by one updateable object.

Configurations: Configurations are only used in transitional semantics, where they represent the states of the finite automaton on which the semantics of the language is based. The set of phrases and the set of values are subsets of the set of configurations; phrases are the initial configurations of the underlying automaton, and values are the terminal configurations. (Memories are treated as attributes that are attached to configurations, not as part of the configurations themselves.)

Objects aren't divided into types beyond these given here. If a language includes different types of values, identifiers, or whatever, this must be represented by a judicious choice of names for constructors.

Except where specified, this thesis uses unambiguous concrete syntax to represent abstract syntax terms. This syntax is usually written in **typewriter font**. The exception is that infix constructors in isolated phrases are written in the normal maths font. Alphanumeric constructors are always written in typewriter font. Integers are written in the usual notation. The boolean values are written **tt** and **ff**. Identifiers are written in typewriter font, enclosed in single quotes (e.g. 'a'), except in a few displayed programs. The empty

environment is left blank. The syntax $e\uparrow(x, v)$ denotes the environment e extended by the binding of v to x .

4.2.3 Evaluation States

A **relational evaluation state** is a triple (p, e, m) of a phrase term, an environment term and a memory term. These terms may contain variables. A **relational program** is a relational evaluation state in which the phrase term is the parsed source of the program, and the environment and memory terms are the initial environment and initial memory for the appropriate language.

A **transitional evaluation state** is a triple (c, e, m) of a configuration term, an environment term and a memory term. A **transitional program** is a transitional evaluation state in which the configuration term is the parsed source of the program, and the environment and memory terms are the initial environment and initial memory for the language in question.

By convention, variables in terms are written as lower case italic letters. The choice of letter indicates the type of a variable; for example p is a phrase variable and e is an environment variable (see appendix B for other examples). Variables over identifiers are written x (primed or subscripted as required). Variables may be primed or subscripted when required.

4.2.4 Semantic Rules

A relational semantic rule consists of a proposition called the **conclusion** and zero or more propositions called the **premises**. A rule may also have a side condition. Each proposition is written as a **relational sequent**, which looks like this:

$$env \vdash subject, lhm \Rightarrow result, rhm$$

The *env*, *subject*, *lhm*, *result* and *rhv* are terms of the abstract syntax. These terms have types just as actual objects do. In fact an object is a closed term of the same type; for example a phrase is a closed phrase term.

A transitional semantic rule consists of a **conclusion** and zero or one **premises**. A rule may also have a side condition. **Transitional sequents** are written with a single arrow instead of a double arrow, as follows:

$$env \vdash subject, lhm \rightarrow result, rhv$$

The intuition underlying the presentation of transitional semantics in this thesis is that each transition represents one step of evaluation of an abstract machine.

The *env* of a sequent (relational or transitional) must be an environment term, the *subject* a phrase term and the *result* a value term. The left hand memory (*lhm*) and right hand memory (*rhv*) must be memory terms.

The *subject*, *lhm* pair is called the left hand side (**lhs**) and the *result*, *rhv* pair is called the right hand side (**rhs**) of the sequent. Also, the *lhs*, *env* pair is called the left state (**lst**) and the *rhs*, *env* pair is called the right state (**rst**) of the sequent.

Given a sequent q , $subject(q)$ denotes the subject of q , $env(q)$ denotes the *env* of q , and so forth. The conclusion of a rule ρ is written $conclusion(\rho)$ and the conclusion of an instance r of ρ is written $conclusion(r)$.

The division of lhs's and rhs's into pairs of a phrase or value term and a memory term makes clear the different rôles of memories and programs. Some of the definitions given later would hold for more general relations between arbitrary objects.

Rules are written with their premises separated from their conclusions by a horizontal line and their side condition to the right. This is an example of a rule with one premise:

$$\frac{\vdash \text{lookup}(e, x), m \Rightarrow v', m}{\vdash \text{lookup}(e\uparrow(x', v), x), m \Rightarrow v', m} \quad x \neq x'$$

This rule specifies that the result of looking up the identifier x in the environment $e\uparrow(x', v)$ is found by looking up the value of x in e . It only applies if $x \neq x'$.

Rules with no premises are called **axioms**. An example is:

$$\overline{\vdash \text{lookup}(e\uparrow(x, v), x), m \Rightarrow v, m}$$

This rule specifies that the result of looking up the identifier x in the environment $e\uparrow(x, v)$ is v . It doesn't have a side condition.

The **subject**, **result**, **env** etc. of a rule are those of its conclusion.

Side conditions allow greater control over which rule matches an evaluation state. They don't affect the evaluation once the matching rule has been chosen.

The syntax used for the side condition in the above example is chosen to be easy to read and understand at this point in the thesis. The proper syntax for side conditions will be described later (in section 4.2.8).

4.2.5 Rule Forms

Rule forms are meta-notation used to discuss rules in general instead of just specific rules. They use metavariables and expressions to denote terms.

Metavariables are either upper case italic letters, which denote any terms, or lower case greek letters, which denote variables. As with variables, the choice of letter indicates the type, and the letters may be primed or subscripted as required (see appendix B for examples).

Expressions are formed by applying constructors or **constructor variables** to metavariables or other expressions. Constructor variables are written a , primed or subscripted as required.

Side conditions in rule forms are denoted by θ .

This is an example rule form for relational rules:

$$\frac{E_1 \vdash P_1, \mu_0 \Rightarrow V_1, \mu_1, \dots, E_k \vdash P_k, \mu_{k-1} \Rightarrow V_k, \mu_k}{E_0 \vdash P_0, \mu_0 \Rightarrow V_{k+1}, M} \theta$$

The $NAME_i$ denote optional names of remote rule sets; these will be introduced in Section 4.2.7. Note the difference between the μ_i , which denote memory variables, and M , which denotes a memory term.

The rule forms permitted by the two styles of structured operational semantics are defined in the relevant chapters. These definitions will include restrictions on the admissible instances of the rule forms.

These restrictions will use the notation $\mathcal{F}(P)$. This notation denotes the set of variables in P . For example, $\mathcal{F}(p_1; p_2) = \{p_1, p_2\}$. This notation applies to expressions and terms of all types, and extends to sequents in the obvious way. Other authors often use this notation to denote the free variables of a term, but since abstract syntax terms don't have quantifiers, all variables in an abstract syntax term are free.

A rule is an instance of a rule form if it is a copy of that rule form with every metavariable replaced with a variable or term (as appropriate) of the same type as the metavariable. Metavariables may not be instantiated to metavariables or expressions. Side conditions are optional.

4.2.6 Substitutions and Instances of Rules

A **substitution** is a map from variables to terms. Substitutions are used to match rules to evaluation states, among other things. They must map variables of one type to terms of that type or certain other types, as listed in the following table:

Type of Variable	Type of Term
value	$\mapsto$ value
phrase	$\mapsto$ phrase or value
memory	$\mapsto$ memory
environment	$\mapsto$ environment or value
configuration	$\mapsto$ configuration, phrase, environment or value

Phrase variables and environment variables may be mapped to value terms because some values are also phrases or environments. Configuration variables may be mapped to phrase or value terms because phrases and values are subsets of configurations. Configuration variables may also be mapped to environment terms because some environments are also values.

If σ is a substitution, $\sigma(P)$ denotes P with σ applied to all variables in P . If a variable isn't in the domain of σ , applying σ to it leaves it unchanged.

If σ and σ' are substitutions then $\sigma \circ \sigma'(P) = \sigma(\sigma'(P))$.

A substitution σ that maps P to P' is **minimal** if for each other substitution σ' that maps P to P' there exists a third substitution σ'' such that $\sigma' = \sigma'' \circ \sigma$.

Two terms P, P' **unify** if there exists a substitution σ such that $\sigma(P) = \sigma(P')$.

One term P is more **general** than another P' , sometimes written $P \succ P'$, if there exists a substitution from the first to the second and no substitution exists from the second to the first.

The previous paragraphs apply to terms of all types.

Substitutions may also be applied to rules and premises. The definition of this is a straightforward extension of the preceding definitions. The result of such an application is an **instance** of a rule, or a premise of such an instance.

Two rules **unify** if their envs, lhms and subjects unify after renaming all variables to be unique to the rule in which they appear. A rule is more **general**

than another if at least one of its subject, lhm and env is more general than that of the other rule, and none is less general than that of the other rule.

A rule ρ with $\text{lst}(P, M, E)$ and side condition θ **weakly matches** an evaluation state (p, e, m) if the following conditions hold:

1. There exists a substitution σ such that $\sigma(E) = e$, $\sigma(M) = m$ and $\sigma(P) = p$.
2. p isn't a value.
3. θ is satisfied.

The rule **strongly matches** (p, e, m) if it also satisfies the following condition:

4. Any other rule satisfying conditions 1, 2 and 3 is more general than ρ .

Strong matching selects a single rule that specifies how an evaluation state should be evaluated. It is used to define the process of evaluation. Weak matching selects a set of rules that could match a certain state, and is used when defining the equivalence between transitional semantics and relational semantics.

4.2.7 Rule Sets

Rules may be partitioned into rule sets. Each rule set should define a particular operation in the language (or in the abstract machine). Examples include finding the value of an identifier in an environment, evaluating an expression, or performing basic operations on integers. Each rule set should contain either transitional rules or relational rules, but not both.

A rule may specify that a phrase must be evaluated in another rule set. This is called a **call** to a **remote** rule set. A call is written in the same way as a premise, with the remote rule set indicated by a superscript to the turnstile. A

call is also known as a **remote premise**. A premise that isn't a call is called a **local premise**.

For example, given a rule set called **LOOKUP** containing the rules for looking up the value of an identifier, a rule for evaluating an identifier in an expression can be written:

$$\frac{\text{LOOKUP} \quad e \vdash \text{lookup}(e, x), m \Rightarrow v, m}{e \vdash x, m \Rightarrow v, m}$$

In a rule form, every turnstile has a superscript that is written in italics, for example *NAME* or *NAME.i*. This superscript may be instantiated to the name of a remote rule set or to nothing.

A rule set *A* refers to another rule set *A'* if any rules in *A* include a call to *A'*.

Rule sets have several uses. For example, they make a specification modular, and hence easier to read and modify. Also, primitive operations such as basic integer operations can be provided as built-in rule sets, removing the need for a special notation for primitive operations.

Rule sets are more important in transitional semantics than in relational semantics, because transitional rules are less flexible. For example, the above rule and the rules in the **LOOKUP** rule set could be written without rule sets, by making a disciplined use of constructors, but the equivalent transitional rules can't be rewritten in this way. The explicit use of rule sets in the relational semantics will simplify the definition of equivalence between the two styles given in Chapter 7.

A **relational specification** is a set of relational rule sets that refer only to each other. Similarly, a **transitional specification** is a set of transitional rule sets that refer only to each other.

4.2.8 Side Conditions

Rules may have side conditions that restrict the phrases to which they can be applied. The example given in Section 4.2.4 was the following rule:

$$\frac{\vdash \text{lookup}(e, x), m \Rightarrow v', m}{\vdash \text{lookup}(e \dagger(x', v), x), m \Rightarrow v', m} \quad x \neq x'$$

A side condition is written as a call to a remote rule set. It isn't added to the premises, because it restricts the use of the rule rather than specifying part of the evaluation. Also for this reason, the rhm of a side condition must be the same as the lhs; both are usually omitted for clarity. Using this notation, the above rule becomes:

$$\frac{\vdash \text{lookup}(e, x), m \Rightarrow v', m}{\vdash \text{lookup}(e \dagger(x', v), x), m \Rightarrow v', m} \quad \text{IDENT} \quad \vdash \text{eq}(x, x') \Rightarrow \text{ff}$$

A side condition is satisfied if its lhs and env are evaluated to its rhs.

In many presentations of structured operational semantics, side conditions are used to specify built-in semantic operations. For example, the following rule specifies that the evaluation of an identifier gives the value of that identifier in the current environment:

$$\frac{}{e \vdash x, m \rightarrow v, m} \quad v = e(x)$$

The lookup of the identifier in the current environment can be expressed as a call to a remote rule set, as was done for the side conditions given above.

However, this condition differs from the ones given above in that it doesn't just determine whether the rule applies, but actually performs some computation. In my theory, this sort of "side condition" must be written as a remote premise.

The remaining side conditions may only affect whether a rule matches an evaluation state.

4.2.9 Summary

This section has introduced the notation used to define both relational and transitional semantics. This notation is similar to that used elsewhere, with the extension of meta-notation used to describe rule forms. The choice of letter for a variable or metavariable indicates its type, while the choice of font indicates whether it is a variable, a metavariable over terms or a metavariable over variables.

Rules are grouped into rule sets, each representing a particular operation. Premises may call remote rule sets. Side conditions are also written as calls to remote rule sets; they differ from premises in that they only affect the choice of which rule applies to a particular evaluation state.

4.3 Chapter Summary

The aim of this thesis is to develop a theory of animation. In Chapter 3 I explained my approach. First I define an evaluation step independently of a particular language, in terms of a semantic representation of a program in mid-evaluation. Then I show how a view can be defined in terms of the same representation. Finally I parameterise the definition of a step on a view.

This chapter has described the formalism, structured operational semantics, that I use in the development of this theory. I use this formalism because it combines operational and inferential approaches to specifying languages. It gives concise definitions that can be reasoned about easily. At the same time it is based on the idea of an abstract machine that underlies each language, and describes a process of evaluation.

I begin the development of the theory by combining the advantages of the two styles of structured operational semantics. Although both styles combine the

inferential and operational features, the relational style emphasises more concise definitions and ease of reasoning, while the transitional style emphasises the idea of a computation step. I produce a theory that combines the two by defining a step in relational semantics that is equivalent to that in transitional semantics.

Later I shall show that in fact this definition of step is not as suitable for animation as we would like. I discuss this, with examples, in Chapter 8. I will also show that the definition of a computation step can be refined to give a definition of an animation step that does meet our expectations.

I could have just presented the definition of an animation step, ignoring the idea of a computation step. However, the explicit representation of a step in transitional semantics was part of my original motivation for choosing to work with structured operational semantics, and I think it is interesting to examine computation steps to see why they are unsuitable. In addition, the equivalence between relational and transitional semantics is interesting in its own right. Finally, the definition of an animation step is fairly complex, and the simpler definition of a computation step provides a stepping stone to understanding it.

Chapter 5

Transitional Evaluations

The preceding chapters have introduced the notions of program animation and of generating animators from a specification of a language. They have shown that animation can support a range of tasks, and have mentioned some features of existing animators that help users perform these tasks. They have also shown that there are several tasks that could be supported by animator generators.

Chapter 3 discussed the need to base the design of an animator generator a theory of animation. Since either the specification of a language or the animator generator itself must provide a definition of an evaluation step, it follows that we need a framework in which such a step can be specified. Similarly, we need a framework in which to specify views of a program in mid-evaluation. This framework consists of a semantic representation of a program in mid-evaluation, and step and display functions defined in terms of this representation.

As I explained in the previous chapter, I have chosen to base my theory of animation on structured operational semantics. This is because this formalism has a straightforward operational reading. In particular, transitional semantics encodes computation steps as transitions. However, relational semantics specifications are more concise and easier to reason about. Therefore I begin the development of the theory by defining an evaluation step for relational semantics that is equivalent to that in transitional semantics. The result is a system that

combines the advantages of relational and transitional semantics. This is the subject of Chapters 5, 6 and 7.

In Chapter 8.2 I review this definition of step and consider its suitability for animation. I conclude that it is too coarse a definition for animation, and develop a refinement of this definition that gives better results. This definition, which I call an animation step, has the property that every computation step is also an animation step; the new definition just adds extra steps to make the animation easier for users to follow. The definition of an animation step is a simple extension of the definition of a computation step.

In Chapters 9 and 10 I show that this theory can describe the features mentioned in Chapter 2. This shows that the theory is powerful enough to describe useful systems. This is also demonstrated by Chapter 11, which gives an overview of how pure Prolog could be specified in terms of the theory. Finally, Chapter 12 describes the design and implementation of LSL and The Animator Generator. These are closely based on the theory, and demonstrate that it can form the basis of a practical system.

This chapter begins the development of a theory of animation. The first task in this development is to define a semantic representation of a program in mid-evaluation and to define a computation step in terms of this representation. As described above, I aim to define in terms of relational semantics a computation step based on the transitions of transitional semantics. This definition of a computation step requires a prior definition in transitional semantics. This chapter gives such a definition.

Readers who are interested in the practical aspects of the thesis and not the theory supporting it may skip this chapter. The next chapter presents the definition of a computation step in relational semantics. (The definitions of computation step in the two styles are shown to be equivalent in Chapter 7, which again is of little interest to the practically-minded reader.)

The first section of this chapter presents the model of evaluation that I use for transitional semantics and defines the admissible forms of transitional rules. These forms are compared to those used by Plotkin [Plo81] and to those of Milner's process calculus [Mil89].

The later sections define **transitional evaluation histories** and **transitional computation steps**. An evaluation history is a semantic representation of a program in mid-evaluation. Computation steps are defined as functions from one evaluation history to another. Evaluation histories are called **evaluations** when this doesn't cause ambiguity.

5.1 Transitional Rules

This section defines the forms of transitional rules that are allowed in the theory. These forms are chosen to fit the model of evaluation that equates transitions of a transitional specification with the transitions of a determinate finite automaton. Several example rules are given to illustrate the use of the forms.

These forms are compared with the rules presented by Plotkin in his paper on transitional semantics [Plo81]. This comparison shows that all the determinate rules considered by Plotkin can be described by the forms with little change. The forms are also compared with the rules in Milner's process calculus [Mil89]. This comparison shows the limitations of these rule forms for describing concurrent systems.

5.1.1 Forms of Transitional Rules

Transitional rules must have one of the following forms. The first and second forms describe operations of the underlying finite automaton. The third form

describes how the evaluation of a configuration can be defined in terms of the evaluation of a sub-configuration.

Rules of the first form are simple axioms:

$$\frac{}{E_0 \vdash C_0, \mu \rightarrow C_1, M} \quad \theta$$

where

$$\begin{aligned} \mathcal{F}(\theta) \cup \mathcal{F}(M) &\subseteq \{\mu\} \cup \mathcal{F}(E_0) \cup \mathcal{F}(C_0) \\ \mathcal{F}(C_1) &\subseteq \mathcal{F}(E_0) \cup \mathcal{F}(C_0) \end{aligned}$$

The conditions associated with this form, and those associated with the other forms, ensure that variables will be instantiated before use. Recall that $\mathcal{F}(P)$ denotes the variables occurring in P (see section 4.2.5).

Rules of the second form call a remote rule set:

$$\frac{E_1 \vdash C_1, \mu \rightarrow \nu, \mu'}{E_0 \vdash C_0, \mu \rightarrow C_2, M} \quad \theta$$

where

$$\begin{aligned} \mathcal{F}(E_1) \cup \mathcal{F}(C_1) &\subseteq \mathcal{F}(E_0) \cup \mathcal{F}(C_0) \\ \mathcal{F}(C_2) &\subseteq \mathcal{F}(E_0) \cup \mathcal{F}(C_0) \cup \{\nu\} \\ \mathcal{F}(\theta) &\subseteq \mathcal{F}(E_0) \cup \mathcal{F}(C_0) \cup \{\mu\} \\ \mathcal{F}(M) &\subseteq \mathcal{F}(E_0) \cup \mathcal{F}(C_0) \cup \{\mu', \nu\} \end{aligned}$$

References of rule sets to each other must form a directed acyclic graph (in other words references to remote rule sets may not mutually recurse). This graph is called the **rule set graph**. This restriction ensures that remote rule sets describe operations of a finite automaton that is more basic than the program being evaluated.

The third form specifies that if a sub-configuration of a configuration is evaluated one step, then the configuration itself is evaluated a corresponding step. If the sub-configuration doesn't evaluate to a value in that step, then the same rule will strongly match the next step of the evaluation. Thus the evaluation of large configurations can be composed of the evaluation of sub-configurations.

$$\frac{E_1 \vdash \gamma, \mu \rightarrow \gamma', \mu'}{E_0 \vdash C_0, \mu \rightarrow C_0[\gamma'/\gamma], \mu'} \quad \theta$$

where

$$\begin{aligned} \gamma &\in \mathcal{F}(C_0) \\ \mathcal{F}(\theta) \cup \mathcal{F}(E_1) &\subseteq \mathcal{F}(E_0) \cup \mathcal{F}(C_0) \end{aligned}$$

and γ may not be a value in any instance of this form.

The lhs of the premise of an instance of the third form must be a variable, as it is substituted for on the rhs of the conclusion. The rhs of the premise must also be a variable. It could theoretically be a configuration, which would restrict the result of the sub-evaluation. However, the same effect can be achieved by side conditions on the rules that weakly match the possible results of this rule, and the theoretical development is easier if the rhs of the premise is a variable.

Rules may not have more than one premise because this would make several operations of the underlying finite automaton directly equivalent to one, which is nonsensical. The premise of a transitional rule τ is sometimes written $\text{premise}(\tau)$; that of an instance t of τ is written $\text{premise}(t)$.

5.1.2 Determinacy Restriction

This thesis only considers determinate specifications. This is ensured by requiring any pair of rules in the same rule set to satisfy at least one of the following conditions:

1. The rules don't unify.
2. Their side conditions are disjoint.
3. One rule is more general than the other.

If either of the first two conditions are true, only one of the rules can match a given evaluation state. If the first two conditions are false but the third condition is true, then the definition of matching rules chooses the least general rule. Thus the specification must be determinate.

5.1.3 Examples

Here are three rules that describe an addition operator. The first two specify that the parameters of the operator are to be evaluated to numbers. The third has a remote premise that refers to a rule set that defines operations on integers:

$$\frac{e \vdash c_1, m \rightarrow c'_1, m'}{e \vdash c_1 + p_2, m \rightarrow c'_1 + p_2, m'}$$

$$\frac{e \vdash c_2, m \rightarrow c'_2, m'}{e \vdash v_1 + c_2, m \rightarrow v_1 + c'_2, m'}$$

$$\frac{\text{INTEGER} \quad \vdash v_1 + v_2, m \rightarrow v, m}{e \vdash v_1 + v_2, m \rightarrow v, m}$$

The following rules describe a while loop. These aren't as straightforward as the rules for addition.

Consider what happens if we take the normal syntax for a while construct and begin evaluating the condition:

$$\frac{e \vdash c_1, m \rightarrow c'_1, m'}{e \vdash \text{while } c_1 \text{ do } p_2, m \rightarrow \text{while } c'_1 \text{ do } p_2, m'}$$

Suppose the condition evaluates to true. We then evaluate the body until it returns the value nil. What do we do next?

$$\frac{}{e \vdash \text{while true do nil, } m \rightarrow \text{while ?? do ??, } m}$$

The original sub-phrases of the while construct have been lost, and so we can't continue.

This problem can be avoided by using a constructor with four arguments, in which the third and fourth arguments hold the original sub-phrases. This gives the following set of rules:

$$\frac{e \vdash c_1, m \rightarrow c'_1, m'}{e \vdash \text{while}(p_1, p_2, c_1, p_2), m \rightarrow \text{while}(p_1, p_2, c'_1, p_2), m'}$$

$$\frac{e \vdash c_2, m \rightarrow c'_2, m'}{e \vdash \text{while}(p_1, p_2, \text{true}, c_2), m \rightarrow \text{while}(p_1, p_2, \text{true}, c'_2), m'}$$

$$\frac{}{e \vdash \text{while}(p_1, p_2, \text{true}, \text{nil}), m \rightarrow \text{while}(p_1, p_2, p_1, p_2), m}$$

$$\frac{}{e \vdash \text{while}(p_1, p_2, \text{false}, p_2), m \rightarrow \text{nil}, m}$$

Other authors often use a transitive closure in their rules for `while`. This isn't allowed by the forms used in this thesis. The two approaches will be discussed when all the examples here are compared to examples from existing transitional semantic specifications (Section 5.1.4).

The following rule defines an assignment operator:

$$\frac{\text{LOOKUP} \quad \vdash \text{lookup}(e, x), m \rightarrow v', m}{e \vdash x := v, m \rightarrow \text{nil}, m + (v', v)}$$

The following rules define a linear lookup of an identifier in an environment:

$$\frac{}{\vdash \text{lookup}(e \dagger(x, v), x), m \rightarrow v, m}$$

$$\frac{}{\vdash \text{lookup}(e \dagger(y, v), x), m \rightarrow \text{lookup}(e, x), m} \quad \text{IDENT} \quad \vdash \text{eq}(x, y) \rightarrow \text{ff}$$

The final group of rules define a simple recursive function call. They assume a construct such as

`func  $x$   $x'$  =  $body$  in  $exp$  end`

as in Section 6.1.4. They also assume that the previous two rules define a rule set that can be called by a remote premise:

$$\frac{e \dagger (x, \text{closure}(e, x, x', p)) \vdash c, m \rightarrow c', m'}{e \vdash \text{func } x x' = p \text{ in } c \text{ end}, m \rightarrow \text{func } x x' = p \text{ in } c' \text{ end}, m'}$$

$$\frac{}{e \vdash \text{func } x x' = p \text{ in } v \text{ end}, m \rightarrow v, m}$$

$$\frac{e \vdash c, m \rightarrow c', m'}{e \vdash \text{call } x c, m \rightarrow \text{call } x c', m'}$$

$$\frac{\text{LOOKUP} \quad \vdash \text{lookup}(e, x), m \rightarrow v_1, m}{e \vdash \text{call } x v, m \rightarrow \text{call } v_1 v, m}$$

$$\frac{e' \dagger (x', v) \dagger (x, \text{closure}(e', x, x', p)) \vdash c, m \rightarrow c', m'}{e \vdash \text{call closure}(e', x, x', c) v, m \rightarrow \text{call closure}(e', x, x', c') v, m'}$$

$$\frac{}{e \vdash \text{call closure}(e', x, x', v') v, m \rightarrow v', m}$$

5.1.4 Comparison with Plotkin's Rules

Most of the transitional rules in Plotkin's original paper on structured operational semantics [Plo81] are described straightforwardly by the rule forms defined above. This section discusses the exceptions.

Plotkin's rules are slightly different from those given here in that he restricts the sets of identifiers, memory locations, and so forth for which his sequents are well-formed. For example, the sequents in the following rule [Plo81, page 95] are only well-defined for identifiers in the sets $\alpha \cup \alpha_0$ and α respectively. This is indicated by the subscripts on the turnstile symbols:

$$\frac{\rho[\rho_0] \vdash_{\alpha[\alpha_0]} \langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle}{\rho \vdash_{\alpha} \langle \rho_0; c, \sigma \rangle \rightarrow \langle \rho_0; c', \sigma' \rangle} \quad (\rho_0 : \alpha_0)$$

In the notation used here, with the adoption of α and α_0 , this becomes:

$$\frac{e \dagger e_0 \vdash_{\sigma \cup \sigma_0} c, m \rightarrow c', m'}{e \vdash_{\sigma} e_0; c, m \rightarrow e_0; c', m'} \quad \mathcal{F}(e_0) = \alpha_0$$

This thesis follows Milner (for example in [MTH90]) by assuming that sequents are well-defined for infinite sets of identifiers and locations, although terms must still be finite.

The first rule in Plotkin's paper that doesn't fit the forms used here is the following [Plo81, page 40]:

$$\frac{\langle c_0, \sigma \rangle \rightarrow \sigma'}{\langle c_0; c_1, \sigma \rangle \rightarrow \langle c_1, \sigma' \rangle}$$

In the notation used here, this becomes:

$$\frac{e \vdash c_0, m \rightarrow , m'}{e \vdash c_0; c_1, m \rightarrow c_1, m'}$$

The result of the premise of this rule is empty; Plotkin allows the lhs or rhs of arrows to be a (value, memory) pair or just a memory.

These is easily represented in the forms presented above by adding a special value constructor `nil` (say) to be returned by statements. (This is illustrated by the examples in Section 5.1.3). If required, the concrete syntax of `nil` can be empty, which gives the same effect as Plotkin's solution.

The rules for `while` [Plo81, page 43] use a transitive closure in a local premise:

$$\frac{\langle b, \sigma \rangle \rightarrow^* \langle tt, \sigma' \rangle}{\langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \langle c; \text{while } b \text{ do } c, \sigma' \rangle}$$

In the notation used here, this becomes:

$$\frac{e \vdash c_1, m \rightarrow \text{true}, m'}{e \vdash \text{while } c_1 \text{ do } c_2, m \rightarrow c_2; \text{while } c_1 \text{ do } c_2, m'}$$

Plotkin's approach gives the correct result for a while loop, but it doesn't fit the underlying intuition used in this thesis, namely that each transition corresponds to one computation step in an abstract machine. Using a transitive closure as a premise would imply that several steps were equivalent to one.

Plotkin's approach also has the disadvantage that it depends on the existence of a sequential operator in the language, rather than specifying the sequencing directly in the semantics.

A `do ... while` loop would require two premises with transitive closures in Plotkin's style. This is getting more like relational semantics, and even farther from the idea of one transition corresponding to one step.

An inferior version of the approach used for `while` in Section 5.1.3 would be to parse a while loop to the usual two argument constructor and then use the following rule to silently translate the two argument form to the four argument form:

$$\frac{}{e \vdash \text{while } p_1 \text{ do } p_2, m \rightarrow \text{while}(p_1, p_2, p_1, p_2), m}$$

Apart from the fact that this operation could be done by the parser, and is therefore redundant, it introduces an extra step into the evaluation. So it fails to provide the desired hidden translation.

The final exceptions from Plotkin's paper considered here are the following rules [Plo81, page 93]. The first specifies the evaluation of an identifier when that identifier is bound to an expressible value, the second when it is bound to an address.

$$\frac{}{\rho \vdash \langle x, \sigma \rangle \rightarrow \langle \text{con}, \sigma \rangle} \quad (\rho x = \text{con})$$

$$\rho \vdash \langle x, \sigma \rangle \rightarrow \langle \text{con}, \sigma \rangle \quad (\rho x = I \text{ and } \sigma I = \text{con})$$

Using the notation given here (but allowing multiple premises), these become:

$$\frac{\text{LOOKUP} \quad \vdash \text{lookup}(e, x), m_0 \rightarrow v, m_1}{e \vdash x, m_0 \rightarrow v, m_1}$$

$$\frac{\text{LOOKUP} \quad \vdash \text{lookup}(e, x), m_0 \rightarrow v, m_1 \quad \text{DEREF} \quad \vdash \text{deref}(m, v), m_1 \rightarrow v', m_2}{e \vdash x, m_0 \rightarrow v', m_2}$$

There are two problems with these rules. The first is that they don't satisfy the determinacy restrictions, and the second is that the second rule has two remote premises.

The easiest way to solve the first problem is to use different constructors for constants and references. This is a reasonable requirement to place on a parser.

If different constructors are used, the second rule can be converted to a rule with only one remote premise by creating a new rule set to do the combined lookup and dereference operations. The number of steps an operation takes in a remote rule set doesn't affect the number of steps taken in the caller's rule set.

If different constructors aren't acceptable, then one of the following solutions will have to be used:

1. There could be one rule for looking up values in an environment and another for dereferencing addresses found by the first rule. This will take two transitions instead of one.
2. The LOOKUP and Deref rule sets could be merged, so that the problem is moved from the main rule set to the new remote rule set. This gives a single transition in the current rule set.

5.1.5 Comparison with Milner's Process Calculus

Milner's process calculus [Mil89] uses transitional semantics to describe concurrent systems. This thesis is only concerned with determinate specifications, but it is worth comparing the transition rules defined above with those used by Milner, to get a feel for the limits of the forms presented above.

The most obvious difference is that Milner's transitions are labelled. These labels specify synchronisation between different processes. To allow labels, the system developed here would have to be embedded in a concurrent framework. However, it is interesting to look at the form of Milner's rules ignoring the labels.

The rules that Milner uses for his choice operator, which uses the $+$ sign, have the following form in the notation used in this thesis:

$$\frac{\vdash c_1 \rightarrow c'_1}{\vdash c_1 + c_2 \rightarrow c'_1}$$

$$\frac{\vdash c_2 \rightarrow c'_2}{\vdash c_1 + c_2 \rightarrow c'_2}$$

These rules are not determinate (that's the point of them!). Allowing them in the above forms would require relaxing the determinacy restriction, and adapting the results of this chapter to handle non-determinate evaluations.

Another of Milner's rules that isn't allowed by the above forms has the following form:

$$\frac{\vdash c_1 \rightarrow c'_1 \quad \vdash c_2 \rightarrow c'_2}{\vdash c_1 | c_2 \rightarrow c'_1 | c'_2}$$

This describes the parallel evaluation of two sub-phrases. The forms allowed in this thesis only allow one premise for a transitional rule; this is to prevent the possibility of parallel evaluations. Extending the syntax to cope with rules like the above is simple, and extending the results of this chapter is probably easier than handling the choice operator, provided that we ignore communication between processes.

Finally, Milner also gives a rule with the following form:

$$\frac{\vdash c \rightarrow c'}{\vdash x \rightarrow c'} \quad x \stackrel{def}{=} c$$

where the side condition means that x is defined to be c . This rule effectively looks up the value of x in an implicit environment and executes the first step in the evaluation of the bound expression. The forms used in this thesis requires the lookup and the step to be separate actions since the lookup is an operation of the abstract machine and each such operation should correspond to a transition.

5.1.6 Summary

The model of evaluation that I use for transitional semantics is that each transition is a transition of a finite automaton. The transitions of this

automaton are described by axioms and by calls to remote rules sets. This model limits rules to no more than one premise.

Most rules in existing definitions fit the restrictions of this model. The main exceptions are rules that refer to the original version of a phrase after that phrase has been evaluated. These rules are easily accommodated by a simple change to the abstract syntax.

5.2 Transitional Evaluation Histories

In transitional semantics a complete evaluation of a program is described by a sequence of stacks of instances of transitional rules. Each stack represents a computation step. The 1st of the bottom instance in the first stack is the program that the evaluation describes, and the result of the bottom instance in the last stack is the value of the evaluation. If the instance t at the bottom of a stack has a local premise, then the conclusion of the instance above it is identical to that premise. The sub-stack above t defines the first step in the evaluation of the subject of the premise of t .

For a theory of animation we want a semantic representation of a program in mid-evaluation, not just of a complete evaluation. Such a representation can be constructed easily as a partial evaluation. I define this to be a sequence of stacks like a complete evaluation, but where the result of the bottom instance of the last stack need not be a value. Complete and initial evaluations can be defined easily as special cases of this general construct.

I call this construct an **evaluation history**, so that when necessary we can distinguish between this construct and the process of evaluation. However, evaluation histories are called **evaluations** when it is unambiguous to do so.

The rest of this section gives a formal definition of this transitional evaluations. It also shows that a complete evaluation of a program is unique.

5.2.1 Consistent Stacks

A **consistent stack** is a non-empty stack of instances of transitional rules such that:

1. Each instance in the stack has a local premise, except for the top instance. Each local premise is identical to the conclusion of the instance above it.
2. The side condition (if any) of each instance is satisfied.
3. Each instance in the stack must be an instance of the rule that strongly matches the 1st of that instance.
4. If the top instance is an instance of a rule of the form

$$\frac{E_1 \vdash C_1, \mu \rightarrow \nu, \mu'}{E_0 \vdash C_0, \mu \rightarrow C_2, M} \text{NAME}$$

then either:

- (a) E_0 , C_0 and μ are closed terms, C_1 isn't a value in the remote rule set and (ν, μ') is the rhs of a complete evaluation of E_1 , C_1 and μ in the remote rule set.
- (b) E_0 , C_0 and μ are closed terms, C_1 is a value in the remote rule set, $\nu = C_1$ and $\mu' = \mu$.

The **subject**, **result**, **env**, etc. of a consistent stack are those of the bottom instance of the stack.

Stacks are ordered from the bottom upwards, so for example the second instance in a stack is the instance above the bottom one.

If s is a stack of instances of transitional rules, then $\#s$ is the height of s , $top(s)$ is the top instance in s , $bottom(s)$ is the bottom instance in s and $[s]$ is s with the bottom instance removed. $[s]$ is undefined on stacks of height 1.

Stacks are written as boxed vertical lists of instances of transitional rules, for example:

$$\begin{array}{c}
 \text{INTEGER} \\
 \frac{\vdash 2 + 3, m \rightarrow 5, m}{\vdash 2 + 3, m \rightarrow 5, m} \\
 \\
 \frac{\vdash 2 + 3, m \rightarrow 5, m}{\vdash (2 + 3) + 4, m \rightarrow 5 + 4, m} \\
 \\
 \frac{\vdash (2 + 3) + 4, m \rightarrow 5 + 4, m}{\vdash ((2 + 3) + 4) + 5, m \rightarrow (5 + 4) + 5, m}
 \end{array}$$

or

$$\begin{array}{c}
 \text{LOOKUP} \\
 \frac{\vdash \text{lookup}(e, 'a'), m \rightarrow 2, m}{e \vdash 'a', m \rightarrow 2, m} \\
 \\
 \frac{e \vdash 'a', m \rightarrow 2, m}{e \vdash 'a' > 'b', m \rightarrow 2 > 'b', m}
 \end{array}$$

5.2.2 Consistent Sequences

A **consistent sequence** is a sequence of consistent stacks such that for each stack s in the sequence:

1. The lhs of s is the same as the rhs of the previous stack.
2. The env of s is the same as that of the previous stack.

Note that the empty sequence satisfies these conditions.

If S is a consistent sequence, then $length(S)$ is the number of stacks in S , $head(S)$ is the first stack in S , $last(S)$ is the last stack in S and $tail(S)$ is the sequence formed by removing the first stack from S . If the height of each stack

in $\mathcal{S}$ is greater than 1 and the bottom instance of each stack is an instance of the same rule, then $[\mathcal{S}]$ is the sequence formed by removing those instances from each stack in $\mathcal{S}$. These functions (except *length*) are undefined on empty sequences, and $[\mathcal{S}]$ is undefined on sequences that don't satisfy the necessary condition.

This thesis presents a consistent sequence as a vertical list of stacks grouped together by a left brace. A horizontal list would be a better presentation, because it would show the correspondence between the lhs of one stack and the rhs of the previous stack, but this wouldn't fit on a page in a readable size of print.

5.2.3 Definition of Evaluation Histories

A **transitional evaluation** of an evaluation state (p, e, m) is a finite consistent sequence T such that $\text{length}(T) > 0$ and $\text{lst}(T) = (p, m, e)$.

5.2.4 Initial and Complete Evaluations

An **initial transitional evaluation** of an evaluation state (p, m, e) is a transitional evaluation T of (p, m, e) such that $\text{length}(T) = 0$.

A transitional evaluation is **complete** if its result is a value.

Lemma 5.1 *A complete transitional evaluation T of an evaluation state (p, m, e) is unique.*

Proof: By double induction. The outermost induction is on the rule set graph; the inner induction is on the length of the evaluation at a node of that graph.

Base case: A rule set that doesn't refer to other rule sets; that is, a rule set that doesn't contain any rules with remote premises.

Base case: $\text{length}(T) = 1$.

$head(T)$ is uniquely determined by p, m and e . This can be shown by a simple induction on the height of $head(T)$:

Let $t = bottom(head(T))$ and τ be the rule instantiated to t .

Base case: $head(T)$ contains one instance.

τ must be an axiom, since the top instance in a stack in an evaluation can't have a local premise, and no rules in this rule set may have remote premises. Therefore the instances of the variables in the rule are determined by (p, m, e) . The determinacy restriction ensures that the choice of rule is unique.

Induction Step: The stack contains more than one instance. τ must have a local premise.

The instances of the variables in τ are uniquely determined by (p, e, m) , except for $rhs(premise(\tau))$. By induction, $[head(T)]$ is uniquely determined by $lst(premise(t))$. Therefore $rhs(premise(t))$ is also uniquely determined.

Induction step: $length(T) > 1$

Let s_i and s_{i-1} be the i^{th} and $i-1^{th}$ stacks in T , and assume that the first $i-1$ stacks in T are uniquely determined by (p, m, e) . Then $lst(s_i) = rst(s_{i-1})$. The choice of rule is unique, by the determinacy restriction. The argument used in the base case shows that s_i is unique.

Thus a transitional evaluation of (p, m, e) is unique for any given length. T is a complete transitional evaluation and so its result is a value. No rules may match a value, so the length of T is fixed and T is unique.

Induction step: A rule set that refers to other rule sets.

Base case: $length(T) = 1$.

If a premise q is remote, any evaluation of $lst(q)$ must be unique, by induction. Therefore $rhs(q)$ must also be unique. The rest of the case proceeds as above.

Induction step: As above.

5.2.5 Summary

Transitional semantics lends itself to a simple representation of a program in mid-evaluation. This representation is a consistent sequence of consistent stacks of instances of rules, in which each stack represents one computation step. Initial and complete evaluations are simple special cases of the general definition.

5.3 Transitional Computation Steps

The definition of a computation step for transitional evaluations is based on the intuition that each stack in an evaluation corresponds to one computation step. Therefore the definition is straightforward.

5.3.1 Definition

A **computation step** for transitional evaluations is a function $\mathcal{E}_T$ such that if T is an transitional evaluation, then $\mathcal{E}_T(T)$ is T extended by a new stack such that the new sequence satisfies the requirements of a transitional evaluation.

5.3.2 Example

If the environment e binds 'a' to 2 and 'b' to 3, the following sequence is a series of transitional evaluations of 'a' > 'b', m and e , such that each evaluation is reached from the previous one by an application of $\mathcal{E}_T$.

The first step looks up the value of 'a' in e :

$$\left\{ \begin{array}{c} \text{LOOKUP} \\ \frac{\vdash \text{lookup}(e, 'a'), m \rightarrow 2, m}{e \vdash 'a', m \rightarrow 2, m} \\ \\ \frac{e \vdash 'a', m \rightarrow 2, m}{e \vdash 'a' > 'b', m \rightarrow 2 > 'b', m} \end{array} \right.$$

The second step looks up the value of 'b':

$$\left\{ \begin{array}{l} \text{LOOKUP} \\ \frac{\vdash \text{lookup}(e, 'a'), m \rightarrow 2, m}{e \vdash 'a', m \rightarrow 2, m} \\ \frac{e \vdash 'a', m \rightarrow 2, m}{e \vdash 'a' > 'b', m \rightarrow 2 > 'b', m} \\ \text{LOOKUP} \\ \frac{\vdash \text{lookup}(e, 'b'), m \rightarrow 3, m}{e \vdash 'b', m \rightarrow 3, m} \\ \frac{e \vdash 'b', m \rightarrow 3, m}{e \vdash 2 > 'b', m \rightarrow 2 > 3, m} \end{array} \right.$$

The third step completes the evaluation:

$$\left\{ \begin{array}{l} \text{LOOKUP} \\ \frac{\vdash \text{lookup}(e, 'a'), m \rightarrow 2, m}{e \vdash 'a', m \rightarrow 2, m} \\ \frac{e \vdash 'a', m \rightarrow 2, m}{e \vdash 'a' > 'b', m \rightarrow 2 > 'b', m} \\ \text{LOOKUP} \\ \frac{\vdash \text{lookup}(e, 'b'), m \rightarrow 3, m}{e \vdash 'b', m \rightarrow 3, m} \\ \frac{e \vdash 'b', m \rightarrow 3, m}{e \vdash 2 > 'b', m \rightarrow 2 > 3, m} \\ \text{INTEGER} \\ \frac{\vdash 2 > 3, m \rightarrow \text{ff}, m}{e \vdash 2 > 3, m \rightarrow \text{ff}, m} \end{array} \right.$$

Lemma 5.2 *If T is an incomplete transitional evaluation, then $\mathcal{E}_T(T)$ is unique.*

Proof: The proof is the same as the innermost induction of the Lemma 5.1, applied to the new stack that is added to T to make $\mathcal{E}_T(T)$.

5.4 Chapter Summary

This chapter has defined the forms of transitional rules considered in this thesis. These rule forms are restricted so that they conform to the model of evaluation, in which every transition is a transition of a determinate finite automaton. This chapter has also defined evaluation histories and computation steps for transitional semantics, and has shown that the specifications permitted by the rule forms must be determinate.

Chapter 6

Relational Evaluations

This chapter defines a computation step in relational semantics. The model of evaluation that underlies this definition is the one developed for transitional semantics in the previous chapter; namely that each transition of a transitional semantic specification is a transition of a determinate finite automaton.

The first section of this chapter describes the form of admissible relational rules and presents the model of evaluation in terms of relational semantics. This form is compared to rules from The Definition of Standard ML [MTH90].

The later sections define **relational evaluation histories** and **relational computation steps**. These are equivalent to the transitional versions defined in the previous chapter. An evaluation history is a semantic representation of a program in mid-evaluation. Computation steps are defined as functions from one evaluation history to another. Evaluation histories are called **evaluations** when this doesn't cause ambiguity.

6.1 Relational Rules

This section defines the form of relational rules that are allowed in the theory, and presents the model of evaluation in terms of this form. Several example rules are given to illustrate the use of the form.

This form is compared with the rules in The Definition of Standard ML[MTH90]. This comparison shows that the operational rules in The Definition of Standard ML can all be described by the form with little change. It also discusses the difference between operational rules and non-operational rules. The formalism that uses this form of rules but that allows non-operational rules is called **natural semantics**.

6.1.1 Form of Relational Rules

Relational rules have the following form:

$$\frac{E_1 \vdash P_1, \mu_0 \Rightarrow V_1, \mu_1, \dots, E_k \vdash P_k, \mu_{k-1} \Rightarrow V_k, \mu_k}{E_0 \vdash P_0, \mu_0 \Rightarrow V_{k+1}, M} \theta$$

where $k \geq 0$, each $NAME_i$ is optional, the line may be single or double, and

$$\begin{aligned} \mathcal{F}(E_1) \cup \mathcal{F}(P_1) &\subseteq \mathcal{F}(E_0) \cup \mathcal{F}(P_0) \\ \mathcal{F}(V_{k+1}) &\subseteq \bigcup_{i=1}^k \mathcal{F}(V_i) \cup \mathcal{F}(E_0) \cup \mathcal{F}(P_0) \\ \mathcal{F}(M) &\subseteq \bigcup_{i=1}^k \mathcal{F}(V_i) \cup \mathcal{F}(E_0) \cup \mathcal{F}(P_0) \cup \{\mu_k\} \\ \mathcal{F}(\theta) &\subseteq \mathcal{F}(E_0) \cup \mathcal{F}(P_0) \cup \{\mu_0\} \\ \bigvee_{j=1}^{k-1} \mathcal{F}(E_{j+1}) \cup \mathcal{F}(P_{j+1}) &\subseteq \mathcal{F}(E_0) \cup \mathcal{F}(P_0) \cup \bigcup_{i=1}^j \mathcal{F}(V_i) \end{aligned}$$

The conditions ensure that all variables are defined before use in evaluations. Recall that $\mathcal{F}(P)$ denotes the variables occurring in P .

References of rule sets to each other must form a directed acyclic graph (that is, references to other rule sets may not mutually recurse). This graph is called the **rule set graph**. As discussed below, this restriction is inherited from the transitional semantic model of evaluation, where it ensures that remote rule sets describe operations of a finite automaton that is more basic than the program being evaluated.

6.1.2 Evaluation Model

The model of evaluation that underlies the theory is based on the one presented for transitional semantics. That model equated each computation step with an axiom or a call to a remote rule set.

When this model is converted to relational semantics, it still treats the evaluation of every remote premise and axiom as a computation step. Most local premises do not correspond to computation steps; they describe sub-evaluations which can be broken down into smaller steps. However, not all computation steps correspond to remote premises or axioms; some correspond to a certain class of local premises.

Some conclusions correspond to computation steps; the most obvious cases being the conclusions of axioms, already mentioned. Another example is the conclusion of the following rule. The rule evaluates two sub-phrases, and then picks the value of the first:

$$\frac{e \vdash p_1, m_0 \Rightarrow v_1, m_1 \quad e \vdash p_2, m_1 \Rightarrow v_2, m_2}{e \vdash p_1 \text{ before } p_2, m_0 \Rightarrow v_1, m_2}$$

The selection of the first value is a step in the evaluation, separate from the sub-evaluations described by the premises.

By contrast, the rule for the if construct on expressions gives a choice of whether to regard the selection of the second value as a step, by an analogy with

the above example, or as a natural consequence of the left to right order of evaluation of the premises:

$$\frac{e \vdash p_1, m_0 \Rightarrow \text{tt}, m_1 \quad e \vdash p_2, m_1 \Rightarrow v, m_2}{e \vdash \text{if}(p_1, p_2, p_3), m_0 \Rightarrow v, m_2}$$

This choice arises whenever the rhs of the last premise is the same as that of the conclusion. Usually the conclusion won't be a step. In this case the rule is called **recursive**. The last premise of a recursive rule is called a recursive premise.

Remote recursive premises are just like other remote premises in that they each correspond to a computation step. Each local recursive premise also corresponds to a computation step, and in this they differ from other local premises. The intuition is that the rule is making a selection (or similar transformation), but is making this selection before the evaluation of the last premise instead of after.

A precise definition of the equivalence between a local recursive premise and a transition in the underlying finite automata is given in the next chapter. There it is presented as a proof of equivalence between the definitions given in this chapter and the last. However, the definition presented in this chapter is the result of the work in the next chapter.

The rule form presented above doesn't distinguish between recursive and non-recursive rules. In this thesis, an example rule will always be recursive if the rhs of its last premise is the same as that of its conclusion. Clearly an implementation of these ideas will need some way of distinguishing recursive rules from non-recursive ones in ambiguous cases.

6.1.3 Determinacy Restriction

Since this thesis only considers determinate specifications, the following determinacy restriction is required.

Any pair of relational rules in the same rule set:

$$\frac{E_1 \vdash^{NAME_1} P_1, \mu_0 \Rightarrow V_1, \mu_1, \dots, E_k \vdash^{NAME_k} P_k, \mu_{k-1} \Rightarrow V_k, \mu_k}{E_0 \vdash P_0, \mu_0 \Rightarrow V_{k+1}, M} \quad \theta$$

$$\frac{E'_1 \vdash^{NAME'_1} P'_1, \mu'_0 \Rightarrow V'_1, \mu'_1, \dots, E'_{k'} \vdash^{NAME'_{k'}} P'_{k'}, \mu'_{k'-1} \Rightarrow V'_{k'}, \mu'_{k'}}{E_0 \vdash P_0, \mu_0 \Rightarrow V'_{k'+1}, M'} \quad \theta'$$

must satisfy at least one of the following:

1. The rules don't unify.
2. Their side conditions are disjoint.
3. One rule is more general than the other.
4. There exists an i such that $1 \leq i \leq \min(k, k')$ and $\forall_{j=1}^i (E_j = E'_j)$, $\forall_{j=1}^i (P_j = P'_j)$, $\forall_{j=1}^i (NAME_j = NAME'_j)$, $\forall_{j=1}^{i-1} (V_j = V'_j)$, and V_i & V'_i don't unify.

This restriction ensures determinacy as follows. If either of the first two conditions are true, only one of the rules can match a given evaluation state. If the first two conditions are false but the third condition is true, then the definition of matching rules chooses the least general rule. If the first three conditions are false but the last is true, then either rule may match. However, when the i^{th} premise is evaluated, only one of the rules will match its result. At that point the correct rule may be substituted for an earlier incorrect choice without affecting the earlier evaluation.

6.1.4 Examples

Here is a relational rule for addition. The first two premises specify that the parameters of the operator are to be evaluated to values. The third is a call to a rule set that defines operations on integers:

$$\frac{e \vdash p_1, m_0 \Rightarrow v_1, m_1 \quad e \vdash p_2, m_1 \Rightarrow v_2, m_2 \quad \text{INTEGER} \quad \vdash v_1 + v_2, m_2 \Rightarrow v, m_3}{e \vdash p_1 + p_2, m_0 \Rightarrow v, m_3}$$

The following rules define a while loop. The iteration is achieved by using a recursive rule:

$$\frac{e \vdash p_1, m_0 \Rightarrow \text{tt}, m_1 \quad e \vdash p_2, m_1 \Rightarrow \text{nil}, m_2 \quad e \vdash \text{while } p_1 \text{ do } p_2, m_2 \Rightarrow \text{nil}, m_3}{e \vdash \text{while } p_1 \text{ do } p_2, m_0 \Rightarrow \text{nil}, m_3}$$

$$\frac{e \vdash p_1, m_0 \Rightarrow \text{ff}, m_1}{e \vdash \text{while } p_1 \text{ do } p_2, m_0 \Rightarrow \text{nil}, m_1}$$

The next rule defines an assignment operator:

$$\frac{\text{LOOKUP} \quad \vdash \text{lookup}(e, x), m \Rightarrow v', m}{e \vdash x := v, m \Rightarrow \text{nil}, m + (v', v)}$$

The following two rules define a linear lookup of an environment:

$$\frac{}{\vdash \text{lookup}(e \uparrow(x, v), x), m \Rightarrow v, m}$$

$$\frac{\vdash \text{lookup}(e, x), m \Rightarrow v', m}{\vdash \text{lookup}(e \uparrow(x', v), x), m \Rightarrow v', m} \quad \text{IDENT} \quad \vdash \text{eq}(x, x') \Rightarrow \text{ff}$$

The final two rules define a simple recursive function call. They assume a construct such as:

`func  $x \ x'$  =  $body$  in  $exp$  end`

which defines a function x with one argument x' , such that a phrase call $x \ v$ occurring in exp will evaluate to $body$ with v substituted for x' and x defined recursively. They also assume that the previous two rules define a rule set that can be called by a remote premise. The recursion is achieved by adding the definition of the function to the env of the appropriate premise.

$$\frac{e \uparrow(x, \text{closure}(e, x', p_1)) \vdash p_2, m_0 \Rightarrow v, m_1}{e \vdash \text{func } x \ x' = p_1 \text{ in } p_2 \text{ end}, m_0 \Rightarrow v, m_1}$$

$$\begin{array}{c}
e \vdash p, m_0 \Rightarrow v, m_1 \quad \text{LOOKUP} \quad \vdash \text{lookup}(e, x), m_1 \Rightarrow \text{closure}(e', x', p'), m_2 \\
\hline
e' \dagger(x', v) \dagger(x, \text{closure}(e', x', p')) \vdash p', m_2 \Rightarrow v', m_3 \\
\hline
e \vdash \text{call } x p, m_0 \Rightarrow v', m_2
\end{array}$$

6.1.5 Comparison with the Definition of Standard ML

The Definition of Standard ML [MTH90] is written in natural semantics. Natural semantics is similar to relational semantics, but its rules can't necessarily be given an operational reading. Indeed, the static semantics (type-checking) of Standard ML contains many such rules. However, the dynamic semantics (run-time semantics) is written entirely in relational semantics. It is interesting to examine both parts of the semantics to explore the limits of the definition of relational semantics.

Most of the rules in the dynamic semantics of Standard ML fit the forms allowed by the theory. An exception is the following pair of rules (nos. 114 and 112):

$$\begin{array}{c}
\frac{s, E \vdash \text{exp} \Rightarrow \text{ref}, s' \quad s', E \vdash p_2 \Rightarrow v, s'' \quad a \notin \text{Dom}(\text{mem of } s'')}{s, E \vdash \text{exp atexp} \Rightarrow a, s'' + \{a \mapsto v\}} \\
\\
\frac{E \vdash \text{exp} \Rightarrow \text{con} \quad \text{con} \neq \text{ref} \quad E \vdash \text{atexp} \Rightarrow v}{E \vdash \text{exp atexp} \Rightarrow (\text{con}, v)}
\end{array}$$

In the notation used here, these become:

$$\begin{array}{c}
\text{MEM} \\
\frac{e \vdash p_1, m_0 \Rightarrow \text{ref}, m_1 \quad e \vdash p_2, m_1 \Rightarrow v, m_1 \quad \vdash \text{next}(m_2), m_2 \Rightarrow v', m_3}{e \vdash p_1 p_2, m_0 \Rightarrow v', m_2 \dagger(v', v)} \\
\\
\text{IDENT} \\
\frac{e \vdash p_1, m_0 \Rightarrow v_1, m_1 \quad \vdash \text{diff}(v_1, \text{ref}), m_1 \Rightarrow \text{tt}, m_2 \quad e \vdash p_2, m_2 \Rightarrow v_2, m_3}{e \vdash p_1 p_2, m_0 \Rightarrow \text{cons } (v_1, v_2), m_3}
\end{array}$$

The first rule only holds if the value of p_1 is `ref`. The second rule only holds if that isn't the case. However, the first rule can test this by making the result of the first premise be `ref`, while the second has to put in an explicit test. Thus

these two rules don't fit the determinacy restriction. The obvious modification is to separate the test in the first rule from the first premise.

Another minor difference is in the handling of value phrases. The dynamic semantics of Standard ML includes an axiom that translates a numeral into the corresponding integer. This thesis allows such rules, but the preferred approach is to make integers be value phrases. Value phrases aren't evaluated. This approach makes animations more natural (although a similar effect could be achieved using the display attributes introduced in Section 10.1.1), and makes languages specifications more concise.

The static semantics of Standard ML contains several rules that aren't admitted by the theory. One class of rule that wouldn't be allowed is illustrated by Rule 27:

$$\frac{C + VE \vdash \text{valbind} \Rightarrow VE}{C \vdash \text{rec } \text{valbind} \Rightarrow VE}$$

In the notation used here, this becomes:

$$\frac{e \upharpoonright v \vdash p \Rightarrow v}{e \vdash \text{rec } p \Rightarrow v}$$

The premise makes sense in an inferential interpretation of the rule: it holds if there is a value v such that the evaluation holds. However, operationally the rule makes no sense. The normal operational reading would say that p is evaluated in the environment $e \upharpoonright v$ – but v isn't known until p is evaluated!

Another rule that isn't admitted by the theory is Rule 13:

$$\frac{C \vdash \text{exp} \Rightarrow \text{exn}}{C \vdash \text{raise } \text{exp} \Rightarrow \tau}$$

In the notation used here, this becomes:

$$\frac{e \vdash p_1, m \Rightarrow \text{exn}, m}{e \vdash \text{raise } p_1, m \Rightarrow v, m}$$

The result, v , is a variable that doesn't occur anywhere else in the rule. The idea is that this rule doesn't generate the actual value but only a placeholder which

must be filled in later. The filling in occurs when a rule specifies that two phrases must give the same value. Rule 11 is an example:

$$\frac{C \vdash \text{exp} \Rightarrow \tau \quad C \vdash \text{ty} \Rightarrow \tau}{C \vdash \text{exp} : \text{ty} \Rightarrow \tau}$$

In the notation used here, this becomes:

$$\frac{e \vdash p_1, m_0 \Rightarrow v, m_1 \quad e \vdash p_2, m_1 \Rightarrow v, m_2}{e \vdash p_1 : p_2, m_0 \Rightarrow v, m_2}$$

In this rule v is determined by e and p_2 ; it then constrains the result of p_1 to be this value. This isn't explicit in the form of the rule, but has to be inferred from the context. It is the opposite of the relational interpretation, which would have v determined by e and p_1 and constraining the result of p_2 !

6.1.6 Summary

The model of evaluation that I use for relational semantics is equivalent to the one presented for transitional semantics in the previous chapter. In relational semantics, each remote premise and each local recursive premise corresponds to a computation step, as does each conclusion of a non-recursive rule. This model of evaluation gives a reasonable interpretation to existing relational semantics such as the dynamic semantics of Standard ML.

6.2 Relational Evaluation Histories

In relational semantics a complete evaluation is presented as a tree of instances of relational rules. Each local premise at each node unifies with the conclusion of the instance at the corresponding child node. The lhs of the instance at the root of the tree is the program that the evaluation describes, and the rhs of that instance is the value and final memory of the evaluation. The tree is called a **inference tree**.

For a theory of animation we need a semantic representation of a program in mid-evaluation, not just of a complete evaluation. Such a representation is suggested by the operational reading of a relational rule. This reading says that to evaluate the lhs, we evaluate each of the sub-expressions in order and then construct the rhs:

$$\frac{e \vdash p_1, m \Rightarrow \text{nil}, m' \quad e \vdash p_2, m' \Rightarrow \text{nil}, m''}{e \vdash p_1; p_2, m \Rightarrow \text{nil}, m''}$$

This reading extends to the construction of an inference tree. To evaluate a program, we first place a matching rule at the root of the tree. Then we recursively evaluate the first premise, match the rhs of the complete sub-evaluation against the rhs of the first premise, and repeat this process for the other premises. Finally we construct the rhs of the root. Thus the tree grows in a depth-first, left-to-right fashion:

$$\frac{\frac{\dots}{e \vdash p_1, m \Rightarrow \text{nil}, m'} \quad \frac{\dots}{e \vdash p_2, m' \Rightarrow \text{nil}, m''}}{e \vdash p_1; p_2, m \Rightarrow \text{nil}, m''}$$

This suggests that a program in mid-evaluation can be represented by a partially instantiated inference tree. Since an inference tree grows in a depth-first left-to-right order, there must be a point in the tree that represents the current extent of the evaluation process. I call this point the **current sequent**. All

variables before the current sequent must be instantiated; all those after it must be uninstantiated.

In the evaluation model that underlies the theory, only some sequents correspond to computation steps. These sequents are called **valid sequents**. The current sequent must always be a valid sequent.

In the evaluation model a computation step corresponds to a remote premise, a recursive local premise or the conclusion of a non-recursive rule. The definition of validity puts this another way. Remote recursive premises and trivial recursive premises are not valid, but the conclusion of the same node "takes their place" as representing a computation step. Also, if a conclusion corresponds to a recursive premise, then its place is taken by the conclusion of the parent node. This is because the evaluation of a recursive rule is complete when the evaluation of its recursive premise is complete. Phrasing the definition of validity this way makes it easier to define and reason about complete relational evaluations. (It also corresponds to the implementation of The Animator Generator.)

As with transitional semantics, I call this construct an **evaluation history**, so that when necessary we can distinguish between this construct and the process of evaluation. Evaluation histories are called **evaluations** when it is unambiguous to do so.

An alternative presentation of the theory could use a simpler definition of a partial inference tree by including more information in the definition of a computation step. The definition of an evaluation history would then be expressed as those partial inference trees that could be generated by repeated applications of a step. This approach would make some parts of the theory harder to express, and as the information has to be somewhere there seems to be little point in moving it around.

6.2.1 Definition

A **relational evaluation** of an evaluation state (p, e, m) is a non-empty finite tree of instances of relational rules, with a particular sequent called the **current sequent**, such that the tree satisfies the properties given below. The properties use the following definitions:

1. Sequents are **ordered** by depth-first traversal, with the conclusion of a node following its premises and with local premises occurring before their corresponding subtrees.
2. A premise of an instance of a relational rule is **trivial** if it is local and has a subject that is a value. A trivial premises doesn't have a corresponding sub-evaluation.
3. The child node **corresponding** to the i^{th} non-trivial local premise of the instance at a parent node is the i^{th} child of the parent, counting both children and local premises from left to right.
4. The **current node** of a relational evaluation is the node that includes the current sequent.
5. A sequent of a relational evaluation is **matched** if it occurs before the current sequent, and is **unmatched** if it occurs after the current sequent. The current sequent is matched if it is a remote premise or a conclusion, and unmatched if it is a local premise. Intuitively a sequent is matched if it has been evaluated.
6. A sequent of a relational evaluation is **valid** if it is a non-recursive remote premise, a non-trivial local recursive premise or a conclusion that doesn't correspond to a recursive premise. Valid sequents correspond to computation steps.

The properties that a relational evaluation must satisfy are:

1. The *lst* of the instance at the root node must be (p, m, e) .
2. The side condition of each instance must be satisfied.
3. The rule instantiated at each node must be the one that strongly matches the *lst* of that node.
4. The current sequent must be valid.
5. All variables that don't occur in a matched sequent or in a premise that corresponds to a matched conclusion must be uninstantiated.
6. All matched non-trivial local premises must have children corresponding to them, and all other premises must not.
7. The rhs of a matched trivial premise must be the same as its lhs, which must be closed.
8. If the child corresponding to a matched non-trivial local premise isn't on the path from the root of the evaluation to the current node, this premise and the conclusion of the instance at the child node must be identical, and all terms in them must be closed.
9. If the child corresponding to a matched non-trivial local premise is on the path from the root of the evaluation to the current node, the *lst* of this premise must be the same as that of the instance at the child node, which must be closed, and the rhs of the premise must unify with that of the child.
10. The rhs of a matched remote premise q must be the result of a complete evaluation in the remote rule set of $lst(q)$, if $subject(q)$ isn't a value in the

remote rule set, and must be the same as $lhs(q)$ if $subject(q)$ is a value in the remote rule set. All terms in a matched remote premise must be closed.

The **subject**, **result**, **env**, etc. of a relational evaluation are those of the instance at the root of the tree.

The current sequent of a relational evaluation $\mathcal{R}$ is written $o(\mathcal{R})$, and the current node is written $O(\mathcal{R})$. The root node is sometimes written $root(\mathcal{R})$.

The **sub-evaluation** $\mathcal{R}[N]$ of a node N in a relational evaluation $\mathcal{R}$ is the subtree rooted at that node. If $o(\mathcal{R})$ is in $\mathcal{R}[N]$ then $o(\mathcal{R}[N]) = o(\mathcal{R})$, otherwise $o(\mathcal{R}[N]) = conclusion(N)$.

If q is a local premise then $\mathcal{R}[q]$ is the sub-evaluation corresponding to q . Similarly, $\mathcal{R}[i]$ is the sub-evaluation corresponding to the i^{th} premise of $root(\mathcal{R})$.

6.2.2 Initial and Complete Evaluations

An **initial relational evaluation** of an evaluation state (p, e, m) is a relational evaluation $\mathcal{R}$ of (p, e, m) such that:

1. If $root(\mathcal{R})$ doesn't have any non-trivial local premises or any valid premises, then $o(\mathcal{R}) = conclusion(root(\mathcal{R}))$.
2. If the first non-trivial premise q of $root(\mathcal{R})$ exists and is valid then $o(\mathcal{R}) = q$.
3. If q exists, is local and isn't valid, then $\mathcal{R}[q]$ is an initial relational evaluation of $lst(q)$.

A relational evaluation $\mathcal{R}$ is **complete** if $o(\mathcal{R}) = conclusion(root(\mathcal{R}))$.

Lemma 6.1 *A complete relational evaluation of an evaluation state (p, e, m) is unique.*

Proof: By double induction. The outermost induction is on the rule set graph; the inner induction is on the depth of the evaluation at a node of that graph.

Let the rule instantiated at the root of the evaluation be ρ :

$$\frac{E_1 \vdash P_1, \mu_0 \Rightarrow V_1, \mu_1, \dots, E_k \vdash P_k, \mu_{k-1} \Rightarrow V_k, \mu_k}{E_0 \vdash P_0, \mu_0 \Rightarrow V_{k+1}, M} \quad \theta$$

Base case: A rule set that doesn't refer to other rule sets; that is, a rule set that doesn't contain any rules with remote premises.

Base case: ρ has no non-trivial premises.

The determinacy restriction guarantees the uniqueness of ρ . Since $\mathcal{R}$ is complete, $o(\mathcal{R}) = \text{conclusion}(\text{root}(\mathcal{R}))$, and so all variables in ρ are instantiated. The instance of ρ is unique, since the instances of all variables in ρ are determined by (p, e, m) .

Induction step: ρ has at least one non-trivial premise.

Since $\mathcal{R}$ is complete, all variables in ρ are instantiated. The instances of E_1, P_1 and μ_0 are determined by e, p and m respectively. By induction, if the instances of E_i, μ_{i-1} and P_i are known, they have a unique evaluation subtree. Therefore the instances of V_i and μ_i are unique. The determinacy restriction ensures that no other rule applies at this point.

Hence if for some j , the instances of the $V_i, 1 \leq i \leq j$, are unique, then so are the instances of P_{i+1} and E_{i+1} . Similarly if the instances of the V_i , for $1 \leq i \leq k$, are unique, so are the instances of V_{k+1} and M . Hence the choice of rule and its instance are unique.

Induction step: A rule set that refers to other rule sets.

Base case: If a premise is remote, any evaluation of the lhs and env of the premise must be unique, by induction. Therefore the rhs of the premise must also be unique. The rest of the case proceeds as above.

Induction step: As above.

6.2.3 Summary

A relational evaluation history is a tree of instances of relational rules. It represents the history of an evaluation. The determinacy restriction on the permitted rule forms ensures that a complete evaluation of a program is unique. The definition of validity encapsulates the notion of a computation step.

6.3 Relational Computation Steps

This section defines a computation step as a map from one relational evaluation to another. It is based on the intuition that the evaluation process extends the evaluation history in a depth-first left-to-right order.

6.3.1 Extension

The **computation step** function for relational evaluations is defined in terms of a relational evaluation being an **extension** of another. Extension formalises the depth-first left-to-right growth of the tree during evaluation. Intuitively, one relational evaluation $\mathcal{R}'$ extends another $\mathcal{R}$ if $\mathcal{R}'$ could be the result of a sequence of computation steps applied to $\mathcal{R}$. Thus it must be possible to embed $\mathcal{R}$ in $\mathcal{R}'$.

Another way of defining this is to say that the part of $\mathcal{R}'$ that corresponds to $\mathcal{R}$ must be identical to $\mathcal{R}$ itself, and that the selected node of $\mathcal{R}'$ must come after that of $\mathcal{R}$ in the left-to-right depth-first ordering. This can be stated formally by the following recursive definition:

If $\mathcal{R}$ and $\mathcal{R}'$ are both relational evaluations of (p, e, m) , and there are j matched premises at the root of $\mathcal{R}$, then $\mathcal{R}'$ extends $\mathcal{R}$ if there are at least j

matched premises at the root of $\mathcal{R}'$ and for all i such that $1 \leq i \leq j$, q is the i^{th} premise at the root of $\mathcal{R}$, q' is the i^{th} premise at the root of $\mathcal{R}'$ and:

1. $o(\mathcal{R})$ is before $o(\mathcal{R}')$ in $\mathcal{R}'$.
2. If q is remote, then $q = q'$.
3. If q is local, then $q = q'$ and $\mathcal{R}[i] = \mathcal{R}'[i]$ if $O(\mathcal{R})$ isn't in $\mathcal{R}[i]$, and $\mathcal{R}'[i]$ extends $\mathcal{R}[i]$ otherwise.

6.3.2 Definition

The definition of a relational computation step combines the notion of one relational evaluation being an extension of another with the notion of validity. Formally, a **computation step** for relational evaluations is a function $\mathcal{E}_R$ such that if $\mathcal{R}$ is a relational evaluation then $\mathcal{E}_R(\mathcal{R})$ is an extension of $\mathcal{R}$ and there are no valid sequents between $o(\mathcal{R})$ and $o(\mathcal{E}_R(\mathcal{R}))$.

To be strict, $\mathcal{E}_R$ is a function from an evaluation to an equivalence class of congruent evaluations. This will be discussed in the Section 6.3.4.

6.3.3 Example

If the environment e binds 'a' to 2 and 'b' to 3, the following sequence is a series of relational evaluations of ('a' > 'b', e, m), such that each evaluation is reached from the previous one by an application of $\mathcal{E}_R$. The current sequent is boxed in each case:

$$\begin{array}{c}
 \text{LOOKUP} \\
 \hline
 \vdash \text{lookup}(e, 'a'), m \Rightarrow 2, m \\
 \boxed{e \vdash 'a', m \Rightarrow 2, m} \\
 \\
 \begin{array}{ccc}
 e \vdash 'a', m \Rightarrow 2, m & e \vdash 'b', m \Rightarrow v_2, m_2 & \text{INTEGER} \\
 & & \vdash 2 > v_2, m_2 \Rightarrow v, m_3
 \end{array} \\
 \hline
 e \vdash 'a' > 'b', m \Rightarrow v, m_3
 \end{array}$$

$$\begin{array}{c}
\text{LOOKUP} \quad \frac{\vdash \text{lookup}(e, 'a'), m \Rightarrow 2, m}{e \vdash 'a', m \Rightarrow 2, m} \quad \text{LOOKUP} \quad \frac{\vdash \text{lookup}(e, 'b'), m \Rightarrow 3, m}{e \vdash 'b', m \Rightarrow 3, m} \\
\hline
e \vdash 'a', m \Rightarrow 2, m \quad e \vdash 'b', m \Rightarrow 3, m \quad \text{INTEGER} \quad \vdash 2 > 3, m \Rightarrow v, m_3 \\
\hline
e \vdash 'a' > 'b', m \Rightarrow v, m_3
\end{array}$$

$$\begin{array}{c}
\text{LOOKUP} \quad \frac{\vdash \text{lookup}(e, 'a'), m \Rightarrow 2, m}{e \vdash 'a', m \Rightarrow 2, m} \quad \text{LOOKUP} \quad \frac{\vdash \text{lookup}(e, 'b'), m \Rightarrow 3, m}{e \vdash 'b', m \Rightarrow 3, m} \\
\hline
e \vdash 'a', m \Rightarrow 2, m \quad e \vdash 'b', m \Rightarrow 3, m \quad \text{INTEGER} \quad \vdash 2 > 3, m \Rightarrow \text{ff}, m \\
\hline
e \vdash 'a' > 'b', m \Rightarrow \text{ff}, m
\end{array}$$

6.3.4 Congruency

The result of $\mathcal{E}_R(\mathcal{R})$ isn't necessarily unique. For example, when extending a tree with a rule that strongly matches a `while` expression there is in general a choice of two legal rules. This is shown in the following trees.

In the first example, the rule for the true case of the `while` loop has been chosen. Eventually this choice will not be allowable, but at this stage the condition has not been fully evaluated, and an arbitrary decision has to be made.

$$\begin{array}{c}
\text{LOOKUP} \\
\hline
\vdash \text{lookup}(e, 'a'), m \Rightarrow \text{ff}, m \\
\boxed{e \vdash 'a', m \Rightarrow \text{ff}, m} \\
\\
\begin{array}{ccc}
e \vdash 'a', m \Rightarrow \text{ff}, m & e \vdash 'b', m \Rightarrow v_2, m_2 & \text{BOOLEAN} \\
& & \vdash \text{ff or } v_2, m_2 \Rightarrow v, m_3
\end{array} \\
\hline
e \vdash 'a' \text{ or } 'b', m \Rightarrow v, m_3 \\
\\
\begin{array}{ccc}
e \vdash 'a' \text{ or } 'b', m \Rightarrow \text{tt}, m_1 & e \vdash \text{nil}, m_1 \Rightarrow \text{nil}, m_2 & e \vdash \text{while } \dots, m_2 \Rightarrow \text{nil}, m_3
\end{array} \\
\hline
e \vdash \text{while } 'a' \text{ or } 'b' \text{ do nil}, m \Rightarrow \text{nil}, m_3 \\
\\
\underline{e \vdash \text{while } 'a' \text{ or } 'b' \text{ do nil}, m \Rightarrow \text{nil}, m_1} \\
\dots
\end{array}$$

The second example shows an evaluation at the same stage, but using the rule for the false case of the while loop:

$$\begin{array}{c}
\text{LOOKUP} \\
\hline
\vdash \text{lookup}(e, 'a'), m \Rightarrow \text{ff}, m \\
\boxed{e \vdash 'a', m \Rightarrow \text{ff}, m} \\
\\
\begin{array}{ccc}
e \vdash 'a', m \Rightarrow \text{ff}, m & e \vdash 'b', m \Rightarrow v_2, m_2 & \text{BOOLEAN} \\
& & \vdash \text{ff or } v_2, m_2 \Rightarrow v, m_3
\end{array} \\
\hline
e \vdash 'a' \text{ or } 'b', m \Rightarrow v, m_3 \\
\\
\begin{array}{c}
e \vdash 'a' \text{ or } 'b', m \Rightarrow \text{ff}, m_1 \\
\hline
e \vdash \text{while } 'a' \text{ or } 'b' \text{ do nil}, m \Rightarrow \text{nil}, m_3
\end{array} \\
\\
\underline{e \vdash \text{while } 'a' \text{ or } 'b' \text{ do nil}, m \Rightarrow \text{nil}, m_1} \\
\dots
\end{array}$$

The need to make arbitrary selections of this nature doesn't contradict the assertion that the relational specifications presented here are determinate; Lemma 6.1 has shown that a complete evaluation is unique. The apparent indeterminacy vanishes as the evaluation progresses. The notion of **congruency** is introduced to formalise this idea. Intuitively two relational evaluations are congruent if their matched sequents are identical, except for the right hand sides of matched local sequents on the path from the root of the tree to the current sequent. These right hand sides will be instantiated later in the evaluation.

Two relational evaluations $\mathcal{R}$ and $\mathcal{R}'$ are **congruent**, written $\mathcal{R} \cong \mathcal{R}'$, if

1. $lst(\mathcal{R}) = lst(\mathcal{R}')$, and if either evaluation is complete, then so is the other and $rhs(\mathcal{R}) = rhs(\mathcal{R}')$.
2. The number of matched premises at the root of $\mathcal{R}$ equals the number at the root of $\mathcal{R}'$.
3. If q is the i^{th} premise of $root(\mathcal{R})$ and q' is the i^{th} premise of $root(\mathcal{R}')$, then:
 - (a) If q is remote, then $q' = q$.
 - (b) If q is local, then $q' = q$ and $\mathcal{R}[i] \cong \mathcal{R}'[i]$.
 - (c) If q is local and is the last matched sequent, then $lst(q') = lst(q)$ and $\mathcal{R}[i] \cong \mathcal{R}'[i]$.
4. If $o(\mathcal{R})$ is a local recursive premise of the root of $\mathcal{R}$, then $o(\mathcal{R}')$ is a local recursive premise of the root of $\mathcal{R}'$ and $lst(o(\mathcal{R})) = lst(o(\mathcal{R}'))$.

Lemma 6.2 *If $\mathcal{R}$ and $\mathcal{R}'$ are initial relational evaluations of (p, e, m) , then $\mathcal{R} \cong \mathcal{R}'$.*

Proof: The proof is by induction on the structure of $\mathcal{R}$.

Base Case: $O(\mathcal{R}) = root(\mathcal{R})$.

$o(\mathcal{R})$ must be either the conclusion, a remote premise or a recursive premise. Only non-recursive trivial premises can be before $o(\mathcal{R})$. The same applies to $\mathcal{R}'$.

The rhs of each trivial premise before $o(\mathcal{R})$ is determined by (p, e, m) . Therefore if $o(\mathcal{R})$ is a premise, $lst(o(\mathcal{R}))$ is determined by (p, e, m) , and if $o(\mathcal{R})$ is the conclusion, $rhs(o(\mathcal{R}))$ is determined by (p, e, m) . The same applies to $\mathcal{R}'$.

Congruence follows immediately.

Induction Step: $o(\mathcal{R})$ is in a sub-evaluation corresponding to a local premise q of the root of $\mathcal{R}$.

Only trivial premises can occur before q . Therefore $lst(q)$ is determined by (p, e, m) . By the determinacy restriction, there must be a premise q' in $\mathcal{R}'$ such that $lst(q) = lst(q')$. By induction, $\mathcal{R}[q] \cong \mathcal{R}'[q']$. Therefore $\mathcal{R} \cong \mathcal{R}'$.

Lemma 6.3 *If $\mathcal{R}$ is a incomplete relational evaluation, $\mathcal{R}' = \mathcal{E}_R(\mathcal{R})$ and $\mathcal{R}'' = \mathcal{E}_R(\mathcal{R})$, then $\mathcal{R}' \cong \mathcal{R}''$.*

Proof: The cases of the definition of congruence are proved in order.

Case 1. The first part of this case is trivial. The second part can be proved using a similar argument to that used in the third case, and is left to the reader.

Case 2. Let there be j matched premises at the root of $\mathcal{R}$.

If there are only j premises in the rule at the root of $\mathcal{R}$, condition 3 of the definition of congruence holds trivially. If there are more than j premises at the root of $\mathcal{R}$, consider the possible cases of the j^{th} premise.

If the j^{th} premise is remote, it must be $o(\mathcal{R})$. By the determinacy restriction, if there are k non-recursive trivial premises after the j^{th} premise in $\mathcal{R}'$, there must be the same number in $\mathcal{R}''$, since the lhs of all such premises is determined by the variables instantiated in $\mathcal{R}$. Similarly, if there isn't a $j + k + 1^{th}$ premise, then there must be $j + k$ premises at the roots of both $\mathcal{R}'$ and $\mathcal{R}''$. If there is a $j + k + 1^{th}$ premise, then there must be $j + k + 1$ matched premises at the roots of both $\mathcal{R}'$ and $\mathcal{R}''$, for if the $j + k + 1^{th}$ premise is remote or recursive, it must be both $o(\mathcal{R}')$ and $o(\mathcal{R}'')$, and if it is local then $o(\mathcal{R}')$ must be in $\mathcal{R}'[j + k + 1]$ and $o(\mathcal{R}'')$ must be in $\mathcal{R}''[j + k + 1]$.

If the j^{th} premise is local and $\mathcal{R}[j]$ isn't complete, then by the definition of completeness $o(\mathcal{R}')$ must be in $\mathcal{R}'[j]$ and $o(\mathcal{R}'')$ must be in $\mathcal{R}''[j]$.

If the j^{th} premise is local and $\mathcal{R}[j]$ is complete, then by the definition of completeness $o(\mathcal{R}')$ can't be in $\mathcal{R}'[j]$ and $o(\mathcal{R}'')$ can't be in $\mathcal{R}''[j]$. The same argument applies as in the remote case.

Case 3. This case holds for all i such that $1 \leq i \leq j$, by the definitions of extension and $\mathcal{E}_R$. (A simple inductive proof is needed if the j^{th} premise is local.)

If there are $j + k + 1$ premises at the roots of $\mathcal{R}'$ and $\mathcal{R}''$, then those between the j^{th} and $j + k + 1^{\text{th}}$ must be non-recursive and trivial, as in case 3. Since the first j premises are identical, the envs and lhs's of the $j + k + 1^{\text{th}}$ premises are identical. If they are remote or recursive, they must be selected, and so the result follows. If they are local and non-recursive, then $\mathcal{R}'[j + k + 1]$ and $\mathcal{R}''[j + k + 1]$ must be initial evaluations of the 1st of the $j + k + 1^{\text{th}}$ premises, and so the result follows by Lemma 6.2.

6.3.5 Summary

A computation step is a function from an evaluation $\mathcal{R}$ to the equivalence class of of minimal extensions to $\mathcal{R}$. Congruency is a transient phenomenon, and it is convenient to treat computation steps as functions from evaluations to evaluations.

The definition of computation step implicitly inherits the model of evaluation used to define evaluations; the key part of both definitions being the definition of validity.

6.4 Chapter Summary

This chapter has defined the relational rule form that is admitted by the theory of animation. It has also defined relational evaluation histories and relational computation steps, based on the model of evaluation presented for transitional semantics in the previous chapter. These definitions use the associated ideas of validity and congruency.

Chapter 7

A Definition of Equivalence

In Chapter 6 I presented a definition of a relational computation step and claimed that it was based on the definition of a transitional computation step presented in Chapter 5. In this chapter I show that the two definitions are indeed equivalent. Readers who are interested in the practical aspects of this thesis but not the underlying theory may skip this chapter.

This equivalence is a direct equivalence. If the two definitions of computation step are repeatedly applied to the same initial program, in the context of equivalent transitional and relational specifications, then they will produce identical sequences of steps. An axiom in one sequence will correspond to an axiom at the same place in the other sequence, and a call to a remote rule set in one sequence will correspond to an identical call at the same place in the other. After each step the two semantic representations of the program in mid-evaluation will be equivalent.

The proof of equivalence of computation steps depends on the definition of equivalence of a transitional specification and a relational specification, and on the definition of equivalence of a transitional evaluation and a relational evaluation. These definitions share the same intuition of what a computation step is.

Two specifications are **equivalent** if there is a one-to-one correspondence between the rule sets in each specification such that each pair of corresponding rule sets **realise** each other. Intuitively, one rule set realises another if any symbolic evaluation of a phrase term in the second can be emulated in the first.

7.1 Equivalent Syntaxes

The discussion of admissible transitional rules in Section 5.1.3 observed that some abstract syntax terms in that formalism need repeated arguments. The example given was the **while** constructor, which contains the original sub-phrases at the same time as copies of those sub-phrases are being evaluated.

This means that the abstract syntax of a transitional specification may be slightly different to that of a relational specification for the same language. In this chapter I assume that there is no such difference. In other words I assume that the syntax for a relational specification also includes these extra arguments when required, even though they are redundant in that specification.

I make this assumption only to simplify the presentation. If the syntaxes did differ, the theory could be modified to use a map ω , called the **syntax map**, from the relational version to the transitional version. For example, a while constructor could be mapped by the definition $\omega(\text{while}(p_1, p_2)) = (\text{while}(p_1, p_2, p_1, p_2))$. All that would be required of the syntax map would be that it satisfied the restriction that for any substitution σ and syntax map ω , $\sigma \circ \omega = \omega \circ \sigma$.

7.2 Transitional Realisation of Relational Rule Sets

This section presents a definition of a transitional realisation of of relational rule sets. Each sequent in a relational rule that corresponds to computation step is mapped to a transitional rule by the realisation. This section also gives several examples of transitional realisations, and discusses how the idea of recursive premises was prompted by the definition of realisation.

7.2.1 Discussion

Consider a relational rule ρ :

$$\frac{E_1 \vdash^{NAME_1} P_1, \mu_0 \Rightarrow V_1, \mu_1, \dots, E_k \vdash^{NAME_k} P_k, \mu_{k-1} \Rightarrow V_k, \mu_k}{E_0 \vdash P_0, \mu_0 \Rightarrow V_{k+1}, M} \quad \theta$$

This symbolically describes the evaluation of (P_0, μ_0, E_0) . Typically each premise evaluates a sub-phrase of the main phrase, with either the last premise or the conclusion performing a final transformation.

If a transitional rule set realises ρ , then there must be a sequence of rules from that rule set that describe the same sub-evaluations as ρ . This sequence is also a symbolic evaluation of (P_0, μ_0, E_0) . The subjects of these transitional rules form a sequence of configuration expressions, C_i .

Since the two symbolic evaluations describe the same phrase, there must be a substitution σ that maps C_1 to P_0 . Also, since the sub-evaluations are the same in both specifications, it must be possible to extend σ to map C_i to appropriate terms formed from the components of ρ .

If ρ has no premises, then the corresponding transitional symbolic evaluation must be an axiom τ_1 such that $\sigma(\tau_1)$ is:

$$\frac{}{E_0 \vdash \sigma(C_0), \mu_0 \rightarrow V_{k+1}, M} \quad \sigma(\theta_{k+1})$$

An axiom is a single step in each specification, so this must be the only rule in the evaluation.

If ρ has more than one premise, and the first premise is local, then it is sufficient for the transitional symbolic sub-evaluation that corresponds to that premise to contain just one rule with a local premise. It must be possible to instantiate the 1st of the premise of the transitional rule to that of the premise of ρ .

More complicated sequences of rules could describe the same sub-evaluation, but in practice these rarely arise. I assume that the case given here covers all practical examples.

If ρ has more than one premise, and the first premise is a call to a remote rule set R' , then the first rule τ_1 of the corresponding transitional sub-evaluation must be a call to an equivalent rule set T' . Thus $\sigma(\tau_1)$ must be

$$\frac{NAME_i \quad E_i \vdash P_i, \mu_{i-1} \rightarrow V_i, \mu_i}{E_0 \vdash \sigma(C_i), \mu_{i-1} \rightarrow \sigma(C_{i+1}), \mu_i} \quad \sigma(\theta_i)$$

Since ρ describes the evaluation of a single construct, $result(\tau_1)$ should be constrained to be an instance of that construct. The simplest way of doing this is to require that $result(\tau_1)$ be less general than $subject(\tau_1)$ (that is, for $subject(\tau_1) \succ result(\tau_1)$). This constraint is also satisfied by transitional rules that have local premises. It yields a transitional symbolic evaluation that gradually evaluates sub-phrases of the main phrase, eventually performing a final transformation (described below). This is the same behaviour as a relational rule. In addition, this restriction will simplify some later work, particularly Lemma 7.4.

The above cases have considered the first premise of ρ . Other premises are treated in the same way, with the possible exception of the last premise.

If the last premise of ρ is not recursive, then τ_k is defined as above, and there must be an axiom τ_{k+1} that describes the same computation step as the conclusion of ρ . Thus $\sigma(\tau_{k+1})$ must be:

$$\frac{}{E_0 \vdash \sigma(C_{k+1}), \mu_{k+1} \rightarrow V_{k+1}, M} \quad \sigma(\theta_{k+1})$$

If the last premise of ρ is recursive and remote, then $(V_k, \mu_k) = (V_{k+1}, M)$ and so $\sigma(\tau_k)$ must be:

$$\frac{\frac{NAME_k}{E_k \vdash P_k, \mu_{k-1} \rightarrow V_k, \mu_k}}{E_0 \vdash \sigma(C_k), \mu_{k-1} \rightarrow V_{k+1}, M} \quad \sigma(\theta_k)$$

If the last premise of ρ is recursive and local, there must be a transitional rule with subject $\sigma(C_k)$ and result P_k . Then the result of a transitional evaluation of P_k will be the result of the whole evaluation of P_0 , as desired. So $\sigma(\tau_k)$ must be:

$$\frac{}{E_0 \vdash \sigma(C_k), \mu_{k-1} \rightarrow P_k, \mu_{k-1}} \quad \sigma(\theta_k)$$

7.2.2 Definition

This subsection formalises the approach discussed in the previous subsection.

A relational rule ρ :

$$\frac{\frac{NAME_1}{E_1 \vdash P_1, \mu_0 \Rightarrow V_1, \mu_1}, \dots, \frac{NAME_k}{E_k \vdash P_k, \mu_{k-1} \Rightarrow V_k, \mu_k}}{E_0 \vdash P_0, \mu_0 \Rightarrow V_{k+1}, M} \quad \theta$$

is **realised** by a transitional rule set if there exist a minimal substitution σ , configuration terms $C_1, \dots, C_k$, transitional rules $\tau_1, \dots, \tau_k$ and if necessary a configuration term C_{k+1} and a transitional rule τ_{k+1} such that:

1. $\sigma(C_1) = P_0$
2. For all i such that $1 \leq i < k$, the i^{th} premise of ρ is remote iff
 $subject(\tau_i) \succ result(\tau_i)$ and

$$\sigma(\tau_i) = \left(\frac{NAME_i \quad E_i \vdash P_i, \mu_{i-1} \rightarrow V_i, \mu_i}{E_0 \vdash \sigma(C_i), \mu_{i-1} \rightarrow \sigma(C_{i+1}), \mu_i} \quad \sigma(\theta_i) \right)$$

and the i^{th} premise of ρ is local iff

$$\sigma(\tau_i) = \left(\frac{E_i \vdash P_i, \mu_{i-1} \rightarrow V_i, \mu_i}{E_0 \vdash \sigma(C_i), \mu_{i-1} \rightarrow \sigma(C_{i+1}), \mu_i} \quad \sigma(\theta_i) \right)$$

where θ_i is the side condition of τ_i .

3. If ρ is not recursive, then τ_k is defined as in case 2, and there exists an axiom τ_{k+1} such that:

$$\sigma(\tau_{k+1}) = \left(\frac{}{E_0 \vdash \sigma(C_{k+1}), \mu_k \rightarrow V_{k+1}, M} \quad \sigma(\theta_{k+1}) \right)$$

4. If ρ has a remote recursive premise then $\sigma(C_{k+1}) = V_k = V_{k+1}$, there is no τ_{k+1} and $\sigma(\tau_k)$ is defined as in case 2.
5. If ρ has a local recursive premise then $E_k = E_0$, there is no C_{k+1} or τ_{k+1} and

$$\sigma(\tau_k) = \left(\frac{}{E_0 \vdash \sigma(C_k), \mu_{k-1} \rightarrow P_k, \mu_{k-1}} \quad \sigma(\theta_k) \right)$$

6. For all i such that $1 \leq i \leq k+1$, if θ holds, so does $\sigma(\theta_i)$.
7. For all i such that $1 \leq i \leq k+1$, there does not exist a transitional rule τ such that $\tau_i \succ \tau$.

The **realisation** of the relational rule consists of the C_i , the τ_i and σ .

A relational rule set is **realised** by a transitional rule set if each rule in the relational rule set is realised by the transitional rule set.

For example, the relational rule given above for addition is realised by the three transitional rules for addition. This gives the following instances of the C_i :

$$C_1 = c_1 + p_2$$

$$C_2 = v_1 + c_2$$

$$C_3 = v_1 + v_2$$

7.2.3 Examples

These are the rules for addition given in Sections 6.1.4 and 5.1.3:

Transitional:

$$\frac{e \vdash c_1, m \rightarrow c'_1, m'}{e \vdash c_1 + p_2, m \rightarrow c'_1 + p_2, m'}$$

$$\frac{e \vdash c_2, m \rightarrow c'_2, m'}{e \vdash v_1 + c_2, m \rightarrow v_1 + c'_2, m'}$$

$$\frac{\text{INTEGER} \quad \vdash v_1 + v_2, m \rightarrow v, m'}{e \vdash v_1 + v_2, m \rightarrow v, m'}$$

Relational:

$$\frac{e \vdash p_1, m_0 \Rightarrow v_1, m_1 \quad e \vdash p_2, m_1 \Rightarrow v_2, m_2 \quad \text{INTEGER} \quad \vdash v_1 + v_2 \Rightarrow v}{e \vdash p_1 + p_2, m_0 \Rightarrow v, m_2}$$

Both sets of rules use local premises to specify that the arguments should be evaluated and then make a call to a remote rule set.

The obvious corresponding rules for division would not satisfy this definition of realisation, since the third transitional rule would have a side condition that the second argument could not be 0. To cope with these rules the definition would have to be extended to make side conditions of transitional rules correspond to remote premises of relational rules.

These are the rules for assignment:

Transitional:

$$\frac{\text{LOOKUP} \quad \vdash \text{lookup}(e, x), m \rightarrow v', m}{e \vdash x := v, m \rightarrow \text{nil}, m + (v', v)}$$

Relational:

$$\frac{\text{LOOKUP} \quad \vdash \text{lookup}(e, x), m_0 \Rightarrow v', m_1}{e \vdash x := v, m_0 \Rightarrow \text{nil}, m_0 + (v', v)}$$

Both rules call a remote rule set.

The following rules define a linear lookup of an identifier in an environment.

Transitional:

$$\frac{}{\vdash \text{lookup}(e \dagger(x, v), x), m \rightarrow v, m}$$

$$\frac{}{\vdash \text{lookup}(e \dagger(x', v), x), m \rightarrow \text{lookup}(e, x), m} \quad \text{IDENT} \quad \vdash \text{eq}(x, x') \rightarrow \text{ff}$$

Relational:

$$\frac{}{\vdash \text{lookup}(e \dagger(x, v), x), m_0 \Rightarrow v, m_0}$$

$$\frac{\vdash \text{lookup}(e, x), m_0 \Rightarrow v', m_1}{\vdash \text{lookup}(e \dagger(x', v), x), m_0 \Rightarrow v', m_1} \quad \text{IDENT} \quad \vdash \text{eq}(x, x') \Rightarrow \text{ff}$$

Both sets of rules traverse the list argument. The transitional rules do this with a transition from a list to its tail. The relational rules use a local premise.

The following rules describe a while loop. The relational rules differ from those in Section 6.1.4 in that they use the same four argument constructor as the transitional rules.

Transitional:

$$\begin{array}{c}
 \frac{e \vdash c_1, m \rightarrow c'_1, m'}{e \vdash \text{while}(p_1, p_2, c_1, p_2), m \rightarrow \text{while}(p_1, p_2, c'_1, p_2), m'} \\
 \\
 \frac{e \vdash c_2, m \rightarrow c'_2, m'}{e \vdash \text{while}(p_1, p_2, \text{tt}, c_2), m \rightarrow \text{while}(p_1, p_2, \text{tt}, c'_2), m'} \\
 \\
 \frac{}{e \vdash \text{while}(p_1, p_2, \text{tt}, \text{nil}), m \rightarrow \text{while}(p_1, p_2, p_1, p_2), m} \\
 \\
 \frac{}{e \vdash \text{while}(p_1, p_2, \text{ff}, p_2), m \rightarrow \text{nil}, m'}
 \end{array}$$

Relational:

$$\begin{array}{c}
 \frac{e \vdash p_1, m_0 \Rightarrow \text{tt}, m_1 \quad e \vdash p_2, m_1 \Rightarrow \text{nil}, m_2 \quad e \vdash \text{while}(p_1, p_2, p_1, p_2), m_2 \Rightarrow \text{nil}, m_3}{e \vdash \text{while}(p_1, p_2, p_1, p_2), m_0 \Rightarrow \text{nil}, m_3} \\
 \\
 \frac{e \vdash p_1, m_0 \Rightarrow \text{ff}, m_1}{e \vdash \text{while}(p_1, p_2, p_1, p_2), m_0 \Rightarrow \text{nil}, m_1}
 \end{array}$$

Both sets of rules use a local premise to specify that the condition should be evaluated first. The transitional rule for this corresponds to a premise that occurs in both relational rules. If the condition evaluates to **ff**, the loop evaluates to **nil**. If the condition evaluates to **tt**, the body of the loop is evaluated and the loop is repeated. This repetition is specified in the same way as the rules for lookup specify list traversal.

7.2.4 Recursive Premises Revisited

The definition of realisation shows clearly that a local recursive premise corresponds to a computation step. This section looks at recursive and non-recursive premises more closely.

Consider the relational rule ρ discussed in Section 6.1.2:

$$\frac{e \vdash p_1, m_0 \Rightarrow \text{tt}, m_1 \quad e \vdash p_2, m_1 \Rightarrow v, m_2}{e \vdash \text{if } p_1 \text{ then } p_2 \text{ else } p_3, m_0 \Rightarrow v, m_2}$$

If ρ is recursive, then the following pair of transitional rules realise ρ :

$$\frac{e \vdash c_1, m \rightarrow c'_1, m'}{e \vdash \text{if } c_1 \text{ then } p_2 \text{ else } p_3, m \rightarrow \text{if } c'_1 \text{ then } p_2 \text{ else } p_3, m'}$$

$$\frac{}{e \vdash \text{if } tt \text{ then } p_2 \text{ else } p_3, m \rightarrow p_2, m}$$

The second rule selects p_2 to be the sub-phrase that will, when evaluated, produce the rhs of the rule. This selection is a computation step, and is represented in a relational evaluation by selecting the recursive premise.

If ρ is not recursive, then the following rules realise ρ :

$$\frac{e \vdash c_1, m \rightarrow c'_1, m'}{e \vdash \text{if } c_1 \text{ then } p_2 \text{ else } p_3, m \rightarrow \text{if } c'_1 \text{ then } p_2 \text{ else } p_3, m'}$$

$$\frac{e \vdash c_2, m \rightarrow c'_2, m'}{e \vdash \text{if } tt \text{ then } c_2 \text{ else } p_3, m \rightarrow \text{if } tt \text{ then } c'_2 \text{ else } p_3, m'}$$

$$\frac{}{e \vdash \text{if } tt \text{ then } v \text{ else } p_3, m \rightarrow v, m}$$

The last of these rules is also a selection; this corresponds to the conclusion of the non-recursive version of ρ . It corresponds to computation step in both specifications.

7.2.5 Trivial Premises

The following lemma is used several times in the development of the theory. It shows that a trivial premise of an instance of a relational rule is not an computation step. A realisation of the relational rule will map the corresponding premise of the rule itself to a transitional rule, but that transitional rule will not appear in the evaluation of the program because the the subject of its premise may not be instantiated to a value.

Lemma 7.1 (The Trivial Lemma) *If the i^{th} premise of an instance of a relational rule ρ is trivial and non-recursive, the substitution instantiating ρ is σ ,*

the substitution used in the transitional realisation of ρ , S_ρ , is σ' , and the configurations in S_ρ are $C_1, \dots, C_k$, then $\sigma \circ \sigma'(C_i) = \sigma \circ \sigma'(C_{i+1})$.

Proof:

If the i^{th} premise is trivial, it must be local. Therefore $\sigma'(C_{i+1}) = \sigma'(C_i)[V_i/P_i]$. But $\sigma(P_i) = \sigma(V_i)$, and the result follows.

7.2.6 Summary

A transitional rule set realises a relational rule set if every evaluation described by the relational set is also described by the transitional set. The transitional realisation of a relational rule consists of a set of transitional rules, a set of configuration terms, and a minimal substitution that satisfy the conditions of Section 7.2.2. These conditions ensure that the two evaluations contain the same computation steps in the same order.

7.3 Relational Realisation of Transitional Rule Sets

Defining sufficient conditions for a relational specification to realise a transitional specification is more complicated. The approach that I use is to create a tree of partially consistent stacks that correspond to the possible sequences of rules that would be instantiated in an evaluation of a given construct, and to show that for each path through the tree there exists a relational rule that describes the same symbolic evaluation.

A **partially consistent stack** is like a consistent stack except that the top instance of the stack may have a local premise or an uninstantiated rhs. In a transitional tree, an instance with a local premise represents the complete

sub-evaluation of its 1st. For example, the transitional rules given for **while** in Section 7.2.3 included one rule to evaluate the conditional and two rules to evaluate the configuration left after the evaluation of the conditional. These form a transitional tree with three nodes; the first rule as the root and the other two as leaves. Thus a local premise in a transitional rule serves the same purpose as a local premise in a relational rule.

7.3.1 Partially Consistent Stacks

A **partially consistent stack** is a stack of instances of transitional rules such that:

1. Each instance in the stack has a local premise, with the possible exception of the top instance.
2. The conclusion of each instance in a stack is identical to the premise of the instance below it (if any).
3. The side condition (if any) of each instance is satisfied.
4. Each instance in the stack must be an instance of the rule that strongly matches the 1st of that instance.
5. If the top instance is an instance of a rule of the form

$$\frac{E_1 \vdash C_1, \mu \rightarrow \nu, \mu'}{E_0 \vdash C_0, \mu \rightarrow C_2, M} \text{NAME}$$

then either:

- (a) ν is a variable.
- (b) E_0, C_0 and μ are closed terms, C_1 isn't a value in the remote rule set and (ν, μ') is the rhs of a complete evaluation of E_1, C_1 and μ in the remote rule set.

- (c) E_0 , C_0 and μ are closed terms, C_1 is a value in the remote rule set,
 $\nu = C_1$ and $\mu' = \mu$.

6. If the top instance has a local premise, then the result and rhm of the premise are variables.

The usual conventions for selecting elements of a stack apply to partially consistent stacks.

A **partially consistent sequence** is a sequence of partially consistent stacks that satisfies the same conditions as a consistent stack, namely for stack s in the sequence:

1. The lhs of s is the same as the rhs of the previous stack.
2. The env of s is the same as that of the previous stack.

Every consistent stack or sequence is also partially consistent.

7.3.2 Sequences as Instances of Stacks

A partially consistent sequence S is an **instance** of a partially consistent stack s if there exists a substitution σ such that the following conditions are satisfied:

1. If there are k instances $t_1, \dots, t_k$ with local premises in s then each stack in S contains k instances $t'_1, \dots, t'_k$ in the same order as the t_i such that $\sigma(t_i) = t'_i$ for all i .
2. If $top(s)$ is an axiom or has a remote premise then $top(head(S)) = \sigma(top(s))$ and:
 - (a) If $subject(top(s)) \succ result(top(s))$, or $result(top(s))$ is a value, then $length(S) = 1$.

- (b) If $\text{subject}(\text{top}(s)) \neq \text{result}(\text{top}(s))$ and $\text{result}(\text{top}(s))$ is not a value, then for each s_i in $\mathcal{S}$, $\#s_i \geq \#s$, and either no stacks in $\mathcal{S}$ of height $\#s$ have a value as the result of their top instances or $\text{last}(\mathcal{S})$ is the only stack that does.

A rule or instance that satisfies the conditions of case 2b is called **expansive**, because if a stack has such an instance or rule at the top then instances of that stack may be of arbitrary length. Similarly, a rule or instance that satisfies the conditions of case 2a is called **non-expansive**. Rules and instances with local premises are neither expansive nor non-expansive.

For example, the following sequence of two stacks:

$$\left\{ \begin{array}{l} \text{INTEGER} \\ \frac{\vdash 2 + 3 \rightarrow 5}{e \vdash 2 + 3, m \rightarrow 5, m} \\ \\ \frac{e \vdash 2 + 3, m \rightarrow 5, m}{e \vdash (2 + 3) + 4, m \rightarrow 5 + 4, m} \\ \\ \frac{e \vdash (2 + 3) + 4, m \rightarrow 5 + 4, m}{e \vdash ((2 + 3) + 4) + 5, m \rightarrow (5 + 4) + 5, m} \\ \hline \text{INTEGER} \\ \frac{\vdash 5 + 4 \rightarrow 9}{e \vdash 5 + 4, m \rightarrow 9, m} \\ \\ \frac{e \vdash 5 + 4, m \rightarrow 9, m}{e \vdash (5 + 4) + 5, m \rightarrow 9 + 5, m} \end{array} \right.$$

is an instance of the stack consisting just the first rule for addition:

$$\boxed{\frac{e \vdash c_1, m \rightarrow c'_1, m'}{e \vdash c_1 + p_2, m \rightarrow c'_1 + p_2, m'}}$$

A partially consistent sequence $\mathcal{S}$ is an **instance** of a partially consistent sequence $\mathcal{S}'$ if $\mathcal{S}$ can be partitioned into instances of each stack in $\mathcal{S}'$, in order.

7.3.3 Transitional Trees

A **transitional tree** of a transitional state (c, m, e) is empty if c is a value. Otherwise it is the smallest tree of partially consistent stacks of height 1 satisfying the following properties:

1. The 1st of the root stack is (c, m, e) .
2. The env of each stack equals the env of its parent stack.
3. The lhs of a stack is an instance of the rhs of its parent stack.
4. Each leaf stack contains an instance t such that either $subject(t) \neq result(t)$ or $result(t)$ is a value.
5. If the instance t at a node has the form:

$$\frac{E_1 \vdash \gamma, \mu \rightarrow \gamma', \mu'}{E_0 \vdash C, \mu \rightarrow C[\gamma'/\gamma], \mu'} \quad \theta$$

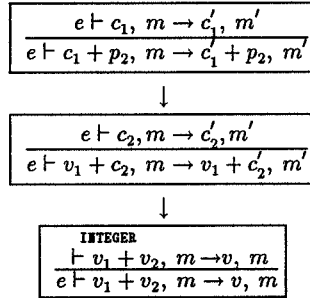
then that node has 1 child for each transitional rule that weakly matches $(C[v/\gamma], \mu', E_0)$, for any v , and each such rule is instantiated by 1 of those children.

6. If the instance t at a node is an axiom or has a remote premise, then that node has 1 child for each transitional rule that weakly matches $rhs(t)$, and each such rule is instantiated by 1 of those children.

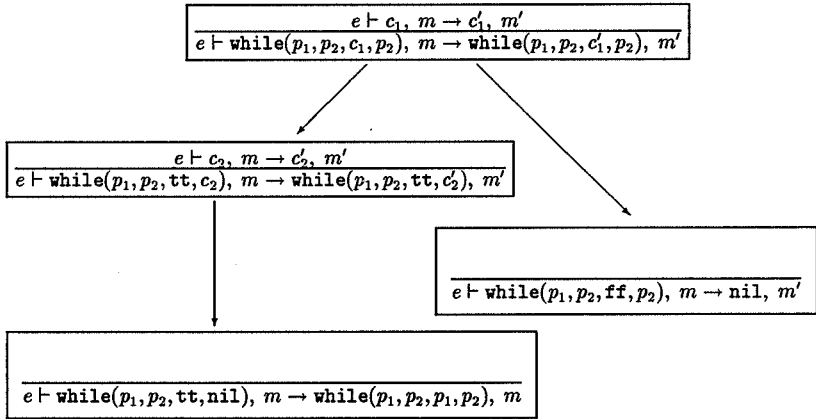
The transitional tree of a transitional rule τ is the transitional tree of the env and lhs of τ .

If a transitional evaluation $\mathcal{T}$ is an instance of a path from the root of a transitional tree, then that path is called the **tree path** of $\mathcal{T}$, and is written $\mathcal{S}_{\mathcal{T}}$.

For example, the transitional tree for $p_1 + p_2$, e and m , using the rules given in Section 5.1.3, is:



The transitional tree for $\text{while}(p_1, p_2, p_1, p_2)$, e and m , using the rules given in Section 5.1.3, is:



The fourth case of the definition of transitional trees ensures that the trees are finite. The exact definition given here may be more strict than necessary, but it does allow a wide range of practical examples.

As an example of how it may be more strict than necessary, consider the following rule in a language where expressions are defined in a separate rule set from statements:

$$\frac{e \vdash^{\text{EXP}} p_1, m \rightarrow \text{tt}, m}{e \vdash \text{while } p_1 \text{ do } p_2, m \rightarrow p_2; \text{while } p_1 \text{ do } p_2, m}$$

The combination of this rule and the rules for the sequencing operator realise the relational rules for the true case of the `while` operator. However, the reverse won't be true, since the definition of relational realisation of transitional rule sets uses transitional trees, and the above rule forms a transitional tree on its own.

Lemma 7.2 *If $\mathcal{T}$ is a transitional evaluation of (c, m, e) , and τ is the transitional rule that strongly matches (c, m, e) , then $\mathcal{T}$ is an instance of a path from the root of the transitional tree of τ .*

Proof: By induction on the length of $\mathcal{T}$. Let s be the stack at the root of the transitional tree. By definition, s contains only τ .

Base Case: $\text{length}(\mathcal{T}) = 1$.

$\text{bottom}(\text{head}(\mathcal{T}))$ is an instance of τ by the definition of transitional evaluation, and so $\text{head}(\mathcal{T})$ is an instance of s .

Induction Step: $\text{length}(\mathcal{T}) > 1$.

The proof proceeds by cases of τ .

1. If τ is expansive, then $\mathcal{T}$ is an instance of s , since all the stacks in a transitional evaluation must have a non-zero height and only the last stack may be of height 1 and have a value as its result.
2. If τ is non-expansive, then $\text{head}(\mathcal{T})$ is the unique prefix of $\mathcal{T}$ that is an instance of s , by the definition of instantiation. $\text{result}(\text{head}(\mathcal{T}))$ is an instance of $\text{result}(\tau)$, and so the root of the transitional tree must have one child that strongly matches $\text{result}(\text{head}(\mathcal{T}))$, by the definition of transitional trees and the existence of further stacks in $\mathcal{T}$. Let this child be $\mathcal{N}$. By induction, $\text{last}(\mathcal{T})$ is an instance of a unique path from the root of the subtree rooted at $\mathcal{N}$, and the result follows.
3. If τ has a local premise, s may be instantiated by a sequence of any length such that the bottom instance of each stack in this sequence is an instance of τ .

Let T' be the longest such prefix of T . If $\text{result}(\text{premise}(\text{bottom}(\text{last}(T'))))$ is not a value, then $T' = T$, since otherwise $\text{result}(\text{last}(T'))$ would be strongly matched by τ and so the next stack in T would have an instance of τ at the bottom. If $\text{result}(\text{premise}(\text{bottom}(\text{last}(T'))))$ is a value and there is another stack in T , then $\text{result}(T')$ must be strongly matched by one of the children of the root. Let this child be $\mathcal{N}$. By induction, the remainder of T is an instance of a unique path from the root of the subtree rooted at $\mathcal{N}$, and the result follows. If $\text{result}(\text{premise}(\text{bottom}(\text{last}(T'))))$ is a value and there isn't another stack in T , the result follows immediately.

7.3.4 Definition of Relational Realisation

With the above definitions, relational realisation can be defined as follows. This definition considers the same cases as the definition of transitional realisation.

A path S from the root of a transitional tree of (c, e, m) to a node in the tree is realised by a relational rule ρ :

$$\frac{E_1 \vdash P_1, \mu_0 \Rightarrow V_1, \mu_1, \dots, E_k \vdash P_k, \mu_{k-1} \Rightarrow V_k, \mu_k}{E_0 \vdash P_0, \mu_0 \Rightarrow V_{k+1}, M} \quad \theta$$

and a minimal substitution σ if the path can be partitioned into sequences $S_1, \dots, S_j$ such that $j \leq k + 1$ and the following conditions hold:

1. For all i such that $1 \leq i \leq k$, $length(S_i) \leq 1$.
2. $(c, m, e) = \sigma(P_0, \mu_0, E_0)$.
3. None of the rules in S have side conditions, except that the one in S_1 (if S_1 is not empty) may have a side condition θ' such that if θ' holds then so does $\sigma(\theta)$.
4. For every i such that $1 \leq i \leq \min(j, k)$, if q is the i^{th} premise of ρ , then one of the following cases holds:
 - (a) q is local and non-recursive and $\sigma(P_i)$ is a value iff S_i is empty.
 - (b) q is local and non-recursive and $\sigma(P_i)$ is not a value iff S_i contains an instance with the premise

$$\sigma(E_i) \vdash \sigma(P_i), \sigma(\mu_{i-1}) \rightarrow \gamma', \sigma(\mu_i)$$
 for some γ' .
 - (c) q is remote iff S_i contains an instance with the premise

$$\sigma(E_i) \vdash \overset{NAME_i}{\sigma(P_i)}, \sigma(\mu_{i-1}) \rightarrow \sigma(V_i), \sigma(\mu_i)$$
 - (d) q is local and recursive iff S_i contains the instance

$$\overline{\sigma(E_0) \vdash C, \sigma(\mu_{k-1}) \rightarrow \sigma(P_k), \sigma(\mu_{k-1})}$$
 for some C .
 - (e) q is recursive iff S_i contains an instance that is a leaf of the transitional tree.
 - (f) q is remote and recursive iff $rhs(S_i) = \sigma(V_k, \mu_k)$.
5. If $j = k + 1$ then one of the following cases holds:

- (a) ρ is not recursive iff $\mathcal{S}_{k+1}$ contains the instance

$$\frac{}{\sigma(E_0) \vdash C, \sigma(\mu_{k-1}) \rightarrow \sigma(V_{k+1}), \sigma(M)}$$

for some C .

- (b) ρ is recursive iff $\mathcal{S}_{k+1}$ is empty.

A relational rule set realises a transitional tree if each path from the root of the tree to a leaf is realised by exactly one rule in the relational rule set.

A transitional rule set is realised by a relational rule set if the transitional tree of the 1st of each rule in the transitional rule set is realised by the relational rule set, and every relational rule realises at least one path from the root of one of these trees to a leaf of that tree. A relational rule may realise more than one path, and more than one relational rule may realise the same path.

For example, the transitional tree given in Section 7.3.3 for $(p_1 + p_2, e, m)$ is realised by a relational rule set containing the following rule:

$$\frac{e \vdash p_1, m_0 \Rightarrow v_1, m_1 \quad e \vdash p_2, m_1 \Rightarrow v_2, m_2 \quad \text{INTEGER} \quad \vdash v_1 + v_2, m_2 \Rightarrow v, m_3}{e \vdash p_1 + p_2, m_0 \Rightarrow v, m_3}$$

This is because there is only one path through the tree, and the stacks in this path are identical to those in the realisation of the relational rule.

The examples given in Section 7.2.3 for transitional realisation of relational rules also hold for relational realisation of transitional rules.

7.3.5 Summary

A relational rule set realises a transitional rule set if every evaluation described by the transitional set is also described by the relational set. Relational realisations are defined in terms of transitional trees, which represent the sequences of rules that might be used to evaluate a program that matches a

certain state. Each node in a path through a transitional tree describes part of an evaluation. Those nodes containing rules with local premises describe subevaluations of arbitrary length, as do leaf nodes that contain expansive rules. Other nodes describe a single step.

The path through a transitional tree that corresponds to a transitional evaluation is called the tree path of that evaluation, and the relational realisation of a tree path is a relational rule and a minimal substitution that satisfy the conditions of Section 7.3.4. These conditions ensure that the two evaluations contain the same computation steps in the same order.

7.4 Equivalent Evaluations

The previous sections have defined an equivalence between a relational specification and a transitional specification. They are equivalent if there is a one-to-one correspondence between the rule sets in each specification such that each pair of corresponding rule sets realise each other.

The following definition of equivalent evaluation histories extends the definition of equivalent specifications to cover instances of the rules involved. The idea is that a transitional evaluation $\mathcal{T}$ and a relational evaluation $\mathcal{R}$ are equivalent if they contain the same steps, using the appropriate models of evaluation.

The definition partitions $\mathcal{T}$ into sub-sequences in such a way that the i^{th} sub-sequence corresponds to the i^{th} premise $\text{root}(\mathcal{R})$. A remote premise of $\text{root}(\mathcal{R})$ corresponds to a single stack in $\mathcal{T}$ that has the same premise on top. For non-trivial, non-recursive local premises and for recursive local premises this correspondence involves an equivalence of sub-evaluations. Non-recursive trivial premises correspond to empty sub-sequences. Finally, a conclusion corresponds to a single stack with an axiom at the top.

7.4.1 Definition

If a transitional specification and a relational specification are equivalent, $\mathcal{T}$ and $\mathcal{R}$ are transitional and relational evaluations in these specifications and ρ is the rule instantiated at the root of $\mathcal{R}$, then $\mathcal{T}$ and $\mathcal{R}$ are **equivalent**, written $\mathcal{T} \sim \mathcal{R}$, if $\mathcal{T}$ can be partitioned into sub-sequences $\mathcal{S}_1, \dots, \mathcal{S}_{j+1}$, where j is the number of matched premises of the root of $\mathcal{R}$, such that the following conditions hold:

1. $lst(\mathcal{R}) = lst(\mathcal{T})$.
2. for all i such that $1 \leq i \leq j$, if q is the i^{th} premise of the root of $\mathcal{R}$ then the following conditions hold:
 - (a) q is trivial and non-recursive iff $\mathcal{S}_i$ is empty.
 - (b) q is local, non-trivial and non-recursive iff $[\mathcal{S}_i]$ is defined and $[\mathcal{S}_i] \sim \mathcal{R}[i]$.
 - (c) q is local and recursive iff $length(\mathcal{S}_i) > 0$, $bottom(head(\mathcal{S}_i))$ is an axiom, $rst(head(\mathcal{S}_i)) = lst(q)$, and if $length(\mathcal{S}_i) > 1$ or $\mathcal{R}[i]$ exists then $tail(\mathcal{S}_i) \sim \mathcal{R}[i]$,
 - (d) q is remote iff $length(\mathcal{S}_i) = 1$, $\#(head(\mathcal{S}_i)) = 1$ and $premise(bottom(head(\mathcal{S}_i))) = q$.
3. $o(\mathcal{R})$ is the local recursive premise of $root(\mathcal{R})$ iff $bottom(head(\mathcal{S}_{j+1}))$ is an axiom, $length(\mathcal{S}_{j+1}) = 1$ and $rst(head(\mathcal{S}_{j+1})) = lst(o(\mathcal{R}))$.
4. If ρ is recursive and $o(\mathcal{R})$ is not the local recursive premise of $root(\mathcal{R})$, then $\mathcal{S}_{j+1}$ is empty.
5. If ρ is recursive then $conclusion(root(\mathcal{R}))$ is matched iff $rhs(\mathcal{T}) = rhs(\mathcal{R})$.
6. ρ is not recursive, and $conclusion(root(\mathcal{R}))$ is matched iff $length(\mathcal{S}_{j+1}) = 1$, $bottom(head(\mathcal{S}_{j+1}))$ is an axiom and $conclusion(bottom(head(\mathcal{S}_{j+1}))) = conclusion(root(\mathcal{R}))$.

7. If ρ is not recursive, and $\text{conclusion}(\text{root}(\mathcal{R}))$ is not matched, then $\mathcal{S}_{j+1}$ is empty.

Lemma 7.3 *If a transitional specification and a relational specification are equivalent then an initial relational evaluation $\mathcal{R}$ of (p, m, e) exists iff there exists an initial transitional evaluation $\mathcal{T}$ of (p, m, e) . If they exist, then $\mathcal{T} \sim \mathcal{R}$.*

The proof requires the result that if $\mathcal{R}'$ and $\mathcal{T}'$ are complete evaluations of a state (p, m, e) in a remote rule set, then $\text{rhs}(\mathcal{R}') = \text{rhs}(\mathcal{T}')$. This will follow from Proposition 7.5, which is proved at the end of this chapter. However, the proof of Proposition 7.5 uses this lemma. It would be tedious to prove the desired result in a separate lemma. The easier alternative is to regard the remaining lemmas and propositions of this chapter as a single inductive proof on the rule set graph. The induction step of this proof is as presented here; the base case is the same except that all references to remote premises and rule sets are ignored.

Proof, part 1: If $\mathcal{T}$ exists then $\mathcal{R}$ exists. By the definition of initial transitional evaluations, $\text{length}(\mathcal{T}) = 1$. The proof is by induction on the depth of $\text{head}(\mathcal{T})$.

Base Case: $\# \text{head}(\mathcal{T}) = 1$.

$\text{head}(\mathcal{T})$ consists of an instance t of a rule τ . Let σ be the minimal substitution that instantiates τ to t . By the definition of equivalent specifications, there exists a relational rule ρ that realises the path $\mathcal{S}_\tau$ that contains just the root of the transitional tree of $\text{lst}(\tau)$, and ρ strongly matches (p, m, e) . Let σ' be the substitution in this realisation.

By the definition of relational realisation, $\sigma'(\rho)$ must have zero or more trivial non-recursive local premises that either precede a premise that isn't trivial, non-recursive and local (and that may be followed by premises of any kind) or that constitute all the premises of $\sigma'(\rho)$. Let the first non-trivial premise (if any) be the $j + 1^{\text{th}}$ premise, and let the $j + 1^{\text{th}}$ premise of ρ (if it exists) be q' .

Consider cases of t :

1. If t has a remote premise q then q' must exist and be remote, and $\sigma'(lst(q')) = lst(premise(\tau))$, by the definition of relational realisation.

$$\begin{array}{ccc} \rho : \frac{\dots q' \dots}{E_0 \vdash P_0, \mu_0 \Rightarrow \dots} & \xrightarrow{\sigma'} & \tau \\ & & \downarrow \sigma \\ & & t : \frac{\dots q \dots}{e \vdash p, m \rightarrow \dots} \end{array}$$

Since $\mathcal{F}(lst(q')) \subset \mathcal{F}(lst(\rho)) \cup \bigcup_{i=1}^j \mathcal{F}(rhs(q_i))$, where q_i is the i^{th} premise of ρ , and $\sigma \circ \sigma'(lst(\rho)) = (p, m, e)$, it follows that $\sigma \circ \sigma'(lst(q')) = lst(q)$. The remote rule sets are equivalent, so $\sigma \circ \sigma'(rhs(q')) = rhs(q)$. The determinacy restriction ensures that this applies for all admissible ρ .

Let $\mathcal{R} = \sigma \circ \sigma'(\rho)$. If ρ is recursive and q' is the recursive premise, then $o(\mathcal{R}) = conclusion(root(\mathcal{R}))$. Otherwise $o(\mathcal{R})$ is the $j + 1^{th}$ premise of $root(\mathcal{R})$. Therefore $\mathcal{R}$ exists.

2. If τ is an axiom then either q' exists and is a local recursive premise or q' doesn't exist and ρ is not recursive, by the definition of relational realisation. In the first case, let q'' be the $j + 1^{th}$ premise of $root(\mathcal{R})$. If q'' is trivial, then $o(\mathcal{R}) = conclusion(root(\mathcal{R}))$, and if q'' isn't trivial then $o(\mathcal{R}) = q''$. In the second case, $o(\mathcal{R}) = conclusion(root(\mathcal{R}))$. Therefore $\mathcal{R}$ exists.

Induction Step: $\#head(T) > 1$.

$bottom(head(T))$ is an instance t of a rule τ , and t has a local premise q . Let σ be the substitution that instantiates τ to t . By the definition of equivalent specifications, there exists a relational rule ρ that realises the path $\mathcal{S}_\tau$ that contains just the root of the transitional tree of the $lst(\tau)$, and ρ strongly matches (p, m, e) . Let σ' be the substitution in this realisation.

By the definition of relational realisation, $\sigma'(\rho)$ must have zero or more trivial non-recursive local premises preceding a non-trivial non-recursive local premise,

possibly followed by other premises. Let the first non-trivial premise be the $j + 1^{th}$ premise, and let the $j + 1^{th}$ premise of ρ be q' .

Now, $\sigma'(lst(q')) = lst(premise(\tau))$, by the definition of relational realisation. Since $\mathcal{F}(lst(q')) \subset \mathcal{F}(lst(\rho)) \cup \bigcup_{i=1}^j \mathcal{F}(rhs(q_i))$, where q_i is the i^{th} premise of ρ , and $\sigma \circ \sigma'(lst(\rho)) = (p, m, e)$ it follows that $\sigma \circ \sigma'(lst(q')) = lst(q)$. The existence of T implies the existence of $[T]$, so by induction $\mathcal{R}[j + 1]$ exists. Therefore $\mathcal{R}$ exists.

Proof, part 2: If $\mathcal{R}$ exists then T exists. The proof is by induction on the depth of $\mathcal{R}$.

Let the rule instantiated at $root(\mathcal{R})$ be ρ , and let the substitution that instantiates it be σ . By the definition of equivalent specifications, there exists a transitional realisation $\mathcal{S}_\rho$ of ρ . Let the substitution, transitional rules and configuration terms in $\mathcal{S}_\rho$ be $\sigma', \tau_1, \dots, \tau_{k+1}$ and $C_1, \dots, C_{k+1}$ respectively.

Base Case: $O(\mathcal{R}) = root(\mathcal{R})$.

By the definition of initial relational evaluations and the restriction on this case of the proof, $o(\mathcal{R})$ is the first non-trivial premise of $root(\mathcal{R})$, if there is such a premise and it isn't a remote recursive premise, and $o(\mathcal{R}) = conclusion(root(\mathcal{R}))$ otherwise. Let there be j non-recursive trivial premises before $o(\mathcal{R})$ (there may also be a recursive premise before $o(\mathcal{R})$, by the definition of validity). By the Trivial Lemma, $\sigma \circ \sigma'(C_{j+1}) = \sigma \circ \sigma'(C_1)$. But $\sigma \circ \sigma'(C_1) = p$, so τ_{j+1} is the rule that strongly matches (p, m, e) , by the definition of transitional realisation.

1. If $o(\mathcal{R})$ is a remote premise and q' is the $j + 1^{th}$ premise of ρ , then τ_{j+1} has a remote premise and $\sigma'(lst(premise(\tau_{j+1}))) = lst(q')$, by the definition of transitional realisation.

$$\begin{array}{ccc}
 \rho : \frac{\dots q' \dots}{\dots} & \xleftarrow{\sigma'} & \tau_{j+1} \\
 \sigma \downarrow & & \\
 \text{root}(\mathcal{R}) : \frac{\dots o(\mathcal{R}) \dots}{e \vdash p, m \Rightarrow \dots} & &
 \end{array}$$

Since $\mathcal{F}(\text{lst}(\text{premise}(\tau_{j+1}))) \subset \mathcal{F}(\text{lst}(\tau_{j+1}))$ and $\sigma \circ \sigma'(\text{lst}(\tau_{j+1})) = (p, m, e)$, it follows that $\sigma \circ \sigma'(\text{lst}(\text{premise}(\tau_{j+1}))) = \text{lst}(o(\mathcal{R}))$. The remote rule sets are equivalent, so $\sigma \circ \sigma'(\text{rhs}(\text{premise}(\tau_{j+1}))) = \text{rhs}(o(\mathcal{R}))$. Therefore $\mathcal{T}$ exists (and the conditions required for equivalence are satisfied).

2. If $o(\mathcal{R}) = \text{conclusion}(\text{root}(\mathcal{R}))$, ρ is recursive and the recursive premise of ρ is remote, then τ_{j+1} has a remote premise, by the definition of transitional realisation. The argument of the previous paragraph shows that $\mathcal{T}$ exists (and that $\mathcal{T} \sim \mathcal{R}$).

3. If $o(\mathcal{R})$ does not fit the above cases, then τ_{j+1} is an axiom, and therefore $\mathcal{T}$ exists.

Induction Step: $O(\mathcal{R}) \neq \text{root}(\mathcal{R})$.

$O(\mathcal{R})$ must be in $\mathcal{R}[q]$, where q is a local, non-trivial, non-recursive premise of $\text{root}(\mathcal{R})$, and all premises of $\text{root}(\mathcal{R})$ before q must be trivial. Let there be j premises before q . Repeating the argument used in the base case, τ_{j+1} is the rule that strongly matches (p, m, e) .

Since q is a local premise, τ_{j+1} has a local premise, by the definition of transitional realisation. Since $\mathcal{F}(\text{lst}(\text{premise}(\tau_{j+1}))) \subset \mathcal{F}(\text{lst}(\tau_{j+1}))$ and $\sigma \circ \sigma'(\text{lst}(\tau_{j+1})) = (p, m, e)$, it follows that $\sigma \circ \sigma'(\text{lst}(\text{premise}(\tau_{j+1}))) = \text{lst}(q)$. By induction, since $[\mathcal{T}]$ exists it follows that $\mathcal{R}[q]$ exists and that $[\mathcal{T}] \sim \mathcal{R}[q]$. (It also follows that the conditions required for equivalence of $\mathcal{T}$ and $\mathcal{R}$ are satisfied).

Proof, part 3: If $\mathcal{R}$ and $\mathcal{T}$ exist then $\mathcal{T} \sim \mathcal{R}$. The proof is by induction on the depth of $\mathcal{R}$, and uses the same definitions as part 2.

Base Case: $O(\mathcal{R}) = \text{root}(\mathcal{R})$.

Let there be j non-recursive trivial premises before $o(\mathcal{R})$. Each of these premises corresponds to an empty sub-sequence of T . As shown in part 2, τ_{j+1} is the rule that strongly matches (p, m, e) .

1. If $o(\mathcal{R})$ is a remote premise, or if $o(\mathcal{R}) = \text{conclusion}(\text{root}(\mathcal{R}))$, ρ is recursive and the recursive premise of ρ is remote, then part 2 shows that $T \sim \mathcal{R}$.
2. If $o(\mathcal{R})$ is a local recursive premise q , or if $o(\mathcal{R}) = \text{conclusion}(\text{root}(\mathcal{R}))$, ρ is recursive and the recursive premise q of $\text{root}(\mathcal{R})$ is local, then τ_{j+1} is an axiom and $\sigma'(\text{rhs}(\tau_{j+1})) = \text{lhs}(q')$, where q' is the recursive premise of ρ , by the definition of transitional realisation. Since $\mathcal{F}(\text{rhs}(\tau_{j+1})) \subset \mathcal{F}(\text{lst}(\tau_{j+1}))$ and $\sigma \circ \sigma'(\text{lst}(\tau_{j+1})) = (p, m, e)$, it follows that $\sigma \circ \sigma'(\text{rhs}(\tau_{j+1})) = \text{lhs}(q)$. Therefore the conditions required for equivalence are satisfied.
3. If $o(\mathcal{R}) = \text{conclusion}(\text{root}(\mathcal{R}))$ and ρ is not recursive, then τ_{j+1} is an axiom and $\sigma'(\text{rhs}(\tau_{j+1})) = \text{rhs}(\text{conclusion}(\rho))$, by the definition of transitional realisation. Since $\mathcal{F}(\text{rhs}(\tau_{j+1})) \subset \mathcal{F}(\text{lst}(\tau_{j+1}))$ and $\sigma \circ \sigma'(\text{lst}(\tau_{j+1})) = (p, m, e)$, it follows that $\sigma \circ \sigma'(\text{rhs}(\tau_{j+1})) = \text{rhs}(o(\mathcal{R}))$. Therefore the conditions required for equivalence are satisfied.

Induction Step: $O(\mathcal{R}) \neq \text{root}(\mathcal{R})$.

This case is the same as the induction step of part 2.

7.4.2 Summary

The definition of equivalence for evaluation histories formalises the notion of which computation steps correspond to which that underlies the definitions of transitional and relational realisations. The lemma shows that initial evaluations of the same program are equivalent.

7.5 Computation Steps Preserve Equivalence

This section shows that $\mathcal{E}_R$ and $\mathcal{E}_T$ are equivalent, that is, evaluating a program will produce the same sequence of computation steps in both a transitional specification and an equivalent relational specification.

This result is shown by Proposition 7.5, which requires the following lemma.

Lemma 7.4 *If the following conditions hold:*

1. *A transitional specification and a relational specification are equivalent and T and $\mathcal{R}$ are equivalent evaluations in those specifications.*
2. *There are j matched premises at $\text{root}(\mathcal{R})$ and $S_1, \dots, S_{j+1}$ are the sub-sequences of T used to show that $T \sim \mathcal{R}$.*
3. *The relational rule instantiated at $\text{root}(\mathcal{R})$ is ρ .*
4. *The configuration terms and transitional rules in the transitional realisation of ρ are $C_1, \dots, C_l$ and $\tau_1, \dots, \tau_l$.*

then the following conditions hold:

1. *For all i such that $1 \leq i \leq k$, S_i is an instance of the stack s_i that contains just τ_i , where $k = j$ if S_{j+1} is empty and $k = j + 1$ otherwise.*
2. *If S_i is not empty, $\text{bottom}(\text{head}(S_i))$ is not an axiom and $\text{result}(\text{premise}(\text{bottom}(\text{last}(S_i))))$ exists and is a value, then $\text{result}(S_i)$ is an instance of C_{i+1} .*

Proof: If S_i is empty, the result follows immediately. If S_i is not empty, let σ be the minimal substitution that maps ρ to $\text{root}(\mathcal{R})$, and let σ' be the minimal

substitution in the transitional realisation of ρ . The proof proceeds by induction on i .

The first part of the proof shows that the result holds for non-empty S_i if $\text{subject}(S_i) = \sigma \circ \sigma'(C_i)$.

With this assumption, τ_i is the rule that strongly matches $\text{subject}(S_i)$, since $\sigma'(\text{subject}(\tau_i)) = C_i$ and the definition of transitional realisation guarantees that there isn't a rule τ such that $\tau_i \succ \tau$. Let q be the i^{th} premise of $\text{root}(\mathcal{R})$ and q' be the i^{th} premise of ρ , if they exist, and consider cases of q .

1. If q is remote, then $\sigma'(\text{premise}(\tau_i)) = q'$, τ_i is non-expansive and $\sigma'(\text{result}(\tau_i)) = \sigma'(C_{i+1})$, from the definition of transitional realisation. Also, $\text{length}(S_i) = 1$, $\# \text{head}(S_i) = 1$ and $\text{premise}(\text{bottom}(\text{head}(S_i))) = q$, from the definition of equivalence. Therefore $\sigma \circ \sigma'(\text{premise}(\tau_i)) = q$. Since $\mathcal{F}(\text{rhs}(\tau_i)) \subset \mathcal{F}(\text{lst}(\tau_i)) \cup \mathcal{F}(\text{rhs}(\text{premise}(\tau_i)))$ it follows that S_i is an instance of s_i and $\text{result}(S_i) = \sigma \circ \sigma'(C_{i+1})$.

2. If q is local and non-recursive, then $\sigma'(\text{premise}(\tau_i)) = q'$ from the definition of transitional realisation. Also, $[S_i] \sim \mathcal{R}[i]$, from the definition of equivalence. Therefore, for every s'_i in S_i , with the possible exception of $\text{last}(S_i)$, $\text{result}(\text{premise}(s'_i))$ is not a value. Therefore for every s'_i in S_i , $\text{bottom}(s'_i)$ is an instance of τ_i . It follows that S_i is an instance of s_i .

If $\text{result}(\text{premise}(\text{bottom}(\text{last}(S_i))))$ is a value and $\text{length}(S_i) = n$, for some n , then $\text{result}(S_i) = (\sigma \circ \sigma'(C_i))[\sigma_n(\gamma')/\sigma_n(\gamma)] \dots [\sigma_1(\gamma')/\sigma_1(\gamma)]$, where $\gamma = \text{subject}(\tau_i)$, $\gamma = \text{result}(\tau_i)$, $\sigma_n(\gamma') = \text{result}(\text{premise}(\text{bottom}(\text{last}(S_i))))$, $\sigma_1(\gamma) = \text{subject}(\text{premise}(\text{bottom}(\text{head}(S_i))))$ and $\sigma_i(\gamma) = \sigma_{i-1}(\gamma') = \text{subject}(\text{premise}(s'_i))$, by the form of τ_i . But $[S_i] \sim \mathcal{R}[i]$, so $\text{result}(q) = \sigma \circ \sigma'(\text{result}(\text{premise}(\text{bottom}(\text{last}(S_i)))))$. Hence $\text{result}(S_i) = (\sigma \circ \sigma'(C_i))[\text{result}(q)/\text{subject}(q)]$. Therefore $\text{result}(S_i) = \sigma \circ \sigma'(C_{i+1})$, from the definition of transitional realisation.

3. If q is local and recursive, then τ_i is an axiom and $\sigma'(rhs(\tau_i)) = lhs(q')$, by the definition of transitional realisation. Also, $bottom(head(S_i))$ is an axiom and if $length(S_i) > 1$ then $rhs(bottom(head(S_i))) = lhs(q)$, by the definition of equivalence. τ_i must be expansive, since it must appear in a path through a transitional tree, this path must be realised by a relational rule, and any axiom appearing in such a path must be expansive unless its result is a value, by the definition of relational realisation. It immediately follows that S_i is an instance of s_i .
4. If q does not exist, then τ_i is an axiom and $\sigma'(rhs(\tau_i)) = rhs(\rho)$, by the definition of transitional realisation. Also, $bottom(head(S_i))$ is an axiom and $rhs(S_i) = rhs(\mathcal{R})$, by the definition of equivalence. It immediately follows that S_i is an instance of s_i .

With this, a simple induction on i completes the proof of the lemma.

Base Case: q is the first non-trivial premise of $root(\mathcal{R})$.

There are $i - 1$ trivial premises before q . By the Trivial Lemma, $C_i = C_1$, and the definition of transitional realisation shows that $\sigma'(C_1) = subject(\rho)$. Therefore $subject(S_i) = \sigma \circ \sigma'(C_i)$, and the above result applies.

Induction Step: q is not the first non-trivial premise of $root(\mathcal{R})$.

By induction the lemma holds for the previous non-trivial premise of $root(\mathcal{R})$. Let that premise be the i^{th} . If it is remote, then $result(premise(bottom(last(S'_i))))$ must be a value, and if it is local, then $[S'_i] \sim \mathcal{R}[i']$ and $\mathcal{R}[i']$ is complete, so $result(premise(bottom(last(S'_i))))$ must again be a value. Therefore $result(S'_i) = \sigma \circ \sigma'(C_{i+1})$. By the definition of equivalent evaluations, $S_{i+1}, \dots, S_{i-1}$ are empty, and so $subject(S_i) = result(S'_i)$. The Trivial Lemma shows that $C_i = C_{i+1}$, and so the above result applies.

7.5.1 The Proof of the Proposition

The previous lemma, and Lemma 7.3 from the previous section, are used in the proof of the proposition itself. This proof also uses the fact that the minimal substitutions instantiating the rule at the root of a relational evaluation and the tree path of a transitional evaluation are uniquely determined by the programs that those evaluations describe. This fact follows from the restrictions on free variables in rules, the requirement that the rhs of a remote premise is the rhs of a complete (and therefore unique) evaluation in the remote rule set, and the restrictions that all terms in a transitional evaluation and terms before the current sequent in a relational evaluation must be closed.

Proposition 7.5 *If a transitional specification and a relational specification are equivalent, $\mathcal{T}$ and $\mathcal{R}$ are transitional and relational evaluations in these specifications and $\mathcal{T} \sim \mathcal{R}$ then $\mathcal{E}_R(\mathcal{R})$ exists iff $\mathcal{E}_T(\mathcal{T})$ exists. If they exist, then $\mathcal{E}_T(\mathcal{T}) \sim \mathcal{E}_R(\mathcal{R})$.*

The proof requires induction on the structure of the rule set graph. The proof for the induction step on this graph is given below; the base case is the same except that it omits the sections that deal with remote premises.

Proof, part 1: If $\mathcal{E}_T(\mathcal{T})$ exists then $\mathcal{E}_R(\mathcal{R})$ exists. Let $\mathcal{S}_1, \dots, \mathcal{S}_{j+1}$ be the sub-sequences of $\mathcal{T}$ used to show that $\mathcal{T} \sim \mathcal{R}$. The proof is by induction on the structure of $\mathcal{T}$. Let $\mathcal{S}_{\mathcal{E}_T(\mathcal{T})}$ be the tree path of $\mathcal{E}_T(\mathcal{T})$ and σ be the minimal substitution that maps $\mathcal{S}_{\mathcal{E}_T(\mathcal{T})}$ to $\mathcal{E}_T(\mathcal{T})$. Let ρ and σ' be the relational rule and minimal substitution in the relational realisation of $\mathcal{S}_{\mathcal{E}_T(\mathcal{T})}$. Let s be the stack added to $\mathcal{T}$ by $\mathcal{E}_T$.

Base Case: $\text{bottom}(\text{head}(\mathcal{S}_j))$ is not an axiom and either $\text{bottom}(\text{last}(\mathcal{T}))$ is an axiom or $\text{result}(\text{premise}(\text{bottom}(\text{last}(\mathcal{T}))))$ is a value.

The restriction that $\text{bottom}(\text{head}(\mathcal{S}_j))$ is not an axiom means that $\text{last}(\mathcal{S}_j)$ doesn't correspond to a sub-evaluation of $\mathcal{R}$, by the definition of equivalence. Hence we can safely consider cases of $\text{bottom}(\text{last}(T))$.

1. If $\text{bottom}(\text{last}(T))$ is an axiom, then $\mathcal{S}_{j+1} = \text{last}(T)$, $o(\mathcal{R})$ must be a local recursive premise and $\text{rst}(T) = \text{lst}(o(\mathcal{R}))$, by the definition of equivalent evaluations and the existence of $\mathcal{E}_T(T)$. Also, s is a transitional initial evaluation of $\text{rhs}(T)$ and $\text{env}(T)$, by the definition of $\mathcal{E}_T$. By Lemma 7.3 there exists an equivalent relational initial evaluation of $\text{rhs}(T)$ and $\text{env}(T)$. Therefore $\mathcal{E}_R(\mathcal{R})$ exists and $\mathcal{E}_T(T) \sim \mathcal{E}_R(\mathcal{R})$.

2. If $\text{bottom}(\text{last}(T))$ has a remote premise, then the j^{th} premise of $\text{root}(\mathcal{R})$ must be a remote premise. If $\text{bottom}(\text{last}(T))$ has a local premise, then the j^{th} premise of $\text{root}(\mathcal{R})$ must be a non-recursive local premise. In either case $\mathcal{S}_{j+1}$ must be empty, from the definition of equivalent evaluations. Assume that there are j' trivial premises of $\text{root}(\mathcal{E}_R(\mathcal{R}))$ immediately following the j^{th} premise. Let q' be the $j + j' + 1^{\text{th}}$ premise of ρ , if it exists.

Now consider cases of $\text{bottom}(s)$.

a. If $\text{bottom}(s)$ has a remote premise q , then q' exists and is remote, by the definition of relational realisation. If $\text{bottom}(s)$ has a local premise q , then q' exists and is local, non-recursive and non-trivial, also by the definition of relational realisation. In either case, $\sigma'(\text{lst}(q')) = \text{lst}(\text{premise}(\tau))$, again by the definition of relational realisation.

Now, $\sigma \circ \sigma'(\text{lst}(\rho)) = \text{lst}(T)$ and both σ and σ' are minimal. But the minimal substitution σ'' that instantiates ρ to $\text{root}(\mathcal{R})$ is uniquely determined by $\text{lst}(\mathcal{R})$, so $\sigma'' = \sigma \circ \sigma'$ over the domain of σ' .

$$\begin{array}{ccc}
\mathcal{S}_{\mathcal{E}_T(T)} & \xleftarrow{\sigma'} & \rho : \frac{\dots q' \dots}{\dots} \\
\sigma \downarrow & & \sigma'' \downarrow \\
\mathcal{E}_T(T) = T, s & & \text{root}(\mathcal{R}) \\
\text{premise}(\text{bottom}(s)) = q & &
\end{array}$$

Since $\mathcal{F}(\text{lst}(q')) \subset \mathcal{F}(\text{lst}(\rho)) \cup \bigcup_{i=1}^j \mathcal{F}(\text{rhs}(q_i))$, where q_i is the i^{th} premise of ρ , it follows that $\sigma \circ \sigma'(\text{lst}(q')) = \text{lst}(q)$. The determinacy restriction ensures that this applies for all admissible ρ .

If q' is remote, the remote rule sets are equivalent, so $\sigma \circ \sigma'(\text{rhs}(q')) = \text{rhs}(q)$. If ρ is recursive and q' is the remote recursive premise, then $o(\mathcal{E}_R(\mathcal{R})) = \text{conclusion}(\text{root}(\mathcal{E}_R(\mathcal{R})))$. Otherwise $o(\mathcal{E}_R(\mathcal{R})) = q$. Therefore $\mathcal{E}_R(\mathcal{R})$ exists and $\mathcal{E}_T(T) \sim \mathcal{E}_R(\mathcal{R})$.

If q' is local, then the sequence containing just $[s]$ is an initial transitional evaluation of $\text{lst}(q)$. Lemma 7.3 shows that there exists an (equivalent) initial relational evaluation of $\text{lst}(q)$. Therefore $\mathcal{E}_R(\mathcal{R})$ exists and $\mathcal{E}_T(T) \sim \mathcal{E}_R(\mathcal{R})$.

b. If $\text{bottom}(s)$ is an axiom, then either q' exists and is a local recursive premise or q' doesn't exist and ρ is not recursive, by the definition of relational realisation. In the second case, $o(\mathcal{E}_R(\mathcal{R})) = \text{conclusion}(\text{root}(\mathcal{E}_R(\mathcal{R})))$. In the other case, if q' is trivial then $o(\mathcal{E}_R(\mathcal{R})) = \text{conclusion}(\text{root}(\mathcal{E}_R(\mathcal{R})))$, and if q' isn't trivial then $o(\mathcal{E}_R(\mathcal{R})) = q'$. Therefore $\mathcal{E}_R(\mathcal{R})$ exists. Equivalence for this case is shown in part 2 of the proof.

Induction Step: Either $\text{bottom}(\text{head}(S_j))$ is an axiom or $\text{premise}(\text{bottom}(\text{last}(T)))$ exists and $\text{result}(\text{premise}(\text{bottom}(\text{last}(T))))$ is not a value.

1. If $\text{bottom}(\text{head}(S_j))$ is an axiom, then $\text{tail}(S_j) \sim \mathcal{R}[q]$, where q is the local recursive premise of $\text{root}(\mathcal{R})$, by the definition of equivalent evaluations. Since $\mathcal{E}_T(T)$ exists, it follows that $\mathcal{E}_T(\text{tail}(S_j))$ exists, from the definition of $\mathcal{E}_T$. By

induction, $\mathcal{E}_R(\mathcal{R}[q])$ exists and $\mathcal{E}_T(\text{tail}(\mathcal{S}_j)) \sim \mathcal{E}_R(\mathcal{R}[q])$. Therefore $\mathcal{E}_R(\mathcal{R})$ exists and $\mathcal{E}_T(T) \sim \mathcal{E}_R(\mathcal{R})$ by the definition of $\mathcal{E}_R$.

2. If $\text{bottom}(\text{head}(\mathcal{S}_j))$ is not an axiom, $\text{premise}(\text{bottom}(\text{last}(T)))$ exists and $\text{result}(\text{premise}(\text{bottom}(\text{last}(T))))$ is not a value, then $\text{premise}(\text{bottom}(\text{last}(T)))$ must be local. Therefore $[\mathcal{S}_j] \sim \mathcal{R}[q]$, by the definition of equivalent evaluations. Also, $\text{bottom}(s)$ must be an instance of the same rule as $\text{bottom}(\text{last}(\mathcal{S}))$, and so $\text{premise}(\text{bottom}(s))$ must be local. Since $\mathcal{E}_T(T)$ exists, it follows that $\mathcal{E}_T([\mathcal{S}_j])$ exists, from the definition of $\mathcal{E}_T$. By induction, $\mathcal{E}_R(\mathcal{R}[q])$ exists and $\mathcal{E}_T([\mathcal{S}_j]) \sim \mathcal{E}_R(\mathcal{R}[q])$. Therefore $\mathcal{E}_R(\mathcal{R})$ exists and $\mathcal{E}_T(T) \sim \mathcal{E}_R(\mathcal{R})$ by the definition of $\mathcal{E}_R$.

Proof, part 2: If $\mathcal{E}_R(\mathcal{R})$ exists then $\mathcal{E}_T(T)$ exists and $\mathcal{E}_T(T) \sim \mathcal{E}_R(\mathcal{R})$. The proof is by induction on the structure of $\mathcal{E}_R(\mathcal{R})$.

Base Case: The last common ancestor in $\mathcal{E}_R(\mathcal{R})$ of $O(\mathcal{R})$ and $O(\mathcal{E}_R(\mathcal{R}))$ is $\text{root}(\mathcal{E}_R(\mathcal{R}))$.

There are two main cases. In the first, $o(\mathcal{R})$ is the recursive local premise of $\text{root}(\mathcal{R})$, and $o(\mathcal{E}_R(\mathcal{R}))$ is in the subtree of $\mathcal{E}_R(\mathcal{R})$ that corresponds to $o(\mathcal{R})$. In the other case, $o(\mathcal{R})$ is or corresponds to one premise of $\text{root}(\mathcal{R})$ and $o(\mathcal{E}_R(\mathcal{R}))$ is another sequent of $\text{root}(\mathcal{R})$ or is in a subtree corresponding to another premise of $\text{root}(\mathcal{R})$. Thus in the first case $\text{root}(\mathcal{R})$ is the start of the computation step, and in the second it is in the middle.

1. If $o(\mathcal{R})$ is the local recursive premise q of $\text{root}(\mathcal{R})$, then $\text{bottom}(\text{last}(T))$ is an axiom, $\text{rst}(T) = \text{lst}(q)$, by the definition of equivalent evaluations. Since $\mathcal{E}_R(\mathcal{R})$ exists, it follows that $(\mathcal{E}_R(\mathcal{R}))[q]$ is an initial relational evaluation of $\text{lst}(q)$. By Lemma 7.3, there exists an equivalent initial transitional evaluation of $\text{env}(T)$ and $\text{rhs}(T)$. Therefore $\mathcal{E}_T(T)$ exists, and $\mathcal{E}_T(T) \sim \mathcal{E}_R(\mathcal{R})$, by the definition of $\mathcal{E}_T$.

2. If $o(\mathcal{R})$ is not the local recursive premise of $root(\mathcal{R})$, then $o(\mathcal{R})$ must either be a remote premise of $root(\mathcal{R})$ or the conclusion of a node that corresponds to a local premise of $root(\mathcal{R})$. In either case let this premise be the j^{th} premise of $root(\mathcal{R})$, and let there be j' non-recursive trivial premises of $root(\mathcal{E}_R(\mathcal{R}))$ immediately following the j^{th} premise.

Let ρ be the rule instantiated at $root(\mathcal{E}_R(\mathcal{R}))$ and let σ be the minimal substitution that maps ρ to $root(\mathcal{E}_R(\mathcal{R}))$. Let the rules and substitution in the transitional realisation of ρ be $\tau_1, \dots, \tau_k$ and σ' respectively. Let q' be the $j + j' + 1^{th}$ premise of ρ , if it exists. If q' exists, let q be the $j + j' + 1^{th}$ premise of $root(\mathcal{E}_R(\mathcal{R}))$.

By Lemma 7.4, there exists a minimal substitution σ'' that maps $\tau_1, \dots, \tau_{j+1}$ to T , and $result(T) = \sigma''(C_{j+1})$. By the Trivial Lemma, $\sigma''(C_{j+j'+1}) = \sigma''(C_{j+1})$, and so $\tau_{j+j'+1}$ is the rule that strongly matches $rhs(T)$ and $env(T)$.

$$\begin{array}{ccc}
 \rho : \frac{\dots q' \dots}{\dots} & \xleftarrow{\sigma'} & \tau_1, \dots, \tau_{j+1} \\
 \sigma \downarrow & & \sigma'' \downarrow \\
 root(\mathcal{E}_R(\mathcal{R})) : \frac{\dots q \dots}{\dots} & & T
 \end{array}$$

Now, $\sigma \circ \sigma'(lst(\tau_1)) = lst(T)$. But σ'' is uniquely determined by $lst(T)$. Therefore, since both σ and σ' are minimal, it follows that $\sigma'' = \sigma \circ \sigma'$ over the domain of σ' .

Consider cases of q' :

- a. If q' is a remote premise then $\tau_{j+j'+1}$ has a remote premise that refers to the same rule set. If q' is a local non-recursive premise then $\tau_{j+j'+1}$ has a local premise. By the definition of transitional realisation, $\sigma'(lst(premise(\tau_{j+j'+1}))) = lst(q')$ in either case. Since $\mathcal{F}(lst(premise(\tau_{j+j'+1}))) \subset \mathcal{F}(lst(\tau_{j+j'+1}))$, it follows that $\sigma \circ \sigma'(lst(premise(\tau_{j+j'+1}))) = lst(o(\mathcal{R}))$.

If q' is remote then $\sigma \circ \sigma'(rhs(premise(\tau_{j+j'+1}))) = rhs(o(\mathcal{R}))$, since the remote rule sets are equivalent. Therefore $\mathcal{T}$ exists and $\mathcal{E}_T(\mathcal{T}) \sim \mathcal{E}_R(\mathcal{R})$.

If q' is local and non-recursive then since $\mathcal{E}_R(\mathcal{R})$ exists, it follows that $(\mathcal{E}_R(\mathcal{R}))[q]$ is an initial relational evaluation of $lst(q)$. By Lemma 7.3, there exists an initial transitional evaluation of $lst(premise(\sigma''(\tau_{j+j'+1})))$. Therefore $\mathcal{E}_T(\mathcal{T})$ exists and $\mathcal{E}_T(\mathcal{T}) \sim \mathcal{E}_R(\mathcal{R})$, by the definition of $\mathcal{E}_T$.

If q' is local and recursive then $\tau_{j+j'+1}$ is an axiom and $\sigma'(rhs(\tau_{j+j'+1})) = lhs(q')$, by the definition of relational realisation. Since $\mathcal{F}(rhs(\tau_{j+j'+1})) \subset \mathcal{F}(lst(\tau_{j+j'+1}))$, it follows that $\sigma \circ \sigma'(rhs(\tau_{j+j'+1})) = lhs(q)$. Therefore $\mathcal{E}_T(\mathcal{T})$ exists and $\mathcal{E}_T(\mathcal{T}) \sim \mathcal{E}_R(\mathcal{R})$.

b. If q' doesn't exist, that is $o(\mathcal{E}_R(\mathcal{R})) = conclusion(root(\mathcal{E}_R(\mathcal{R})))$ and ρ is not recursive, then $\tau_{j+j'+1}$ is an axiom and $\sigma'(rhs(\tau_{j+j'+1})) = rhs(conclusion(\rho))$, by the definition of relational realisation. Since $\mathcal{F}(rhs(\tau_{j+j'+1})) \subset \mathcal{F}(lst(\tau_{j+j'+1}))$, it follows that $\sigma \circ \sigma'(rhs(\tau_{j+j'+1})) = rhs(o(\mathcal{E}_R(\mathcal{R})))$. Therefore $\mathcal{E}_T(\mathcal{T})$ exists and $\mathcal{E}_T(\mathcal{T}) \sim \mathcal{E}_R(\mathcal{R})$.

Induction Step: The last common ancestor in $\mathcal{E}_R(\mathcal{R})$ of $O(\mathcal{R})$ and $O(\mathcal{E}_R(\mathcal{R}))$ is in $(\mathcal{E}_R(\mathcal{R}))[q]$, where q is a non-trivial local premise of $root(\mathcal{E}_R(\mathcal{R}))$. Let q be the j^{th} premise of $root(\mathcal{E}_R(\mathcal{R}))$, and let $\mathcal{S}_j$ be the j^{th} sub-sequence of $\mathcal{T}$ used to show that $\mathcal{T} \sim \mathcal{R}$. Note that the $j + 1^{th}$ sub-sequence in that proof must be empty.

If q is non-recursive, then $[\mathcal{S}_j] \sim \mathcal{R}[q]$, by the definition of equivalent evaluations. Since $\mathcal{E}_R(\mathcal{R})$ exists and both $O(\mathcal{R})$ and $O(\mathcal{E}_R(\mathcal{R}))$ are in $(\mathcal{E}_R(\mathcal{R}))[q]$, $\mathcal{E}_R(\mathcal{R}[q])$ must exist, by the definition of $\mathcal{E}_R$. By induction, $\mathcal{E}_T([\mathcal{S}_j])$ exists and $\mathcal{E}_T([\mathcal{S}_j]) \sim \mathcal{E}_R(\mathcal{R}[q])$. Therefore $result(premise(bottom(last(\mathcal{T}))))$ can't be a value, and so $rhs(\mathcal{T})$ and $env(\mathcal{T})$ must be strongly matched by the rule instantiated by $bottom(last(\mathcal{T}))$. Let the new instance of this rule be t . Now

$lst(premise(t)) = lst(bottom(last(\mathcal{E}_T([S_j])))$). Therefore $\mathcal{E}_T(T)$ exists and $\mathcal{E}_T(T) \sim \mathcal{E}_R(\mathcal{R})$, by the definition of $\mathcal{E}_T$.

If q is recursive, then $tail(S_j) \sim \mathcal{R}[q]$. Since $\mathcal{E}_R(\mathcal{R})$ exists and both $O(\mathcal{R})$ and $O(\mathcal{E}_R(\mathcal{R}))$ are in $(\mathcal{E}_R(\mathcal{R}))[q]$, $\mathcal{E}_R(\mathcal{R}[q])$ must exist. By induction, $\mathcal{E}_T(tail(S_j))$ exists and $\mathcal{E}_T(tail(S_j)) \sim \mathcal{E}_R(\mathcal{R}[q])$. Therefore $\mathcal{E}_T(T)$ exists and $\mathcal{E}_T(T) \sim \mathcal{E}_R(\mathcal{R})$, by the definition of $\mathcal{E}_T$.

7.6 Chapter Summary

This chapter has defined an equivalence between transitional and relational specifications and evaluation histories, based on their underlying models of evaluation. With these definitions it has shown that the definitions of computation steps for the two formalisms are equivalent.

Chapter 8

Relational Animations

In the preceding chapters I have developed part of a theory of animation. This theory is defined in terms of relational semantics, and uses a definition of a computation that is based on the model of evaluation that underlies transitional semantics. In this model a computation step corresponds to a transition of an abstract machine. The motivation for using this model was that it is implicit in the definition of transitional semantics, and thus provides a strong semantic underpinning to the theory of animation.

In the chapter I discuss the suitability of computation steps for animation, and conclude that they are not as suitable as we would like. I then refine the definition of a computation step to give a definition which better fits the requirements of animation. I call this definition an **animation step**. I finish the chapter with a discussion of how this choice of step affects the choice of semantic formalism.

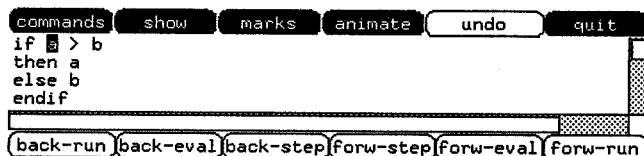
8.1 Computation Steps in Animations

The theory of animation presented in the previous chapters was developed from a semantic viewpoint. The definition of a step was based on the idea that the transitions used in transitional semantics correspond to an intuitive notion of a computation step, and that this definition would be suitable for an animator. However, this intuition must be tested. In practice, it gives steps that are too coarse for animation.

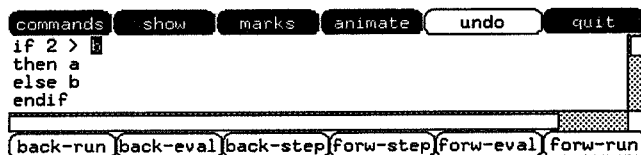
For example, the sequence of displays below animates a program using computation steps. At each step the program is displayed with part of the program highlighted. The highlighted part is called the **focus**, and is the next phrase to be evaluated.

In terms of the definition of a relational evaluation history, the focus is the subject of the sequent that corresponds to a computation step. That is, the focus of a relational evaluation $\mathcal{R}$ is the subject of the first remote premise, recursive premise or conclusion of a non-recursive rule before $o(\mathcal{R})$ (including $o(\mathcal{R})$). This definition translates the definition of validity back into the underlying notion of computation step (cf. Section 6.2).

The current sequent of the initial evaluation is a remote premise which looks up the value of 'a' in the environment:



The current sequent after the second step does the same for 'b':



The current sequent after the third step evaluates $2 > 3$ with a remote premise that calls the built-in operation on integers:

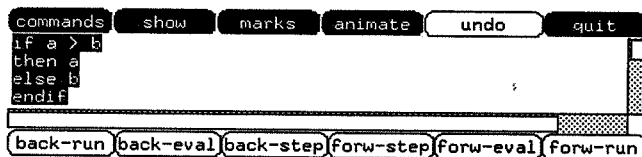


If we compare this sequence of displays with an established animator such as ZSTEP [Lie84], we find two problems:

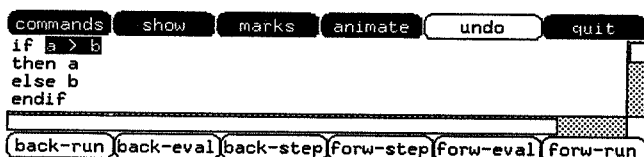
1. After a computation step has evaluated a sub-phrase, the animator moves straight to the sub-phrase that will be evaluated next. It doesn't highlight the result of that sub-phrase before moving on to the sub-phrase that will be evaluated next. The user's eye is drawn away from the current phrase before all the useful information about that phrase has been presented. This is particularly noticeable if the next sub-phrase is some distance away in the program text.
2. A computation step will jump straight to the next sub-phrase to be evaluated. If that sub-phrase is much more deeply nested than the current phrase, the jump could confuse the user. By contrast, ZSTEP will guide the user to the next sub-phrase by highlighting the intermediate sub-phrases in turn. This is less likely to confuse the user, and more likely to reveal the structure of the program.

For example, animating the same program with an animator similar to ZSTEP would produce the sequence of displays shown below. This animation produces eight displays instead of three, and corrects both the faults described above.

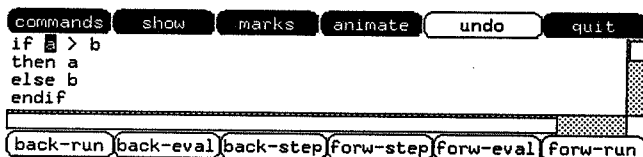
The animation starts by highlighting the while program:



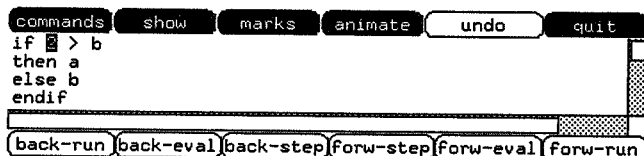
The first step begins a “zoom” towards the first sub-phrase to be evaluated:



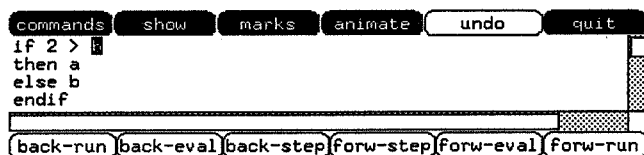
The next step zooms into the next phrase to be evaluated. The ‘zoom’ effect is pronounced with larger programs:



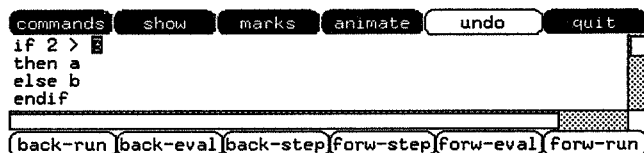
The next display shows the result of the computation step:



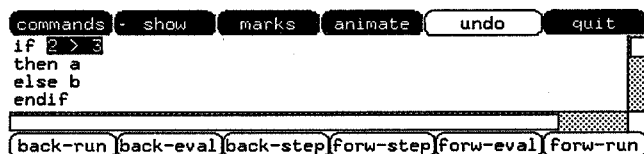
The fifth display shows the next phrase to be evaluated. There is no “zoom” in this case because ‘b’ is a sibling of ‘a’ in the abstract syntax tree:



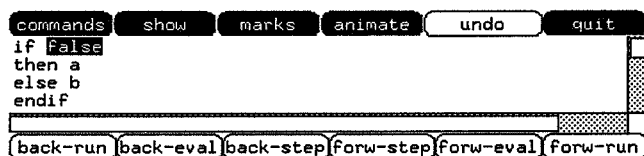
The next display shows the result of the computation step:



The next display shows the next phrase to be evaluated.



The last display shows the result of the computation step:



This comparison indicates that computation steps aren't suitable for animation (although they may have other uses). It also suggests that a definition of an evaluation step for animation should be based on the requirements of the users.

In the absence of a definitive behavioural study on which definition of an

evaluation step suits users best, I base my definition of an evaluation step on that provided by ZSTEP. I call this definition an animation step. I have generated an animator for Proc that incorporates this definition of step. A few people have used this animator and have found it satisfactory. In Chapter 10 I show how this definition can be parameterised on the definition of a view, and then in Chapter 11 I show that given an appropriate view it can mimic the steps produced by an existing Prolog animator. Chapter 13 includes a discussion of other possible definitions of a step.

To recap, the previous chapters have developed a definition of a computation step from purely semantic principles. This definition is not suitable for animations, although it may have other uses. An animation step must be based on the needs of users.

Although computation steps are not suitable for animation, the theory developed to define them can easily be extended to define an animation step. The theory developed in the preceding chapters is not wasted; it forms a useful introduction to a basis for the final definition.

8.2 Relational Animation Histories

Animation steps can be defined in terms of relational semantics. The forms of rules allowed remain unchanged. The definition of relational evaluation histories needs only slight alterations to support the new definition of step; These alterations are described below. The resulting construct is called a relational animation history, and is also called an **animation** when it is unambiguous to do so.

“Zooming in” to the next phrase requires only a small change to the definition of validity. This change lets the current sequent be any non-trivial

local premise, in addition to the existing possibilities. Then the definition of a step as a minimal extension will move the current sequent down a branch of the inference tree one node at a time, producing the desired zoom.

Highlighting the results of sub-phrases is harder. My solution is to add a boolean b to the definition of a relational evaluation. If the current sequent is the conclusion of a non-recursive node or is a remote premise, then its subject is the focus if $b = \text{ff}$ and its result is the focus if $b = \text{tt}$. So the sequents that correspond to computation steps that return a value are visited twice, once before the computation step and once after.

Also, the evaluation of a recursive remote premise can no longer be substituted by the conclusion of the node, as was done for relational evaluations. This is because if the conclusion were selected before the call to the remote rule set then the variables on the rhs of the premise would be instantiated too soon. Therefore the current sequent may be a remote recursive premise if $b = \text{ff}$. Similarly, the current sequent may be a conclusion that corresponds to a recursive premise if the current node is not recursive and $b = \text{ff}$.

One further refinement is needed. With the above definition an animation of a program could produce consecutive animation histories that were displayed the same way. The first would have a focus that was the subject of a local premise. The second would have a focus that was the subject of the conclusion of the corresponding node. Since the subjects of these sequents must be identical, by definition, the two animation histories will be displayed the same. This would clash with the requirement that applying an animation step should produce an animation history that is displayed differently from the original.

A similar clash could arise if the first animation history was as described in the previous paragraph, and the second had a focus that was the subject of the remote recursive premise of the corresponding node. Often the subject of a remote recursive premise of a rule is the same as the subject of the rule itself,

with some sub-phrases replaced with their values, as in step 6 of the above example. If these sub-phrases are already evaluated, the two subjects will be displayed the same way. Even if the subjects of the premise and the rule aren't related in this way, they will often still be displayed the same way, as in the example rule for evaluating identifiers. I assume that this situation will always produce identical displays.

This discussion leads to the following restriction:

In a relational animation, if the current sequent is a recursive remote premise or a conclusion, b may only be false if the current node has at least one non-trivial premise before the current sequent.

Thus in the above example steps 2 and 4 show the result of the remote recursive premise, skipping the display of its subject. The restriction does not apply to local recursive premises because selecting such a premise with an identical subject is a computation step, and allows animation of a potentially infinite loop.

8.2.1 Definition

A **relational animation** $(\mathcal{R}, b)$ of a program (p, m, e) is a possibly empty finite tree $\mathcal{R}$ of instances of relational rules with a particular sequent called the **current sequent** and a boolean b , satisfying the properties below. The properties use the same auxiliary definitions of **ordered**, **trivial**, **corresponding**, and **current node** as those for relational evaluations (see Section 6.2.1), augmented by the following:

- 5'. A sequent of a relational animation is **matched** if it occurs before the current sequent and is **unmatched** if it occurs after the current sequent.

The current sequent is matched if $b = \text{tt}$ and the current sequent is a

remote premise or a conclusion, and is unmatched if $b = \mathbf{ff}$ or the current sequent is a local premise.

- 6'. A sequent of a relational animation is **valid** if it is a remote premise, a non-trivial local premise, a conclusion of a non-recursive node or a conclusion that doesn't correspond to a recursive premise.

The properties that a relational animation $(\mathcal{R}, b)$ must satisfy are:

1. If $\mathcal{R}$ is non-empty then $lst(\mathcal{R}) = (p, m, e)$.
2. The side condition of each node must be satisfied.
3. The current sequent must be valid.
4. The rule instantiated at each node must be the one that strongly matches the lst of that node.
5. All variables that don't occur in a matched sequent or in a premise that corresponds to a matched conclusion must be uninstantiated.
6. All matched non-trivial local premises must have children corresponding to them, and all other premises must not.
7. The rhs of a matched trivial premise must be the same as its lhs, which must be closed.
8. If the child corresponding to a matched local non-trivial premise isn't on the path from the root of the animation to the current node, then this premise and the conclusion of the instance at the child node must be the same, and all terms in them must be closed.

9. If the child corresponding to a matched local non-trivial premise is on the path from the root of the animation to the current node, then the *lst* of this premise must be the same as those of the instance at the child node, which must be closed, and the rhs of the premise must unify with that of the child.
10. The rhs of a matched remote premise q must be the result of a complete animation in the remote rule set of $lst(q)$, if $subject(q)$ isn't a value in the remote rule set, and must be the same as $lhs(q)$ if $subject(q)$ is a value in the remote rule set. All terms in a matched remote premise must be closed.
11. If the current sequent is a remote recursive premise then $b = ff$.
12. If the current premise is a conclusion that corresponds to a recursive premise then $b = ff$.
13. If $b = ff$ then the current sequent must be the conclusion of a non-recursive node or a remote premise.
14. If $b = ff$ and the current sequent is a recursive remote premise or a conclusion, then the current node must include at least one non-trivial premise before the current sequent.

The current sequent of a relational animation is written $o(\mathcal{R}, b)$, and the current node is written $O(\mathcal{R}, b)$. The *root* of a relational animation $(\mathcal{R}, b)$ is $root(\mathcal{R})$.

The focus of a non-empty relational animation $(\mathcal{R}, b)$ is $subject(o(\mathcal{R}, b))$ if $o(\mathcal{R}, b)$ is a local premise or $b = ff$, and is $result(o(\mathcal{R}, b))$ if $o(\mathcal{R}, b)$ is a remote premise or a conclusion and $b = tt$. The focus of the empty relational animation of (p, m, e) is p .

Let $(\mathcal{R}, b)$ be a relational animation and $\mathcal{N}$ be a node in $\mathcal{R}$. Then the sub-animation $(\mathcal{R}, b)[\mathcal{N}]$ rooted at $\mathcal{N}$ is defined by cases:

1. If $O(\mathcal{R}, b)$ is in the subtree rooted at $\mathcal{N}$, the sub-animation is $(\mathcal{R}[\mathcal{N}], b)$.
2. If $O(\mathcal{R}, b)$ isn't in the subtree rooted at $\mathcal{N}$, the sub-animation is $(\mathcal{R}[\mathcal{N}], \text{tt})$.

8.2.2 Initial and Complete Animations

An **initial relational animation** is an empty tree. As defined above, the focus of the initial relational animation of (p, m, e) is p .

A relational animation $(\mathcal{R}, b)$ is **complete** if either $o(\mathcal{R}, b) = \text{conclusion}(\text{root}(\mathcal{R}))$ and $b = \text{tt}$ or p is a value.

Lemma 8.1 *A complete relational animation of a (p, m, e) is unique.*

This follows immediately from Lemma 6.1.

8.2.3 Summary

The definition of a relational animation is similar to that of a relational evaluation. The new definition of validity makes animation steps “zoom in” to the next phrase to be evaluated. The boolean variable makes them show the result of each phrase. The idea of the focus of an animation defines what an animator should display.

8.3 Relational Animation Steps

Animation steps are defined in terms of relational animations in much the same way that computation steps are defined in terms of relational evaluations. The differences are the addition of the boolean b and the definition of an initial animation as an empty tree. The new definition of validity affects the result of the definition, but not the definition itself.

8.3.1 Extension

If $(\mathcal{R}, b)$ and $(\mathcal{R}', b')$ are both relational animations of (p, e, m) , and there are j matched premises at the root of $\mathcal{R}$, then $(\mathcal{R}', b')$ **extends** $(\mathcal{R}, b)$ if at least one of the following conditions holds:

1. $\mathcal{R}$ is empty.
2. $\mathcal{R} = \mathcal{R}'$, $b = \text{ff}$ and $b' = \text{tt}$.
3. There are at least j matched premises at the root of $\mathcal{R}'$ and for all i such that $1 \leq i \leq j$, q is the i^{th} premise at the root of $\mathcal{R}$, q' is the i^{th} premise at the root of $\mathcal{R}'$ and:
 - (a) $o(\mathcal{R}, b)$ is before $o(\mathcal{R}', b')$ in $\mathcal{R}'$.
 - (b) If q is remote, then $q = q'$.
 - (c) If q is local, then $q = q'$ and $(\mathcal{R}, b)[i] = (\mathcal{R}', b')[i]$ if $O(\mathcal{R}, b)$ isn't in $(\mathcal{R}, b)[i]$, and $(\mathcal{R}', b')[i]$ extends $(\mathcal{R}, b)[i]$ otherwise.

$(\mathcal{R}', b')$ **minimally extends** $(\mathcal{R}, b)$ if there doesn't exist a relational animation $(\mathcal{R}'', b'')$ such that $(\mathcal{R}', b')$ extends $(\mathcal{R}'', b'')$ and $(\mathcal{R}'', b'')$ extends $(\mathcal{R}, b)$.

8.3.2 Definition

A **animation step** for relational animations is a function $\mathcal{A}_R$ such that if $(\bar{\mathcal{R}}, b)$ is a relational animation then $\mathcal{A}_R(\bar{\mathcal{R}}, b)$ is a minimal extension of $(\bar{\mathcal{R}}, b)$.

This definition has the property that every computation step is also an animation step.

8.3.3 Example

If the environment e binds 'a' to 2 and 'b' to 3, the following sequence is a series of relational animations of 'a' > 'b', m and e , such that each animation is reached from the previous one by an application of $\mathcal{A}_R$. The current sequent is boxed in each case.

After the first step, the animation selects the first sub-phrase to be evaluated:

$$\begin{array}{c}
 b = \text{ff} \\
 \hline
 \boxed{e \vdash 'a', m \Rightarrow v_1, m_1} \quad e \vdash 'b', m_1 \Rightarrow v_2, m_1 \quad \text{INTEGER} \vdash v_1 > v_2, m_2 \Rightarrow v, m_3 \\
 \hline
 e \vdash 'a' > 'b', m \Rightarrow v, m_3
 \end{array}$$

The second step evaluates the sub-phrase:

$$\begin{array}{c}
 \text{LOOKUP} \\
 \hline
 \vdash \text{lookup}(e, 'a'), m \Rightarrow 2, m \\
 \boxed{e \vdash 'a', m \Rightarrow 2, m} \\
 \hline
 e \vdash 'a', m \Rightarrow 2, m \quad e \vdash 'b', m \Rightarrow v_2, m_2 \quad \text{INTEGER} \vdash 2 > v_2, m_2 \Rightarrow v, m_3 \\
 \hline
 e \vdash 'a' > 'b', m \Rightarrow v, m_3
 \end{array}$$

The third step advances to the next sub-phrase:

$$\begin{array}{c}
 \text{LOOKUP} \\
 \frac{\vdash \text{lookup}(e, 'a'), m \Rightarrow 2, m}{e \vdash 'a', m \Rightarrow 2, m} \qquad b = ff \\
 \\
 \frac{e \vdash 'a', m \Rightarrow 2, m \quad \boxed{e \vdash 'b', m \Rightarrow v_2, m_2} \quad \text{INTEGER} \vdash 2 > v_2, m_2 \Rightarrow v, m_3}{e \vdash 'a' > 'b', m \Rightarrow v, m_3}
 \end{array}$$

The fourth step evaluates the sub-phrase:

$$\begin{array}{c}
 \text{LOOKUP} \quad \text{LOOKUP} \\
 \frac{\vdash \text{lookup}(e, 'a'), m \Rightarrow 2, m}{e \vdash 'a', m \Rightarrow 2, m} \quad \frac{\vdash \text{lookup}(e, 'b'), m \Rightarrow 3, m}{\boxed{e \vdash 'b', m \Rightarrow 3, m}} \quad b = tt \\
 \\
 \frac{e \vdash 'a', m \Rightarrow 2, m \quad e \vdash 'b', m \Rightarrow 3, m \quad \text{INTEGER} \vdash 2 > 3, m \Rightarrow v, m_3}{e \vdash 'a' > 'b', m \Rightarrow v, m_3}
 \end{array}$$

The fifth step selects the phrase itself:

$$\begin{array}{c}
 \text{LOOKUP} \quad \text{LOOKUP} \\
 \frac{\vdash \text{lookup}(e, 'a'), m \Rightarrow 2, m}{e \vdash 'a', m \Rightarrow 2, m} \quad \frac{\vdash \text{lookup}(e, 'b'), m \Rightarrow 3, m}{e \vdash 'b', m \Rightarrow 3, m} \quad b = ff \\
 \\
 \frac{e \vdash 'a', m \Rightarrow 2, m \quad e \vdash 'b', m \Rightarrow 3, m \quad \boxed{\text{INTEGER} \vdash 2 > 3, m \Rightarrow ff, m}}{e \vdash 'a' > 'b', m \Rightarrow ff, m}
 \end{array}$$

The sixth step produces the final result:

$$\begin{array}{c}
 \text{LOOKUP} \quad \text{LOOKUP} \\
 \frac{\vdash \text{lookup}(e, 'a'), m \Rightarrow 2, m}{e \vdash 'a', m \Rightarrow 2, m} \quad \frac{\vdash \text{lookup}(e, 'b') \Rightarrow m_3 m}{e \vdash 'b', m \Rightarrow 3, m} \quad b = tt \\
 \\
 \frac{e \vdash 'a', m \Rightarrow 2, m \quad e \vdash 'b', m \Rightarrow 3, m \quad \text{INTEGER} \vdash 2 > 3, m \Rightarrow ff, m}{\boxed{e \vdash 'a' > 'b', m \Rightarrow ff, m}}
 \end{array}$$

8.3.4 Congruence

As with the definition of computation step, the definition of animation step is really a function from relational animations to equivalence classes of relational animations, but this doesn't affect the final result of the program.

The definition of congruence for relational animations is nearly the same as that for relational evaluations. One difference is that the boolean variables in the two animations must be the same. The other is that the current sequent may be any local premise.

Two relational animations are **congruent**, written $(\mathcal{R}, b) \cong (\mathcal{R}', b')$, if:

1. $lst(\mathcal{R}) = lst(\mathcal{R}')$ and if either animation is complete then so is the other and $rhs(\mathcal{R}) = rhs(\mathcal{R}')$.
2. $b = b'$.
3. The number of matched premises of $root(\mathcal{R})$ and of $root(\mathcal{R}')$ are the same.
4. If q is the i^{th} premise of $root(\mathcal{R})$ and q' is the i^{th} premise of $root(\mathcal{R}')$, then:
 - (a) If q is remote and is not the last matched sequent, then $q' = q$.
 - (b) If q is remote and is the last matched sequent, then $lst(q') = lst(q)$ and both premises refer to the same remote rule set.
 - (c) If q is local and is not the last matched sequent, then $q' = q$ and $(\mathcal{R}, b)[i] \cong (\mathcal{R}', b')[i]$.
 - (d) If q is local and is the last matched sequent, then $lst(q) = lst(q')$ and $(\mathcal{R}, b)[i] \cong (\mathcal{R}', b')[i]$.
5. If $o(\mathcal{R}, b)$ is the i^{th} premise of $root(\mathcal{R})$, then $o(\mathcal{R}', b')$ is the i^{th} premise of $root(\mathcal{R}')$ and $lst(o(\mathcal{R}, b)) = lst(o(\mathcal{R}', b'))$.

Lemma 8.2 *If $(\mathcal{R}, b)$ is an incomplete initial relational animation, $(\mathcal{R}', b') = \mathcal{A}_R(\mathcal{R}, b)$ and $(\mathcal{R}'', b'') = \mathcal{A}_R(\mathcal{R}, b)$, then $(\mathcal{R}', b') \cong (\mathcal{R}'', b'')$.*

Proof: The proof follows that of Lemma 6.2.

Lemma 8.3 *If $(\mathcal{R}, b)$ is an incomplete relational animation, other than an initial animation, $(\mathcal{R}', b') = \mathcal{A}_R(\mathcal{R}, b)$ and $(\mathcal{R}'', b'') = \mathcal{A}_R(\mathcal{R}, b)$, then $(\mathcal{R}', b') \cong (\mathcal{R}'', b'')$.*

Proof: The proof follows that of Lemma 6.3.

8.3.5 Summary

The definition of relational animation steps is similar to that of relational computation steps. The difference are the new definition of validity, the changes to handle the boolean variable, and the extra case for empty initial animations.

8.4 Choice of Formalism

The decision to base the theory of animation on structured operational semantics was partly based on the idea that this formalism encodes an idea of an evaluation step. The definition of a computation step was based on this idea. Now we have seen that this definition of a step is not suitable for animation. How does this affect the choice of formalism?

Structured operational semantics is still a suitable base for the theory. It still has an operational reading, and still provides concise specifications. The definitions introduced in this chapter show that an animation step can be defined in terms of relational semantics, with only a few changes from the definition of a computation step. Furthermore, every computation step is also an animation step, so the idea of a transition as a computation step has been augmented rather than rejected. Finally, Chapter 9 will show that a range of

views can be specified in terms of relational semantics. All this shows that relational semantics is a suitable base for a theory of animation.

Nevertheless, now that we are using a definition of a step that is based on the user's requirements rather than on a semantic intuition, we could consider other semantic formalisms in a similar way. For example, denotational semantic specifications are concise and easy to reason about. We might find that a suitable definition of a step, such as treating each function application as a step, would give the same effects as the definitions presented here. Similarly, it may be possible to define a view of a program in terms of highlighting the current redex.

A particular interesting comparison would be between the theory presented here and a theory based on definitional interpreters. A definitional interpreter is defined as a set of rewrite rules from one state of an abstract machine to another. Plotkin's original development of structured operational semantics was partly motivated by a wish to remove what he saw as unnecessary steps from definitional interpreters, and to concentrate only on those that actually performed some computation. Hence our intuition that transitions in transitional semantics encode computation steps. It would be intriguing to see if the extra steps that Plotkin eliminated from abstract machines correspond to the steps that we have put back to give better animations.

For example, a well-known definitional interpreter is the SMC machine [Mil76], that was mentioned in Chapter 4. It was developed to define simple imperative languages. It consists of three stacks:

- S: The value stack, used for storing values while evaluating expressions.
- M: The memory, used to store the values of variables.
- C: The control, used to store the part of the source not yet evaluated.

Addition can be defined by the following set of rewrite rules:

$$\langle S, M, (p_1 + p_2).C \rangle \Rightarrow \langle S, M, p_1.p_2 + .C \rangle$$

$$\langle v_2.v_1.S, M, +.C \rangle \Rightarrow \langle v_1 \oplus v_2.S, M, C \rangle$$

where $.$ is used to separate stack elements and $\oplus$ denotes integer addition (as opposed to $+$, which denotes the addition of arbitrary integer-valued phrases).

Many of the rules that define the SMC machine analyse the program source without performing any computation. For example, the first rule for addition just puts the plus operator and its arguments on the stack in the right order, and the rule for integers just moves an integer from the control stack to the value stack. Also, the machine keeps the whole of the remaining program around on the control stack rather than specifying operations in isolation. This can make proofs about the language more complicated than necessary. These disadvantages were what motivated Plotkin to develop transitional semantics.

Clearly there is scope for comparing different formalisms on their suitability for developing a theory of animation. An important part of such comparisons will focus on the definition of an animation step, as discussed above. However, they must also consider the ease with which views and advanced operations can be defined. The next chapters discuss these aspects of the current theory.

8.5 Chapter Summary

The theory developed in the previous chapters included a definition of a computation step. This definition is based on the transitions of a determinate finite automaton, as encoded by transitional semantics. This chapter has shown that computation steps are not suitable for animation. Animation steps are better because they show the result of each phrase and because they guide the user to the next phrase to be evaluated by zooming in to it instead of jumping straight to it. These differences make animation steps easier to follow.

The existing theory can be used as the basis of a definition of relational animations and animation steps. Relational animations differ from relational evaluations in that every non-trivial local premise is valid, and a boolean variable is used to make an animation stop twice for each computation step. The definitions of step and congruence are straightforward versions of the same definitions for relational evaluations. The definition of an animation step is such that every computation step is also an animation step. Experience with animators based on this definition have found it satisfactory.

An equivalent definition of animation steps can be given in transitional semantics. I present such a definition, with a proof of equivalence, in Appendix A. That definition is much more complicated than the simple definition of computation step in transitional semantics. This is not surprising, given that computation steps are based on the transitional semantic formalism and that animation steps are not. However, the fact that animation steps can be defined in transitional semantics does suggest that the definition is not dependent on relational semantics.

This completes the definition of a step for the basic view of a program in which the whole program is displayed and the result of every phrase is shown. The following chapters show how different views can be defined in terms of this theory, and how the definition of a step can be parameterised on a view. They also show how advanced operations can be defined in terms of the theory.

Chapter 9

Views

The preceding chapters have shown how an animation step can be specified in terms of relational semantics. That development assumed a basic view of a program, which replaced each phrase by its result as it was evaluated. However, Chapter 2 showed that an animator can support users better if it provides several views of the structure of a program, each showing a different type of proximity or level of granularity. This chapter shows that several structural views can be defined in terms of this theory.

As explained in Chapter 3, the theory of animation defines a step independently of the language being animated. However, the theory defines views specific to each language, as part of the specification of that language. This chapter shows how views can be specified in terms of relational animations.

Chapter 10 will show that the definition of an animation step for the basic view can be parameterised on another view. This will allow any of the views described in this chapter to be animated.

The views presented in this chapter are classified in the following ways:

1. By whether they are static or dynamic.
2. By what they present (source, environment, memory or semantics).

Source views are further classified by their proximity and granularity, as defined in Chapter 2.

Static views are constructed from the abstract syntax of a program and aren't changed by the evaluation. However, an animator may highlight the current phrase, as in MacPascal [Hue84], DBXTool [Sun] and Dock [TAK82]. This gives a simple form of animation.

Dynamic views are constructed from the program state during evaluation. The displays that they generate show the changes in the program as it is evaluated. A simple example is the view shown by ZSTEP [Lie84]. This is the basic view that was assumed in the previous chapters. It displays the source of the program with each phrase replaced by its value when that value is known, and highlights the current phrase or value.

The views described above are **source views**; they display the source of the program. It is also possible to present views of the environment and of the memory, called **environment views** and **memory views** respectively. Memory views are essential for monitoring the evaluation of programs with side-effects. The desire to see the contents of the memory is often the main motivation for the development of debugging tools. Environment views may be static or dynamic, but memory views must be dynamic. Some views show a combination of the environment and the memory.

One of the advantages of generating an animator from a semantic description of a language is that it can show the evaluation of a program in semantic terms. In other words it can display **semantic views**. Semantic views are intrinsically dynamic, because semantics describe an evaluation process.

The notation developed in this chapter is my own. Chapter 12 will show that it forms the basis for the display rules in LSL, the input language for The Animator Generator. Appendix B includes a glossary of this notation.

9.1 Static Source Views

Static views are specified by a display rule for each constructor. This rule specifies how an instance of that constructor should be displayed. It is called a **constructor display rule**. For textual views these display rules are similar to the unparse rules of the Synthesizer Generator [RT84], the layout rules of Cépage [Mey87] or the pretty printing rules of Centaur [BmL87]. Graphical views require more powerful rules.

9.1.1 Lexical Proximity: Lexical Views

The simplest view of a program is its source code. This can either be reconstructed from the internal form or recovered from the source files. The second approach uses more memory in an implementation because the abstract syntax must be annotated with pointers into the source files, but it gives better results. Novices in particular prefer to see the code that they actually wrote [Raj86].

Lexical views correspond to lexical proximity. They present one piece of code following another if the two are lexically adjacent. They correspond to the expression level of granularity.

The following constructor display rule specifies that an instance of the **plus** constructor should be displayed in conventional arithmetic form:

plus:

\$1 " + " \$2

The first part of the rule is the name of the constructor that the rule applies to. The rest of the rule gives a sequence of tokens to be displayed. Strings are displayed literally, so " + " displays the plus sign with some space on each side.

The symbols \$1 and \$2 refer to the first and second arguments of the abstract syntax node.

If we assume that there is a similar rule for the `times` constructor, then the following phrase:

```
plus (2, plus (times (3, 4), 5))
```

would be displayed as:

$$2 + 3 * 4 + 5$$

The highlighting of the current phrase can be built-in to the display algorithm.

Most programs should be indented to show their structure. Some pretty-printing systems do this by allowing rules to include commands which set and reset the level of indentation. A more declarative way of specifying indentation is to put a sub-phrase into a box and put some space in front of the box. The sub-phrase will be formatted with the left side of the box as its margin. This approach is similar to, although simpler than, that of the `TEX` [Knu84] text formatting system.

For example, if boxes are denoted by braces surrounding the expression to be boxed, the display of a procedure can be specified as:

func:

```
"func " $1 " " $2 " =\n " {$3} "\nim\n " {$4} "\nend\n"
```

This indents the third and fourth arguments of the constructor. So, for example:

```
func (x', x, times (x, x), times (2, call (x', x)))
```

would be displayed as:


```
func x' x =  
    x * x  
in  
    2 * x'(x)  
end
```

In the above example, the sequence `\n` in a string denotes a newline character. Since strings containing only spaces or newlines are quite common in display rules, and too many quote characters can cause confusion, the symbols `$s` and `$n` will sometimes be used in their stead. This makes the above rule become:

```
func:  
    "func " $1 $s $2 " = " $n $s$s { $3 } $n  
    "in" $n $s$s { $4 } $n "end"
```

9.1.2 Lexical Proximity: Outline Views

Often it is useful to show the overall structure of a program, hiding some of the detail. One way of doing this is to display an **outline view**. Outline views display only certain parts of a program, such as showing module or procedure headers without showing their bodies. For example, a static outline view of a parser written in C++ might look like this:

```
char get_char ();  
char* get_filename ();  
token scan ();  
void error (char* msg, char* filename, int line);  
tree* parse ();
```

As in other static views, the current procedure can be highlighted. Static outline views correspond to lexical proximity at different levels of granularity. The above example is a procedure level lexical view.

A first attempt at specifying this view could use the rule given in the previous section for displaying procedures and the empty rule for all other constructors.

This approach would work for a language such as Pascal, in which procedure bodies must be blocks and blocks may not be identifiers. The specification for such a language could have an empty rule for blocks and a non-empty rule for identifiers, and any identifiers in a block would never be reached.

However, this approach wouldn't work for Proc, because if the rule for identifiers was empty, then it wouldn't display the identifiers in procedure headers, and if the rule wasn't empty then it would display all identifiers. Neither behaviour is the one desired.

To specify this view in Proc we need some way of providing context-sensitive information. This can be done by parameterising the display rules. If we require that each display rule for this view takes a parameter called *show*, the rule for the *func* Constructor can be written as follows:

func:

```
"func" $1[show=tt] $s $2[show=tt] " = " $n $s$s
$3[show=ff] $n "in" $n $s$s $4[show=ff] $n "end" $n
```

and the rules for identifiers (the *id* constructor) can be written as follows:

```
id[show=tt]:
```

```
$1
```

```
id[show=ff]:
```

The notation `$1[show=tt]` specifies that the first argument should be displayed with the parameter *show* passed the value *tt*. The rules for the *id* constructor are selected according to the value of the *show* parameter.

An extension of this syntax could allow patterns to use the ordering relations over integers. This could be used to bracket infix expressions. Each expression would pass its precedence to each sub-expression.

In the outline view given here, the rules for all the other constructors are the same; they don't display anything. The specification can be simplified by using a default rule, to be used for constructors which aren't given a specific rule.

9.1.3 Structural Proximity: Static Call Graphs

A **static call graph** shows which procedures call which. One way of displaying a call graph is to list the procedures called by a certain procedure below that procedure and indented relative to it. With this approach, the example program used in the outline view above would be displayed as follows:

```
parse
  get_filename
    get_char
  scan
    get_char
  error
```

A call graph can instead be displayed on a graphics screen as a tree or a directed acyclic graph of procedure names connected with lines, as in InterLisp [Tei78]. The graph approach avoids redundancy, since each procedure name is only displayed once. A static call graph is a procedure level structural view.

A textual display of a call graph can be specified in a similar way to an outline view. However, a graphical display of a call graph is more complicated, because it can't be formatted at the same time as the abstract syntax is traversed. This is because the position of a node of the graph may depend on the layout of parts of the program that have yet to be traversed.

Graphical views such as this can be implemented using a layout language. The display rules will specify the graph in terms of this language. The display will then be constructed in two passes; the first will traverse the abstract syntax to produce a description in the layout language, and the second will produce the display from this description.

The notation for boxes used in Section 9.1.1 is a simple example of a layout language. The position of elements in a box is constrained by the box and elements following the box are positioned relative to the box. The display must be constructed in two passes, so that the width of the box is known before the elements that follow it are drawn,

A more sophisticated language is needed for graphs. The more sophisticated the language, the more sophisticated the views it can produce. The only fixed requirement is that it should support highlighting of the current phrase. One possibility would be an adaptation of an existing language such as PIC [Ker81].

The following example specifies how a program is displayed as a tree that represents the call graph of the program:

func:

```
LET self = #{ $1 }
      tmp = ⟨ self ABOVE $3[ parent = self ] ⟩
in⟨⟨ LINE parent TO tmp LEFT $4 ⟩
```

It uses a few simple constructions. The angle brackets are used for grouping, and are not displayed. The notation `LET self = #{ $1 }` creates a box from the display of `$1` and names the box `self`; it doesn't display it. `self ABOVE $3[ parent = self ]` positions `self` above the display of the third argument of the constructor, centring both. The rule gives this the name `tmp`; again, it isn't displayed. `LINE parent TO tmp` draws a line from the box `parent` (a parameter)

to the display of *tmp*. Finally, LEFT \$4 draws the display of the fourth argument to the right of the display of *tmp*.

In this view only part of the current phrase is highlighted. This is indicated by preceding that part with a # symbol. Thus only the name of the current procedure is highlighted, instead of the whole subtree rooted at this node. This notation must override the default behaviour.

Displaying a call graph as a directed acyclic graph would require a more sophisticated display language. An associative array would be useful for this; it could be used to store nodes of the graph indexed by their procedure names.

Other structural views include parse trees and control flow graphs. In imperative languages, control flow graphs are statement level views and parse trees are usually expression level views. These views can be specified in the same way as call graphs.

9.1.4 Referential Proximity: Static Slices

A particularly useful view of an imperative program is a **slice**. A slice is a mini-program consisting of those lines of the main program which affect the value of a given imperative variable. It is useful because there is strong evidence that people often mentally construct slices when debugging [Wei82]. A slice corresponds to referential proximity, usually at the statement level.

As an example of a slice, consider the following program:

```
sum := 0;
count := 0;
read p;
while p <> -1 do
    sum := sum + p;
    count := count + 1;
    read p;
end;
if sum > max_sum then
    max_sum := sum;
return count;
```

A slice of this program on 'p' produces the following program:

```
read p;
while p <> -1 do
    read p;
end;
```

A slice on 'max_sum' produces:

```
sum := 0;
read p;
while p <> -1 do
    sum := sum + p;
    read p;
end;
```

Weiser, who first did experiments to show that people mentally create slices, also designed an algorithm that produces near-minimal slices from simple Pascal programs [Wei81]. This algorithm is reasonably complicated. Since most display languages don't handle arbitrary programming problems very well, a slice

generator is best implemented as another part of the programming environment. The generated slice can be passed as a set of instances of constructors to be displayed.

Display rules can then be used to specify a view that will display an arbitrary slice, by passing an implicit parameter to each rule. This parameter will be true when the instance of the constructor is in the slice. If this parameter is called *visible*, the rules for displaying identifiers would look like this:

```
id[visible=tt]:
```

```
  $1
```

```
id[visible=ff]:
```

Sometimes the display of a node will depend on whether the sub-animations of that node include any selected nodes. For example, if the sequencing operator is displayed as an infix semi-colon, then it shouldn't be displayed if there are no selected nodes after the current node. This case can be handled using the following display rules for the sequencing constructor:

```
seq[visible=tt, prefix=tt]:
```

```
  " ; " $n $1[prefix = ff] $2[prefix = tt]
```

```
seq[visible=tt, prefix=ff]:
```

```
  $1[prefix = ff] $2[prefix = tt]
```

```
id[visible=ff]:
```

```
  $2[prefix = tt]
```

These rules assume that an instance of the sequence constructor will only appear in the slice if the display of the first argument is not empty. If the first argument is empty, the sequence operation is redundant. Therefore it displays the two arguments in turn, specifying that the second argument must display the semicolon if it is non-empty.

All display rules for nodes that could appear on the right of a sequencing operator must handle the *prefix* parameter correctly. In many imperative languages only other sequencing operators and the null statement have to do this, but in an expression-based language this will have to be done by almost every display rule.

If it is not possible to assume that the first argument will be empty, the rule would need some way of determining whether the first argument displayed anything. This would require a more powerful language.

9.1.5 Summary

Static views can be specified by display rules over the abstract syntax. For simple textual views these rules are similar to those of a pretty-printer. More complex rules are needed if a system is to support a wider range of views. In some cases they have to be parameterised on the results of another tool such as a slice generator.

9.2 Static Environment Views

If environments are bindings of identifiers to values, there are several possible ways of displaying one. The simplest is as a list of the current bindings – identifiers on the left, values on the right. This can be specified by the following rule:

env:

\$1 \$t \$2 \$n \$3

This rule introduces the notation `$t`, which stands for the string `"\t"`, the string containing the tab character.

The above rule is fine for simple values but is less suitable for arrays or records, as these values might be too large to display easily. There are several ways to tackle this problem, including the following:

1. By default the system could display just the first part of a record or array and to give the user a way of displaying the rest, such as horizontally scrolling the relevant window or popping up the whole value in a separate window.
2. The system may allow the user to set a parameter that restricts the depth to which the abstract syntax is traversed. Each recursive call of a display rule for a value would decrement this parameter. A display rule for which the parameter equals zero wouldn't display anything.
3. The system could limit the values that can be displayed to simple values while allowing composite identifiers such as `'record_name.field_name'` on the left hand side of the display. Users could then choose the parts of the value that they wish to see.

There is also a choice of which identifiers to display in an environment view. One option is to display all the identifiers currently in scope. Another is to display all identifiers, with some indication of the scope of each. A third approach is to let users select which identifiers they want to see. This can be specified by parameterising the view on a list of identifiers. The pattern matching notation would have to be extended to include predicates over terms, for example:

```
env[is_selected($1)]:
    $1 $t $2 $n $3
```

This facility must include a way of handling the scope of variables. For example, a user might select the global identifier ‘a’; in this case the system should not display any local identifiers with the same name (unless they are also selected).

As with static program views, it is possible to highlight static environment views. Usually the desired behaviour will be to highlight an identifier and its associated value when the current phrase is structurally equal to that identifier. If the notation $\# = \$1 =$ tests whether the current phrase is equal to $\$1$ and highlights the following display expression if so, then the display rule for environments would become:

```
env[is_selected($1)]:
    <# = $1 = <$1 $t $2> > $n $3
```

If environments can appear as sub-phrases in a source view, for example as part of a closure or as the result of a declaration, they may take up too much space on the screen. It may be sensible to parameterise their display rules so that they are displayed in an abbreviated form in such cases.

9.3 Dynamic Views

A dynamic view is constructed from an animation history. A rule is given for each semantic rule and constructor to specify how an instance of that semantic rule or constructor should be displayed. These rules are called **semantic display rules** and **constructor display rules** respectively. The constructor display rules are defined in the same way as those for static views and are not discussed any further.

9.3.1 Lexical Proximity: Simple Lexical Views

A dynamic lexical view displays the source of the program with evaluated phrases replaced by their values. This display is constructed by traversing the animation, displaying each node as described below.

If a sub-phrase of the subject corresponds to the subject of a local premise it is displayed as follows:

1. If the premise is unmatched, then the sub-phrase is displayed normally.
2. If the premise is matched and the corresponding sub-animation is complete, then the sub-phrase is replaced by its value.
3. If the premise is matched and the corresponding sub-animation isn't complete, then the sub-phrase is replaced by the display of the sub-animation.

This is the algorithm that produced the second set of displays in Chapter 8. This is a source view, so memories and environments are ignored.

For example, if the semantic rule for the sequencing operator is:

$$\frac{e \vdash p_1, m_0 \Rightarrow v_1, m_1 \quad e \vdash p_2, m_1 \Rightarrow v_2, m_2}{e \vdash p_1; p_2, m_0 \Rightarrow \text{nil}, m_2}$$

then the corresponding semantic display rule for the dynamic lexical view is:

Sequence:

\$2 = *subject*

\$1 = *subject*

\$0 = *result* ??0 #(*sequence*(@1, @2))

\$\$ = \$0

As with a constructor display rule, a semantic display rule begins with the name of the object for which it specifies a display. A semantic display rule then

has several equations, one for each sequent and one for the overall display of the instance of the rule.

The symbols \$1 and \$2 in the above display rule refer to the first and second premises of the semantic rule. \$0 refers to the conclusion, and \$\$ refers to the entire rule. Each line of the display rule defines the display of the part of the semantic rule referred to on the left of the equals sign. Thus in this example the rule is displayed by displaying the conclusion. The variables *subject* and *result* display the subject and result of the corresponding sequent, so the first two premises of this rule are displayed by displaying their subjects.

The right hand side of an equation is similar to the body of a constructor display rule, but can include variables from the semantic rule. It may also use a wider range of operators. As in a constructor display rule, strings are displayed literally and angle brackets are used for grouping. The # symbol is used for highlighting; it highlights the expression that follows it if the focus of the animation is the subject of the sequent being displayed. Another symbol, ##, highlights the expression that follows it if the focus is the result of the sequent being displayed. The special variables *subject* and *result* are assumed to be preceded by # and ## respectively.

The ??0 in the display of the conclusion tests whether the evaluation of the conclusion is complete. If it is complete, the expression on the left of the ??0 symbol is displayed; if it isn't then the expression on the right is displayed. This notation can be used for sub-animations as well; the symbol ??1 would test the completeness of the sub-animation corresponding to the first premise, and so on.

The symbol @1 displays the sub-animation corresponding to the first premise, if it exists, or the premise (defined by the second line) if the sub-animation doesn't exist.

Finally, the expression `sequence(@1, @2)` is displayed by the constructor

display rule for `sequence` with the arguments of the constructor replaced by the display of the first and second premises or subtrees as appropriate.

Thus if the evaluation of this node is complete, the result is displayed, and is highlighted if the node is the current node. If the evaluation is not complete, the phrase is displayed in source form, with the sub-phrases displayed according to the display rules for the nodes in the sub-animations.

9.3.2 Lexical Proximity: More Complicated Examples

The method of the previous section handles simple cases, but some rules are more complicated. One example is the following semantic rule for the `while` constructor:

$$\frac{e \vdash p_1, m_0 \Rightarrow \text{tt}, m_1 \quad e \vdash p_2, m_1 \Rightarrow \text{nil}, m_2 \quad e \vdash \text{while } p_1 \text{ do } p_2, m_2 \Rightarrow \text{nil}, m_3}{e \vdash \text{while } p_1 \text{ do } p_2, m_0 \Rightarrow \text{nil}, m_3}$$

If the recursive premise of this rule is matched or is the current sequent, the display of this rule should be the display of that premise only. This has the effect of only displaying the current iteration of the while loop. The other premises should be ignored.

This can be specified by the following semantic display rule:

While:

```
$3 = #(<while(@1,@2)>)
$2 = subject
$1 = subject
$0 = result ??0 @3
$$ = $0
```

This is similar to the display rule for the sequential operator, but if there is a

sub-animation corresponding to the third premise, the display of the current node is completely replaced with the display of that sub-animation.

Another complication arises with the following semantic rule for function calls:

$$\frac{e \vdash p, m_0 \Rightarrow v, m_1 \quad \text{LOOKUP} \quad \vdash \text{lookup}(e, x) \Rightarrow \text{closure}(e', x, x', p') \quad e' \dagger (x', v) \dagger (x, \text{closure}(e', x, x', p')) \vdash p', m_1 \Rightarrow v', m_2}{e \vdash \text{call } x \ p, m_0 \Rightarrow v', m_2}$$

In this rule p' first appears as part of the result of the second premise. Therefore the syntactic representation of `call x p` won't display the full evaluation of this node.

There are two common ways of displaying procedure calls: the first is to expand the procedure in-line (e.g. APT [Raj86]); the other is to display only the current procedure (e.g. ZSTEP [Lie84]). In each case p' is displayed; in the first case the call is probably displayed as well and in the second case the screen is cleared before p' is displayed. An obvious optimisation of the second case is to traverse only that part of the animation history below the first procedure call between the current node and the root.

The following display rule specifies that only the current procedure should be displayed:

Call:

```
$2 = result ??2 #x
$1 = subject
$0 = result ??0 #(<call(@2, @1)>)
$$ = !3 ?!3 $0
```

Two new symbols are used here: `!3` displays the sub-animation corresponding

to the third premise, like @3, but clears the screen first. ?!3 tests whether the current node is in the sub-animation corresponding to the third premise. Thus this rule displays the body of the procedure while it is being evaluated, and the call at other times.

A third complication is illustrated by one of the rules for the `lookup` constructor:

$$\frac{\vdash \text{lookup}(e, x), m \Rightarrow v', m_1}{\vdash \text{lookup}(e \dagger(x, v), x), m \Rightarrow v', m_1} \quad \text{IDENT} \quad \vdash \text{eq}(x, x') \Rightarrow \text{ff}$$

In the views that we have considered so far, the `lookup` constructor has always been evaluated in a remote rule set. But we can also consider viewing the `LOOKUP` rule set itself. A possible view of the `lookup` operation is to show the environment as a list of pairs, with the highlight moving along the list until it reaches an identifier that matches the one being searched for. For example, if we are looking for the value bound to the identifier 'a' in the environment $(\text{'a'}, 1) \dagger (\text{'b'}, 2) \dagger (\text{'c'}, 3) \dagger (\text{'d'}, 4)$ we would have the following display (where the box indicates highlighting):

$$\text{lookup}((\text{'a'}, 1); (\boxed{\text{'b'}}, 2); (\text{'c'}, 3); (\text{'d'}, 4), \text{'a'})$$

This is a display of the following animation tree:

$$\frac{\vdash \text{lookup}((\text{'a'}, 1) \dagger (\text{'b'}, 2), \text{'a'}), m \Rightarrow v', m_1}{\vdash \text{lookup}((\text{'a'}, 1) \dagger (\text{'b'}, 2) \dagger (\text{'c'}, 3), \text{'a'}), m \Rightarrow v', m_1}$$

$$\frac{\vdash \text{lookup}((\text{'a'}, 1) \dagger (\text{'b'}, 2) \dagger (\text{'c'}, 3), \text{'a'}), m \Rightarrow v', m_1}{\vdash \text{lookup}((\text{'a'}, 1) \dagger (\text{'b'}, 2) \dagger (\text{'c'}, 3) \dagger (\text{'d'}, 4), \text{'a'}), m \Rightarrow v', m_1}$$

To produce this display the first `lookup` rule must be treated differently from the others, as the display of that rule must include the string "lookup (" and the identifier being searched for.

This can be specified using the following rule:

Lookup:

```

$1          = e
$0[first = tt] = result ??0 "lookup (" @1[first = ff] " :: (" #x', v ")", x ")"
$0[first = ff] = @1[first = ff] " :: (" #x', v ")"
$$          = $0

```

This uses parameters, similar to those used to specify certain static views. Instead of whole rules including patterns to match against the parameters, patterns are part of equations. The notation `@1[first = ff]` displays the sub-animation corresponding to the first premise, if it exists, or the first sequent otherwise, with the parameter *first* set to *ff* in each case. `$1[first = ff]` and `&1[first = ff]` have the obvious similar meanings when they appear in display expressions.

A final complication arises when a language contains some constructs with boring results. For example, Pascal statements are evaluated for their side-effects and users aren't interested in the fact that (in relational semantics) their result is *nil*. However, the results of Pascal expressions are interesting. This problem isn't resolved until Section 10.1.1, because it involves changes to the definition of step as well as views.

The views discussed in this section have been specified by traversals of the entire animation tree. If a system is required to store all this information it will use a large amount of storage. In practice most systems store information corresponding to the sub-animations of the procedures on the path from the root of the animation to the current animation. This restricts the range of views that can be displayed, but allows the standard views of the current procedure and the procedure call stack.

9.3.3 Temporal Proximity: Traces

A **trace** is the dynamic equivalent of an outline view. For example, a procedure level trace shows the procedures called by an evaluation, in the order that they are called. Traces correspond to temporal proximity.

If a trace is constructed as the program is evaluated, it can't display the unevaluated part of the program (unless the program is so simple that it only has one path of evaluation). An alternative is to evaluate the whole program first and then construct the animation from information stored during the evaluation [EB86,Moh88].

Traces can be specified as a traversal of the animation tree, similar to dynamic lexical views. The problems that arise with static outline views arise with traces too, and can be solved the same way (see Section 9.1.2).

Defining a trace in terms of a traversal of the animation tree after every step is obviously inefficient, and doesn't fit the intuitive idea of a trace as a permanent record of an evaluation that is only ever appended to. A system could provide an efficient implementation of traces so that only new additions to the tree would have to be traversed at each step.

9.3.4 Referential Proximity: Dynamic Slices

A dynamic slice is a trace of the statements in a slice. It corresponds to referential proximity at the statement level.

Dynamic slices can be specified as a traversal of the animation history, if this traversal is parameterised on a set of visible nodes. The problems found with dynamic lexical views and static slices also apply to dynamic slices, and can be dealt with by the techniques outlined in the appropriate sections.

9.3.5 Structural Proximity: Dynamic Call Graphs

A **dynamic call graph** shows which procedures have called which. It starts with just the main procedure and grows as new procedures are called. The dynamic call graph of a completed evaluation will differ from the static call graph of the same program in that the static graph will include procedures that are not called during the evaluation and the dynamic call graph will show the depth to which recursive procedures have been called. A dynamic call graph is a procedure level structural view.

Static and dynamic call graphs differ in the way they handle recursion. When a recursive instance of a procedure is called, a static call graph will highlight the previous occurrence of that procedure. There can't be a subgraph for recursive instances in a static call graph or the graph would be infinite. Instead, when a recursive instance of a procedure is called, a static call graph will highlight the previous occurrence of that procedure. In contrast, a dynamic call graph will expand by adding the recursive instance to the graph in the place where it is called. The size of the dynamic graph is limited by the size of the evaluation.

Dynamic call graphs can be specified as a traversal of the animation tree. They combine the problems of dynamic lexical views and static call graphs. The techniques from the relevant sections can be used to overcome these problems.

If the system stores only the information corresponding to procedures on the path from the root of the tree to the current procedure, as suggested in Section 9.3.2, a dynamic call graph will be limited to displaying the procedure call stack. In addition to this information, a system could keep track of which procedures have called which other procedures, ignoring the evaluation of the procedure bodies. This would be enough information to create a dynamic call graph without requiring as much storage as the whole evaluation tree. This could be an acceptable compromise between efficiency and providing a wide range of views.

Dynamic call graphs are particularly useful for animating Prolog programs. Prolog is a difficult language to animate because it backtracks. This means that the actions of a particular part of the program depend on whether that part of the program has been reached before. Some Prolog animators use lexical views with extra information about backtracking [Byr80,Plu88,Raj86], but this approach is often hard to follow. Other Prolog animators display the search tree [EB86,DC86]. This shows the backtracking information more clearly, but can sometimes obscure the detail of the current procedure. Pain and Bundy give a good introduction to the different ways of displaying Prolog animations [PB87]. Chapter 11 sketches an approach to specifying Prolog in terms of this theory.

Both functional and logic programming languages would benefit from the ability to highlight all occurrences of a phrase as a procedure parameter in a dynamic call graph, as allowed in the Dewlap system [DC86]. This can be implemented by parameterising the view on a phrase (or a set of phrases) and testing each phrase for equality with the parameter. Ideally the user would be able to select the phrase(s) to be the parameter(s) using a pointing device.

9.3.6 Structural Proximity: Procedure Call Stacks

A restricted version of a dynamic call graph is a stack of procedure calls, with the main procedure of the program at the bottom and the current procedure at the top. This is called a **call stack**. Most debuggers provide a call stack view, partly because it corresponds to the way that most languages are implemented and partly because it shows something of the context of the evaluation of the current procedure. Procedure call stacks are a restricted case of a dynamic call graph, and are therefore procedure level views of structural proximity.

A call stack view can be generated explicitly by traversing the animation, ignoring all nodes except instances of the rule for procedure calls:

Call:

$$\$ \$ = \langle \text{call } (p_1, v) \ \$n \ \&3 \rangle \ ?!3 \ \text{""}$$

9.3.7 Hybrid Source Views

Hybrid source views are possible. One possibility is to show a temporal view of the parts of the program that have been evaluated and a lexical view of the remaining parts. Another possibility is to have different types of view at different levels of granularity.

Rajan [Raj86] argues that procedures should be expanded in-line at their point of call, so that users can see the context of the current procedure. This combines temporal proximity at the procedure level with lexical proximity at smaller levels. A more common way of displaying procedures is to display only the innermost procedure in full, using the stack to display outer procedures. This combines structural proximity at the procedure level with lexical proximity at lower levels.

9.3.8 Summary

Dynamic source views can be specified by display rules for each semantic rule and constructor. The notation for semantic display rules can be used to vary the display of the animation depending on how far the evaluation has progressed. A parameter passing mechanism allows the display of a node to depend on its context and also allows a view to be parameterised on the results of another part of the programming environment.

9.4 Dynamic Environment Views

A dynamic environment view is a view of the current environment. If $(\mathcal{R}, b)$ is an animation, then the **current environment** of $(\mathcal{R}, b)$ is $env(o(\mathcal{R}, b))$ if $o(\mathcal{R}, b)$ is a local premise or a conclusion, and the env of $O(\mathcal{R}, b)$ if $o(\mathcal{R}, b)$ is a remote premise.

A dynamic environment view can be specified in the same way as a static environment view (see Section 9.2). It has the extra complication that the user might want to see the contents of locations instead of their addresses. For example, in Pascal users are more interested in the current value of an identifier than the memory location used to hold that value. Conversely, sometimes users want to know which identifiers reference the same location.

A primitive approach of showing the contents of a location is to allow users to evaluate an expression in the current environment and memory. Users can then evaluate an identifier, which will return the value stored in the corresponding location. This approach is used by many symbolic debuggers, but it is a replacement for an adequate view of the current environment rather than being such a view. Users have to enquire about the identifiers that they are interested in after each step, instead of having the display change automatically.

To enable users to see either the contents of a location or the address of the location, a view could be parameterised on a list of identifiers for which to show the contents of the corresponding locations. Alternatively, a view could display both the address and contents at the same time. A more sophisticated method is to display a view of the memory (see below) and draw arrows from each identifier to the appropriate location or object in the memory view. This approach is used by PV [BCH⁺85].

9.5 Memory Views

A memory view is a view of the current memory. If $(\mathcal{R}, b)$ is a relational animation, then the **current memory** of $(\mathcal{R}, b)$ is defined as follows:

1. if $o(\mathcal{R}, b)$ is a remote premise and $b = \text{ff}$ then the current memory is $lhm(o(\mathcal{R}, b))$.
2. if $o(\mathcal{R}, b)$ is a remote premise and $b = \text{tt}$ then the current memory is $rhbm(o(\mathcal{R}, b))$.
3. if $o(\mathcal{R}, b)$ is a local premise then the current memory is $lhm(o(\mathcal{R}, b))$.
4. if $o(\mathcal{R}, b)$ is a conclusion and $b = \text{ff}$ then the current memory is the rhbm of the last premise of $O(\mathcal{R}, b)$.
5. if $o(\mathcal{R}, b)$ is a conclusion and $b = \text{tt}$ then the current memory is $rhbm(o(\mathcal{R}, b))$.

In the fourth case, the last premise is guaranteed to exist by the definition of relational animations.

A simple view of a typical memory is a list of locations with their contents. Animations of assembly language programs should have memory views displaying the contents of registers, condition words and accumulators. This sort of view is often used with imaginary computers for teaching basic concepts (e.g. [Eis79, Lea84]) but can also be used with a real computer architecture (e.g. [Raj86, Fai79]).

A view of a memory as a simple list of locations or a collection of registers can be specified in the same way as a simple view of an environment. Techniques

to restrict the locations displayed can be based on those used to restrict the identifiers displayed in an environment view.

The simple approach suffers from the problem that it doesn't display compound objects clearly, as described for simple environment views in Section 9.2. In addition, it's difficult to understand complex objects by following a maze of pointers through the memory.

More sophisticated animators display the memory in terms of the objects stored in it, such as trees or graphs. Such views often highlight the last object to be referenced. Some examples of this sort of view are:

1. A program to sort an array can be viewed as a sequence of elements, with pointers into the array viewed as markers alongside the sequence [BCH⁺85].
2. A program that manipulates a tree can be viewed by displaying that tree [BS81].
3. A finite state automaton can be viewed as a graph [Rei87].
4. A program simulating a bank can be viewed as a queue of customers waiting in front of a counter [Lan84].
5. A simulation can be viewed as an activity diagram [BWLN84].

Some of these views combine memory and environment views, showing which identifiers are bound to which object in the memory. Since these views are program-specific, most of the animators providing such views have to be pre-programmed; each program contains code or special comments that control the display. Notable exceptions are Graphtrace, Provide, PV and Incense.

Graphtrace [GKS83] allows users to specify interactively their preferred views of Pascal data structures. For each pointer in a record type, users can tell

Graphtrace to print instances of that pointer horizontally, to print them vertically or to print them in the direction that works best given the constraints of the other pointers. A graph drawing program uses this information to produce a suitable display. Users can specify several views for each program.

Provide [Moh88] and PV [BCH⁺85] are more sophisticated; they allow views to be created with bitmap editors. These can then be attached to the variables of a compiled program. Changes in the value of these variables is reflected in the views.

Incense [Mye80] associates layout positions with each element of a type. Each argument has a fixed size rectangle in which it must display itself, so recursive structures are displayed with the items getting smaller as the depth increases. Once displayed, the parts of a structure can be selected and repositioned with a mouse or similar pointing device.

Displaying compound objects in memory often requires an algorithm specific to the type of the objects. This is usually best done by another part of the programming environment. For simple cases, the layout sub-language of the specification language could provide a generic program for drawing graphs that could use the type information stored in a type environment to produce simple graphs.

9.6 Semantic Views

One of the advantages of generating an animator from a semantic description of a language is that it may be able to show the evaluation of a program in semantic terms. The nature of this display depends on the semantic formalism chosen. A display of an inference tree can be generated in a similar manner to a dynamic call graph:

Seq:

```

$2 = <env " | - " subject " ==> " result>
$1 = <env " | - " subject " ==> " result>
$0 = <env " | - " subject " ==> " result>
$$ = $0 OVER (@1 " " @2)

```

The notation `$0 OVER (@1 " " @2)` puts `$0` above `@1 " " @2`, centring both and separating them with a horizontal line.

9.7 Abbreviations

When several constructors or semantic rules are displayed the same way in a particular view, or when a constructor or semantic rule is displayed the same way in several views, it should be possible to combine the appropriate display rules into one. For example, in the LSL specification for Proc the rules for the `null_env` and `null_memory` constructors don't display anything in any view. Using LSL notation, where the constructor display rule for each view of a constructor is listed under that constructor and headed by an arrow that incorporates the view name, it would be desirable to specify this as follows:

```
null_env, null_memory:
```

```
-view1->
```

```
-view2->
```

```
-view3->
```

```
" "
```

```
;
```

This could be taken further by specifying a default display to be used in all views except where explicitly overridden:

```
null_env, null_memory:
```

```
" "
```

```
-view48->
```

```
"()"
```

```
;
```

It is less likely that two different semantic rules will have exactly the same displays. However, semantic display rules often have a similar structure, differing only in that they use different constructors. In such cases it should be possible to parametrise a display rule on the constructor. For example, the following display rule specifies the display of several integer operations:

Plus, Minus, Times, Divide, Modulus:

```
-view1->
```

```
$3 = #(a(@1,@2))
```

```
$2 = subject
```

```
$1 = subject
```

```
$0 = result ??0 #(a3)
```

```
-view2->
```

```
⋮
```

WHERE *a* IS

Plus: plus, Minus: minus, Times: times,

Divide: divide, Modulus: modulus

This example also omits the specification of \$\$; this can be taken to be \$0 if it isn't explicitly defined.

Another way to abbreviate semantic display rules is to give each view two default display rules, one for sequents and one for rules. The default rule for rules is used whenever a display rule isn't explicitly given for a semantic rule, and the default rule for sequents is used whenever a display rule requires the display of a sequent but doesn't specify how to display it.

The default display rule for a sequent can be defined as a normal semantic display rule with only \$\$ defined and the name of the rule replaced by the keywords DEFAULT SEQUENT. For example:

DEFAULT SEQUENT:

-source->

$$$ = subject$

-semantics->

$$$ = \langle env \ " \} - \ " \ subject \ " ==> \ " \ result \rangle$

The default display rule for an instance of a semantic rule can be defined as a normal semantic display rule with the name of the rule replaced by the keywords DEFAULT RULE. For example:

DEFAULT RULE:

-source->

$$$ = \0

-semantics->

$$$ = \$0 \text{ OVER } [@i \ \$s]$

The notation [@i \$s] denotes all the arguments to the constructor or all the sequents of the semantic rule, separated by spaces. Any separator may be replaced by any string.

9.8 Chapter Summary

The internal representation of a program can be displayed in many ways, including source text, parse trees, outline views, call graphs and memory views. These views can be classified by whether they are static or dynamic, and by whether they show the source, the environment, the memory or the semantics. Source views can be further classified by type of proximity and level of granularity.

Chapter 2 showed that animators support tasks better if they provide a range of views that correspond to different levels of proximity and granularity. This chapter has shown how several views can be defined in a specification language, using abstract syntax for static views and relational semantics for dynamic views. These definitions use display rules, which map instances of the syntactic or semantic rules into a layout language. Parameterising the display rules allows them to access information from the contexts of these instances, and also provides a way to interface an animator to specialised software tools such as a slice generator.

All the views mentioned in Chapter 2 can be specified in terms of relational semantics. This shows that this part of the theory is powerful enough to describe practical systems. The next chapter will show how the definition of a step can be parameterised on a view, so that any of the views described in this chapter can be animated. It will also show how advanced operations can be defined in terms of relational semantics.

Chapter 10

Operations

The preceding chapters have developed a formal semantic basis for a theory of animation. The theory includes a language-independent definition of an animation step and a notation for specifying language-specific views. An animator based on this theory would let users see an animation of a program, either by stepping through it themselves or by running it automatically.

In this chapter I show that many advanced operations provided by existing animators and debuggers can also be defined in terms of this theory. In particular I show that all the operation mentioned in Chapter 2, where they were shown to help the tasks supported by animation, can be defined in terms of the theory. This will complete the development of the theory, and show that it is powerful enough to describe useful systems.

In this chapter, the step defined in Chapter 8 is called the **basic step**. Larger steps and view-specific steps can be constructed from repeated applications of basic steps. Operations such as setting break points and changing the current memory can be defined directly in terms of relational animation histories.

This chapter sometimes refers to those semantic rules or premises that describe the evaluation of a certain language construct. A rule or sequent is said

to **describe** a constructor if that constructor is outermost in the subject of the rule or sequent.

10.1 View-specific Steps

The basic view used in most of the examples in this thesis replaces all phrases with their results as the program is evaluated. Not all views share this behaviour. For example, in many imperative languages statements don't return a value, and are best highlighted once only. Another example is a dynamic call graph, which just displays the procedure name at each node of the graph, and doesn't replace it with the value of the procedure (if indeed it has one).

Similarly, some constructs aren't interesting at all. For example, in an outline view, only procedure headers will be displayed. Another example is that the binary sequencing operator just determines the order of evaluation of its sub-phrases, and doesn't perform any computation itself. A sequence of displays that show the traversal of a tree of binary sequencing operations presents a confusing model to the user, who expects the statements to be evaluated as a linear sequence.

A basic requirement of an animation is that each step must change the semantic representation of the program in a way that is reflected in the display of the selected view of the program. However, the definition of a basic step doesn't take account of the phrases that aren't displayed by a view. Therefore the changes that it makes to the semantic representation of the program won't necessarily be shown on the display. So the definition of a step must be customized to each view. As discussed in Chapter 3, this could be done by parameterising the definition of a step on a view, by repeatedly applying the fundamental definition of step until the view changes, or by incorporating a definition of a step in the specification of each view.

This section begins by discussing two ways in which a view may specify that certain phrases should be hidden or should not be replaced by their results. Then it defines a step that is parameterised on such views. It also discusses the problem of choosing which definition of step to use when there are several views of a program visible at once. The section finishes by discussing how this definition relates to the three possible approaches mentioned in the previous paragraph.

10.1.1 Display Attributes

This variety of views can be provided by assigning one of the **display attributes** DYNAMIC, STATIC or HIDDEN to each constructor for each source view. Constructors are called static, hidden or dynamic if they are assigned the corresponding attribute. Static and dynamic constructors should not be confused with static and dynamic views.

These attributes have the following meanings:

HIDDEN: Nodes that describe a hidden constructor are never displayed. An animator shouldn't stop at sequents that describe hidden constructors.

DYNAMIC: Nodes that describe a dynamic constructor are displayed with the result updating the phrase in place when the phrase is evaluated. An animator should always stop at valid sequents that describe dynamic constructors.

STATIC: Nodes that describe a static constructor are displayed without the result updating the phrase in place when the phrase is evaluated. An animator should stop at valid sequents that describe static constructors except when the boolean of the animation is false. If the current sequent of the animation describes a static constructor then the focus of the animation should always be the subject of that sequent.

The subject of the root node of an animation history should never be hidden. This ensures that there is at least one sequent where the animator may stop. It also means that the focus of a complete animation is always the conclusion of the root node. If the root node describes a hidden constructor then it should be treated as static.

There are other conceivable attributes. Only experience will tell which are needed.

10.1.2 View Sets

The second way that a view may specify that certain phrases should be hidden is to explicitly list each constructor used in the view. This list is called the view set. A view set is a more flexible approach than assigning a display attribute to each constructor, because some instances of a constructor may be in the view set while others may be omitted. Thus a view set can provide a list of phrases for a particular program. This means that this mechanism is a way of interfacing an animator with an external tool such as a slice generator.

Omitting a phrase from the view set has the same effect as assigning the `HIDDEN` attribute to that phrase. Therefore we can define a view-specific step in terms of display attributes alone, by associating an attribute to each phrase as well as each constructor. An assignment of a `HIDDEN` attribute, from either the view set or the outermost constructor of the phrase, will override any other attribute assigned from another source.

For the reasons given in the previous subsection, the subject of the root node of an animation may never be hidden. If it is assigned the `HIDDEN` attribute it should be treated as `STATIC`.

10.1.3 Definition

Adding display attributes to phrases has the effect that the places where an animator should stop now depend on the particular view being animated. Therefore the definition of an animation step must be parameterised by a view.

The new definition is similar to the definition of a basic step. The main change is the addition of a notion of the validity of an animation itself, rather than of sequents in the animation. The definition of extension is also slightly modified; this is because it must handle the case where all phrases in a sub-evaluation are hidden. (This case is a generalisation of condition 14 in the definition of a basic step.)

If $\mathcal{V}$ is a view then a relational animation $(\mathcal{R}, b)$ is $\mathcal{V}$ -valid if $o(\mathcal{R}, b)$ is valid and either:

1. $o(\mathcal{R}, b)$ describes a static constructor and $b = \mathbf{tt}$.
2. $o(\mathcal{R}, b)$ describes a dynamic constructor.

If $(\mathcal{R}, b)$ and $(\mathcal{R}', b')$ are both $\mathcal{V}$ -valid relational animations, then $(\mathcal{R}', b')$ $\mathcal{V}$ -extends $(\mathcal{R}, b)$ if the following conditions hold:

1. $(\mathcal{R}', b')$ extends $(\mathcal{R}, b)$.
2. If $o(\mathcal{R}, b)$ is a local premise and $o(\mathcal{R}', b') = \text{conclusion}((\mathcal{R}', b')[o(\mathcal{R}, b)])$.

$(\mathcal{R}', b')$ **minimally** $\mathcal{V}$ -extends $(\mathcal{R}, b)$ if there is no $\mathcal{V}$ -valid relational animation $(\mathcal{R}'', b'')$ such that $(\mathcal{R}', b')$ $\mathcal{V}$ -extends $(\mathcal{R}'', b'')$ and $(\mathcal{R}'', b'')$ $\mathcal{V}$ -extends $(\mathcal{R}, b)$.

A **view-specific step** for a view $\mathcal{V}$ is written $\mathcal{A}_{R, \mathcal{V}}$. If $(\mathcal{R}, b)$ is a $\mathcal{V}$ -valid relational animation then $\mathcal{A}_{R, \mathcal{V}}(\mathcal{R}, b)$ is a minimal $\mathcal{V}$ -extension of $(\mathcal{R}, b)$.

Since an animation is only $\mathcal{V}$ -valid if its current sequent is valid, it follows that a view-specific step is equivalent to repeated applications of the basic step function.

Using this definition it is still possible that a step won't change the display. If the step maps an animation with a non-recursive local premise as the selected sequent to one that has the corresponding conclusion as the selected sequent, and this conclusion describes a static constructor, then the two animations may be displayed the same way. It isn't clear whether this behaviour is desirable or not. Unlike the case of a dynamic constructor, users may find it intuitive that, for example, the current procedure in a call graph could be evaluated in a single step without changing the display.

If this behaviour is found to be undesirable, there are two ways in which it could be changed. The first is to require the animator to display a status line, so that the difference between the two states is clear. The other is to change the definition of extension. This will have the effect that phrase evaluations will no longer always be possible when the selected node is a local premise (see below).

10.1.4 Highlighting Multiple Views

An animator may display several views at once. Each view may have different view sets or different assignments of display attributes to constructors. Therefore their corresponding step functions may give different results. One way of choosing which function to use is to let the user select one of the views and to use the definition of step associated with that view. Another way is to choose the definition which will result in the smallest extension of the animation for the current phrase. These approaches could be combined, with the "smallest extension" approach being used if the user doesn't select a particular view.

Whichever approach is taken, each view $\mathcal{V}$ should be updated when and only when the focus of the animation is in $\mathcal{V}$. If the definition of step is chosen for

another view, and results in a larger step than the definition corresponding to $\mathcal{V}$, then $\mathcal{V}$ should highlight the last phrase to be the focus in $\mathcal{V}$ during the evaluation of the step. This results in the current procedure always being highlighted in a call graph, for example.

It may be useful to indicate those views that don't display the current phrase, to show that the information they display isn't necessarily current. This could be done by changing the background or highlight for such views. This feature may be supplied automatically, or the language designer could have to specify whether it applies to each view.

10.1.5 Discussion

In Chapter 3 I mentioned three ways in which a view-specific step could be defined:

1. Each view could contain an appropriate definition of a step.
2. There could be a single definition of a step which was parameterised on the current view.
3. A view-specific step could be a repeated application of a fundamental definition of a step, stopping when it causes the display specified by the view to change.

I also claimed that I would give a definition of a step which would be parameterised on a view. In other words I claimed that I would take the second option. To what extent is this true?

The definition given above is certainly parameterised on a view. However, the specification of a view has been extended to include display attributes, which control the behaviour of the step function. The definition doesn't just take a set

of display rules and produce a view-specific step; it takes some information that is added to the view for the purpose of defining a view-specific step.

Nevertheless, the definition clearly does not fall into the first category. The specification of a view does not define a step from scratch, and it is not clear how it could do so. The display attributes only serve to parameterise an existing definition. So the definition combines elements of the first two approaches.

It is also the case that a view-specific step must be equivalent to a series of basic steps. Therefore the definition can be viewed as falling under the third approach. In this way of looking at the definition, display attributes encode when the display generated by a view will change. Thus the definition combines elements of all three approaches, but is best presented as an example of the second approach.

10.1.6 Summary

In the development of the definition of an animation step I deliberately used the basic view, because it gives more information than the others that I have considered. This resulted in the definition of a basic animation step. Other views don't show all the phrases of a program or don't replace each phrase by its result. The effect of this can be described by assigning a display attribute to each phrase or constructor in a view. The definition of an animation step can be parameterised on a view, giving a view-specific step.

10.2 Breaks, Marks and Interrupts

Section 2.3.3 showed that users often want to skip part of the evaluation of a program, so that they can concentrate on parts that they suspect to be buggy. This section discusses several operations that let them do this.

10.2.1 Break Points

A common way of examining part of an evaluation is to set a **break point** on the piece of code in question. The animator then stops when that piece of code is about to be evaluated, and the user can check the current environment and memory. An animator must provide a command to evaluate a program until it reaches the break point, or the n^{th} next break point, where n is given by the user. This program can be evaluated to that point silently, or automatically animated.

Users may set break points in many ways. The most common method is to specify a line of the source code (e.g. DBX [Sun] and MacPascal [Hue84]). A more flexible system is provided by the Masterscope facility of InterLisp [TM81, Tei78]. This allows a user to set break points at particular occurrences of function calls or operator applications. More flexible still is the facility provided by the Centaur system [mDD⁺85]. This allows a user to select a node of the abstract syntax using a mouse.

It should also be possible to set break points at all parts of the abstract syntax which match a certain pattern. Examples include all procedure calls, or all calls of a certain procedure, or all calls of that procedure with certain arguments. This facility is provided by the Prolog SPY package [Byr80] and the InterLisp Masterscope package [TM81].

Break points may easily be described in the theory of animation. A break point itself is simply a tag on a phrase. Running to the next break point consists of repeated by applying the step function until the current phrase has a break point set. Break points should be ignored at nodes which aren't in the selected view or the selected definition of a step.

10.2.2 Marks

A related feature is the ability to set **marks** during the evaluation, and run the evaluation backwards to a given mark. Marks differ from break points in that they are set at specific points of the evaluation rather than at parts of the abstract syntax. For example, a break point could be set at the beginning of a function, in which case the animator would stop whenever that function was called. By contrast, a mark would be set for a particular call of the function, and the evaluation could later be unwound to that point.

Marks can help people to experiment with a program. They can set a mark, change the current memory and see what happens, safe in the knowledge that they can run the evaluation backwards to the mark and undo the change to the memory.[ST83]

Like break points, marks may be implemented as a tag, this time on the focus of the animation. One way of running to the most recent mark is to repeatedly apply a reverse evaluation step (defined below in Section 10.3.2) until the focus reaches a marked term. As with break points, marks should be ignored on terms which can't be the focus in the selected view.

An alternative way of defining a mark is to push a copy of the current animation history onto a stack. Running back to the last mark then consists of popping the stack. Certain primitive implementations could implement marks this way by writing a saved state of the animator to a file.

10.2.3 Break Conditions

Another way to examine part of the evaluation is to run the evaluation until a specified condition (called a **break condition**) becomes true. This condition might be an assertion or invariant failing, a value being assigned to a certain memory location, or the stack exceeding a certain depth. This feature was first

provided by the AIDS system [Gri70] in 1969. It is becoming more common, with some authors suggesting that it should actually be built into new languages [HK85,PN81].

Break conditions differ from break points in that they are expressed in terms of data instead of code. If the data concerned is an argument to a function, this provides more control than setting a break point on that function, since correct calls of the function will be ignored. If the data is a memory location, the condition provides a form of control that can't be expressed with break points.

There are several ways that users can set break conditions, including the following:

1. Users can specify that the program should be run until a the contents of a specified memory location are changed, or until a specified value is assigned to that location. This can be implemented by comparing the contents of the memory component of the focus before the step function is applied with those of the memory component of the focus after the step is applied.
2. Another possibility is to allow users to set arbitrary expressions in the source language that must be evaluated in the current environment and memory after each step, such that the evaluation stops if they return a specified value. Such a condition could be specified as a simple equation or relation. This has a straightforward implementation. After each step the animator must evaluate the condition to see if it returns the specified value. usually the condition will not be allowed to perform any side-effects; if it does then he animator must undo them before continuing with the animation.
3. A third approach is for users to specify patterns that are matched against the subject of each instance of a rule in the animation, stopping the evaluation if the match succeeds. For example, the pattern `8652 := 2` will

detect an assignment of the value 2 to the location 8652. Similarly, the pattern `call foo 2` will detect any call of the function `foo` with argument 2.

Patterns can be extended by allowing them to contain wild cards. For example, if `_` is a wild card pattern, `8652 := _` will detect any assignment to location 8652, and `call foo _` will detect any call of the function `foo`.

Ideally, it should be possible to specify a memory location by an identifier that refers to it. The user interface should convert this into the address of the location, so that the animator will catch assignments via other identifiers that refer to the same location. However, it should not check assignments to that location when the variable is out of scope. For example, in a conventional implementation of an imperative language a stack location may be used by many different identifiers in different scopes.

10.2.4 Interrupts

One problem that can occur when skipping part of the evaluation is that the program might get into an infinite loop. Therefore animators should provide a way to interrupt the evaluation. Interrupts can also be used when the user spots a mistake during an automatic animation of the program. They can be implemented by polling after every n steps.

Since users sometimes interrupt a program to find that it is running correctly, just more slowly than they expected, It should be possible to resume the last command after interrupting an evaluation.

10.2.5 Break Levels

A more advanced version of resuming the last command after an interrupt or break is the notion of **break level** found in interactive Lisp systems such as

InterLisp [Tei78]. In these systems an interrupt or break point suspends the current evaluation, giving the user access to all the normal commands. The user may perform any desired commands, such as examining and changing the current memory, and then either resume or abort the original evaluation. If another interrupt occurs in this break level, the user is given another break level. The break levels form a stack; each may resume the command in the previous break level.

10.2.6 Summary

There are several ways that users may specify when an evaluation should stop. These features let users skip uninteresting parts of the evaluation and concentrate on interesting parts. They can easily be specified in terms of relational animations and animation steps.

10.3 Other Advanced Operations

This section shows how several other advanced operations can be defined in terms of relational animations. First it describes phrase evaluation and reverse evaluation. Together these support Lieberman's bug location procedure, which was described in Section 2.3.3. Then it describes a facility that lets users select which remote rule sets they want to see animated. Finally it describes augmented relational animations, which allow users to change the current memory or result of a phrase.

10.3.1 Phrase Evaluation

Users can be given the option of evaluating the current phrase in one go rather than stepping through the evaluation of that phrase. Many animators give this

choice for procedure calls (e.g. DBX [Sun] and Spy [Byr80]), but it can be extended to every phrase. I call this a **phrase evaluation**.

Phrase evaluation can be used to skip the uninteresting parts of an evaluation. It can also be used in conjunction with reverse evaluation (described below), marks (described above) or an undo facility to implement Lieberman's technique for bug location (see Section 2.3.3).

The phrase evaluation function is written $\mathcal{P}_R$ and is the same for every view. It is defined on a relational animation $(\mathcal{R}, b)$ as follows:

1. If $o(\mathcal{R}, b)$ is a local premise q then $\mathcal{P}_R(\mathcal{R}, b)$ is a minimal extension $(\mathcal{R}', b')$ of $(\mathcal{R}, b)$ such that $((\mathcal{R}', b')[q])$ is a complete animation.
2. If $b = \text{ff}$ then $\mathcal{P}_R(\mathcal{R}, b) = \mathcal{A}_R(\mathcal{R}, b)$.
3. If $(\mathcal{R}, b)$ is the initial relational animation of (p, m, e) then $\mathcal{P}_R(\mathcal{R}, b)$ is the complete relational animation of (p, m, e) .
4. If the above cases don't apply then $\mathcal{P}_R(\mathcal{R}, b) = (\mathcal{R}, b)$.

In section 10.1.3 I remarked that a view-specific step won't change the display when $o(\mathcal{R}, b)$ is a non-recursive local premise that describes a static constructor and there is no $\mathcal{V}$ -valid sequent between $o(\mathcal{R}, b)$ and the corresponding conclusion. This is also the case for phrase evaluation. If the definition of extension used for view-specific steps was changed as suggested in that section, then if $(\mathcal{R}', b') = \mathcal{P}_R(\mathcal{R}, b)$ then $o(\mathcal{R}', b')$ would not be the conclusion that corresponds to $o(\mathcal{R}, b)$. Users would probably find this confusing.

An animator should also provide a command to evaluate the program completely. This is called a **total evaluation**, and is written $\mathcal{C}_R$. $\mathcal{C}_R(\mathcal{R}, b)$ is the complete animation that extends of $(\mathcal{R}, b)$.

10.3.2 Reverse Evaluation

Reverse evaluation is the ability to step backwards through an evaluation as well as forwards. It was first suggested by Grishman [Gri70], who advocated it for experimenting with different paths of an evaluation, undoing choices as necessary. (Neal [Nea87] supports the provision of undo facilities as encouraging people to learn by experimenting.)

Zelkowitz [Zel73] advocated the use of reverse evaluation to find the context in which a run-time fault occurred, thus reducing the amount of trace information printed. Brna et al. [Brn88] make a similar observation about its use in Prolog debuggers.

Lieberman's technique for bug location described in Section 2.3.3 is an important use of reverse evaluation. This technique says that if the evaluation of a phrase gives an erroneous result, then the user should run the evaluation backwards to the start of the evaluation of that phrase and examine it in more detail. Alternatively the user could examine the evaluation of the phrase by stepping through it backwards.

Ideally all forward evaluation commands should have reverse evaluation counterparts. However, reverse evaluation is inefficient to implement, and may not always be possible (for example, it is difficult to implement if the animator is linked to a compiler rather than an interpreter). Some of the same functionality can be obtained using marks instead, as described in the next section.

Reverse evaluation can be specified easily in terms of forward steps. As usual with relational semantics, these definitions are only unique up to congruency.

A reverse animation step is written $\overline{\mathcal{A}_R}$. If $(\mathcal{R}, b)$ is a relational animation, then $\overline{\mathcal{A}_R}(\mathcal{R}, b)$ is the relational animation such that $\mathcal{A}_R(\overline{\mathcal{A}_R}(\mathcal{R}, b)) \cong (\mathcal{R}, b)$.

If $\mathcal{V}$ is a view, then a reverse step for that view is written $\overline{\mathcal{A}_{R,\mathcal{V}}}$. If $(\mathcal{R}, b)$ is a relational animation, then $\overline{\mathcal{A}_{R,\mathcal{V}}}(\mathcal{R}, b)$ is the relational animation such that $\mathcal{A}_{R,\mathcal{V}}(\overline{\mathcal{A}_{R,\mathcal{V}}}(\mathcal{R}, b)) \cong (\mathcal{R}, b)$.

A reverse phrase evaluation is written $\overline{\mathcal{P}_R}$. If $(\mathcal{R}, b)$ is a relational animation, then $\overline{\mathcal{P}_R}(\mathcal{R}, b)$ is the smallest relational animation such that $\overline{\mathcal{P}_R}(\overline{\mathcal{P}_R}(\mathcal{R}, b)) \cong (\mathcal{R}, b)$.

Unlike the definitions of $\overline{\mathcal{A}_R}$ and $\overline{\mathcal{A}_{R,\mathcal{V}}}$, the definition of $\overline{\mathcal{P}_R}$ must specify that $\overline{\mathcal{P}_R}(\mathcal{R}, b)$ is the smallest animation of those possible. This because the rhs of a recursive premise is the same as the rhs of the rule or instance. Therefore if $\alpha(\mathcal{R}, \mathbf{tt})$ is the conclusion of a recursive instance, then $\alpha(\overline{\mathcal{P}_R}(\mathcal{R}, b))$ is the premise that corresponds to that conclusion.

For example, consider the following excerpt of an animation history, where all the instances are recursive and the box indicates the current sequent:

$$\frac{\dots}{e \vdash \text{while } \dots \text{ do } \dots, m \Rightarrow \text{nil}, m'} \\ \frac{\dots \quad e \vdash \text{while } \dots \text{ do } \dots, m \Rightarrow \text{nil}, m'}{e \vdash \text{while } \dots \text{ do } \dots, m \Rightarrow \text{nil}, m'} \\ \frac{\dots \quad e \vdash \text{while } \dots \text{ do } \dots, m \Rightarrow \text{nil}, m'}{\boxed{e \vdash \dots, m \Rightarrow \text{nil}, m'}}$$

This could be the result of applying $\mathcal{P}_R$ to the following animation:

$$\frac{\dots \quad \boxed{e \vdash \text{while } \dots \text{ do } \dots, m \Rightarrow \text{nil}, m'}}{e \vdash \text{while } \dots \text{ do } \dots, m \Rightarrow \text{nil}, m'} \\ \frac{\dots \quad e \vdash \text{while } \dots \text{ do } \dots, m \Rightarrow \text{nil}, m'}{e \vdash \dots, m \Rightarrow \text{nil}, m'}$$

But it could also be the result of applying $\mathcal{P}_R$ to this animation:

$$\frac{\dots \quad \boxed{e \vdash \text{while } \dots \text{ do } \dots, m \Rightarrow \text{nil}, m'}}{e \vdash \dots, m \Rightarrow v, m'}$$

Therefore the definition of $\overline{\mathcal{P}_R}$ must define which of these animations is chosen.

A reverse total evaluation is written $\overline{\mathcal{C}_R}$. $\overline{\mathcal{C}_R}(\mathcal{R}, b)$ is the initial relational animation of $lst(\mathcal{R})$.

10.3.3 Remote Rule Sets

The definitions of step all treat a sub-evaluation in a remote rule set as a single step. However, sometimes users will want to see these sub-evaluations. For example, a Prolog animator might specify the matching of a clause against a goal in a separate rule set. Sometimes users may want to ignore this process, and at other times they may want to see it.

A degree of control can be provided by assigning display attributes to rule sets. These are similar to those assigned to constructors, but have the following meanings:

HIDDEN: An animator shouldn't stop at a call to a hidden rule set.

STATIC: An animator should stop at a call to a static rule set, as in the existing definitions of step, unless the constructor described by the call is hidden.

DYNAMIC: If the call doesn't describe a hidden constructor, then an animator should animate the evaluation of the call in the remote rule set. This animation should begin with the animator stopping at the remote premise. The next animation step should display the subject of the premise, and the following steps should animate the evaluation of that phrase in the remote rule set. Once that evaluation is complete, another step should switch back to the original rule set. If the constructor described by the call is dynamic the animator should show the result of the call.

If the call describes a hidden constructor, then the call should be treated as if it was a call to a hidden rule set.

TRANSPARENT: An animator should not animate the evaluation of the call in the remote rule set, but should animate any calls made to static or dynamic rule sets during that evaluation. For example, a specification of Prolog may include a call to search the database for a clause that unifies with the current goal. This operation may be divided into two sub-operations; one to find a clause that defines the predicate of the goal, and the other to perform the unification:

$$\frac{\begin{array}{c} \text{FIND} \\ \vdash \text{find}(e, p) \Rightarrow v' \end{array} \quad \begin{array}{c} \text{UNIFY} \\ \vdash \text{unify}(p, v') \Rightarrow v \end{array}}{\text{LOOKUP} \\ \vdash \text{lookup}(e, p) \Rightarrow v}$$

A view may want to animate the unification without animating the call to the **FIND** operation or the call to the **LOOKUP** operation itself. CODA is an example of a Prolog animator which behaves in this way.

More control could be provided by assigning display attributes to individual calls, allowing them to override the values for the rule set. For example, this would allow a user to see the lookup of all identifiers except procedure names.

10.3.4 Changing an Evaluation

Section 2.3.3 described how a user might want to change the result of part of a program and the side-effects caused by that part. One possible use of this feature is to recover from an error so that the rest of the evaluation can be examined. Another use is to avoid performing a lengthy sub-evaluation by substituting the correct result.

Users should be able to examine and change the current state at any stage of an evaluation, and should also be able to undo any such alteration. They should also be able to evaluate an phrase in the current environment and memory. A

common use for this is to examine the contents of the memory using symbolic names instead of locations. If such an evaluation is allowed to alter the memory, there should be some way of undoing these side-effects and recovering the main track of the evaluation.

To satisfy the definition of a relational animation, any such change would have to change other parts of the animation history. This change may be non-determinate; for example, if the result of an addition is changed then there are several corresponding changes that could be made to the summands. The changes could also affect the program text, which would usually not be desired.

Therefore the definition of an animation must be augmented to allow changes to the current sequent. The new definition introduces **user-defined terms** and **user-defined axioms**. These must be closed.

A **augmented relational animation** is a relational animation with the following modifications:

1. The lhm of a remote or non-trivial local premise may be replaced with a user-defined memory.
2. The rhm of a non-recursive remote premise may be replaced with a user-defined memory.
3. The rhm of the last premise of a non-recursive node may be replaced with a user-defined memory.
4. The rhm of the conclusion of a node that doesn't correspond to a recursive premise may be replaced with a user-defined memory.
5. The result of a non-recursive remote premise or a conclusion that doesn't correspond to a recursive premise may be replaced by a user-defined value. A node containing a user-defined value must still be an instance of a

relational rule, except that if the result of the conclusion is user-defined, that value need not be an instance of the result of the rule.

6. The child corresponding to a non-trivial local premise may be a user-defined axiom. The conclusion of this axiom and the premise must be the same, but the result and rhm may be user-specified. If the rhm isn't user-specified, it must be the same as the lhm.
7. A remote premise may have a corresponding child that consists of a user-defined axiom. This must match the restrictions given in the previous case.

The memories listed above are those that can be the current memory of an animation, as defined in Section 9.5. Thus the current memory can be changed at any time during an animation. Some care may be required; for example it may be possible to change the lhm of a premise to a value that doesn't satisfy the side-conditions of any rule that matches that premise.

The results listed in case 5 are those that can be the focus of an animation $(\mathcal{R}, b)$. Whenever the focus of an animation is a result of a premise or node, that result may be updated. As with user-defined memories care must be taken not to change a result to one that doesn't match any rules.

User-defined axioms allow the user to change the entire rhs of a premise without evaluating it. This operation should be available only when the premise is selected, and if the premise is remote then only when the boolean of the animation is false.

When the current sequent is a conclusion and the boolean of the animation is false, then the result of the conclusion can't be changed and the conclusion can't be replaced by a user-defined axiom. However, an animator could provide an operation which changes the rhs of the conclusion in lieu of a step, setting the boolean to true in the process.

The existing definitions of step apply to augmented animations. Stepping backwards over a user-defined memory or value erases that object.

10.3.5 Summary

Several advanced operations can be defined in terms of relational animations. These include phrase evaluation and reverse evaluation, selecting which rule sets to be displayed, and changing the current memory and result.

10.4 Chapter Summary

In this chapter I have shown that the basic definition of an animation step can be parameterised on the definition of a view. This has the effect that stepping through a view of a program will cause the corresponding display to be changed after almost every step. The only case in which the display won't be changed seems to be a natural case.

I have also shown that many other advanced operations can be defined in terms of my theory of animation. In particular, it can describe all the operations mentioned in Chapter 2 as helping the tasks that an animator can support. This demonstrates that the theory is sufficiently powerful to describe useful systems.

Some of these operations, such as phrase evaluation and reverse evaluation, are simple extrapolations of the definition of an animation step. Others, such as break points, can be defined directly in terms of relational animations. Some, such as changing the value of the current memory, are defined in terms of augmented relational animations.

This concludes the development of my theory of animation. The theory now includes a definition of an animation step, the ability to define multiple program-independent views, and definitions of many advanced operations. The

next chapter sketches an approach to specifying pure Prolog in terms of this theory. The following chapters present an implementation of an animator generator based on the theory, and discuss possible extensions.

Chapter 11

An Example Language: Prolog

In this chapter I consider how the theory defined in the preceding chapters could be used to specify a real programming language. The language I consider is Prolog, because there are several existing animators for Prolog with which a specification can be compared, and because it tests the theory on a different type of language from the imperative and eager functional languages considered so far. Since a full specification of Prolog would be a project in its own right, I sketch the outline of a specification of pure Prolog. I give the semantic rules for this subset of the language, and compare a simple example animation with the display produced by the CODA animator .

11.1 Review

The preceding chapters have defined a theory of animation. Such a theory is a prerequisite for the design of an animator generator. This is because an animator generated by such a system requires a definition of an evaluation step and a specification of how to display a program in mid-evaluation. These can be defined in terms of a semantic representation of a program in mid-evaluation.

Chapter 6 showed how a program in mid-evaluation can be represented in terms of relational semantics. A relational evaluation is a tree of instances of relational semantic rules. Repeated applications of the computation step function make the tree grow in a depth-first left-to-right fashion. The definition of a computation step is based on the transitions in transitional semantics, as defined in Chapter 5. The definitions of step in the two styles are shown to be equivalent in Chapter 7.

Chapter 8 refines the definition of a computation step to give a definition of an animation step. Animation steps are better suited for animation because they let a view highlight the result of each phrase as it evaluated and because they guide the user to the next phrase to be evaluated instead of jumping straight to it. The definition of animation steps is a straightforward development of the definition of computation steps.

Chapter 9 shows how a range of views can be defined in terms of in terms of relational animations. A view is specified by a display rule for each constructor and semantic rule. Each display rule specifies how an instance of the constructor or semantic rule will be displayed.

Chapter 10 shows how the definition of an animation step can be customised to particular views by using display attributes. It also shows that all the operations deemed important in Chapter 2 can be defined in terms of the theory. Several operations can be defined in terms of animation steps; others can be defined directly in terms of relational animations.

Section 6.1.5 shows that the form of relational rules admitted by the theory is adequate to describe all the rules in the definition of the dynamic semantics of Standard ML. Chapters 9 and 10 show that the theory can describe all the views and operations regarded as important to support the tasks described in Chapter 2. This implies that the theory is powerful enough to describe real systems.

The next two chapters support this claim. In this chapter I sketch an approach to specifying pure Prolog in terms of the theory. This shows that the theory can describe logic programming features as well as the imperative and eager functional features considered so far. In addition, the overview shows how we could write display rules that would produce the same display as an existing animator for Prolog.

Chapter 12 will describe the design and implementation of the prototype of an animator generator based on the theory. This shows that the theory has practical use as the basis of an implementation.

11.2 Discussion

Most of the examples in this thesis use the toy language Proc, which is an imperative language. If the theory of animation is a general theory, it should be able to describe other languages, including those from different paradigms. Hence it would be useful to attempt to specify a range of languages in terms of the theory. This chapter makes a small but useful start on this project by showing how pure Prolog might be specified.

Prolog is both a good and a bad choice of language with which to test the theory. On the one hand, there are several existing Prolog animators that example animations can be compared against [EB86,DC86,Byr80,Plu88]. Also, Prolog is a logic programming language, and so specifying it tests the generality of the theory. On the other hand, there isn't a standard definition of Prolog. Implementations can vary radically in their syntax, built-in operations and semantics.

In this chapter I present an outline of a specification, ignoring these problems. I don't give any built-in operations, and I consider only pure Prolog, without operations such as cut, assert, retract or disjunction. I also ignore user-generated

failures, so that the evaluation terminates after the first solution is discovered. The resulting specification captures the core issues involved in a semantics and associated display rules for a simple view of pure Prolog. This specification could form the core of a complete specification of a particular implementation.

I begin by giving the semantic rules for pure Prolog. Then I present an example animation of a simple program, showing how the definition of an animation step and an appropriate choice of display attributes produce a sequence of steps identical to that of CODA. Then I discuss how the notation for semantic display rules must be extended to allow us to specify a view similar to that of CODA [Plu88].

To show that the specification and the definition of an animation step give animations similar to those of CODA, I will show how the following simple program is displayed.

```
location(Person, Place) :-  
    at(Person, Place).
```

```
location(Person, Place) :-  
    visit(Person, Other),  
    at(Other, Place).
```

```
at(alan, room19).
```

```
? location(alan, X).
```

11.3 Semantics

A Prolog program consists of a database of clauses and a list of goals to be evaluated. The goals are evaluated left to right, by unifying them with the heads of clauses in the database. When a unification succeeds, the list of goals in the body of the clause is evaluated in the same way. Thus the evaluation is a depth-first traversal of a search tree.

The semantic rules given below assume the existence of several remote rule sets. The operations used are described here. These sample calls show the successful results of a call; in addition the first two calls shown may return *fail*.

$$\text{FIND} \\ e_1 \vdash \text{find}(p_1) \Rightarrow \text{success}(\text{clause}(p_2, p_3), e_2)$$

This call finds the first clause in the database e_1 with the same head predicate as the goal p_1 . It returns that clause, which has head p_2 and body p_3 , and the rest of the database. This call may return *fail*.

$$\text{UNIFY} \\ \vdash \text{unify}(p_1, p_2) \Rightarrow \text{subst}(v_1)$$

This call unifies the terms p_1 and p_2 . It returns the unifying substitution or *fail*.

$$\text{LIST} \\ \vdash \text{append}(p_1, p_2) \Rightarrow p_3$$

This call appends the list p_2 to the end of the list p_1 . It can never fail.

$$\text{SUBST} \\ \vdash v_1 \circ v_2 \Rightarrow v_3$$

This call returns the composition of the substitutions v_1 and v_2 . It can never fail.

$$\text{SUBST} \\ \vdash \text{restrict}(v_1, p_2) \Rightarrow v_3$$

This call returns the substitution v_1 restricted to the variables in p_2 . It can never fail.

$$\stackrel{\text{SUBST}}{\vdash} \text{apply}(v_1, p_1) \Rightarrow p_2$$

This call applies the substitutions v_1 to the term p_1 , returning the updated term. It can never fail.

The names of the remote rule sets will be omitted for the rest of this chapter, to make the rules and animation histories easier to read. The constructor names ensure that this doesn't introduce any ambiguity.

The first semantic rule specifies how a goal p_1 is evaluated in the context of a database e_1 . The evaluation finds the first clause in the database with the same head predicate as p_1 and then unifies the goal with that clause. Then it applies the unifying substitution to the body of the clause and evaluates the result, returning an answer substitution. Finally it returns the composition of the answer substitution with the unifying substitution. It also returns the rest of the database, which is used for backtracking, as explained below.

$$\frac{\begin{array}{l} e_1 \vdash \text{find}(p_1) \Rightarrow \text{success}(\text{clause}(p_2, p_3), e_2) \\ \vdash \text{unify}(p_1, p_2) \Rightarrow \text{subst}(v_1) \\ \vdash \text{apply}(v_1, p_3) \Rightarrow p_4 \\ \text{choice}(e_1, e_1) \vdash p_4 \Rightarrow \text{subst}(v_2) \\ \vdash \text{restrict}(v_2 \circ v_1, p_1) \Rightarrow v_3 \end{array}}{e_1 \vdash \text{goal}(p_1) \Rightarrow \text{success}(v_3, e_2)}$$

The next three rules specify what happens if a failure occurs during the evaluation of a goal. The first of these specifies that if the evaluation of the clause body fails, then the goal is evaluated in the context of the rest of the database. In other words, the evaluation tries to use the next matching clause in the database to evaluate the goal.

$$\begin{array}{c}
e_1 \vdash \text{find}(p_1) \Rightarrow \text{success}(\text{clause}(p_2, p_3), e_2) \\
\vdash \text{unify}(p_1, p_2) \Rightarrow \text{subst}(v_1) \\
\vdash \text{apply}(v_1, p_3) \Rightarrow p_4 \\
\text{choice}(e_1, e_1) \vdash p_4 \Rightarrow \text{fail} \\
\frac{e_2 \vdash \text{goal}(p_1) \Rightarrow v_2}{e_1 \vdash \text{goal}(p_1) \Rightarrow v_2}
\end{array}$$

The next rule specifies that the evaluation also tries the next matching clause if the goal doesn't unify with the first matching clause.

$$\begin{array}{c}
e_1 \vdash \text{find}(p_1) \Rightarrow \text{success}(\text{clause}(p_2, p_3), e_2) \\
\vdash \text{unify}(p_1, p_2) \Rightarrow \text{fail} \\
\frac{e_2 \vdash \text{goal}(p_1) \Rightarrow v_1}{e_1 \vdash \text{goal}(p_1) \Rightarrow v_1}
\end{array}$$

The last rule for the goal constructor specifies that if there are no more matching clauses then the evaluation of the goal fails.

$$\frac{e \vdash \text{find}(p) \Rightarrow \text{fail}}{e \vdash \text{goal}(p) \Rightarrow \text{fail}}$$

The remaining rules specify how a list of goals should be evaluated. Evaluation proceeds from left to right until the list is empty. If a goal fails, then the evaluation tries to find another solution for the previous goal, and continues the evaluation from there. This process is called backtracking. It has the effect that a goal may be re-evaluated after it has returned a successful result. This re-evaluation must ignore all clauses in the database that have matched the goal in previous evaluations. Therefore the environment contains both the full database and a version of the database with previously matched clauses removed. The first goal of the list is evaluated in the reduced version of the database; the others are evaluated in the full database. This pair of databases is called a choice point.

The first of these rules specifies that a list of goals is evaluated by evaluating the first goal, and if that succeeds then evaluating the rest of the list. If the

whole evaluation succeeds the result is the composition of the substitutions returned by each goal.

$$\begin{array}{c}
 e_2 \vdash \text{goal}(p_1) \Rightarrow \text{success}(v_1, e_3) \\
 \vdash \text{apply}(v_1, p_2) \Rightarrow p_3 \\
 \text{choice}(e_1, e_1) \vdash p_3 \Rightarrow \text{subst}(v_2) \\
 \vdash \text{restrict}(v_2 \circ v_1, p_1) \Rightarrow v_3 \\
 \hline
 \text{choice}(e_1, e_2) \vdash \text{cons}(p_1, p_2) \Rightarrow \text{subst}(v_3)
 \end{array}$$

The next rule specifies that if the evaluation of the first goal fails then the evaluation of the whole list fails.

$$\frac{e_2 \vdash \text{goal}(p_1) \Rightarrow \text{fail}}{\text{choice}(e_1, e_2) \vdash \text{cons}(p_1, p_2) \Rightarrow \text{fail}}$$

The next rule specifies that if the evaluation of the first goal succeeds but the evaluation of the rest of the list fails, then the list is re-evaluated with the evaluation of the first goal ignoring any previously matched clauses. This is where backtracking occurs.

$$\begin{array}{c}
 e_2 \vdash \text{goal}(p_1) \Rightarrow \text{success}(v_1, e_3) \\
 \vdash \text{apply}(v_1, p_2) \Rightarrow p_3 \\
 \text{choice}(e_1, e_1) \vdash p_3 \Rightarrow \text{subst}(v_2) \\
 \text{choice}(e_1, e_3) \vdash \text{cons}(p_1, p_2) \Rightarrow v_2 \\
 \hline
 \text{choice}(e_1, e_2) \vdash \text{cons}(p_1, p_2) \Rightarrow v_2
 \end{array}$$

The last rule specifies that the empty list evaluates to the empty substitution.

$$\frac{}{\epsilon \vdash \text{nil} \Rightarrow \text{subst}(\{\})}$$

11.4 An Example Animation

The following example animates the simple program give above. At each step the display shows how this program is displayed by the CODA system (up to renaming of variables). To keep the animation small I assume that the details of the unification process are hidden; this is user-selectable in CODA. In this view the constructors *goal*, *nil*, *find*, *append* and *o* are hidden; *cons* and *unify* are static. The display rules for this view will be given in the next section.

Phrases of the program that appear in the animation are written in *italic*. To keep the size of the sequents manageable, I use the abbreviations listed in the following table:

<i>e</i> :	The complete database.
<i>p'</i> :	<i>location(Person, Place)</i>
<i>p''</i> :	<i>at(Person, Place)</i>
<i>e'</i> :	<i>e</i> without the first clause.
<i>p'''</i> :	<i>at(alan, room19)</i>
<i>e''</i> :	<i>e</i> without <i>p'''</i> .
<i>v</i> :	$\{Person \mapsto alan, Place \mapsto X\}$

The initial animation history is an empty tree, and the animator just displays the initial goal list. CODA doesn't generate a display for the initial animation history; instead it automatically applies the first step. This minor difference in behaviour doesn't affect the rest of the animation. A display of the initial goal list in the style of CODA would be as follows. The $\Rightarrow$ arrow indicates that the first goal (and in this case, the only goal) of the list is the next to be evaluated.

$\Rightarrow$ *location(alan, X)*.

The first step produces the following animation. There isn't enough space to show it as a tree, but for this program the animation has only one path of any

interest at all, so the space limitation is unimportant. The upper node of the two below corresponds to the first premise of the lower node.

$$\begin{array}{c}
 e \vdash \text{find}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(\text{clause}(p', p''), e') \\
 \boxed{\vdash \text{unify}(\text{location}(\text{alan}, X), p') \Rightarrow \text{subst}(v_1)} \\
 \vdash \text{apply}(v_1, p'') \Rightarrow p_4 \\
 \text{choice}(e, e) \vdash p_4 \Rightarrow \text{subst}(v_2) \\
 \vdash \text{restrict}(v_2 \circ v_1, \dots) \Rightarrow v_3 \\
 \hline
 e \vdash \text{goal}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(v_3, e')
 \end{array}$$

$$\begin{array}{c}
 e \vdash \text{goal}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(v_1, e_2) \\
 \vdash \text{apply}(v_1, \text{nil}) \Rightarrow p_3 \\
 \text{choice}(e, e) \vdash p_3 \Rightarrow \text{subst}(v_2) \\
 \vdash \text{restrict}(v_2 \circ v_1, \dots) \Rightarrow v_3 \\
 \hline
 \text{choice}(e, e) \vdash \text{cons}(\text{location}(\text{alan}, X), \text{nil}) \Rightarrow \text{subst}(v_3)
 \end{array}$$

CODA produces the following display at this point. The first line is the goal to be evaluated, and the rest of the display is the first matching clause. The `*>` arrow signifies that the evaluation is about to begin the unification process:

```

location(alan, X).

*> location(Person, Place) :-
    at(Person, Place).

```

The next step performs the unification and applies the resulting substitution. Recall that v is an abbreviation for the substitution $\{Person \mapsto \text{alan}, Place \mapsto X\}$:

$$\begin{array}{c}
e \vdash \text{find}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(\text{clause}(p', p''), e') \\
\vdash \text{unify}(\text{location}(\text{alan}, X), p') \Rightarrow \text{subst}(v) \\
\vdash \text{apply}(v, p'') \Rightarrow \text{cons}(\text{at}(\text{alan}, \text{Place}), \text{nil}) \\
\boxed{\text{choice}(e, e) \vdash \text{cons}(\text{at}(\text{alan}, \text{Place}), \text{nil}) \Rightarrow \text{subst}(v_2)} \\
\vdash \text{restrict}(v_2 \circ v, \dots) \Rightarrow v_3 \\
\hline
e \vdash \text{goal}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(v_3, e')
\end{array}$$

$$\begin{array}{c}
e \vdash \text{goal}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(v_1, e_2) \\
\vdash \text{apply}(v_1, \text{nil}) \Rightarrow p_3 \\
\text{choice}(e, e) \vdash p_3 \Rightarrow \text{subst}(v_2) \\
\vdash \text{restrict}(v_2 \circ v_1, \dots) \Rightarrow v_3 \\
\hline
\text{choice}(e, e) \vdash \text{cons}(\text{location}(\text{alan}, X), \text{nil}) \Rightarrow \text{subst}(v_3)
\end{array}$$

CODA updates the previous display to show the effect of the unification. The $\Rightarrow$ arrow indicates that the evaluation is about to evaluate the first (and only) goal in the body of the clause:

`location(alan, X).`

`location(alan, X) :-`

`=> at(alan, X).`

The next step begins the evaluation of the next goal. The evaluation stops before the goal is unified with the first matching clause. In this animation the second node from the top corresponds to the fourth premise of the node beneath it:

$$\begin{array}{c}
e \vdash \text{find}(\text{at}(\text{alan}, \text{Place})) \Rightarrow \text{success}(\text{clause}(p''', \text{nil}), e'') \\
\boxed{\vdash \text{unify}(\text{at}(\text{alan}, \text{Place}), p''') \Rightarrow \text{subst}(v_1)} \\
\vdash \text{apply}(v_1, \text{nil}) \Rightarrow p_4 \\
\text{choice}(e, e) \vdash p_4 \Rightarrow \text{subst}(v_2) \\
\vdash \text{restrict}(v_2 \circ v_1, \dots) \Rightarrow v_3 \\
\hline
e \vdash \text{goal}(\text{at}(\text{alan}, \text{Place})) \Rightarrow \text{success}(v_3, e'') \\
\\
e \vdash \text{goal}(\text{at}(\text{alan}, \text{Place})) \Rightarrow \text{success}(v_1, e_2) \\
\vdash \text{apply}(v_1, \text{nil}) \Rightarrow p_3 \\
\text{choice}(e, e) \vdash p_3 \Rightarrow \text{subst}(v_2) \\
\vdash \text{restrict}(v_2 \circ v_1, \dots) \Rightarrow v_3 \\
\hline
\text{choice}(e, e) \vdash \text{cons}(\text{at}(\text{alan}, \text{Place}), \text{nil}) \Rightarrow \text{subst}(v_3) \\
\\
e \vdash \text{find}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(\text{clause}(p', p''), e') \\
\vdash \text{unify}(\text{location}(\text{alan}, X), p') \Rightarrow \text{subst}(v) \\
\vdash \text{apply}(v, p'') \Rightarrow \text{cons}(\text{at}(\text{alan}, \text{Place}), \text{nil}) \\
\text{choice}(e, e) \vdash \text{cons}(\text{at}(\text{alan}, \text{Place}), \text{nil}) \Rightarrow \text{subst}(v_2) \\
\vdash \text{restrict}(v_2 \circ v, \dots) \Rightarrow v_3 \\
\hline
e \vdash \text{goal}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(v_3, e') \\
\\
e \vdash \text{goal}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(v_1, e_2) \\
\vdash \text{apply}(v_1, \text{nil}) \Rightarrow p_3 \\
\text{choice}(e, e) \vdash p_3 \Rightarrow \text{subst}(v_2) \\
\vdash \text{restrict}(v_2 \circ v_1, \dots) \Rightarrow v_3 \\
\hline
\text{choice}(e, e) \vdash \text{cons}(\text{location}(\text{alan}, X), \text{nil}) \Rightarrow \text{subst}(v_3)
\end{array}$$

CODA replaces the previous display with the display of the evaluation of this goal:

at(alan, X).

*> at(alan, room19).

The next step performs the unification and applies the unifying substitution to the body of the clause. Since the body is `nil`, which is a hidden constructor, the evaluation continues to the conclusion of the `cons` node:

$$\begin{array}{l}
 e \vdash \text{find}(\text{at}(\text{alan}, \text{Place})) \Rightarrow \text{success}(\text{clause}(p''', \text{nil}), e'') \\
 \quad \vdash \text{unify}(\text{at}(\text{alan}, \text{Place}), p''') \Rightarrow \text{subst}(\{X \mapsto \text{room19}\}) \\
 \quad \vdash \text{apply}(\{X \mapsto \text{room19}\}, \text{nil}) \Rightarrow \text{nil} \\
 \text{choice}(e, e) \vdash \text{nil} \Rightarrow \text{subst}(\{\}) \\
 \quad \vdash \text{restrict}(\{\} \circ \{X \mapsto \text{room19}\}, \dots) \Rightarrow \{X \mapsto \text{room19}\} \\
 \hline
 e \vdash \text{goal}(\text{at}(\text{alan}, \text{Place})) \Rightarrow \text{success}(\{X \mapsto \text{room19}\}, e'')
 \end{array}$$

$$\begin{array}{l}
 e \vdash \text{goal}(\text{at}(\text{alan}, \text{Place})) \Rightarrow \text{success}(\{X \mapsto \text{room19}\}, e'') \\
 \quad \vdash \text{apply}(\{X \mapsto \text{room19}\}, \text{nil}) \Rightarrow \text{nil} \\
 \text{choice}(e, e) \vdash \text{nil} \Rightarrow \text{subst}(\{\}) \\
 \quad \vdash \text{restrict}(\{\} \circ (\{X \mapsto \text{room19}\}, \dots) \Rightarrow (\{X \mapsto \text{room19}\}) \\
 \hline
 \boxed{\text{choice}(e, e) \vdash \text{cons}(\text{at}(\text{alan}, \text{Place}), \text{nil}) \Rightarrow \text{subst}(\{X \mapsto \text{room19}\})}
 \end{array}$$

$$\begin{array}{l}
 e \vdash \text{find}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(\text{clause}(p', p''), e') \\
 \quad \vdash \text{unify}(\text{location}(\text{alan}, X), p') \Rightarrow \text{subst}(v) \\
 \quad \vdash \text{apply}(v, p'') \Rightarrow \text{cons}(\text{at}(\text{alan}, \text{Place}), \text{nil}) \\
 \text{choice}(e, e) \vdash \text{cons}(\text{at}(\text{alan}, \text{Place}), \text{nil}) \Rightarrow \text{subst}(v_2) \\
 \quad \vdash \text{restrict}(v_2 \circ v, \dots) \Rightarrow v_3 \\
 \hline
 e \vdash \text{goal}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(v_3, e')
 \end{array}$$

$$\begin{array}{l}
 e \vdash \text{goal}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(v_1, e_2) \\
 \quad \vdash \text{apply}(v_1, \text{nil}) \Rightarrow p_3 \\
 \text{choice}(e, e) \vdash p_3 \Rightarrow \text{subst}(v_2) \\
 \quad \vdash \text{restrict}(v_2 \circ v_1, \dots) \Rightarrow v_3 \\
 \hline
 \text{choice}(e, e) \vdash \text{cons}(\text{location}(\text{alan}, X), \text{nil}) \Rightarrow \text{subst}(v_3)
 \end{array}$$

Since the evaluation of the sub-goal is complete, CODA returns to the display of the parent goal. The `=>` shows that the evaluation of the body of the clause is now complete:

```
location(alan, room19).
```

```
location(alan, room19) :-  
    at(alan, room19) => .
```

The next step completes the animation:

$$\begin{array}{l}
e \vdash \text{find}(\text{at}(\text{alan}, \text{Place})) \Rightarrow \text{success}(\text{clause}(p''', \text{nil}), e'') \\
\quad \vdash \text{unify}(\text{at}(\text{alan}, \text{Place}), p''') \Rightarrow \text{subst}(\{X \mapsto \text{room19}\}) \\
\quad \vdash \text{apply}(\{X \mapsto \text{room19}\}, \text{nil}) \Rightarrow \text{nil} \\
\quad \text{choice}(e, e) \vdash \text{nil} \Rightarrow \text{subst}(\{\}) \\
\quad \vdash \text{restrict}(\{\} \circ \{X \mapsto \text{room19}\}, \dots) \Rightarrow \{X \mapsto \text{room19}\} \\
\hline
e \vdash \text{goal}(\text{at}(\text{alan}, \text{Place})) \Rightarrow \text{success}(\{X \mapsto \text{room19}\}, e'')
\end{array}$$

$$\begin{array}{l}
e \vdash \text{goal}(\text{at}(\text{alan}, \text{Place})) \Rightarrow \text{success}(\{X \mapsto \text{room19}\}, e'') \\
\quad \vdash \text{apply}(\{X \mapsto \text{room19}\}, \text{nil}) \Rightarrow \text{nil} \\
\quad \text{choice}(e, e) \vdash \text{nil} \Rightarrow \text{subst}(\{\}) \\
\quad \vdash \text{restrict}(\{\} \circ \{X \mapsto \text{room19}\}, \dots) \Rightarrow \{X \mapsto \text{room19}\} \\
\hline
\text{choice}(e, e) \vdash \text{cons}(\text{at}(\text{alan}, \text{Place}), \text{nil}) \Rightarrow \text{subst}(\{X \mapsto \text{room19}\})
\end{array}$$

$$\begin{array}{l}
e \vdash \text{find}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(\text{clause}(p', p''), e') \\
\quad \vdash \text{unify}(\text{location}(\text{alan}, X), p') \Rightarrow \text{subst}(v) \\
\quad \vdash \text{apply}(v, p'') \Rightarrow \text{cons}(\text{at}(\text{alan}, \text{Place}), \text{nil}) \\
\quad \text{choice}(e, e) \vdash \text{cons}(\text{at}(\text{alan}, \text{Place}), \text{nil}) \Rightarrow \text{subst}(\{X \mapsto \text{room19}\}) \\
\quad \vdash \text{restrict}(\{X \mapsto \text{room19}\} \circ v, \dots) \Rightarrow \{X \mapsto \text{room19}\} \\
\hline
e \vdash \text{goal}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(\{X \mapsto \text{room19}\}, e')
\end{array}$$

$$\begin{array}{l}
e \vdash \text{goal}(\text{location}(\text{alan}, X)) \Rightarrow \text{success}(\{X \mapsto \text{room19}\}, e') \\
\quad \vdash \text{apply}(\{X \mapsto \text{room19}\}, \text{nil}) \Rightarrow \text{nil} \\
\quad \text{choice}(e, e) \vdash \exists \Rightarrow \text{subst}(\{\}) \\
\quad \vdash \text{restrict}(\{\} \circ \{X \mapsto \text{room19}\}, \dots) \Rightarrow \{X \mapsto \text{room19}\} \\
\hline
\boxed{\text{choice}(e, e) \vdash \text{cons}(\text{location}(\text{alan}, X), \text{nil}) \Rightarrow \text{subst}(\{X \mapsto \text{room19}\})}
\end{array}$$

At the end of an animation CODA shows the result substitution. An alternative display shows the goal with the answer substitution applied:

`location(alan, room19) => .`

```
location(alan, room19) :-
    at(alan, room19) => .
```

This example shows that the definition of an animation step, together with the display attributes for the constructors, gives the same steps as CODA. The next section discusses the generation of the displays from the animation histories.

11.5 Display Rules

The CODA-like displays given in the previous section are hard to specify using the notation for display rules developed in Chapter 9. The reason seems to be that Prolog's use of unification has the effect that a variable may be instantiated by an operation that occurs after the point in the evaluation where the variable is to be displayed.

For example, an initial attempt to specify the display of a goal might use the rule shown below. This rule displays the goal to be evaluated above the matching clause. If the body of the clause is being evaluated, its display is generated from that sub-valuation. Otherwise it is displayed by the appropriate variable.

Goal:

$$\begin{aligned}
 \$\$ &= p_1 \$n\$n \\
 &\langle \&4 \ ?!4 \\
 &\langle p_3 \ ??2 \ p_2 \rangle
 \end{aligned}$$

The problem with this rule is that the variables in the goal are not updated by the unification of the goal with the head of the clause, nor when the variables are unified with values in the evaluation of the body of the clause.

One way of rectifying this situation would be to change the semantic rules to carry more information around with them. the evaluation of each goal could include a term representing the parent goal, and each new substitution could be applied to the parent. However, this approach is unsatisfactory. The semantic rules should specify the evaluation of a language, and should not be encumbered with display information. The excess information would result in specifications that were harder to read and to reason about.

A better approach is to pass the information about in the display rules themselves, using parameters. The display rule for a goal list can take the parent goal and the evaluated part of the list as parameters. If the display rule can apply a substitution to a term then it can produce the desired display.

Chapter 9 showed how parameters could be added to display rules. However, the need to apply a substitution to a term requires a new construct in the display language. Substitutions are defined as part of the language specification, and the only operations on them are those of the language specification. Therefore the display rules need a way of invoking a semantic operation on terms.

The obvious way to do this is to add a construct that calls an operation in a remote rule set – in this case, applying a substitution. I write this construct as follows:

$$[\overset{\text{SUBST}}{\vdash} \text{ apply}(v_1, p_2) \Rightarrow p_5] (\dots)$$

This means that the substitution v_1 is applied to the phrase p_2 to produce the result p_5 ; this is then used in the following expression.

With this construct we can specify the above displays with the following display rules. Both display rules are fairly complicated, and stretch the concise notation that I've used to the limits of readability.

The first display rule displays the evaluation of a goal. If the body of the clause is being evaluated, that sub-evaluation is displayed with the goal and an

empty list as parameters, as discussed above. Since the goal has been unified with the head of the clause, the unifying substitution is applied to the goal before it is passed as a parameter. If the body of the clause is not being evaluated yet, then the display is constructed from the relevant parts of the current node. Again, if the goal has been unified with the head of the clause then the substitution is applied to the goal before it is displayed. The equations are used here simply to structure the display expressions; they have no other significance. In particular, there is no highlighting in this view.

Goal:

$$\$\$ = \$4 \text{ ?!4 } \langle \$3 \text{ ??2 } \$2 \rangle$$

$$\$2 = p_1 \text{ \$n\$n " * > " } p_2 \text{ " : - " } \$n \text{ \$t } \{p_3\} \text{ " . "}$$

$$\$3 = [\text{SUBST } \vdash \text{ apply}(v_1, p_2) \Rightarrow p_5] p_5 \text{ \$n\$n } p_5 \text{ " : - " } \$n \text{ \$t " } => \text{ "{p}_4\text{ " . "}$$

$$\$4 = [\text{SUBST } \vdash \text{ apply}(v_1, p_2) \Rightarrow p_5] \&4[p_5, \text{nil}]$$

The second display rule displays a list of goals. If the first goal in the list is being evaluated, then the display of the list is the display of that goal. Otherwise the current substitution is applied to the parameters and they are used to form what is effectively the display of the parent goal, as described above. The three different equations correspond to the cases when the evaluation of the tail of the list has not begun, is in progress, or has finished.

Cons[goal,list]:

```

$$ = !1 ?!1 <$2 ?!3 <$4 ??? $3>
      SUBST
$2 = [⊢ apply(v1, goal) ⇒ goal']
      SUBST
      [⊢ apply(v1, list) ⇒ list']
      &3[goal',list]
      SUBST
$3 = [⊢ apply(v1, goal) ⇒ goal']
      SUBST
      [⊢ apply(v1, list) ⇒ list']
      goal' $n $n goal' " : -" $n $t {list' $n " => " p3} "."
      SUBST
$4 = [⊢ apply(v2 ◦ v1, goal) ⇒ goal']
      SUBST
      [⊢ apply(v2 ◦ v1, list) ⇒ list']
      SUBST
      [⊢ apply(v2, p3) ⇒ p'3]
      goal' $n $n goal' " : -" $n $t {list' $n p'3} " => ."

```

One can imagine other ways of defining the semantics of Prolog and the associated display rules for this view, but it is likely that they will all need some way of calling a semantic operation in the display rule. For example, an approach based on a common implementation technique would involve looking up the current value bound to a variable in an environment. This would still need a call to the corresponding semantic operation in the display rule.

One problem with this technique is that it is now possible to write display rules that will fail because the call to a semantic operation is badly defined. Similarly, it is possible to write display rules that will fail to terminate. This situation is no worse than having to write an animator from scratch, but is less pleasant than the secure position that we had before introducing this construct.

11.6 Chapter Summary

The semantics of pure Prolog can be defined by a small number of relational semantic rules and associated operations. The definition of animation step, in conjunction with an appropriate choice of display attributes, produces a sequence of steps equivalent to that of CODA [Plu88], an existing animator for Prolog. A new construct is needed to produce the same displays as CODA from the semantic model of animation.

This exercise shows that the theory of animation developed in the previous chapters can be extended to programming languages other than the imperative and eager functional ones that were used as examples in its development. However, the theory may need some extensions to cope with some of the features found in these languages. This suggests that it would be useful to experiment with specifying languages in a range of paradigms. Such experimentation will be easier with an implementation of an animator generator based on the theory, such as that described in the next chapter.

Chapter 12

The Animator Generator

This chapter describes the design and implementation of an animator generator based on the theory presented in the preceding chapters. This implementation is called The Animator Generator(with capital letters). It takes a specification of a language L , written in a specification language called LSL, and generates an animator for L . This animator provides multiple views and most of the operations defined in Chapter 10. It demonstrates that the theory can describe realistic animators, although it is not production quality software. It also provides a tool for experimenting with specifications of different languages.

12.1 Review of the Theory

The theory presented in the preceding chapters is based on relational semantics. In this formalism a language is specified by an abstract syntax and some semantic rules. These rules are grouped into rule sets such that each set describes a particular operation.

The theory defines a fundamental animation step that is independent of a particular language. This step is defined in terms of an inference tree. Repeated

applications of the step function construct the tree in a left-to-right depth-first order, in accordance with the operational reading of of the semantic rules.

A view of a program in mid-evaluation can be specified by display rules for each constructor and semantic rule in the language. These display rules specify how an instance of a constructor or semantic rule is displayed. Several views can be defined for each language.

The definition of a step can be parameterised on a view by assigning display attributes for that view to each constructor and rule set. If a constructor is not displayed by the view then it should be given the `HIDDEN` attribute. An animator should not stop at a hidden constructor. If the result of a constructor is not interesting then that constructor should be given the `STATIC` attribute. An animator should only stop once at the conclusion of a node that describes a static constructor, and should display the subject of that conclusion. If the result of a constructor is interesting then it should be assigned the `DYNAMIC` constructor and the animator should stop twice at the conclusion of nodes that describe that constructor. First it should display the subject, then the result.

Several advanced operations are defined in the theory. Phrase evaluation, reverse evaluation, break points and marks are defined in terms of view-specific steps. Changing the current memory and result is defined in terms of augmented relational animation histories.

The relational rules admitted by the theory can describe the dynamic semantics of Standard ML and the semantics of pure Prolog. The range of views and operations defined in the theory covers those provided by most existing program-independent animators. The rest of this chapter shows that the theory can form the basis of the implementation of an animator generator. Together, these facts demonstrate the generality and practicality of the theory.

12.2 Designing the Implementation

Animators can be implemented in many ways, including the following:

Monitors: Most commercial debuggers monitor the state of a compiled program. This provides a reasonably fast evaluation, and ensures that the object code being debugged is the code that the compiler produces.

Tracers: Tracers print a record of procedure calls and side effects of selected variables. These traces must be animated after the program has finished.

Interpreters: Interpreters provide flexible system for program development and debugging. They are also quick to implement. For these reasons many advanced programming environments are still interpreted, even though interpreters tend to be slow..

The aim of The Animator Generator is to illustrate and test the theory. It is not required to interface to existing compilers, nor to be particularly efficient. Therefore I have chosen the most flexible and easiest to implement of these forms: an interpreter.

The interpreter is a straightforward implementation of the theory. Each animator stores an encoded version of the abstract syntax, semantic rules and display rules. The step function (and the operations defined in terms of it) build an inference tree from instances of the semantic rules. The nodes of this tree contain a little extra information for the display routines, but otherwise they are basically the same as the trees in the formal definition of relational animation histories.

The display routines traverse encoded versions of the display rules at each node and constructor. The variables in these display rules point to the

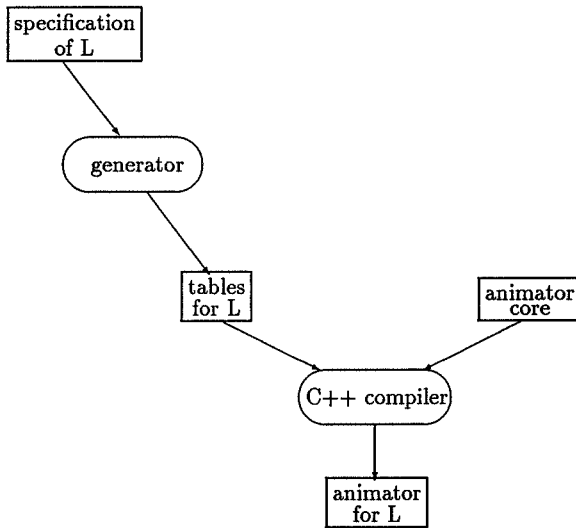
appropriate sub-components of the node or constructor. The extra information mentioned in the previous paragraph tells the display algorithm whether the current object is or has been selected. The display algorithm uses this to decide which branch of a conditional to take and when to highlight an object.

The implementation optimises the encoding of stack views. the language designer may indicate that a premise of a semantic rule is the start of the evaluation of a new procedure. When an instance of such a premise is evaluated, the current node is pushed onto a stack. Two special types of view use this stack to display the procedure call stack and just the current procedure.

The specification language is called LSL, which stands for Language Specification Language. LSL is a straightforward translation into ASCII of the notation used in this thesis. Each LSL specification includes specifications of abstract syntax, grammar, semantic rules, display rules and display attributes. LSL supports several of the constructions use to define views in Chapter 9. It doesn't provide all of them, but it could be extended to do so.

The Animator Generator demonstrates that the concepts developed in this thesis can produce a viable implementation. It isn't particularly sophisticated; it lacks some user interface features, provides only dynamic, alphanumeric views, and needs an interface to an editor. However, it does produce usable animators from language specifications, and these animators support multiple views as well as most of the operations listed in Chapter 10. All the pictures of displays in this thesis are produced by The Animator Generator.

The Animator Generator generates animators from specifications written in LSL. The system has two parts; the generator proper and the core animation functions:



The generator processes the language description and produces:

1. A YACC description of a parser for the language.
2. Arrays of sequents, terms, rules, and so forth.
3. Display functions.

The animator for a language comprises the parser produced from the YACC description, the arrays listed above and a core of routines describing the animation operations independent of the particular language. This core of each animator does most of the work.

12.3 LSL

An LSL specification consists of a header followed by a sequence of variable definitions, rule set definitions and view definitions. Each rule set defines constructors, semantic rules, and possibly views for these rules and constructors (the view definitions can be written outside of any rule set instead). As in the theory, the rule sets form a hierarchy. This is ensured by allowing remote premises to reference only rule sets that have been previously defined. The main rule set defines the grammar of the language.

Appendix C gives a specification of Proc in LSL. The examples presented in this chapter are taken from that specification.

12.3.1 Semantic Rules

Semantic rules are relational rules, as defined in Chapter 6. A simple ASCII notation is used to write them. Each semantic rule is given a name. The abstract syntax terms contained in the rules are built from constructors and variables. By convention, constructor and variable names are written entirely in lower case and semantic rule names are capitalised.

For example, the following rule:

$$\frac{\vdash \text{lookup}(e, x), m \Rightarrow v', m}{\vdash \text{lookup}(e\uparrow(x', v), x), m \Rightarrow v', m} \quad \text{IDENT} \quad \vdash \text{eq}(x, x') \Rightarrow \text{ff}$$

is written in LSL as follows:

```

Lookup_env2:
(|- lookup_env (e, y), m ==> v1, m1)
-----
(|- lookup_env (add_env (e, x, v), y), m ==> v1, m1)
PROVIDED
ID ( |- equal (x, y) ==> false)

```

The only notation in LSL that isn't a straightforward ASCII version of the notation used in this thesis affects local premises; these can be enclosed in square brackets. This is because each animator maintains a stack of nodes of the animation tree, analogous to the procedure call stack. When the animator encounters a premise enclosed in square brackets while it is animating a program, it pushes the current node onto the stack. The stack is used to provide an efficient implementation of views of the procedure call stack.

Each rule set also defines the set of constructors described by rules in that rule set, and the types of all constructors used in that rule set, whether described by rules or just appearing in premises.

Constructors must be assigned one of the types PHRASE, VALUE, ENV, MEMORY, VALUE ENV or VALUE PHRASE. Variables must be assigned one of the types PHRASE, VALUE, ENV or MEMORY.

All constructors and variables are global, as are the types of variables, but each assignment of types to constructors is local to a rule set. Thus a constructor may be an environment in one rule set and a phrase or value in another; for example a rule set defining declarations will produce values which are mappings of identifiers to values, while a rule set that evaluates expressions will use the same mappings as environments.

The root rule set must include **parse rules** for the language. These are similar to the rules used in the YACC parser generator. Each rule specifies a

term that a successful parse of the rule will create. For example, this is the rule from the grammar of Proc that parses while loops:

```
phrase:      WHILE phrase DO phrase ENDWHILE :
              while ($2, $4)
```

The \$2 and \$4 symbols refer to the second and fourth objects on the right hand side of the rule, in other words the two sub-phrases. Thus a successful parse of this rule will return the appropriate term of the abstract syntax. Non-terminals of the grammar must be declared using the `%token%` construct, for example:

```
%token WHILE DO ENDWHILE
```

The root rule set must also contain a default initial memory and environment. All programs are evaluated in these unless the user specifically overrides them.

LSL provides little support for lexical analysis. The Animator Generator comes with a default lexical analyser. The root rule set can specify keywords to customise the default analyser to a specific language, using the `%keyword` construct:

```
%keyword WHILE "while"
```

However, there is no way to change the definition of what constitutes a word or a special symbol from within LSL. Language definers may provide their own lexical analyser written in C or C++ (perhaps by modifying the sample one).

LSL provides **built-in operations** on identifiers, integers and booleans. These are available as rule sets that may be called from LSL rules. The language definer may define constructors with the same names as the built-in constructors; the system resolves such overloading by noting the rule set of the

premise containing the constructor. This allows the basic operations to be defined easily on phrases of the language as well as the built-in values.

12.3.2 Views

Each view must be given a name and a type at the head of the LSL specification. There are five types of view: SOURCE, ENV, MEMORY, STACK, and TOPSTACK. Source views display the entire evaluation tree. Memory and environment views display the current memory and environment respectively. Stack views display those nodes of the evaluation tree that have been pushed onto the stack. Topstack views display the top entry of the stack.

Each source view and each topstack view specifies:

1. How instances of constructors are displayed.
2. How instances of semantic rules are displayed.
3. The default display attributes of each rule set.

A stack view specifies the first two of these; environment and memory views just specify the first.

The display of instances of constructors and semantic rules is specified by constructor display rules and semantic display rules, similar to those presented in Chapter 9. However, LSL is less powerful than that language. The operators presented in Chapter 9 to show the feasibility of specifying certain views are supported in LSL, except for the `LINE ... TO ...`, `LET name = ...` and `LEFT` operators. LSL doesn't allow views to be parameterised. Semantic display rules may not be combined with `WHERE` clauses, and there are no default display rules. Also, the special variables *subject* and *result* are not implemented; they must be replaced with the appropriate constructor applications and highlighting

operations. Boxes are always aligned horizontally at the top and vertically at the left or centre, and views always grow downwards.

To simplify the specification of semantic views, the strings " $\vdash$ " and " $\Rightarrow$ " can be replaced by the symbols $\vdash$ and $\Rightarrow$ respectively.

A stack view will display all entries on the stack without the language definer explicitly specifying this. This usually means that a semantic display rule in a stack view shouldn't display subtrees of the current node.

Different views for the same constructor or semantic rule are introduced with the name of each view embedded in an arrow, for example:

```
call:
    -source->
        "call " $1 $s $2
```

The `-source->` arrow introduces the display of the constructor `call` for the view called `source`. Other views could follow this one by adding more arrows and equations. If two or more views display this constructor in the same way, they may share the same display expression, preceding it by a list of arrows, one for each view. A default display expression may be given before the first arrow; this is used for all views except those explicitly defined to be different.

The initial display is determined by picking a rule that matches the program and displaying the tree that consists of a single instance of that rule, with the `SELECTED-START` flag set. If the program is also a value, then the display is that specified by a special rule called the **value display rule**. For example:

VALUE:

subject

Each rule set may be given a display attribute for each topstack and source view. This is specified at the start of the rule set. If a display attribute isn't given, all rule sets except the root are assumed to be static. The root rule set is always dynamic, whatever display attribute is specified. The user may toggle static and dynamic attributes assigned to rule sets while the animator is in use. The TRANSPARENT attribute is not supported.

Each constructor may also be given a display attribute for each view. This is specified in the display rules for that constructor. If an attribute is omitted, the constructor is assumed to be dynamic in that view.

12.3.3 Summary

LSL is a straightforward translation of into ASCII of the notation used in this thesis. Abstract syntaxes and semantic rules are defined as in the theory, with the addition of special support for stack views. Parse rules are defined using a notation based on that of YACC. Textual views can be specified using a subset of the notation described in Chapter 9. Graphical views are not supported. Display attributes for each source view can be assigned to constructors and rule sets, to control the behaviour of the step functions in that view.

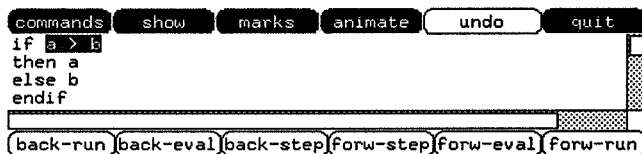
12.4 Using an Animator

The Animator Generator and the animators it produces run under the UNIX operating system and the X window system. It requires a mouse or similar pointing device and three mouse buttons (or equivalents).

12.4.1 Getting Started

Like most UNIX programs, animators are started from a shell. Since they don't have an interface to an editor, the program to be evaluated must either be in a file or entered from the shell window. The same applies to any initial environment or memory provided by the user.

Once an animator has parsed the program, it creates an X window displaying a view of the program. This window may be moved, resized, and so forth, using the X window manager. The window contains some buttons which are used to control the animator, and looks like this:



The white buttons are **command buttons**; clicking with the mouse on a white button executes the named command. The black buttons are **menu buttons**; pressing a mouse button while the cursor is on a black button pops up a menu of commands or choices. These menus stay on the screen while the mouse button is held down and select the entry indicated by the cursor when the button is released. Nothing happens if the button is released when the mouse isn't over the menu.

The central window displays the first source or topstack view names in the header of the LSL specification. The “*show*” menu can be used to create windows that display the first stack, env or memory view. These windows are like the ones above, but without the buttons. For example:



Expert users may change the view in each window to another view of the appropriate type.

12.4.2 Command Buttons

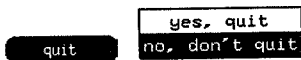
The white buttons perform the following commands:

- forw-step** The definition of step for the current view.
- forw-eval** Phrase evaluation.
- forw-run** Evaluation to the next mark, or the end of the animation.
- back-step** The reverse step for this view.
- back-eval** Reverse phrase evaluation.
- back-run** Reverse evaluation to the nearest mark, or the start of the animation.
- undo** Returns the animator to the state before the previous command. Further undo commands undo the effect of the commands before that. Once another command is executed, undo commands will undo the effect of the new command, then of the previous undo commands, and so on.

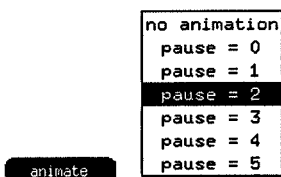
Any **forw-eval**, **forw-run**, **back-eval** or **back-run** command may be interrupted by clicking with the mouse over the main window.

12.4.3 Menu Buttons

The black buttons pop up the following menus:



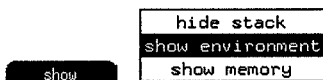
The quit menu contains two selections. "yes, quit" exits the animator immediately. "no, don't quit" does nothing.



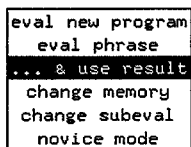
The **animate** menu controls whether phrase evaluations and evaluations to the next mark are automatically animated, and the number of seconds to pause after each step if animation is selected. The longer the pause setting, the slower the animation.



The **marks** menu contains a single entry. This is “**set mark**” if there is no mark set at the current position, and “**clear mark**” if there is a mark set at the current position. Selecting the entry sets or clears a mark at the current position. Marks are also set by the “**change _____**” commands (see the **commands** menu below).



The **show** menu can be used to display or hide the windows that show the current environment, memory and stack. Each entry in this menu corresponds to one of these windows. This entry is “**show _____**” if the window is currently hidden and “**hide _____**” if it is currently displayed.



commands

The **commands** menu contains entries for several commands that are used less often than those linked to the white buttons.

The “**eval new program**” entry reads a new phrase, initial environment and initial memory from the keyboard and evaluates those, discarding the current animation.

The “**eval phrase**” entry reads a new phrase from the keyboard and evaluates it in a new window with the current environment and memory. On completion of the new animation, the animator continues with the current animation unchanged.

The “**...& use result**” entry evaluates a new phrase as above, but replaces the result value and memory of the current phrase of the current animation with the result of the new animation.

The “**change memory**” entry prompts the user to enter a new memory and uses it to replace the current memory. It also sets a mark at the current position.

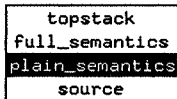
The “**change result**” entry prompts the user to enter a value and uses it to replace the result of the selected sequent. It also sets a mark at the current position. This entry is only available if the current position is the rhs of a sequent.

The “**change sub-eval**” entry prompts the user to enter a value and a memory and uses these to replace the rhs of the selected sequent. Then it marks that sequent as evaluated. It also sets a mark at the (new) current position. This entry is only available if the current position is the lhs of a sequent.

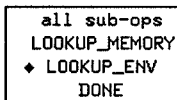
The “**expert mode**” entry makes the advanced features of the animator available, if they aren’t already. These advanced features are described below. This entry is only displayed in novice mode.

The “novice mode” entry makes the advanced features of the animator unavailable. This entry is only displayed in expert mode.

12.4.4 Advanced Features



If the advanced features are enabled and the right mouse button is pressed over the central area of a window, the animator pops up a menu of views. Selecting an entry from this menu changes the view used to display the animation in that window. There are separate menus for each window, listing the source and topstack views, environment views, stack views or memory views as appropriate.



If the advanced features are enabled and the middle mouse button is pressed over the central area of the main window, the animator pops up a menu of sub-operations. This menu is a list of all rule sets that may be selected in the current program view.

This menu behaves differently from the others; it stays on the screen until the “DONE” entry is clicked on. While it is on the screen the mouse buttons may be used to select or deselect entries. Clicking on an entry with the left mouse button selects that group (or all groups); clicking with the right mouse button unselects that group (or all groups bar the main one). Selected groups are indicated by a small diamond before the name.

A selected rule set is given the DYNAMIC attribute; each sub-animation in that rule set is stepped through and is displayed to the user. An unselected rule set is given the STATIC attribute; each sub-animation in that rule set is evaluated in one step and is not displayed to the user.

12.4.5 Summary

Animators produced by The Animator Generator are controlled by mouse-clicks. The white buttons performs the basic step operations, and the black buttons pop-up menus which offer a wider range of actions and some ways to configure the animator. The animated source is displayed in the main window, and the current environment, memory and call stack can be displayed in secondary windows. In advanced mode users can select different views and can select which rule sets are displayed, by clicking the appropriate mouse button over the relevant window.

Animators provide most of the operations described in Chapter 10, including view-specific steps, marks, interrupts, changing the current memory and result, phrase evaluation and reverse evaluation. They don't support break points, break conditions or resuming the last command after an interrupt.

12.5 Implementation

The Animator Generator is written in about 12,000 lines of C++, using the YACC parser generator and the InterViews window library. It runs under UNIX and the X window system. It has two parts; the generator proper and the core animation functions.

12.5.1 The Generator

The generator translates the LSL grammar rules to YACC and the other LSL rules to C++ tables. It consists of a parser, a set of types for internal tree structures, and a set of routines for writing out these trees as tables.

The parser translates LSL grammar rules to YACC directly. It translates the `%token` and `%keyword` constructs of LSL to their YACC equivalents and stores

them in one file, and translates the rules themselves to their YACC equivalents and stores them in a second file. These files are then combined with some standard code to give a complete YACC specification.

For example, the following parse rule from the grammar of Proc:

```
phrase:      WHILE phrase DO phrase ENDWHILE :
              while ($2, $4)
```

is translated to the following YACC rule:

```
phrase:      WHILE phrase DO phrase ENDWHILE
              { $$ = yyconst ("while", $2, $4, NULL); }
```

An animator's parser needs to know whether it is reading a memory, environment, phrase or value. So the generator adds the rule:

```
accept:      P phrase | E env | M memory | V value ;
```

to the YACC file and defines the tokens P, E, M, and V. Then whenever an animator calls a parser, it pushes the appropriate token onto the input stream, so that the parser reads that token first and so reads the desired type of object.

If the language designer is using the lexical analyser provided with The Animator Generator, then the parser adds a line to recognise a keyword to that analyser for each %keyword construct in the LSL specification.

The parser translates the other LSL constructs into trees formed from C++ classes and pointers. As it translates it checks that the specification is legal in certain respects, such as requiring the rhs of a recursive rule to be the same as the rhs of its last premise.

These pointers can't be stored in files, so the generated tables consist of arrays of structures with pointers replaced with integer offsets. The first thing an animator does is to convert these back into trees. These trees are described in the following sections.

Once the tables have been generated, a UNIX makefile calls YACC to translate the grammar rules to C. Then it compiles the output of YACC and the C++ tables with the core animation functions to produce an animator. (C and C++ produce compatible object code.)

12.5.2 Operations

A relational animation is stored as a tree of instances of semantic rules, with pointers to the current sequent and node. Each node of the tree contains pointers to the rule that is instantiated at that node, to the env, lhs and rhs of the instance, to a list of subtrees and to a list of copies of the variables in the rule. Each node also contains an index into the list of subtrees that indicates which subtree is being evaluated (if any), and a status flag that can have one of the following values:

SELECTED.START The node is the selected node, and the evaluation of the node has not begun (see below).

SELECTED.PREMISE The selected sequent is a remote premise of this node.

NOT-SELECTED This node has not been fully evaluated, but it is not the selected node. The selected node is in the subtree of this node indicated by the index into the list of subtrees.

SELECTED.LEFT The selected sequent is the conclusion of this node but the conclusion has not been evaluated.

SELECTED_RIGHT The selected sequent is the conclusion of this node and the conclusion has been evaluated.

DONE This node is fully evaluated and is no longer selected.

Remote premises also have a status flag, which is set to **SELECTED_LEFT** if the premise has not been evaluated and **SELECTED_RIGHT** or **DONE** if it has been evaluated.

It can be seen from the above that the definition of validity used in the animator differs slightly from that in the definition of a relational animation. One change is that the boolean of a relational animation is no longer required, as the information is incorporated into the status flag of the current node or premise. The other change is that the animation no longer stops at local premises, but creates the subtree corresponding to them and stops there, with the status flag set to **SELECTED_START**.

The status flags are used because this information is needed by the display algorithm. Stopping at the subtree rather than the local premise made some parts of the animator easier to write.

The basic step function $\mathcal{A}_R$ (see Section 8.2) is implemented as a loop, each step of which is defined by cases of the current sequent. There are more cases to this loop than the valid cases listed above. The loop finishes when the animation is valid and the rule instantiated at the current node is in a dynamic rule set and doesn't describe a hidden constructor. The test for validity checks for the special case of selecting the conclusion of the node that the step started from, and the special case of static constructors.

If the current sequent is a non-trivial local premise, the loop creates a child node corresponding to it, sets the current sequent to the conclusion of the child and the status flag of the child to

SELECTED.START. It also sets the status flag of the parent to NOT_SELECTED and sets its index appropriately.

If the current sequent is a conclusion and that node's status flag is set to SELECTED.START, the loop selects the first premise of the child if there is one, or the conclusion otherwise. If the conclusion is selected, the status flag is set to SELECTED.LEFT.

If the current sequent is a trivial local premise, the loop unifies the rhs with the lhs, if necessary updating the choice of rule instantiated at this node, and sets the current sequent to the next premise of this node if there is one or to the conclusion otherwise. If the conclusion is selected, the status flag is set to SELECTED.LEFT.

If the current sequent is a remote premise and its subject is a value in the remote rule set, then it is treated as a trivial local premise.

If the current sequent is a remote premise, its subject isn't a value in the remote rule set and that rule set is selected, then it is treated as a non-trivial local premise. If the rule set is built-in, it is evaluated by dedicated code.

If the current sequent is a conclusion and the status flag is set to SELECTED.LEFT, the rhs of the conclusion is unified with the rhs of the corresponding premise in the parent node, the choice of the rule at the parent node is updated if necessary and the status flag is set to SELECTED.RIGHT.

If the current sequent is a conclusion and the status flag is set to SELECTED.RIGHT, the current sequent is set to the next premise of the parent if it exists or the conclusion of the parent otherwise. The index and status flags of the parent are set appropriately.

If the selected sequent is a premise, the status flag of the current node is set to SELECTED.PREMISE. This is done here instead of in the main loop so that it

can be used in the test for validity to see if the current node has a valid premise before the current sequent.

The above algorithm refers to updating the choice of rule at a node. This is because a constructor may be described by more than one rule. These rules are stored in a list attached to the constructor. When a new node is created, the first rule on that list is instantiated. After each premise is evaluated, the result of the sub-animation must be unified with the rhs of the premise. If this unification fails, the other rules describing that constructor are tried until one is found that fits. Normally, the determinacy restriction ensures that once a rule on the list has failed to unify at a node it will never unify at that node again. However, if a user-defined result is added at some point during the evaluation, it may be necessary to search all the rules that describe the constructor.

Updating the rule entails changing the pointer to the rule and updating the copies of variables, making sure that instantiated variables aren't affected.

A reverse step is implemented by making the forward step function store a list of valid sequents, and simply retreating to the most recent of these. The indices of each node between the current node and the stored node must be reset, and all variables instantiated by the previous forward step must be uninstantiated.

View-specific steps are the smallest step operations available to the user. As shown in Chapter 10, they are composed of one or more basic steps. Basic steps are applied repeatedly until the current sequent describes a selected constructor and its display attribute is satisfied. The display attributes of constructors are stored centrally and are easily updated and checked. If the current sequent before the step began was a local premise and the current sequent after the step is the conclusion of the corresponding subtree, the step function is called once more.

Phrase evaluation is defined by repeated applications of the view-specific step. Static and hidden remote rule sets are treated the same way, as are side-conditions.

Running to the next mark is defined as repeatedly applying the view-specific step functions until the current sequent is one with a mark set.

Automatic animation consists of repeated applications of the view-specific step with the display being updated and the animation pausing after each application.

Interrupts are implemented by checking after each view-specific step to see if a mouse button has been clicked. over the main window. If it has, the current operation is abandoned.

The undo command is implemented by keeping a list of commands. Each entry on the list consists of a flag indicating the direction of evaluation performed by the command, and the number of view-specific steps that it made. The undo command simply performs the same number of view-specific steps in reverse. This is fragile; it breaks whenever the current view is changed to one with different display attributes, or when the display attributes are changed directly. It also doesn't cope with the change commands.

The change commands replace the appropriate objects with ones supplied by the user. At the time of writing, this is implemented by unifying the new objects with copies of the corresponding terms in the rule instantiated at the current node. This is overly restrictive.

12.5.3 Views

Each constructor and each node in the animation tree has an associated display rule. This is stored as a list of parse trees in which each parse tree corresponds to a particular view. The current view for each window is stored as an integer, so finding the required parse tree from a list is a simple indexing operation.

The display algorithm interprets the appropriate display rule. If this rule refers to subtrees or abstract syntax terms, the algorithm incorporates interpretations of the display rules for the appropriate objects.

The result returned by the display algorithm is a parse tree of the display language. This describes a box, which describes a list of rows, each of which is a list of boxes or strings. Each row corresponds to a row of text or a horizontal line, and each box corresponds to an area with its own margins, so that a new row in a box starts at the left margin of that box rather than the left margin of the screen.

Strings in a display rule are translated directly to strings in the display language. A list of items is translated to a row, and a box to a box. Newlines start a new row in the nearest enclosing box.

The OVER and ABOVE constructs create a box in which the left argument is positioned above the right argument, and separate their arguments with a horizontal line and a blank row respectively.

The highlight operators test the status flag at a node to see if it is the current node, and produce special markers to switch reverse video on and off. The conditional operators test the value of the index and status flags at each node.

The query operators also test the status flags at a node, to determine which sub-expression to interpret.

Once this parse tree has been generated by the display algorithm, another algorithm converts it to a screen display. This algorithm is straightforward.

The animator keeps a list of nodes that have been pushed onto the call stack. Stack views display all of these, one above the other.

12.5.4 Summary

The implementation of The Animator Generator is fairly straightforward. It is an interpreter that operates on structures that correspond to those defined in the theory of animation. Most of the commands are defined as repeated applications of the basic step operation, which is described in detail above. Views are constructed by traversing the internal representation of the display rule for each constructor or semantic rule, following pointers to the internal form of the evaluation when required.

12.6 Shortcomings

The Animator Generator demonstrates the feasibility of generating an animator from a formal specification, but it isn't intended to be production-quality software. The animators that it produces don't support all the notation for specifying views described in Chapter 9. Nor do they support all the operations defined in Chapter 10. They are also rather slow, and have a poor user interface.

12.6.1 Operations

Although the generated animators provide most of the required operations, they don't provide a way to set break points or to specify break conditions. There is no way to resume an interrupted command; it is always aborted. The speed of an automatic animation must be set in advance and cannot be altered while the animator is running.

The undo facility is limited to white button commands; it can't undo changes to the current memory or result. There is no way to change the current environment. The ability to evaluate a phrase in the current memory and environment and use the result in the main animation is limited by users having

to decide whether to use the result before beginning the sub-animation. It would be easier if they could choose when the sub-animation is complete and they know the result.

The built-in values and operations are limited to integers, identifiers and booleans. Most languages will need more than this, such as floating point numbers, characters or bitsets.

12.6.2 Views

LSL can only specify a limited range of views. It can't specify static views or program-specific views. It doesn't support graphics, which rules out flow-charts and graphs and restricts the ways that trees can be displayed. There is no interface with an environment that could be used to produce slices, outline views or type-specific views. Memory and environment views must display the entire memory or environment.

Many of the display operators described in Chapter 9 aren't available in LSL. LSL doesn't support default semantic display rules, and doesn't let display rules be combined with WHERE clauses.

There is little control over layout. Although text can be put into boxes, these boxes are always horizontally aligned at the top and vertically aligned at the left or centre. Also, the display always grows from the top downwards.

Although the keyword system allows the sample lexical analyser to be customised easily, any major changes require the language definer to write a lexical analyser in C or C++. LSL should be extended to provide automatic generation of lexical analysers from regular expressions [Les75,HK86,Heu86,HL87].

12.6.3 The User Interface

The user interface is visually unappealing and suffers from many minor problems. The most annoying defect is that the window doesn't always show the current phrase; the focus of attention can move off the screen with no indication of the direction taken.

Another obvious limitation is that trees made with the OVER and ABOVE display operators aren't centred, and the line printed by the OVER operator has a fixed length.

Animators don't have an interface to an editor. This limits the usefulness of the "change ____" and "eval ____" commands, since users must supply a complete program or memory instead of editing the current one. A closely-coupled editor could be used to set marks or break points, whereas the existing provision is primitive.

The interface also needs a better way of displaying error messages, and a way to recover from errors. At present all errors terminate the animator.

There should be a way to limit the depth to which terms are displayed in each view. Semantic views in particular tend to be unmanageable with even small programs. More sophisticated algorithms than a simple limit to the depth of display would provide even more flexibility.

12.6.4 Summary

The Animator Generator is intended to demonstrate the viability of the ideas in this thesis. It does this, but provides only the minimum facilities required. It lacks some desirable features and can only provide a limited range of views. In addition, the user interface and the specification of lexical analysis could be improved.

12.7 Chapter Summary

This chapter described the implementation of The Animator Generator and LSL. It included a brief introduction to using a generated animator. It also discussed some of the shortcomings of the generated animator.

This prototype demonstrates the viability of the ideas presented in this thesis. It is a useful tool for language designers and for teaching relational semantics. The next chapter includes a discussion of ways in which the implementation could be improved.

Chapter 13

Future Work

This thesis has presented a new area of research: the semantic specification of animation. It has explored some of the basic issues, and has defined a framework for discussing views and animation operations in terms of semantics. This chapter discusses some ways in which this work could be developed further.

The chapter begins by discussing how the practical development of the theory should be based on experience with specifying other languages. Then it considers some extensions to the definition of relational semantics, extensions to the notation used for defining views, and some extra operations that require extensions to the theory. Finally it considers ways of improving the implementation of The Animator Generator.

13.1 Other Languages

To perform its function an animator generator should be able to produce animators for a wide variety of languages. The immediate task is to test The Animator Generator on a range of language specifications to determine which language features the current theory can and can't support.

Most of the examples in this thesis use the toy language Proc which includes imperative and functional features. A good test of The Animator Generator would be to generate animators for languages with different methods of evaluation. Chapter 11 gives an overview of a specification of pure Prolog, and discussed how we could specify a view similar to that of CODA [Plu88]. This is only one possible view of Prolog; there are several other animators that provide different views [Plu88,EB86,PB87,DC86].

Many of these animators display the Prolog search tree [EB86,DC86]. This is because Prolog evaluations switch context frequently, and a view of the search tree gives the user some idea of the context of the current goal. Most of these views use graphics, and would be a useful guide for which graphical features should be added to the notation for specifying views.

This range of views also offers scope for comparing methods of specifying languages. It should be possible to define two relational semantics of Prolog in terms of two different views, so that the objects in each version of the semantics correspond to the objects in one of the views. For example, one semantics could have an AND/OR tree as its phrases [EB86,DC86], while another could use a procedure based model [Byr80,Plu88]. It would be interesting to see if the view specifications of LSL would allow each view to be specified in terms of the semantics based on the other view.

Another possibility is a lazy language, that is, one in which the arguments to a function are only evaluated when they are first used. Many people find it difficult to follow the evaluation of programs in lazy languages, because when an argument to a function is first used, the evaluation of that function is suspended while the argument is evaluated. If the argument itself is a function application, its evaluation may be suspended in turn while its arguments are evaluated. Thus the focus of attention effectively switches between several co-routines and users lose track of where they are.

It could be argued that animation is not appropriate for a lazy functional language, since it is a mathematically pure language. However, animation is likely to be useful for teaching and for finding cases of infinite recursion, and will probably have other uses as well.

Another feature that LSL might find hard to handle is raising and catching exceptions. It could specify a literal interpretation of the exception mechanism of Standard ML, in which exceptions are propagated up the syntax tree from the point where they are raised. However, an efficient implementation would jump from that point to the corresponding exception handler, and that is harder to specify in LSL. A similar problem exists with *gotos*, and is compounded by the unstructured nature of *gotos*.

13.2 Semantics

The theory of animation includes a formal definition of relational semantics. This definition is not fixed in stone, and some changes might make it more flexible. This section discusses some of these. It also discusses the possibility of defining animation in terms of other formalisms.

13.2.1 Removing Recursive Premises

The need to label rules as either recursive or non-recursive is inelegant. Another way of achieving the same effect would be to replace each recursive premise with a *SUCH-THAT* clause. For example, the relational rule for addition could be rewritten:

$$\text{SUCH-THAT} \frac{e \vdash p_1, m_1 \Rightarrow v_1, m_1 \quad e \vdash p_2, m_1 \Rightarrow v_2, m_2}{e \vdash p_1 + p_2, m_1 \Rightarrow v, m_3}$$

INTEGER

$$\vdash v_1 + v_2, m_2 \Rightarrow v, m_3$$

This is similar to the way that a final computation is often written as a side-condition in the literature. This is most often seen in transitional semantics, but extends easily to the relational style. When read as an inference rule the SUCH-THAT clause should be treated as a premise.

Operationally, the SUCH-THAT clause should be read as a final computation that corresponds to the conclusion. This gives credence to the apparently arbitrary heuristic adopted in Chapter 8, which assumed that the subject of a remote recursive premise is displayed the same way as the subject of the rule.

13.2.2 Adding Predicate Premises

Relational rules could be extended by allowing predicates as well as premises above the line. In the inferential reading, these would be treated the same as other premises. However, in the operational reading they would be treated as conditions that must be true at the corresponding point in the evaluation. Testing a condition would not count as an animation step.

This addition would make more transitional specifications and relational specifications equivalent. This is because the side condition of a transitional rule could correspond to a predicate of a relational rule. For example, a transitional rule for division that required the second argument to be non-zero could now form part of a rule set that was equivalent to a relational rule for division.

13.2.3 Non-determinism

Non-determinism was omitted from this thesis to simplify the development of the theory. However, it should be straightforward to add. It can be introduced by having more than one rule describing the same phrase. For example, the following two rules define an explicit non-deterministic operator:

$$\frac{}{e \vdash p_1 \oplus p_2, m \Rightarrow p_1, m}$$

$$\frac{}{e \vdash p_1 \oplus p_2, m \Rightarrow p_2, m}$$

Another form of non-determinism is implicit non-determinism of the order of evaluation of premises. Many languages don't define the order of evaluation of arguments to numeric expressions, on the grounds that leaving the order of evaluation undefined allows compilers more freedom in optimising such expressions. The following two rules show how this could be specified:

$$\frac{e \vdash p_1, m_0 \Rightarrow v_1, m_1 \quad \overset{\text{INTEGER}}{e \vdash p_2, m_1 \Rightarrow v_2, m_2 \vdash v_1 + v_2, m_2 \Rightarrow v, m_3}}{e \vdash p_1 + p_2, m_0 \Rightarrow v, m_3}$$

$$\frac{e \vdash p_2, m_0 \Rightarrow v_2, m_1 \quad \overset{\text{INTEGER}}{e \vdash p_1, m_1 \Rightarrow v_1, m_2 \vdash v_1 + v_2, m_2 \Rightarrow v, m_3}}{e \vdash p_1 + p_2, m_0 \Rightarrow v, m_3}$$

Dijkstra's overlapping guards [Dij75] are more complicated. One way of representing them is to first select one branch of the construct and evaluate its guard. Then if the guard evaluates to `tt`, evaluate the rest of that branch. If the guard evaluates to `ff`, that branch should return an exception value. The exception must be caught by the construct, which will then select another branch.

This would require a notation for selecting one sub-phrase from an n -ary construct. In this thesis such constructs have been built from infix operators, but it would be clearer to use set notation. If this were done, then a guarded command construct could be defined by the following rules:

$$\frac{e \vdash p_i, m_0 \Rightarrow v, m_1 \quad \overset{\text{EQ}}{\vdash \text{eq}(v, \text{fail}), m_1 \Rightarrow \text{ff}, m_2}}{e \vdash p_1 \oplus \dots \oplus p_n, m_0 \Rightarrow v, m_2} \quad i \in 1..n$$

$$\frac{e \vdash p_i, m_0 \Rightarrow v, m_1 \quad \overset{\text{EQ}}{\vdash \text{eq}(v, \text{fail}), m_1 \Rightarrow \text{tt}, m_2} \quad e \vdash p_1 \oplus \dots \oplus p_n \setminus p_i, m_2 \Rightarrow v, m_3}{e \vdash p_1 \oplus \dots \oplus p_n, m_0 \Rightarrow v, m_3} \quad i \in 1..n$$

The definition and proofs of equivalence presented in Chapter 6 should extend to this simple non-determinism straightforwardly. The proposition that “steps

preserve equivalence” is stated in the form of a bisimulation, which is a standard method for showing the equivalence of non-deterministic systems [Par81,Mil89].

13.2.4 Using Variables to Represent Identifiers

The theory could be extended by taking an idea from the Edinburgh Logical Framework [HHP87]. This is a formalism for defining inference systems. The feature in question is the ability to represent variables in the system being defined by variables of the framework itself. In the theory of animation, this would mean that identifiers in an object language would be represented by variables in the specification language. Environments and memories would be bindings of variables to values. The advantage of this approach is that variable lookup and substitution would be available for free in any specification.

13.2.5 Unification

In the present theory, all variables in the lhs of a sequent must be instantiated before that sequent is evaluated, and all variables in the rhs of a conclusion must be instantiated by the evaluation of the node. This has the effect that the implementation can use one-way pattern matching instead of full unification. A possible extension would be to allow full unification.

One rule that would be admitted by this change is rule 13 of the static semantics of Standard ML, which was discussed in Section 6.1.5. In our notation this rule is as shown below. The result of this rule is a variable v that is instantiated later in the evaluation:

$$\frac{e \vdash p_1, m \Rightarrow \text{exn}, m}{e \vdash \text{raise } p_1, m \Rightarrow v, m}$$

The addition of rules such as this may make it easier to specify languages that make implicit use of unification, such as the type inference of Standard ML

or logic programming languages such as Prolog. They may also make it easier to specify lazy evaluation. These languages will be even easier to specify if this change is combined with the idea in the previous paragraph, so that a specification of Prolog (for example) could use a built-in definition of unification.

13.2.6 Other Formalisms

As mentioned in Chapter 8, there is scope for developing theories of animation in terms of other semantic formalisms, such as denotational semantics or definitional interpreters. It would be interesting to compare the ease of defining views and operations in different formalisms.

It would be particularly interesting to compare the definition of an animation step in the current theory with the rewrite rules of a definitional interpreter. Structured operational semantics was originally developed partly to avoid defining steps that didn't perform any computation. So comparing those animation steps that aren't also computation steps with those rewrite rules that aren't computation steps might give us some insight into the relation between transitional semantics, relational semantics and definitional interpreters.

Appendix A defines an animation step in terms of transitional semantics, and shows it to be equivalent to the definition for relational semantics. The transitional definition of an animation step is much more complicated than the transitional definition of a computation step. Transitional semantics doesn't seem to be as flexible as relational semantics, which suggests that it isn't as good a framework for comparing animation operations.

13.2.7 Summary

The definition of relational semantics used in the theory is flexible enough to describe a wide range of languages. However, it could be extended; both in

minor ways (such as removing recursive premises or adding predicates as premises) and major ways (such as allowing non-determinism or representing identifiers by variables of the specification language). It would also be interesting to examine definitions of animation in other semantic formalisms.

13.3 Views

This section discusses several ways of extending the notation used for specifying views to make it more powerful or more concise.

13.3.1 Graphics

The most obvious improvement is to improve the layout language to provide facilities similar to those of PIC [Ker81]. Of these, it should at least support more ways to align boxes, and ways to create the displays upwards from the bottom of the screen instead of downwards from the top. The graphical views displayed by several existing Prolog animators are a guide to the features that would be useful.

13.3.2 Pattern Matching

The notation could also be extended by allowing constructor display rules to apply only when the instance of the constructor has certain arguments. For example, if a simple environment is to be displayed as follows:

$$(a,1) + (b,2) + \dots$$

then there should be a special rule for the case where the first argument of `add_env` is `null_env`, for example:

```

add_env(null_env, -, -) :
    "(" $2, $3 ")"

add_env(-, -, -) :
    $1 " + (" $2, $3 ")"

```

13.3.3 New Display Attributes

One way to extend the range of views supported by the theory is to add more display attributes. For example, a language designer might want each occurrence of a construct to be evaluated completely in one step. This could be specified with a new display attribute, perhaps called EVAL-ALL.

Another, rather contorted, example of a display attribute is demonstrated by the following construct, which combines a procedure call with an if statement, so that the procedure p_2 is only called if the condition p_1 is true:

$$\begin{array}{c}
 \frac{e \vdash p_1, m_0 \Rightarrow \text{ff}, m_1}{e \vdash \text{condcall } p_1 \ x \ p_2, m_0 \Rightarrow \text{nil}, m_1} \\
 \\
 \begin{array}{c}
 e \vdash p_1, m_0 \Rightarrow \text{tt}, m_1 \\
 e \vdash p_2, m_1 \Rightarrow v, m_2 \\
 \text{LOOKUP} \\
 \vdash \text{lookup}(e, x), m_2 \Rightarrow \text{closure}(e', x, x', p'), m_3 \\
 e' \vdash (x', v) \vdash (x, \text{closure}(e', x, x', p')) \vdash p', m_3 \Rightarrow v', m_4 \\
 \hline
 e \vdash \text{condcall } p_1 \ x \ p_2, m_0 \Rightarrow v', m_4
 \end{array}
 \end{array}$$

A dynamic call graph shouldn't extend the display of a node containing an instance of this rule until the current sequent is the fourth premise of the second rule, because only then is the procedure actually called. The animation shouldn't stop at a premise that describes `condcall` in a dynamic call graph, because if the condition evaluates to `ff` then the call graph won't be extended. Thus a display attribute for this construct would have to specify the premise that the animator should stop at. This behaviour can't be specified with the existing attributes.

Another way to extend the range of views with display attributes is to allow the attribute of a constructor to depend on the context of that constructor. For example, if an assignment operator returns a value (as in the C language), a language definer may wish the operator to be displayed as a dynamic construct when the value is required, but as a static construct when it is acting as a statement.

This could be done by giving an INHERITED attribute to the assignment constructor. The sequencing operator could then force sub-phrases with the INHERITED attribute to be displayed as if they were STATIC, and those operators expecting values from their arguments could force sub-phrases with the INHERITED attribute to be DYNAMIC.

A possible extension to this scheme would be to deduce the status of INHERITED attributes from the semantic rules, by assuming that sub-phrases expected to return nil are normally STATIC, and that other sub-phrases are normally DYNAMIC.

Another way to extend the range of views with display attributes is to allow display attributes to be attached to specific premises instead of specific constructors. For example, a designer might decide that looking up a procedure identifier in a procedure call is boring, but that looking up identifiers in expressions is worthwhile. In this case the lookup premise in the rule for procedure calls could be hidden without affecting lookup premises elsewhere.

The display attributes listed above are only some suggestions out of many possibilities. More views and languages need to be specified to determine the range of display attributes needed.

13.3.4 Abbreviations

As experience is gained with LSL and The Animator Generator, it is likely that certain patterns of rules will occur frequently. The notation should be extended with shorthand for such patterns. The abbreviation of several semantic display rules with WHERE clauses that was described in Chapter 9 is the result of this sort of experience.

13.3.5 Language Independence

This section suggests two approaches to making specifications of views less language dependent. The first is to deduce the existence of a procedure call stack directly from the semantic rules. The second is to define semantic display rules independently of the language being displayed. These are both highly conjectural ideas.

The Animator Generator already includes a feature for simplifying the specification of views of the procedure call stack. It may be possible to take this further by detecting procedure calls automatically. For example, in the specification of Proc the third premise to the semantic rule for `call` is the only local premise in the entire specification with a subject that isn't a sub-phrase of the subject of the rule:

$$\frac{
 \begin{array}{c}
 e \vdash p, m_0 \Rightarrow v, m_1 \quad \text{LOOKUP} \vdash \text{lookup}(e, x), m_1 \Rightarrow \text{closure}(e', x', p'), m_2 \\
 e' \dagger (x', v) \dagger (x, \text{closure}(e', x', p')) \vdash p', m_2 \Rightarrow v', m_3
 \end{array}
 }{
 e \vdash \text{call } x \ p, m_0 \Rightarrow v', m_2
 }$$

If this condition, or a similar one, detects the places where procedures are called in the specifications of most other languages, it could be taken as a default rule for selecting the premises that add entries to the call stack.

It may also be possible to make the specification of dynamic views largely independent of the language being described. For example, the semantic view of Proc could largely be specified by two default display rules, one for sequents and another for semantic rules. The only rules that wouldn't be covered by these defaults are the rules for *if*.

Until the condition is evaluated, the rules for *if* produce the same display, with the subject of the second premise replaced with question marks. The idea of this display is that it doesn't matter which rule is chosen to begin the evaluation of an *if* construct, and the subject of the second premise is revealed when it is known (after the condition is evaluated and only one rule can be used at the current node). If a notation can be developed for specifying this case, then this view could be specified independently of the language.

It would be necessary to experiment with a range of languages to see how many views can be defined independently of a language, and how many languages they could be used with. It may be necessary to have other components in a language-independent specification, such as one default rule for recursive rules and another for non-recursive rules.

13.3.6 Program-Specific Views

Chapter 9 mentioned several memory views that were specific to the program being animated. These included displaying a representation of a graph as a graph on the screen, and displaying an array as a sequence of elements. These views show the entities of the problem domain modelled by the program in terms of the problem domain instead of the terms of the programming language. Chapter 2 gave evidence that such views helped people understand someone else's programs and to learn specific programming techniques.

Chapter 9 also mentioned some animators that allowed users to define views for a specific program. Garden [Rei87] and Graphtrace [GKS83] let users define

views for specific types in the program. PV [BCH⁺85] lets users attach predefined graphic objects, such as dials and sliders, to specific values in the program.

It would be interesting to see if these approaches could be made part of a language specification. For example, it might be possible to interface a graph drawing package to a specification of a language by declaring that in that language, memory locations may be the nodes of graphs, and that values of pointer types may be the edges of graphs. This would produce a tool similar to Graphtrace.

13.3.7 Summary

The notation used to specify views could be extended in several ways as we gain experience with new languages and views. The most important extension is probably the inclusion of graphical features. Other extensions include finer control of the choice of display rule and the addition of new display attributes. The most challenging extensions would be the language independent specification of views and the provision of program-specific views.

13.4 Operations

This section discusses several ways that the theory could be extended to describe new operations.

13.4.1 Defining an Animation Step

This section suggests some possible alternatives to the current definition of an animation step, and then discusses the problem of judging between competing definitions.

At present, the definition of a step is unaltered when it enters (or leaves) a sub-animation that pushes (or pops) its parent onto (or off) the stack. In some views it may be desirable to make this push or pop a step, since it could result in a change to the display. This could be done with another display attribute. Alternatively it could be made automatic for topstack views.

The definition of an animation step assumes that a call to a remote rule set is an operation in the underlying model of evaluation. However, the subject of a remote premise could be a value in the remote rule set. In this case it might be better to ignore the call, treating it in the same way as a trivial local premise.

Another extension would be to relax the restriction that rule sets form a static hierarchy. This would enable one rule set to define declarations and another to define expressions even for a language like Standard ML in which expressions can contain declarations and vice versa. The proofs that use this restriction can be rewritten fairly easily. The problem is that this extension would require a change in the underlying model of evaluation, in which calls to remote rule sets represent operations of an abstract machine.

The current definition doesn't support semantic views well (nor was it designed to). For example, when a node corresponds to a local recursive premise, the last step from that node could move the focus to a node some distance away. This jump is difficult for the user to follow. We might be able to define a fundamental step in terms of a semantic view instead of the basic source view, provided that we could still parameterise it on display attributes to get a suitable behaviour for source views.

A basic question raised by this thesis is: "Is there a definition of an animation step that is independent of the language to be animated?". One approach to answering this question is to generate animators for several languages, as advocated in Section 13.1. This raises a problem, however: how do we know what makes a good definition of an animation step in each language?

There are several possible candidates even for a simple language like Proc. In addition to the alternatives suggested above, users might want a definition which allows them to select local premises with subjects that are values, instead of ignoring these values. They might want a definition that highlights a parent phrase between evaluating its sub-phrases. Or they might want a definition which moves directly to the next sub-phrase to be evaluated instead of gradually zooming in to it.

A good definition is one that makes sense both formally and informally. Discussion among users and language definers is one way of determining what makes a good definition. In the end the acceptability of any definition is established by this social process [MW85,MLP79]. However, such debates can often be based on little evidence.

Another approach is to conduct experiments to see which definitions are helpful to users. Such experiments could use various psychological techniques, such as hypothesis testing [Shn80,Bro80,Kep82,Mon84] or protocol analysis [NS72]. This approach can't completely replace the social process, but it might make the debate better informed.

One conjecture motivating this thesis, but not examined by it, is that a user's model of a language is most effective when it closely matches the semantic model of that language, and that using a semantics based animator will encourage the development of a "correct" model. Psychological techniques could be used to model users' understanding of programming and examine these conjectures. These models could even be used to develop new semantic formalisms which correspond to a user's "natural" models. This would be applying the conjecture in reverse.

13.4.2 Changing the Current Environment

The theory doesn't allow users to change the current environment. This is made difficult by the tree structure of animation histories. For example, assume that the following program fragment is being evaluated in an environment that assigns 4 to 'b' and that the current sequent describes $2 \times 'b'$:

$$1 \times 'b' + \boxed{2 \times 'b'} + 3 \times 'b'$$

If the user changes the value of 'b' to 5, one would expect that change to apply to the sequent describing $3 \times 'b'$ as well as the current sequent. But the tree structure of the above phrase has the following (abbreviated) form:

$$\begin{array}{r} 1 \times 'b' \qquad \boxed{2 \times 'b'} \\ \hline 1 \times 'b' + 2 \times 'b' \qquad 3 \times 'b' \\ \hline 1 \times 'b' + 2 \times 'b' + 3 \times 'b' \end{array}$$

where all phrases initially share the same environment. If this environment is changed in the current sequent, it wouldn't affect the one describing $3 \times 'b'$. This implies that the environment should be changed globally.

However, if the environment globally is changed globally, then the evaluation of $1 \times 'b'$ becomes inconsistent. So if the environment is changed globally, then the environment of each completed sub-animation should be protected from this change.

Furthermore, if the environment is changed globally and this program fragment is part of a local block which defines 'b', any global redefinition of 'b' might affect another identifier called 'b' declared in an outer scope. So in fact the environment should be changed where the current binding of 'b' was first introduced. This implies that the animator must have some knowledge of scope, and that changes to an environment must preserve scope.

The scope structure could be indicated by introducing a new constructor type SCOPE. Constructors of type SCOPE should be outermost in the env of

every sequent. The animator would then have to co-operate with an editor in allowing parts of the current environment to be changed; new parts would have to replace the old, leaving the overall structure of the environment unchanged.

The animator should store the old version of the environment at the point where it was changed. The reverse step function would then look for stored versions and restore them when necessary.

An alternative approach is to implement each env as part of the lhs and rhs of the sequent, in the same way as memories. Then it would be possible to change a right hand environment without affecting the left. This approach needn't change the way rules are written; it could be an internal transformation.

13.4.3 Input and Output

There are some obvious language features that aren't supported by LSL. One of these is I/O. Since I/O both depends on the hardware available and varies greatly from language to language, it is perhaps best to support it by allowing the language definer to write "built-in" rule sets which are linked with the animator object code. LSL would need to be extended to specify an interface to the implementation language. Most, if not all, semantic formalisms treat I/O poorly; until a good formalism is found this ad-hoc approach will have to suffice.

13.4.4 Bug Location Algorithms

The Animator Generator could be extended to support Shapiro's bug location algorithm [Sha82,Per86]. This is defined for any language with functions or equivalent constructs. Each sequent of an LSL rule can be regarded as a function for this algorithm, giving a much finer level of detail than that envisaged by Shapiro. Alternatively, either the language definer or the user could specify a subset of the constructors of the language to be considered by the algorithm.

13.4.5 Symbolic Evaluation

The Animator Generator could also be extended to support symbolic evaluation [CR83]. Symbolic evaluation replaces some part of the input or program by a variable. This variable can't be evaluated, so the evaluator evaluates those phrases that it can, leaving those involving the variable as abstract syntax terms. The user can then see what happens in the general case to the part of the input or program represented by the variable, instead of only seeing what happens for particular values.

When a phrase that includes a variable is tested in an *if* statement, an evaluator must either choose one branch of the *if* statement, or record both branches to form a tree which the user can traverse. The first option is best only used interactively, since the user can specify which branch to take. The second option can be used either interactively or in batch mode. It must be designed to avoid infinite loops! Implementing the second approach in The Animator Generator would require an extension to the definition of an animation history to support multiple branches, and to the operations so that users could examine the different evaluation paths.

13.4.6 Summary

The theory already describes most of the operations that are provided by existing animators. Possible extensions include a facility for changing the current environment, support for I/O, adapting Shapiro's bug location algorithm, and allowing symbolic evaluation. In addition, some details of the fundamental definition of an animation step could be examined more closely.

13.5 Implementation

The Animator Generator is a practical tool for designing small languages and for providing an animator quickly. To be useful in other situations, it would have to produce animators that were fast enough and powerful enough to compete with hand-crafted animators. This requires a better implementation than the one developed for this thesis.

The current implementation could be improved by extending the range of features it supports, by making it more efficient, and by incorporating it into a full programming environment. The shortcomings listed in Section 12.6 also need to be eradicated.

13.5.1 Including Static Semantics

An LSL specification defines the syntax and dynamic semantics of a language. However, most languages have constraints on syntactically legal programs, such as identifiers having to be declared before use, and many have both a static semantics that specifies type-checking and a dynamic semantics that specifies evaluation. LSL could be extended to support these.

Syntax constraints are usually defined using attribute grammars. The grammar part of LSL could easily be extended to support attributes. An alternative approach is that used in HPL84 [KNPS88]; a general symbol table is used as a model for a set of commands to describe syntax constraints.

Static semantics can also be described by attribute grammars, but it would be more appropriate to expand LSL to specify both static and dynamic semantics using relational rules. This is the approach taken by the Centaur system [mDD⁺85]. Bradley has developed a semantic formalism which handles

several stages of evaluation in an integrated manner [Bra87]. Possibly the theory could be extended in this way.

13.5.2 Concrete Syntax

LSL would be more pleasant to use if terms could be written in concrete syntax instead of abstract syntax. The easiest way would be to allow strings to replace terms and to parse these strings with the parser that is generated from the LSL grammar rules.

This would impose a three-phase structure on the generator. The parser would have to be generated first and then linked with the rest of the generator code. The resulting program would then generate the C++ tables as at present. This scheme also requires that the parser be able to cope with all fragments of the language.

13.5.3 Efficiency

The Animator Generator is slow, and could be speeded up. Apart from general programming improvements, or linking it to a compiler generator, there are some specific optimisations that could be considered.

One optimisation involves temporal views. The display of an animation is recomputed after every step for each view. Temporal views remain unchanged by a step apart from new material which is added at the end. Indeed, in most debuggers a temporal view is normally implemented as a trace written to a file or to the screen. It should be possible to add a new type of view called TRACE, which doesn't recompute the display after every step but stores a trace to a file instead. The display algorithm would have to keep track of which material in the file corresponded to which node, but the contents of the file would only have to be changed if the current memory, environment or result were changed.

Another way to speed up the display process would be to compile the LSL rules into C++ functions, rather than interpreting them at run-time.

Another optimisation saves space rather than time. The Animator Generator stores the entire evaluation tree, which takes up large amounts of memory. Most animators and debuggers store only the current environment, the current memory and the path from the root of the tree to the selected node. This greatly reduces memory use, at the cost of being unable to support reverse evaluation. An animator could provide the option of storing or not storing the tree, or of storing only part of the tree.

13.5.4 Programming Environments

The Animator Generator would be much more powerful and easy to use if it were integrated with a programming environment. This section discusses some of the tools that would be useful in such an environment.

One possibility is a structure editor. This could be generated from an LSL specification. Many structure editors include facilities for checking syntactic constraints using attribute grammars. This would be an easy extension to LSL grammars.

Another possibility is a compiler generator that would generate a compiler from an LSL specification. The design and implementation of such a beast is a major task. Dam and Jensen made a start with their paper [DJ86], but most work on compiler generation is based on denotational semantics instead of relational semantics. A compiler generator could be linked to The Animator Generator so that an animator generated by The Animator Generator would run code compiled by the compiler, but this would be even harder. It's a desirable goal, however, as the resulting animators could be used on much larger programs than existing animators.

A programming environment could also provide several different views. It might include a tool for generating slices, or one for generating call graphs. These could be integrated with The Animator Generator, perhaps using the ideas of view sets and display attributes.

Another possible tool in an LSL environment is a theorem prover. Semantics of languages are often used to prove useful properties of these languages, or of translation schemes involving those languages [Des86]. Thus it would be useful to have a theorem prover that would support relational semantic rules as specified in LSL.

In a similar vein, Da Silva has developed a transformation technique that takes a relational semantic specification and produces an equivalent specification in which each constructor is only described by a single rule [dS]. This removes the need for the determinacy restriction and the treatment of congruence in my theory. This technique could be implemented as part of an environment for manipulating relational semantic specifications. Such an environment could also include a rule editor: an editor that allows language definers to edit “typeset” versions of rules instead of ASCII approximations. A similar tool could take a specification written in LSL and produce typeset output.

13.5.5 Summary

The Animator Generator adequately demonstrates the ideas of this thesis, and is a useful tool for testing the generality of the theory, but it is not production quality software. It could be improved by eradicating the shortcomings listed in Section 12.6, by supporting more features, and by being making it more efficient. Advances to the theory to include new operations and new features for specifying views should also be incorporated into The Animator Generator. Ideally The Animator Generator would be made part of a larger programming environment.

13.6 Chapter Summary

The theory of animation presented in this thesis has defined a new area of research. This chapter has outlined several ways in which this work could be continued and developed. These include testing The Animator Generator on more languages, extending the descriptive power of the notation for specifying views, extending the theory to describe more operations, improving the implementation and integrating The Animator Generator with other language independent systems.

Chapter 14

Review and Conclusion

This thesis has presented a theory of program animation based on structured operational semantics. It has also described an implementation of an animator generator based on this theory. This chapter reviews the key points of the thesis.

14.1 Animation

Program animation combines research in debuggers and research in aids for teaching programs. Animation can help people learn to program, to learn a new language, to understand other people's code and to debug programs. It does this by letting people compare their model of a language or program with the behaviour of a correct model, phrase by phrase.

The fundamental operation of animation is the evaluation step. Each step must change the representation of the program in a way that is reflected in the display of the program. This shows users how each phrase of the program is evaluated.

Animation presents the user with a view of the program. There are many possible views of a program, including lexical views, outlines, call graphs,

memory views and semantic views. Animators help users perform the above tasks if they present a range of views, including source views that correspond to different types of proximity and levels of granularity.

Animators also help users perform these tasks if they provide a range of advanced operations. These include phrase evaluation, break points, reverse evaluation, and the ability to change the current memory.

14.2 Generating Animators

This thesis shows that we can generate animators from the specification of the semantics and views of a language, just as we can generate parsers and structure editors from the syntax of a language, or compilers from its syntax and semantics. Most existing language-independent systems don't provide animation facilities. Even Centaur and PSG, which provide some support, lack a clear definition of the operations that they provide. They also only support a single view.

An animator must incorporate a definition of an evaluation step and at least one view for the language being animated. Therefore an animator generator must either provide such definitions itself or require that they are included in the specification of each language. In either case we need a way of defining them in terms of a program in mid-evaluation. Therefore we need a theory of animation based on a semantic representation of a program in mid-evaluation.

In this thesis I have shown how such a representation can be defined in relational semantics as an inference tree. I have shown how evaluation steps and views can be defined in terms of these trees. I have also shown how this framework can be used to discuss different definitions of step. I require the specification of each language to include specifications of the views for that language, and have defined most operations in a language independent manner.

14.3 Defining a Step

Different views display different parts of the program. Since each step should change the representation in a way that is shown by the display of the program, the size of a step depends on the selected view. Either the specification of a view must include a definition of a step, or the definition of a step must be parameterised on a view, or the animator should repeatedly apply a fundamental definition of a step until the view changes.

I have chosen to define a fundamental definition of a step and to augment the specification of a view with display attributes. These attributes define when a view-specific step must stop. Thus in one sense the definition of a view-specific step is part of the specification of the view, in another in is a common definition that is parameterised on a view, and in a third it consists of repeated applications of a fundamental step.

A fundamental step can be defined in terms of a basic view, in which every phrase is replaced with its value as the program is evaluated. This shows more of the program than other non-semantic views, and so steps for other views can be defined in terms of the fundamental step. The fundamental step is a minimal extension of a relational animation. The extent of the step is controlled by the definition of validity, which ensures that each fundamental step changes the representation of the program in a way that is reflected by the basic view.

Plotkin's original presentation of structured operational semantics encoded an intuitive definition of computation steps as transitions. It seems reasonable to examine the suitability of this definition for animation. In Chapters 5 to 7 I define a computation step in relational semantics that is equivalent to that encoded in transitional semantics. I prove that the definitions are equivalent by defining an equivalence between specifications in relational and transitional

semantics, and then between relational and transitional evaluation histories, and showing that applying the step functions to equivalent evaluation histories produces equivalent results. This definition of a computation step in relational semantics combines the advantages of compositionality of relational semantics with the explicit model of evaluation of transitional semantics.

Unfortunately computation steps don't produce desirable animations. They don't show the result of a phrase before advancing to the phrase that will be evaluated next. They also jump straight to the next phrase instead of guiding the user through the structure of the program.

However, the definition can be refined to give a definition that produces better animations. This new definition is called an animation step. It is a straightforward adaption of the definition of a computation step, and has the property that every computation step is also an animation step. A few people have used an animator that uses this definition of a step and have found it satisfactory. As discussed in the previous Chapter, it would be worth experimenting with other definitions to find the best.

14.4 Specifying Views

A wide range of views can be specified by giving a display rule for each constructor and semantic rule. Each display rule specifies how an instance of the corresponding constructor or semantic rule is displayed, in terms of a layout language. The more powerful the layout language, the wider the range of views that can be specified.

Sample features of a layout language include displaying string literals, displaying arguments of terms or variables of rules, enclosing graphical expressions in boxes, drawing lines from one box to another, and centring expressions. Extra facilities for semantic display rules include testing whether

sub-animations are empty, in progress or completed, and highlighting expressions if the subject or result of the instance of the rule is selected.

Display rules may be parameterised so that the display of some constructs can depend on their context. This is useful for several views, such as procedure-level outline views in Proc. It also provides a mechanism for interfacing an animator to specialised software tools such as slice generators.

Views can be classified by whether they display source, environments, memories or semantics and by whether they are static or dynamic. Source views may be further classified by type of proximity and level of granularity. Chapter 9 describes several views in these terms and shows how they can be defined using the above technique. This helps to demonstrate that the theory can describe practical systems.

14.5 Advanced Operations

To be practical animators must provide more operations than stepping. Chapter 2 mentions several operations that contribute to an animator's support of users in their tasks. Chapter 10 shows how all these operations can be defined in terms of the theory. Several operations, such as phrase evaluation, reverse evaluation and break points can be defined in terms of view-specific steps. Others, such as changing the current memory, can be defined in terms of augmented animation histories.

14.6 Implementation

The Animator Generator shows that the theory can form the basis of a useful software tool. Its design follows the theory closely; it models a program in mid-evaluation as a tree of instances of relational rules. It produces animators that provide most of the advanced operations defined in Chapter 10.

The specification language of The Animator Generator is called LSL. An LSL specification includes the abstract syntax, parse rules, semantic rules, display rules and display attributes for the object language. It uses a straightforward ASCII translation of the notation used in this thesis.

The Animator Generator is a useful tool for language designers and for generating simple animators quickly. It is a prototype rather than a production quality system. The generated animators have a poor user interface, and to compete with hand-crafted animators it would need an interface to a compiler generator. However, it does illustrate that a practical system can be based on the theory of animation.

14.7 Assessing the Theory

The theory presented in this thesis has a firm semantics base, can describe all the operations deemed important to the tasks considered in Chapter 2, and can describe a wide range of views. It admits all the rules used to define the dynamic semantics of Standard ML, with the slight modifications discussed in Section 6.1.5. Chapter 11 suggests that it will also be flexible enough to describe pure Prolog. The existence of The Animator Generator demonstrates that it can form the basis of a practical implementation.

The choice of relational semantics has proved successful. The operational reading of the rules and the obvious semantic model of a program in mid-evaluation have lead to a reasonably simple theory. Relational semantic specifications are concise and easy to reason about.

One of the reasons for choosing structured operational semantics has not paid off. The definition of computation steps encoded in transitional semantics and translated to relational semantics in Chapters 5 to 7 turned out to be inappropriate for animation. However, the definition of a computation step in relational semantics was easily refined to give the definition of an animation step. This definition has the property that every computation step is also an animation step, so that the sequence of computation steps in an evaluation can be projected from the sequence of animation steps, should this be required. This work also shows that the theory provides a framework within which different definitions of step can be compared.

It would be interesting to compare this theory with definitions of steps and views in other semantic formalisms. Such a comparison might give some insight into the relative strengths of the different formalisms. In particular it would be interesting to see if the rewrite rules of a definitional interpreter correspond to animation steps.

The decision to define a step independently of a particular language or view seems to have paid off. The definition of animation steps, suitably parameterised on a view, produces reasonable animations for imperative languages, eager functional languages and logic programming languages. We need to experiment with other languages to see how general this language-independent definition really is.

The decision to make specifications of views be language-specific has also given good results. This approach gives flexibility across the range of languages and views. We need to experiment with specifying more languages to find a

sufficiently powerful set of display primitives for our purposes. If we can define some views independently of a particular language then it would be desirable to do so, but it is not obvious that this will be possible.

The above discussion has shown that the choices made in developing the theory of animation seem to have borne fruit. The resulting theory is powerful enough to describe a range of languages and views, and forms the basis of a prototype implementation. It also forms a framework which we can use to compare definitions of animation operations.

14.8 Conclusion

In this thesis I have presented a formal theory of animation. This theory shows how an animator for a language can be derived from a formal specification of that language. The specification of the language must include a specification of at least one view or the language in mid-evaluation. The theory includes a notation for specifying such views. The theory defines animation operations independently of a particular language.

The theory is powerful enough to describe a range of views and operations that have been shown to be useful by existing language-specific animators or by psychological research. The thesis includes some suggestions as to how imperative languages, eager functional languages, and logic programming languages could be specified in terms of the theory.

I have used the theory to design and implement The Animator Generator. This takes specifications of a language in LSL and produces an animator for that language. Both *lsl* and The Animator Generator are clearly modelled on the theory. Although The Animator Generator is only a prototype, it has been used in teaching, and it demonstrates that the theory can form the basis of a practical implementation.

Appendix A

Equivalent Animations

This appendix defines animation steps in terms of transitional semantics and proves that this definition is equivalent to the definition for relational semantics. The definition of an animation step is much more complicated than the definition of an evaluation step. This suggests that relational semantics is a better framework for comparing definitions of animation operations.

The first two sections of this appendix define animation histories and animation steps in terms of transitional semantics. These definitions are similar to those for evaluation histories and computation steps given in Chapter 5. The last section shows that the definitions are equivalent. It uses the definition of equivalent specifications developed in Chapter 7, and the proof follows the same approach as the proof of equivalence for computation steps that was given in Chapter 7.

A.1 Transitional Animation Histories

Animation steps gradually “zoom in” on the phrase to be evaluated. So a transitional animation is like a transitional evaluation except that the last stack in the sequence may be incomplete, and an animation step may build this stack one rule at a time as it moves the focus to smaller phrases of the language. The last stack is separated from the rest of the sequence, and a transitional animation is written (S, s) . If s or S is empty, the animation history is written $(S, .)$ or $(., s)$. The notation $S@s$ denotes the sequence formed by appending s to S .

Animation steps also show the result of a phrase in place, replacing the evaluated phrase. Therefore the animation of most axioms and remote premises must take two steps: one to make the focus be the subject of the sequent and one to make it the result. I represent this as follows.

Let (S, s) be a transitional animation. If s is not empty, then the focus of the animation is $subject(premise(top(s)))$, or $subject(top(s))$ if $top(s)$ is an axiom. This is the phrase that is about to be evaluated. The evaluation of a remote premise or axiom usually produces $(S@s, .)$. When the stack of a transitional animation (S', s') is empty, then the the focus is $result(top(last(S)))$ if $top(S)$ is an axiom and $result(premise(top(last(S))))$ if it has a remote premise. Thus in $(S@s, .)$ the focus is the result of the phrase that has just been evaluated.

If a rule has a remote premise and a value result this convention only works if the rhs of the premise is the same as that of the rule. This because only the result of the premise will become the focus of the animation. If a rule with a remote premise and a value result is in a rule set that is realised by a relational rule set, then the definition of relational realisation tells us that the two values will always be the same. I assume that we are not interested in rules that don't fit this restriction. If it were required, a more general treatment could introduce

a boolean variable to indicate which result should be displayed, similar to the variable in a relational animation.

The preceding paragraphs outline the basic ideas behind a definition of a transitional animation history. However, we want to represent an animation by a sequence of animation histories, each being the minimal extension of the previous one. The definition needs several refinements before such a sequence gives the exact behaviour that we desire.

The first refinement concerns the growth of the stack. If the stack is empty after each phrase as been evaluated, and grows one instance at a time, then the display will keep highlighting the whole program and zooming in the the current phrase time and time again. Therefore we must only allow $top(s)$ to have a local premise, if the subject of that premise is a different phrase from the result of the corresponding premise (if any) in $last(S)$.

The second refinement applies to axioms that don't have value results. If such an axiom is in a rule set that is realised by a relational rule set, then it corresponds to the local recursive premise of a relational rule. A relational animation will display the subject of a recursive premise. This corresponds to the result of the transitional axiom. The subject of the transitional axiom will never be displayed. Therefore the top instance of the stack may never be such an axiom.

The last two refinements prevent an animation from generated consecutive displays that are identical, except in the special case of an empty loop or similar construct.

If $top(s)$ is an axiom, then its subject is displayed. Therefore $top(s)$ may only be an axiom if the subject of the premise of the rule below it could not have been displayed and $result(top(last(S)))$ couldn't have been displayed.

If a rule has a remote premise and a result that is a value, then the subject of its premise will often be displayed in the same way as the subject of the rule

itself. This is the same feature noted for relational animations with remote recursive premises. I assumes that such rules always will be displayed in this way. Therefore $top(s)$ may only be an instance of such a rule if the subject of the premise of the rule below it could not have been displayed.

These refinements make the definition of an animation history significantly more complicated. However, they are necessary to give the behaviour that we desire.

A.1.1 Auxiliary Definitions

The informal description of an animation history given above included some concepts that need to be defined precisely before we can formalise the definition of an animation history itself.

Two terms differ if they occur in different parts of a parent term. Two terms may differ even if they are structurally identical. Difference is a comparison of occurrence rather than structure. It is used in the following definition.

The next definition is used to control the growth of the stack in an animation history (S, s) . If S is empty, then every instance added to s is an animation step. Similarly, if $\#(last(S)) < \#s$, then every new instance added to s is an animation step. These steps implement the “zooming in” to the next phrase to be evaluated. However, if $\#(last(S)) \geq \#s$, then we don’t always want to “zoom in”. As discussed above, we want to avoid “zooming in” from the whole program with each new stack. We can do this by requiring that the result of the $n + 1^{th}$ instance in $last(S)$ differs from $subject(premise(top(s)))$.

This definition is also used in the cases of the definition that correspond to the third and fourth refinements. These are the ones that prevent two adjacent animation histories being identical. They both use the idea of the subject of the

premise of the instance below the top instance being one that could have been displayed in a previous animation.

If (S, s) is a transitional animation history and $1 < n < \#s$ then the premise q of the n^{th} instance in s is valid if one of the following conditions holds:

1. $\text{length}(S) = 0$.
2. $\#(\text{last}(S)) \leq n$.
3. $\#(\text{last}(S)) > n$ and $\text{subject}(q)$ differs from the result of the $n + 1^{\text{th}}$ instance in $\text{last}(S)$.

A.1.2 Definition of Animation Histories

The following definition formalises the above discussion. The fourth case corresponds to the first refinement mentioned in that discussion. The fifth case corresponds to the second and third refinements, and the the sixth case corresponds to the fourth refinement.

A **transitional animation** of a program (c, m, e) is a consistent sequence S and a partially consistent stack s , such that:

1. If $\text{length}(S) > 0$, then $\text{lst}(S) = (p, m, e)$. If $\text{length}(S) = 0$ and $\#s > 0$, then $\text{lst}(s) = (p, m, e)$.
2. All terms on the rhs of a remote premise in s must be closed.
3. If $\text{length}(S) > 0$ and $\#s > 0$ then $\text{lst}(s) = \text{rst}(S)$.
4. If $\text{top}(s)$ has a local premise then that premise must be valid.
5. If $\text{top}(s)$ is an axiom and $\#s = n$, then $\text{result}(\text{top}(s))$ must be a value, and the premise of the $n - 1^{\text{th}}$ instance in s must not be valid.

6. If $top(s)$ has a remote premise, $result(top(s))$ is a value and $\#s = n$, and the premise of the $n - 1^{th}$ instance in s must not be valid.

The **env**, **subject**, **lhm**, **lhs** and **lst** of a transitional animation (S, s) are those of S if S isn't empty, or those of s if S is empty. The **result**, **rhbm**, **rhs** and **rst** of (S, s) are those of s if s isn't empty, or those of S if s is empty. All these functions are undefined if both S and s are empty.

The focus of a transitional animation (S, s) of (c, m, e) is defined as follows:

1. If s is empty, S is not empty and $top(S)$ is an axiom then the focus is $result(top(last(S)))$.
2. If s is empty, S is not empty and $top(S)$ has a premise then the focus is $result(premise(top(last(S))))$.
3. If s is not empty and $top(S)$ is an axiom then the focus is $subject(top(last(S)))$.
4. If s is not empty and $top(S)$ has a premise then the focus is $subject(premise(top(last(S))))$.
5. If both s and S are empty then the focus is c .

A.1.3 Initial and Complete Animations

The initial transitional animation of a program (p, m, e) is an empty sequence and an empty stack.

A transitional animation (S, s) is **complete** if the result of S is a value. (For this to hold s must be empty.)

Lemma A.1 *A complete transitional animation $(S, \cdot)$ of a phrase (c, m, e) is unique.*

This follows immediately from Lemma 5.1.

A.1.4 Transitional Trees Revisited

A transitional animation (S, s) is an instance of a partially consistent stack s' if the partially consistent sequence $S@s$ is an instance of s' . If (S, s) is an instance of a path from the root of a transitional tree, then this path is called the **tree path** of (S, s) , and is written $S_{(S, s)}$.

Lemma A.2 *If (S, s) is a transitional animation of (c, m, e) and τ is the transitional rule that strongly matches (c, m, e) then (S, s) is an instance of a path from the root of the transitional tree of τ .*

Proof: The proof follows that of Lemma 7.2.

A.1.5 Summary

The difference between the definitions of transitional animation histories and transitional evaluation histories resembles the difference between relational animation histories and transitional evaluation histories. The animation histories are more complex, to reflect the smaller size of the animation steps. They support the ideas of zooming in to the next sub-phrase and of showing the result of each sub-phrase as it is evaluated.

Transitional animation histories show this greater detail by separating the last stack of the history from the rest and letting it grow a rule at a time.

A.2 Transitional Animation Steps

This section defines a transitional animation step, gives a small example of an animation, and proves some simple facts about the definition. The definition is based on the idea of a minimal extension of an animation history, in much the same way as the definition of relational animation steps.

A.2.1 Definition

If (S, s) and (S', s') are transitional animations, (S', s') **extends** (S, s) if one of the following conditions holds:

1. $S@s$ is a prefix of S' .
2. $S = S'$ and $\#s' > \#s$.
3. S, s , and S' are all empty, and $\#s'$ is not empty.

(S', s') **minimally extends** (S, s) if there is no transitional animation (S'', s'') such that (S'', s'') extends (S, s) and (S', s') extends (S'', s'') .

A **animation step** for transitional animations is a function $\mathcal{A}_T$ such that if (S, s) is a transitional animation, then $\mathcal{A}_T(S, s)$ is the minimal extension of (S, s) .

A.2.2 Example

If the environment e binds 'a' to 2 and 'b' to 3, the following sequence is a series of transitional animations of 'a' > 'b', m and e , such that each animation is reached from the previous one by an application of $\mathcal{A}_T$.

After the first step, the animation selects the first sub-phrase to be evaluated, in this case 'a':

$$s \quad \boxed{\frac{e \vdash 'a', m \rightarrow c'_1, m'}{e \vdash 'a' > 'b', m \rightarrow c'_1 > 'b', m'}}$$

The second step evaluates the call to the remote rule set. Note that a transitional animation with an empty sequence and a stack containing the two instances shown here would be illegal, by case 6 of the definition of transitional animations.

$$s \quad \boxed{\begin{array}{c} \text{LOOKUP} \\ \frac{\vdash \text{lookup}(e, 'a'), m \rightarrow 2, m}{e \vdash 'a', m \rightarrow 2, m} \\ \\ \frac{e \vdash 'a', m \rightarrow 2, m}{e \vdash 'a' > 'b', m \rightarrow 2 > 'b', m} \end{array}}$$

empty

The third step selects the next sub-phrase to be evaluated, in this case 'b':

$$s \quad \boxed{\begin{array}{c} \text{LOOKUP} \\ \frac{\vdash \text{lookup}(e, 'a'), m \rightarrow 2, m}{e \vdash 'a', m \rightarrow 2, m} \\ \\ \frac{e \vdash 'a', m \rightarrow 2, m}{e \vdash 'a' > 'b', m \rightarrow 2 > 'b', m} \end{array}}$$

$$s \quad \boxed{\frac{e \vdash 'b', m \rightarrow c'_2, m'}{e \vdash 2 > 'b', m \rightarrow 2 > c'_2, m'}}$$

The fourth step evaluates the call to the remote rule set.

$$\begin{array}{c}
 S \left\{ \begin{array}{l}
 \text{LOOKUP} \\
 \frac{\vdash \text{lookup}(e, 'a'), m \rightarrow 2, m}{e \vdash 'a', m \rightarrow 2, m} \\
 \\
 \frac{e \vdash 'a', m \rightarrow 2, m}{e \vdash 'a' > 'b', m \rightarrow 2 > 'b', m} \\
 \\
 \text{LOOKUP} \\
 \frac{\vdash \text{lookup}(e, 'b'), m \rightarrow 3, m}{e \vdash 'b', m \rightarrow 3, m} \\
 \\
 \frac{e \vdash 'b', m \rightarrow 3, m}{e \vdash 2 > 'b', m \rightarrow 2 > 3, m}
 \end{array} \right. \\
 s \quad \text{empty}
 \end{array}$$

The fifth step selects the overall phrase:

$$\begin{array}{c}
 S \left\{ \begin{array}{l}
 \text{LOOKUP} \\
 \frac{\vdash \text{lookup}(e, 'a'), m \rightarrow 2, m}{e \vdash 'a', m \rightarrow 2, m} \\
 \\
 \frac{e \vdash 'a', m \rightarrow 2, m}{e \vdash 'a' > 'b', m \rightarrow 2 > 'b', m} \\
 \\
 \text{LOOKUP} \\
 \frac{\vdash \text{lookup}(e, 'b'), m \rightarrow 3, m}{e \vdash 'b', m \rightarrow 3, m} \\
 \\
 \frac{e \vdash 'b', m \rightarrow 3, m}{e \vdash 2 > 'b', m \rightarrow 2 > 3, m}
 \end{array} \right. \\
 s \quad \begin{array}{l}
 \text{INTEGER} \\
 \frac{\vdash 2 > 3, m \rightarrow \text{ff}, m'}{e \vdash 2 > 3, m \rightarrow \text{ff}, m'}
 \end{array}
 \end{array}$$

The last step produces the final result:

$$\begin{array}{c}
 S \left\{ \begin{array}{l}
 \text{LOOKUP} \\
 \frac{\vdash \text{lookup}(e, 'a'), m \rightarrow 2, m}{e \vdash 'a', m \rightarrow 2, m} \\
 \frac{e \vdash 'a', m \rightarrow 2, m}{e \vdash 'a' > 'b', m \rightarrow 2 > 'b', m} \\
 \text{LOOKUP} \\
 \frac{\vdash \text{lookup}(e, 'b'), m \rightarrow 3, m}{e \vdash 'b', m \rightarrow 3, m} \\
 \frac{e \vdash 'b', m \rightarrow 3, m}{e \vdash 2 > 'b', m \rightarrow 2 > 3, m} \\
 \text{INTEGER} \\
 \frac{\vdash 2 > 3, m \rightarrow \text{ff}, m}{e \vdash 2 > 3, m \rightarrow \text{ff}, m}
 \end{array} \right. \\
 s \qquad \qquad \text{empty}
 \end{array}$$

Lemma A.3 *If (S, s) is an incomplete transitional animation, then $\mathcal{A}_T(S, s)$ is unique.*

The proof is similar to that of Lemma 5.2.

Lemma A.4 *If (S, s) is a transitional animation of (c, m, e) and $(S', s') = \mathcal{A}_T(S, s)$ then:*

1. *If s is not empty then either $\text{length}(S') = \text{length}(S) + 1$ and s' is empty or $S' = S$ and $\#s' = \#s + 1$.*
2. *If s is empty then either $\text{length}(S') = \text{length}(S) + 1$ and s' is empty or $S' = S$ and s' is not empty.*

Proof, part 1: s is not empty. Consider cases of $\text{top}(s)$:

If $\text{top}(s)$ is an axiom or has a remote premise then $\mathcal{A}_T(S, s) = (S @ s, \cdot)$, by the definition of $\mathcal{A}_T$, and the result follows immediately.

If $\text{top}(s)$ has a local premise then from the definition of transitional animations one of the following conditions holds:

1. $\#last(S) < \#s$
2. S is empty
3. $\#last(S) \geq n$ and either t is an axiom or $result(premise(t))$ differs from $subject(premise(top(s)))$, where t is the n^{th} instance in $last(S)$.

Let s'' be s extended by a new instance t' . Since $\#s'' = \#s + 1$, the conditions above also hold for s'' .

By the definitions of $\mathcal{A}_T$ and transitional animations, if t' is an axiom or has a remote premise then $\mathcal{A}_T(S, s) = (S@s'', \cdot)$, and if t' has a local premise then $\mathcal{A}_T(S, s) = (S, s'')$. In either case the result follows immediately.

Proof, part 2: s is empty. The proof uses induction on the structure of $last(S)$.

Base Case: Either S is empty or $bottom(last(S))$ is an axiom or $result(premise(bottom(last(S))))$ is a value.

If S is empty, then (S, s) is an initial animation of (c, m, e) . In this case, let τ be the transitional rule that strongly matches (c, m, e) . If S is not empty, let τ be the transitional rule that strongly matches $rst(S)$. In both cases let s'' be the stack that contains just the appropriate instance τ , and consider cases of τ .

1. If τ is an axiom, then if S is empty, $bottom(last(S))$ is an axiom or $result(bottom(s))$ is not a value, then $\mathcal{A}_T(S, s) = (S@s'', \cdot)$; otherwise $\mathcal{A}_T(S, s) = (S, s'')$ and $\#s'' = 1$, by the definition of $\mathcal{A}_T$.
2. If τ has a remote premise, then if $result(bottom(s''))$ is a value and either S is empty or $bottom(last(S))$ is an axiom, then $\mathcal{A}_T(S, s) = (S@s'', \cdot)$; otherwise $\mathcal{A}_T(S, s) = (S, s'')$ and $\#s'' = 1$, by the definition of $\mathcal{A}_T$.
3. If τ has a local premise, then $\mathcal{A}_T(S, s) = (S, s'')$ and $\#s'' = 1$, by the definition of $\mathcal{A}_T$.

Induction Step: $result(premise(bottom(last(S))))$ exists and is not a value. Let τ be the transitional rule that strongly matches $rst(S)$. Let t be the corresponding instance of τ .

The condition on the induction step ensures that t is an instance of the same rule as $bottom(last(S))$, which has a local premise. Therefore $lhs(premise(t))$ doesn't differ from $rhs(premise(bottom(last(S))))$. Thus a stack containing just t would not be consistent. Therefore $\#s'' > 1$.

By induction $\mathcal{A}_T([last(S)], \cdot) = (S^\dagger, s^\dagger)$, where either $length(S^\dagger) = 2$ and $s^\dagger$ is empty, or $S^\dagger = S'$ and $s^\dagger$ is not empty.

If $s^\dagger$ is empty then $\mathcal{A}_T(S, s) = (S@s'', \cdot)$, where $[s''] = last(S^\dagger)$. If $s^\dagger$ is not empty then $\mathcal{A}_T(S, s) = (S, s'')$, where $[s''] = s^\dagger$. Both cases follow from the definition of $\mathcal{A}_T$, and the result follows immediately.

A.2.3 Summary

A transitional animation step is defined as a minimal extension of a transitional animation history. Most of the complexity of the process is included in the definition of an animation history, and this makes the definition of a step comparatively straightforward. Animation steps are determinate, and extend an animation in a limited number of ways.

A.3 Equivalent Animations

This section defines when a transitional animation and a relational animation are equivalent. It then proves that applying a relational and a transitional animation step to the (empty) initial animations of the same program in equivalent specifications produces equivalent animations. This definition and

proof are similar to ones given in Chapter 7, but have extra cases to handle the more complicated definitions of animations compared to evaluations.

This work uses the definition of equivalent specifications that was given in Chapter 7. This shows that the definitions of animations doesn't change the functional meaning of a semantics.

A.3.1 Definition

The definition of equivalence for animations is similar to that for evaluations (see Section 7.4). It is more complex because it has to allow for partially consistent stacks in transitional animations and the boolean variable in relational animations. The first extra case is case 2(c), which is a special case of case 2(b). The other extra cases are case 3, which applies when $o(\mathcal{R}, b)$ is a non-recursive local premise, and case 8, which applies when $o(\mathcal{R}, b) = \text{conclusion}(\text{root}(\mathcal{R}))$ and $b = \text{ff}$.

If a transitional specification and a relational specification are equivalent, $(\mathcal{S}, s)$ and $(\mathcal{R}, b)$ are non-empty transitional and relational animations in these specifications and ρ is the rule instantiated at the root of $(\mathcal{R}, b)$, then $(\mathcal{S}, s)$ and $(\mathcal{R}, b)$ are equivalent, written $(\mathcal{S}, s) \approx (\mathcal{R}, b)$, if $\mathcal{S}$ can be partitioned into subsequences $\mathcal{S}_1, \dots, \mathcal{S}_{j+1}$, where j is the number of matched premises of $\text{root}(\mathcal{R}, b)$, such that the following conditions hold:

1. $\text{lst}(\mathcal{R}, b) = \text{lst}(\mathcal{S}, s)$.
2. For all i such that $1 \leq i \leq j$, if q is the i^{th} premise of $\text{root}(\mathcal{R})$ then the following conditions hold:
 - (a) If q is trivial and non-recursive then $\mathcal{S}_i$ is empty.
 - (b) If q is local, non-trivial and non-recursive, and either $o(\mathcal{R}, b)$ is not in $(\mathcal{R}, b)[i]$ or s is empty then $[\mathcal{S}_i]$ is defined and $([\mathcal{S}_i], \cdot) \approx (\mathcal{R}, b)[i]$.

- (c) If q is local, non-trivial and non-recursive, $o(\mathcal{R}, b)$ is in $(\mathcal{R}, b)[i]$ and s is not empty then either $[S_i]$ and $[s]$ are defined and $([S_i], [s]) \approx (\mathcal{R}, b)[i]$ or S_i is empty and $(\cdot, [s]) \approx (\mathcal{R}, b)[i]$.
 - (d) q is local and recursive iff $length(S_i) > 0$, $bottom(head(S_i))$ is an axiom, $rst(head(S_i)) = lst(q)$, and if $length(S_i) > 1$ or $(\mathcal{R}, b)[i]$ is not empty then $(tail(S_i), s) \approx (\mathcal{R}, b)[i]$,
 - (e) q is remote iff $length(S_i) = 1$, $\#(head(S_i)) = 1$ and $premise(bottom(head(S_i))) = q$. If $O(\mathcal{R}, b) = q$ then s is empty.
3. $o(\mathcal{R}, b)$ is a remote premise of $root(\mathcal{R})$ and $b = \mathbf{ff}$ iff S_i is empty, $\#s = 1$, $premise(bottom(s))$ is a remote premise that refers to the same remote rule set as $o(\mathcal{R}, b)$, $lst(premise(bottom(s))) = lst(o(\mathcal{R}, b))$, and $rhs(premise(bottom(s))) = rhs(o(\mathcal{R}, b))$.
 4. $o(\mathcal{R}, b)$ is a non-recursive local premise of $root(\mathcal{R})$ iff $premise(bottom(s))$ is a local premise, $\#s = 1$ and $lst(premise(bottom(s))) = lst(o(\mathcal{R}, b))$.
 5. $o(\mathcal{R}, b)$ is the local recursive premise of $root(\mathcal{R})$ iff $length(S_{j+1}) = 1$, $bottom(head(S_{j+1}))$ is an axiom, $lst(S_{j+1}) = lst(o(\mathcal{R}, b))$ and s is empty.
 6. If ρ is recursive and $o(\mathcal{R}, b)$ is not the local recursive premise of $root(\mathcal{R})$, then S_{j+1} is empty.
 7. If ρ is recursive then $conclusion(root(\mathcal{R}))$ is matched iff $rhs(S) = rhs(\mathcal{R})$ and s is empty.
 8. ρ is not recursive, $o(\mathcal{R}, b) = conclusion(root(\mathcal{R}))$ and $b = \mathbf{tt}$ iff $length(S_{j+1}) = 1$, $bottom(head(S_{j+1}))$ is an axiom and $conclusion(bottom(head(S_{j+1}))) = o(\mathcal{R}, b)$.
 9. ρ is not recursive, $o(\mathcal{R}, b) = conclusion(root(\mathcal{R}, b))$ and $b = \mathbf{ff}$ iff S_{j+1} is empty, $bottom(s)$ is an axiom and $conclusion(bottom(head(s))) = o(\mathcal{R}, b)$.

10. If ρ is not recursive, and $o(\mathcal{R}, b) \neq \text{conclusion}(\text{root}(\mathcal{R}))$, then $\mathcal{S}_{j+1}$ is empty.

In addition, the empty transitional animation is equivalent to the empty relational animation.

Lemma A.5 *If a transitional specification and a relational specification are equivalent, and $(\mathcal{S}, s)$ and $(\mathcal{R}, b)$ are initial animations of (p, m, e) in those specifications, then $\mathcal{A}_T(\mathcal{S}, s)$ exists iff $\mathcal{A}_R(\mathcal{R}, b)$ exists. If they exist, then $\mathcal{A}_T(\mathcal{S}, s) \approx \mathcal{A}_R(\mathcal{R}, b)$.*

This proof is similar to, but more complicated than, the proof of Lemma 7.3, and is omitted for brevity.

A.3.2 Summary

The definition of equivalence for animation histories resembles that for evaluation histories, but is complicated by the extra detail in the definitions of animations compared to those of evaluations.

The definition uses the definition of equivalent specifications developed in Chapter 7. The notion of equivalent animations doesn't alter the semantics of the underlying specifications.

A.4 Step Functions Preserve Equivalence

This section shows that $\mathcal{A}_R$ and $\mathcal{A}_T$ are equivalent, in other words that a transitional animation of a program and a relational animation of the same program will produce identical sequences of animation steps.

This result is shown by Proposition A.7, which requires the following lemma.

Lemma A.6 *If the following conditions hold:*

1. A transitional specification and a relational specification are equivalent and (S, s) and (R, b) are equivalent animations in those specifications and are not equivalent animations.
2. There are j matched premises at $\text{root}(R)$ and $S_1, \dots, S_{j+1}$ are the sub-sequences of S used to show that $(S, s) \approx (R, b)$.
3. The relational rule instantiated at $\text{root}(R)$ is ρ .
4. The configuration terms and transitional rules in the transitional realisation of ρ are $C_1, \dots, C_l$ and $\tau_1, \dots, \tau_l$.

then the following conditions hold:

1. For all i such that $1 \leq i \leq k$, S_i is an instance of the stack s_i that contains just τ_i , where $k = j$ if S_{j+1} is empty and $k = j + 1$ otherwise.
2. If S_i is not empty, $\text{bottom}(\text{head}(S_i))$ is not an axiom and $\text{result}(\text{premise}(\text{bottom}(\text{last}(S_i))))$ exists and is a value, then $\text{result}(S_i)$ is an instance of C_{i+1} .

This proof is similar to, but more complicated than, the proof of Lemma 7.4, and is omitted for brevity.

Proposition A.7 *If a transitional specification and a relational specification are equivalent, (S, s) and (R, b) are a transitional animation and a relational animation in these specifications and $(S, s) \approx (R, b)$, then $A_T(S, s)$ exists iff $A_R(R, b)$ exists. If they exist, then $A_T(S, s) \approx A_R(R, b)$.*

The proof is similar to, but more complicated than, the proof of Proposition 7.5, and is omitted.

A.5 Appendix Summary

This appendix has defined transitional animation histories and transitional animation steps, which introduce the same complications introduced for relational semantics by relational animation histories.

This appendix has also shown that animations of a program in equivalent specifications produce the same sequence of animation steps. In other words, the definition of animation step given here is independent of the chosen formalism. It uses the definition of equivalent specifications given in Chapter 7, so the definitions of animation steps have not altered the underlying semantics.

Appendix B

A Guide to the Notation

This is a quick reference to the notation used throughout the thesis.

B.1 Terms and Metavariables

The notation for variables, terms, metavariables and expressions used in abstract syntaxes, rules and rule forms is given in the following table:

	Objects	Variables	Terms	Metavariables	Expressions
Configurations	$plus(3, 5)$	c	$plus(3, v)$	γ, C	$a(c, c')$
Values	$2, 'a'$	v	tt	ν, V	$a(V)$
Phrases	$plus(3, 5)$	p	$plus(3, v)$	π, P	P
Environments	$('a', 3)$	e	$e \dagger(p, v)$	ε, E	$E \dagger E'$
Memories	(v, v')	m	$m \dagger(v, v')$	μ, M	M
Constructors	$plus$	a	$-$	$-$	$-$
Identifiers	$'a'$	x	$-$	$-$	$-$
Booleans	tt, ff	b	$-$	$-$	$-$
Integers	$2, -1$	i, j, k, l, m, n	$-$	$-$	$-$

All variables (in this and other sections) may be primed or subscripted as required. The text also uses variables subscripted by integer variables (e.g. P_i); this indicates a variable from a set of appropriately numbered variables.

B.2 Substitution

The following table gives the notation for substitutions:

$\mathcal{F}(C)$	(Free) Variables of C
$C[\gamma'/\gamma]$	Substitution of γ' for γ in C
σ	Substitutions
$\sigma \circ \sigma'$	Composition of substitutions

$\mathcal{F}$ and $[\gamma'/\gamma]$ can be applied to any term, sequent or rule; the configurations shown here are examples.

B.3 Sequents and Rules

Sequents have the following forms:

	Local	Remote
Relational:	$env \vdash subject, lhm \Rightarrow result, rhm$	$env \vdash^{NAME} subject, lhm \Rightarrow result, rhm$
Transitional:	$env \vdash subject, lhm \rightarrow result, rhm$	$env \vdash^{NAME} subject, lhm \rightarrow result, rhm$

The variables used for sequents and rules are as follows:

q	Sequents
τ	Transitional Rules
t	Instances of Transitional Rules
ρ	Relational Rules
r	Instances of Relational Rules

The next table lists the operations used on sequents and rules.

$env(q)$	The env of q .
$subject(q)$	The subject of q .
$lhm(q)$	The lhm of q .
$lhs(q)$	The lhs of q .
$lst(q)$	The lst of q .
$result(q)$	The result of q .
$rhv(q)$	The rhv of q .
$rhs(q)$	The rhs of q .
$rst(q)$	The rst of q .
$conclusion(\rho)$	The conclusion of ρ .

conclusion is defined on all rules and instances of rules. A relational rule is used here as an example.

B.4 Stacks and Sequences

The variables used for stacks and sequences are as follows:

s	Stacks
$\mathcal{S}$	Sequences

The next table lists the operations on stacks and sequences.

$\#s$	The height of s .
$top(s)$	The top instance of s .
$bottom(s)$	The bottom instance of s .
$[s]$	s minus $bottom(s)$.
$head(\mathcal{S})$	The first stack in $\mathcal{S}$.
$last(\mathcal{S})$	The last stack in $\mathcal{S}$.
$\mathcal{S}@s$	$\mathcal{S}$ with s appended.
$tail(\mathcal{S})$	$\mathcal{S}$ minus $head(\mathcal{S})$.
$[\mathcal{S}]$	$\mathcal{S}$ minus $bottom(s) \forall s \in \mathcal{S}$.

$head$, $last$, $tail$ and $[\mathcal{S}]$ are undefined on empty sequences. $[s]$ is undefined if $\#s > 1$. $[\mathcal{S}]$ is undefined if $[s]$ is undefined for any stack in $[\mathcal{S}]$ or if the bottom instances of the stacks in $\mathcal{S}$ aren't instances of the same rule.

The operations on sequents also apply to stacks and sequences.

B.5 Relational Evaluations and Animations

The following table gives the notation for relational evaluation histories, relational animation histories, and the corresponding step functions:

$\mathcal{R}$	Relational Evaluations
(p, m, e)	Relational Evaluation States
N	Nodes of Trees
$\mathcal{R}[N], \mathcal{R}[i]$	Relational Sub-evaluations
$\mathcal{E}_R$	Relational Evaluation Step
$(\mathcal{R}, b)$	Relational Animations
$(\mathcal{R}, b)[N], (\mathcal{R}, b)[i]$	Relational Sub-animations
$\mathcal{A}_R$	Relational Animation Step
$\mathcal{R} \cong \mathcal{R}', (\mathcal{R}, b) \cong (\mathcal{R}', b')$	Congruence
$o(\mathcal{R}), o(\mathcal{R}, b)$	Current Sequents
$O(\mathcal{R}), O(\mathcal{R}, b)$	Current Nodes

The operations on sequents also apply to evaluations and animations.

B.6 Equivalence

The following table gives the notation for transitional evaluation histories, transitional animation histories, the corresponding step functions, and the equivalences between relational and transitional systems:

ω	Syntax Map
$\mathcal{T}$	Transitional Evaluations
(c, m, e)	Transitional Evaluation States
$\mathcal{E}_T$	Transitional Evaluation Step
$\mathcal{S}_\rho$	Realisations of Relational Rules
$\mathcal{S}_T$	Tree path of $\mathcal{T}$
$\mathcal{T} \sim \mathcal{R}$	Evaluation Equivalence
$(\mathcal{S}, s), (\mathcal{S}, \cdot)$	Transitional Animations
$\mathcal{A}_T$	Transitional Animation Step
$(\mathcal{S}, s) \approx (\mathcal{R}, b)$	Animation Equivalence

The operations on sequents also apply to evaluations and animations.

B.7 Animation Operations

This table lists the notation used for the animation operations defined on relational animations.

$\mathcal{A}_{R,v}$	View-specific Step
$\mathcal{P}_R$	Phrase Evaluation
$\mathcal{C}_R$	Total Evaluation
$\overline{\mathcal{A}}_R$	Reverse Animation Step
$\overline{\mathcal{A}}_{R,v}$	Reverse View-specific Step
$\overline{\mathcal{P}}_R$	Reverse Phrase Evaluation
$\overline{\mathcal{C}}_R$	Reverse Total Evaluation

B.8 LSL Display Expressions

This table lists the operators that can be used in display rules in LSL. It omits the operators mentioned in Chapter 9 that aren't in LSL. It also omits the basic tokens such as string literals, the special variables *subject*, *result*, etc.

$\&i$	The i^{th} subtree of the current node.
$\$i$	The i^{th} premise of the current node or the i^{th} argument of a term.
$\$0$	The conclusion of the current node.
$\$\$$	The current node.
$\$n$	A newline character.
$\$s$	A space character.
$\$t$	A tab character.
$@i$	Equivalent to $\&i ?i \$i$.
$x ?i y$	x if the evaluation of the i^{th} premise has started, y otherwise.
$x ?!i y$	x if the i^{th} premise is being evaluated, y otherwise.
$x ??i y$	x if the evaluation of the i^{th} premise has finished, y otherwise.
$\#x$	x , highlighted if the focus is the subject of this sequent.
$\#\#x$	x , highlighted if the focus is the result of this sequent.
$\#!x$	x , highlighted if the focus is the subject or the result of this sequent.
$\{x\}$	x in a box.
$\langle x \rangle$	x – the brackets indicate grouping.
$x \text{ OVER } y$	x positioned above y , seperated by a line.
$x \text{ ABOVE } y$	x positioned above y , seperated by a space.

Appendix C

The Specification of Proc

This appendix gives an annotated specification of Proc, the language used for all the examples in this thesis. The specification is almost complete; each omission is almost exactly the same as a part that is included.

Proc is an expression-based language; each program is an expression that returns a value. A few constructs (**while** and **assign**) are evaluated for their side-effects and return the value **null**. Proc provides functions of one argument, while loops, conditional expressions, simple integer operations, and little else.

C.1 The Header

The header gives the name of the language and the names and types of the views of that language.

The **stack**, **env** and **memory** views of Proc are simple views of those types. The **topstack** view is a source view of the procedure on top of the stack. The **source** view is a source view of the whole animation tree. The **plain_semantics** view displays the relational semantic proof tree shorn of memories and environments, and the **full_semantics** view shows the entire proof tree.

LANGUAGE proc

VIEWS

stack: STACK

source, plain_semantics, full_semantics: SOURCE

topstack: TOPSTACK

env: ENV

memory: MEMORY

C.2 The LOOKUP_ENV Rule Set

This rule set defines the operation of looking up the value of an identifier in an environment. Environments are defined as an unordered list of bindings; the lookup operation uses linear search.

The specification first defines some variables to use in the rules. These have global scope, and so they are defined outside the rule set:

VARs

x, y, arg: PHRASE

e, e1: ENV

v, v1: VALUE

m, m0, m1: MEMORY

The first part of the rule set specifies the default display attribute for the rule set in each view.

BEGIN LOOKUP_ENV

VIEWS

source, topstack: HIDDEN

full_semantics plain_semantics: STATIC

The next part of the rule set defines the new constructors that it will use, with their types. These constructors have global scope, but the types given here are local to this rule set. Even built-in constructors such as `true` and `false` must be defined before they can be used.

CONSTRUCTORS

lookup_env (ENV, PHRASE): PHRASE

false true: VALUE

add_env (ENV, PHRASE, VALUE): ENV

equal (PHRASE, PHRASE): PHRASE

Finally the rule set includes the semantic rules. These are LSL equivalent of the following rules from the main text:

$$\frac{\frac{}{\vdash \text{lookup}(e\uparrow(x, v), x), m \Rightarrow v, m}}{\vdash \text{lookup}(e, x), m \Rightarrow v', m}}{\vdash \text{lookup}(e\uparrow(x', v), x), m \Rightarrow v', m} \quad \text{IDENT} \quad \vdash \text{eq}(x, x') \Rightarrow \text{ff}$$

Variables may not repeated in the lhs of an LSL rule, so both rules need side conditions:

RULES

Lookup_env1:

`(|- lookup_env (add_env (e, x, v), y), m ==> v, m)`

PROVIDED

`ID ( |- equal (x, y) ==> true)`

Lookup_env2:

`(|- lookup_env (e, y), m ==> v1, m1)``(|- lookup_env (add_env (e, x, v), y), m ==> v1, m1)`

PROVIDED

`ID ( |- equal (x, y) ==> false)`

END LOOKUP_ENV

The rule set could include display rules at this point, but all the display rules for Proc are given after all the rule sets.

C.3 The LOOKUP_MEMORY Rule Set

This rule set is similar to the LOOKUP_ENV rule set. The main difference is in the definition of the constructors and their types. Constructors that have already been defined are assigned types for this rule set in the TYPES section:

CONSTRUCTORS

`lookup_memory (MEMORY, PHRASE): PHRASE``add_memory (MEMORY, PHRASE, VALUE): MEMORY`

TYPES**false: VALUE****true: VALUE****equal (PHRASE, PHRASE): PHRASE**

The semantic rules are the obvious equivalents of those in the LOOKUP_ENV rule set. A better implementation would update values in place instead of extending the list with a new binding.

C.4 The PROC Rule Set

This is the main rule set of the specification. It defines the semantics of Proc expressions.

First the specification defines some more variables.

VARs**cond, arg1, arg2: PHRASE****v2: VALUE****m2, m3: MEMORY**

The first part of the rule set follow the same format as above, but define many more constructors. Most of these are obvious counterparts to the language constructs, **if**, **while**, **func**, **call**, etc. The **deref** constructor is a language construct that looks up the value of an identifier in the current memory. The **null_env** and **null_memory** constructors stand for the empty environment and empty memory. The **null** constructor is the value returned by side-effecting constructs (**while** and **assign**). The **id** and **int** constructors are built-in.

The `while` constructor takes two arguments; the four argument version used in the thesis is not needed here because this specification is not proven equivalent to a transitional semantics.

BEGIN PROC

CONSTRUCTORS

if (PHRASE, PHRASE, PHRASE): PHRASE

int: VALUE PHRASE

plus minus times divide modulus (PHRASE, PHRASE): PHRASE

less greater (PHRASE, PHRASE): PHRASE

func (PHRASE PHRASE PHRASE PHRASE): PHRASE

call (PHRASE PHRASE): PHRASE

closure (ENV PHRASE PHRASE PHRASE): PHRASE

while assign sequence (PHRASE PHRASE): PHRASE

deref (PHRASE PHRASE): PHRASE

null_env: ENV

null_memory: MEMORY

null: VALUE

id: PHRASE

TYPES

false: VALUE PHRASE

true: VALUE PHRASE

equal (PHRASE PHRASE): PHRASE

Since this is the main rule set, it must include a grammar and a default environment and memory. Proc includes both defaults.

DEFAULT ENV : null_env

DEFAULT MEMORY : null_memory

Proc uses the simple lexical analyser provided with The Animator Generator. Therefore the first part of the grammar is a list of keywords. This is the only way that the lexical analyser can be tailored to a language. Note that NULL can't be used as the symbol for a keyword because it would clash with the C identifier, and END can't be used as it would clash with the LSL identifier.

GRAMMAR

%keyword IF	"if"
%keyword THEN	"then"
%keyword ELSE	"else"
%keyword ENDIF	"endif"
%keyword TRUE	"true"
%keyword FALSE	"false"
%keyword FUNC	"func"
%keyword IN	"in"
%keyword End	"end"
%keyword CALL	"call"
%keyword WHILE	"while"
%keyword ENDWHILE	"endwhile"
%keyword DO	"do"
%keyword ASSIGN	"assign"
%keyword NUL	"null"

The parse rules for Proc are ambiguous. This ambiguity is resolved by giving precedences to some of the symbols, as in YACC [Joh78]. The last precedence rule binds tightest, so `assign x 1+1; 2` is parsed as `(assign x (1+1)); 2`.

```
%left ';'
%nonassoc ASSIGN
%nonassoc '='
%left '>' '<'
%left '+' '-'
%left '*' '/' '%'
```

The parse rules are like those in YACC. Each left hand side is a list of keywords, single characters and non-terminals. A successful parse by a certain rule returns the value following the colon, where $\$i$ means the value of the i^{th} item in that rule. Unlike YACC, no side-effecting actions may be made while a rule is being parsed.

```
env:          '(' ')' :      null_env
            |      env '+' '(' id ',' value ')' : add_env ($1, $4, $6)
            ;

value:        id :      $1
            |      int :      $1
            |      TRUE :      true
            |      FALSE :      false
            |      NUL :      null
            ;
```

```

phrase:      id :      $1
|
|      int :      $1
|
|      TRUE :      true
|
|      FALSE :      false
|
|      IF phrase THEN phrase ELSE phrase ENDIF :
|
|
|              if ($2, $4, $6)
|
|      IF phrase THEN phrase ENDIF : if ($2, $4, null)
|
|      phrase '+' phrase :      plus ($1, $3)
|
|      phrase '-' phrase :      minus ($1, $3)
|
|      phrase '*' phrase :      times ($1, $3)
|
|      phrase '/' phrase :      divide ($1, $3)
|
|      phrase '%' phrase :      modulus ($1, $3)
|
|      phrase '>' phrase :      greater ($1, $3)
|
|      phrase '<' phrase :      less ($1, $3)
|
|      phrase '=' phrase :      equal ($1, $3)
|
|      WHILE phrase DO phrase ENDWHILE :      while ($2, $4)
|
|      '!' id :      deref ($2)
|
|      ASSIGN id phrase :      assign ($2, $3)
|
|      phrase ';' phrase :      sequence ($1, $3)
|
|      FUNC id id '=' phrase IN phrase End :
|
|              func ($2, $3, $5, $7)
|
|      CALL '(' id phrase ')' : call ($3, $4)
|
|      '(' phrase ')' : $2
|
;

memory:      '(' ')' :      null_memory
|
|      memory '+' '(' id ',' value ')' :
|
|              add_memory ($1, $4, $6)
|
;

```

This rule set ends with the semantic rules. The only integer operation shown is `plus`; the others are similar.

Recursive rules are declared as such. This is optional, because The Animator Generator automatically assumes that a rule is recursive if the rhs of its conclusion is the same as that of its last premise, unless the rule is specified to be non-recursive. Including the declarations helps to spot typing mistakes.

RULES

Deref: RECURSIVE:

```
LOOKUP_MEMORY ( |- lookup_memory (m, x), m ==> v, m1)
```

```
----
```

```
(e |- deref (x), m ==> v, m1)
```

The `Var` rule shows how the `id` constructor is used. The arguments of the constructor is the identifier itself, but this is not an LSL term, and so it must always be used as an argument to this constructor. Two different identifiers can be matched by using different arguments.

Var: RECURSIVE:

```
LOOKUP_ENV ( |- lookup_env (e, id (arg)), m ==> v, m1)
```

```
----
```

```
(e |- id (arg), m ==> v, m1)
```

If_false: RECURSIVE:

```
(e |- cond, m0 ==> false, m1)
```

```
(e |- arg2, m1 ==> v, m2)
```

```
----
```

```
(e |- if (cond, arg1, arg2), m0 ==> v, m2)
```

If_true: RECURSIVE:

```
(e |- cond, m0 ==> true, m1)
```

```
(e |- arg1, m1 ==> v, m2)
```

```
----
```

```
(e |- if (cond, arg1, arg2), m0 ==> v, m2)
```

Plus: RECURSIVE:

```
(e |- arg1, m0 ==> v1, m1)
```

```
(e |- arg2, m1 ==> v2, m2)
```

```
INTEGER ( |- plus (v1, v2), m2 ==> v, m3)
```

```
----
```

```
(e |- plus (arg1, arg2), m0 ==> v, m3)
```

Sequence:

```
(e |- arg1, m0 ==> v1, m1)
```

```
(e |- arg2, m1 ==> v2, m2)
```

```
----
```

```
(e |- sequence (arg1, arg2), m0 ==> null, m2)
```

While2:

```
(e |- arg1, m0 ==> false, m1)
```

```
----
```

```
(e |- while (arg1, arg2), m0 ==> null, m1)
```

While1: RECURSIVE:

```
(e |- arg1, m0 ==> true, m1)
```

```
(e |- arg2, m1 ==> null, m2)
```

```
(e |- while (arg1, arg2), m2 ==> v, m3)
```

```
----
```

```
(e |- while (arg1, arg2), m0 ==> v, m3)
```

The **Assign** rule shows that the rhm of a conclusion may be an arbitrary expression.

Assign:

```
(e |- arg1, m0 ==> v1, m1)
----
(e |- assign (x, arg1), m0 ==> null, add_memory (m1, x, v1))
```

The **Func** rule evaluates the body of the function in the environment that binds the function name to a closure containing the function definition. Thus the function may be recursive.

Func: NON_RECURSIVE:

```
(add_env (e, x, closure (e, x, y, arg1)) |- arg2, m0 ==> v, m1)
----
(e |- func (x, y, arg1, arg2), m0 ==> v, m1)
```

The third premise of the **Call** rule is enclosed in square brackets to show that the current node must be added to the stack while this premise is evaluated.

Call: RECURSIVE:

```
(e |- arg2, m0 ==> v, m1)
LOOKUP_ENV ( |- lookup_env (e, x), m1 ==> closure (e1, x, y, arg1), m2)
[(add_env (add_env (e1, y, v), x, closure (e1, x, y, arg1)) |-
                                arg1, m2 ==> v1, m3)]
----
(e |- call (x, arg2), m0 ==> v1, m3)
```

END PROC

C.5 Constructor Display Rules

The rest of the specification consists of the display rules. First are the constructor display rules. A few examples are included here; the other rules are similar. If no display attribute is specified, it is assumed to be DYNAMIC.

DISPLAY CONSTRUCTORS

```
int id:
    BUILT_IN
;

deref:
    "! " $1
;

if:
    "if " $1 $n "then " $2 $n "else " $3 $n "endif"
;

false:
    "false"
;
```


The sequence constructor is hidden; the animator never stops at an instance of a rule describing this constructor.

sequence:

HIDDEN

\$1 ";" \$n \$2

;

lookup_env:

"lookup_env (" {\$1}, \$2 ")"

;

null_env null_memory:

""

;

add_env add_memory:

\$1 "(" \$2, \$3 ")" \$n

;

closure:

"closure (" {\$1}, \$2, \$3, {\$4} ")"

;

func:

STATIC

"func " \$1 \$s \$2 " =" \$n\$s\$s {\$3} \$n "in" \$n\$s\$s {\$4} \$n "end"

;

C.6 Semantic Display Rules

The default display rule for sequents is similar to the one described in Chapter 9. LSL doesn't allow default display rules for semantic rules.

DEFAULT SEQUENT:

```
-full_semantics->
  <{Env} |- {Subject}, {Lhm} ==> {Result}, {Rhm}>
-plain_semantics->
  < |- {Subject} ==> {Result}>
-source->
-topstack->
  Result ?? Subject
;
```

The source view for the Plus rule says that if the node has been evaluated, it displays the result; otherwise it displays the constructor with the arguments replaced with the corresponding sub-animations. If the focus is the subject of the conclusion or of the remote premise, the display is highlighted.

Plus:

```
-full_semantics->
-plain_semantics->
  $$ = @0 OVER <@1 " " @2 " " @3>
-source->
-topstack->
  $3 = #<plus (@1 @2)>
  $0 = #!<v ?? @3>
;
```

Sequence:

```
-full_semantics->
-plain_semantics->
  $$ = @0 OVER <@1 " " @2>
-source->
-topstack->
  $0 = #!<sequence (@1 @2)>
;
```

Assign:

```
-full_semantics->
-plain_semantics->
  $$ = @0 OVER @1
-source->
-topstack->
  $0 = #!<assign (x, @1)>
;
```

Var:

```
-full_semantics->
  $1 = <"lookup_env (" |- #<lookup_env (e arg)>, {m} ==> v , {m1}" ">
  $$ = @0 OVER @1
-plain_semantics->
  $1 = <"lookup_env (" |- #<lookup_env (e arg)> ==> v ">
  $$ = @0 OVER @1
-source->
-topstack->
  $0 = #!<v ?? arg>
;
```

Lookup_env1:

```

-full_semantics->
-plain_semantics->
  $$ = @0 OVER ""
-source->
-topstack->
  $0 = #!<v ?? <lookup_env (add_env (e, x, v), y)>>
;

Lookup_env2:
-full_semantics->
-plain_semantics->
  $$ = @0 OVER @1
-source->
-topstack->
  $0 = #!<v ?? <lookup_env (add_env (e, x, v), y)>>
;

```

The source view for the If... rules are slightly more complicated than the ones seen so far; if the second premise is being evaluated then that sub-animation is displayed. Also note how the second premise is displayed in the semantic views.

If_true:

-full_semantics->

\$2 = <{e} |- {arg1}, {m1} ==> v, {m2}> ??1 <{e} |- "???" ==> v>

\$\$ = @0 OVER <<@1 " " @2> ?1 "">

-plain_semantics->

\$2 = < |- {arg1} ==> v> ??1 <{e} |- "???" ==> v>

\$\$ = @0 OVER <<@1 " " @2> ?1 "">

-source->

-topstack->

\$2 = arg1

\$0 = #!<@2 ?!2 <v ?? <if (@1 @2 arg2)>> >

;

If_false:

-full_semantics->

\$2 = <{e} |- {arg2}, {m1} ==> v, {m2}> ??1 <{e} |- "???" ==> v>

\$\$ = @0 OVER <<@1 " " @2> ?1 "">

-plain_semantics->

\$2 = < |- {arg2} ==> v> ??1 <{e} |- "???" ==> v>

\$\$ = @0 OVER <<@1 " " @2> ?1 "">

-source->

-topstack->

\$2 = arg2

\$0 = #!<@2 ?!2 <v ?? <if (@1 arg1 @2)>> >

;

While1:

-full_semantics->

-plain_semantics->

\$\$ = @0 OVER <@1 " " @2 " " @3>

-source->

-topstack->

\$3 = #<while (@1 @2)>

\$0 = #!<v ?? @3>

;

While2:

-full_semantics->

-plain_semantics->

\$\$ = @0 OVER @1

-source->

-topstack->

\$0 = #!<null ?? <while (@1 arg2)> >

;

Func:

-full_semantics->

-plain_semantics->

\$\$ = @0 OVER @1

-source->

-topstack->

\$0 = #!<v ?? <func (x, y, arg1, @1)>>

;

Call:

```
-full_semantics->
  $2 = <"lookup (" |- #<lookup_env (e x)>, {m1} ==>
                                ##<closure (e1, x, y, arg1), {m2} ")">>
  $$ = @0 OVER <@1 "      " @2 "      " @3>
-plain_semantics->
  $2 = <"lookup (" |- #<lookup_env (e x)> ==>
                                ##<closure (e1, x, y, arg1) ")">>
  $$ = @0 OVER <@1 "      " @2 "      " @3>
-source->
-topstack->
  $2 = ##<call (x, v)> ?? #x
  $0 = #!<v1 ?? <@3 ?!3 <@2 ??2 call (@2, @1)>>>
-stack->
  $0 = #!<call (x, v)>
;
```

The value display rule is used when the program is a value and doesn't have to be evaluated.

VALUE:

```
-full_semantics->
-plain_semantics->
  $$ = @0 OVER ""
-source->
-topstack->
  $$ = $0
;
```

Bibliography

- [All86] C. M. Allwood. Novices on the computer: A review of the literature. *IJMMS*, 25:633–658, 1986.
- [Bae75] R. Baecker. Two systems which produce animated representations of the execution of computer programs. In *Papers of the Fifth Technical Symposium on Computer Science Education, SIGCSE Bulletin*, 7(1), pages 158–167, Feb 1975.
- [BBA76] A. Barr, M. Beard, and R. C. Atkinson. The computer as a tutorial laboratory: The Stanford BIP project. *IJMMS*, 8:567–596, 1976.
- [BCH⁺85] Getchen P. Brown, Richard T. Carling, Christopher F. Herot, David A. Kramlich, and Paul Souza. Program visualization: Graphical support for software development. *Computer*, 18(8):27–35, Aug 1985.
- [Ber] Dave Berry. The Animator Generator: User manual. Forthcoming.
- [BM83] P. Bayman and R. E. Mayer. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *CACM*, 26:677–679, Sep 1983.
- [BmL87] P. Borrás, D. Clément, and B. Lang. Basic principles of a pretty printing formalism. In *Generation of Interactive Programming*

Environments: ESPRIT project 348, second annual review report, Jan 1987.

- [BOM81] B. Du Boulay, T. O'Shea, and J. Monk. The black box inside the glass box: Presenting concepts to novices. *IJMMS*, 14:237-249, 1981.
- [BPB87] Alan Bundy, Helen Pain, and Paul Brna. Evaluating Prolog environments: A progress report. D.A.I. Research Paper 351, Dept. of Artificial Intelligence, University of Edinburgh, 1987.
- [Bra87] Laurette Bradley. A treatment of languages with stages of evaluation. In M. Main, A. Melton, M. Mishove, and D. Schmidt, editors, *Proceedings of the Third Workshop on Mathematical Foundations of Programming Language Semantics, LNCS 298*, pages 425-443, 1987.
- [Brn88] Paul Brna. Improving Prolog environments: A review of available tools. D.A.I. Research Paper 386, Dept. of Artificial Intelligence, University of Edinburgh, 1988.
- [Bro80] R. E. Brooks. Studying programmer behaviour experimentally: The problems of proper methodology. *CACM*, 23(4):207-213, 1980.
- [Bro83] R. Brooks. Towards a theory of the comprehension of computer programs. *IJMMS*, 18(6):543-554, June 1983.
- [BS81] Marc. H. Brown and Robert Sedgewick. Progress report: Brown University instructional computing laboratory. In *Papers of the 12th SIGCSE Technical Symposium on Computer Science Education, SIGCSE Bulletin, 10(1)*, pages 91-101, Feb 1981.

- [BS85] Marc. H. Brown and Robert Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28-39, 1985.
- [BS86a] Rolf Bahlke and Gregor Snelting. Context sensitive editing with PSG environments. In *Workshop on Advanced Programming Environments*. Springer-Verlag, LNCS 244, June 1986.
- [BS86b] Rolf Bahlke and Gregor Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM TOPLAS*, 8(4):547-576, Oct 1986.
- [BSM87] Rolf Bahlke, Gregor Snelting, and Bernhard Moritz. A generator for language-specific debugging systems. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, *SIGPLAN Notices* 22(7), pages 92-101, July 1987.
- [BWLN84] Graham Birtwistle, Brian Wyvill, Danny Levinson, and Radford Neal. Visualising a simulation using animated pictures. In *Proceedings of the Conference on Simulation in Strongly Typed Languages*, pages 57-61. SCS, 1984.
- [Byr80] L. Byrd. Understanding the control flow of Prolog programs. In S. Tarnlund, editor, *Proceedings of the Logic Programming Workshop*, 1980.
- [Cec84] David R. Cecil. *Debugging BASIC Programs*. TAB Books, 1984.
- [CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.
- [CR83] Lori A. Clark and Debra J. Richardson. The application of error-sensitive testing strategies to debugging. In *ACM*

SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging, SIGPLAN Notices 18(8), pages 45–52, 1983.

- [Dam] Mads Dam. Personal communication.
- [DC86] A. D. Dewar and J. G. Cleary. Graphical display of complex information within a Prolog debugger. *IJMMS*, 25(5):503–522, Nov 1986.
- [Des84] T. Despeyroux. Executable specification of static semantics. In *Semantics of Data Types*. Springer Verlag, LNCS 173, 1984.
- [Des86] J. Despeyroux. Proof of translation in natural semantics. In *Proceedings of the First ACM Conference on Logic in Computer Science*, 1986.
- [Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, Aug 1975.
- [DJ86] Mads Dam and Frank Jensen. Compiler generation from relational semantics. In *European Symposium on Programming*, pages 1–30. Springer-Verlag, LNCS 213, 1986.
- [DM83] Sarah A. Douglas and Thomas P. Moran. Learning text editor semantics by analogy. In *Proceedings of CHI '83*, pages 207–211. ACM, 1983.
- [dS] Fabio da Silva. Personal communication.
- [EB86] M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine TPM. Tech. Report 21, Human Cognition Research Lab., Open University, 1986.

- [Eis79] M. Eisenstadt. A friendly software environment for psychology students. *AISB Quarterly*, 34, 1979.
- [Fai79] R. Fairley. ALLADIN: Assembly language driven debugging interpreter. *IEEE Transactions on Software Engineering*, 5(4):426–428, Jul 1979.
- [GKS83] S. C. Getz, G. Kalligiannis, and S. R. Schach. A very high level interactive graphical trace for the Pascal heap. *IEEE Transactions on Software Engineering*, 9(2):179–185, 1983.
- [Gou75] J. D. Gould. Some psychological evidence on how people debug computer programs. *IJMMS*, 7:151–182, 1975.
- [Gri70] R. Grishman. The debugging system AIDS. In *Proceedings of the Spring Joint Computer Conference*, vol. 36, pages 59–64. AFIPS, 1970.
- [Gro75] J. F. Gross. Video augmented computer science. *SIGCSE Bulletin*, 7(4):47–49, Dec 1975.
- [Heu86] V. P. Heuring. The automatic generation of fast lexical analysers. *Software Practice and Experience*, 16(9):801–808, Sep 1986.
- [HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings of the Second Annual Symposium on Logic In Computer Science*, pages 194–204. IEEE, 1987.
- [Hir84] Stuart H. Hirshfield. Program synthesis as a tool for teaching programming. *SIGCSE Bulletin*, 16(2):4–6, Jun 1984.
- [HK85] J. Heering and P. Klint. Towards monolingual programming environments. *ACM TOPLAS*, 7(2):183–213, 1985.

- [HK86] J. Heering and P. Klint. A syntax definition formalism. Report CS-R8633, Centre for Mathematics and Computer Science, Amsterdam, 1986.
- [HL87] R. Nigel Horspool and Michael R. Levy. *Mkscan* – an interactive scanner generator. *Software Practice and Experience*, 17(6):369–379, June 1987.
- [Hue84] Jon Hueras. *Macintosh Pascal Users' Guide*. Apple Computer Ltd., Cupertino CA, 1984.
- [Joh78] S. C. Johnson. Yacc: Yet another compiler compiler. UNIX Programmer's Manual Vol. 2B, 1978.
- [Kah82] Hank Kahney. An in-depth study of the cognitive behaviour of novice programmers. Technical Report 5, Human Cognition Research Lab., Open University, 1982.
- [KB84] D. E. Kieras and S. Bornir. The role of a mental model in learning to operate a device. *Cognitive Science*, 8(3):255–273, 1984.
- [Kep82] G. Keppel. *Design and Analysis: A Researcher's Handbook*. Prentice-Hall, 1982.
- [Ker81] Brian W. Kernighan. PIC – a language for typesetting graphics. In *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, *SIGPLAN Notices* 16(6), Jun 1981.
- [KNPS88] K. Koskimies, O. Nurmi, J. Paakki, and S. Sippu. The design of a language processor generator. *Software Practice and Experience*, 18(2):107–136, Feb 1988.

- [Knu84] Donald E. Knuth. *The T_EX book*. Addison-Wesley, Reading, Massachusetts, 1984.
- [Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.
- [Lan84] Laurent Langlois. Simulation visualization with SIMSEA, a general purpose animation language. In *Proceedings of the Conference on Simulation in Strongly Typed Languages*, pages 62–67. SCS, 1984.
- [Lau79] S. Lauesen. Debugging techniques. *Software Practice and Experience*, 9(1), Jan 1979.
- [Lea84] Thomas R. Leap. Animations of computers as teaching aids. In *Papers of the 15th SIGCSE Technical Symposium on Computer Science Education, SIGCSE Bulletin*, 16(1), pages 84–90, Feb 1984.
- [Les75] M. E. Lesk. LEX – a lexical analyser generator. Computer Science Technical Report 39, Bell Labs, 1975.
- [Lie84] H. Lieberman. Towards better debugging tools for LISP. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 247–255, 1984.
- [Luo89] Z. Luo. Ecc, an extended calculus of constructions. In *Proceedings of the Fourth Annual Symposium on Logic In Computer Science*, pages 385–395. IEEE, 1989.
- [May81] R. E. Mayer. The psychology of how novices learn computer programming. *Computing Surveys*, 13(1):121–141, 1981.
- [mDD⁺85] D. Clément, J. Despeyroux, T. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. INRIA Research Report RR 416, INRIA Sophia-Antipolis, June 1985.

- [Mey87] Bertrand Meyer. Cépage: Towards computer-aided design of software. Technical Report TR-CE-7/GP, Interactive Software Engineering, Inc., 1987.
- [Mil76] Robin Milner. Program semantics and mechanised proof. In *Foundations of Computer Science II*, pages 3–44. Math. Centre Amsterdam Tracts 82, 1976.
- [Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [Mit] Kevin Mitchell. Personal communication.
- [MLP79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *CACM*, 22(5):271–280, May 1979.
- [Moh88] T. G. Moher. PROVIDE: A process visualisation and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, June 1988.
- [Mon84] Douglas C. Montgomery. *Design and Analysis of Experiments (Second Edition)*. Wiley, 1984.
- [MS76] T. G. McDaniel and E. J. Schweppe. An interactive turing/post/mixed machine simulator. *SIGCSE Bulletin*, 8(2):36–40, June 1976.
- [MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT, 1990.
- [MW85] Donald Mackenzie and Judy Wajcman. *The Social Shaping of Technology: How the Refrigerator Got Its Hum*. OUP, 1985.

- [Mye80] Brad A. Myers. Displaying data structures for interactive debugging. Technical Report CSL-80-7, Xerox Palo Alto Research Center, Jun 1980.
- [Nea87] Lisa Rubin Neal. Cognition-sensitive design and user modelling for syntax-directed editors. In *Proceedings of CHI+GI '87*, pages 99-102. ACM, 1987.
- [Nor82] D. A. Norman. Some observations on mental models. In A. Stevens, editor, *Mental Models*, pages 7-19. Erlbaum, 1982.
- [NS72] A. Newell and H. A. Simon. *Human Problem Solving*. Prentice-Hall, 1972.
- [Pan81] H. Pangratz. COMPI - an instructive model for elementary computer science education. *SIGCSE Bulletin*, 13(2):12-16, June 1981.
- [Par81] David Park. A predicate transformer for weak fair iteration. Theory of Computation Report 36, Dept. of Computer Science, University of Warwick, May 1981.
- [PB87] H. Pain and A. Bundy. What stories should we tell novice Prolog programmers? In R. Hawley, editor, *Artificial Intelligence Programming Environments*. Ellis Horwood, 1987.
- [Per86] L. M. Pereira. Rational debugging in logic programming. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming, Springer-Verlag LNCS 225*, 1986.
- [PF88] J. Patrick and L. FitzGibbon. Structural displays as learning aids. *IJMMMS*, 28:625-635, 1988.

- [Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.
- [Plu88] D. Plummer. Coda: An extended debugger for prolog. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 496-511, 1988.
- [PN81] B. Plattner and J. Nievergelt. Monitoring program execution: A survey. *Computer*, 14(11):76-93, Nov 1981.
- [Raj86] Tim Rajan. APT: A principled design for an animated view of program execution for novice programmers. Tech. Report 19, Human Cognition Research Laboratory, Open University, Dec 1986.
- [Rei87] S. P. Reiss. Visual languages and the GARDEN system. In P. Gorny and M. J. Tauber, editors, *Visualization in Programming*, pages 178-198. Springer-Verlag, LNCS 282, 1987.
- [RT84] Thomas Reps and Tim Teitelbaum. The Synthesizer Generator. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 42-48, Apr 1984.
- [SBE83] E. Soloway, J. Bonar, and K. Ehrlich. Cognitive strategies and looping constructs: An empirical study. *CACM*, 26:853-860, 1983.
- [SEBG82] E. Soloway, K. Ehrlich, J. Bonar, and J. Greenspan. What do novices know about programming? In B. Shneiderman and A. Badre, editors, *Directions in Human Computer Interactions*. Ablex, Norwood N.J., 1982.

- [Sha82] E. Y. Shapiro. *Algorithmic Program Diagnosis*. ACM, 1982.
- [Shn80] B. Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop, 1980.
- [SS86] J. C. Spohrer and E. Soloway. Alternatives to construct based programming misconceptions. In *Proceedings CHI' 86*. ACM, 1986.
- [ST83] R. Seidner and N. Tindall. Interactive debug requirements. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging, SIGPLAN Notices 18(8)*, 1983.
- [Sun] Sun Microsystems. *Debugging Tools for the Sun Workstation*.
- [TAK82] K. Takahashi, T. Ase, and M. Kobayashi. Visual aid for fortran program debugging. In *Proceedings of the Sixth International Conference on Software Engineering*, pages 414–415, 1982.
- [TdB86] J. Taylor and J. B. H. du Boulay. Why novices may find learning Prolog hard. In J. Rutkowska and C. Crook, editors, *The Child and the Computer: Issues for Developmental Psychology*. Wiley, 1986.
- [Tei78] Warren Teitelman. The InterLisp reference manual. Technical report, Xerox Palo Alto Research Center, Dec 1978.
- [TM81] Warren Teitelman and Larry Masinter. The InterLisp programming environment. *Computer*, 14(4):25–34, April 1981.
- [Tra79] M. Tratner. A fundamental approach to debugging. *Software Practice and Experience*, 9:97–99, 1979.
- [TSP83] R. Tischler, R. Schauffer, and C. Payne. Static analysis of programs as an aid to debugging. In *ACM SIGSOFT/SIGPLAN*

Software Engineering Symposium on High Level Debugging, SIGPLAN Notices 18(8), pages 155–158, 1983.

- [Ves85] Iris Vessey. Expertise in debugging computer programs: A process analysis. *IJMMS*, 23(5):459–494, Nov 1985.
- [War86] Robert Ward. *Debugging C*. Que Corp., Indiana, 1986.
- [Wei81] M. Weiser. Program slicing. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 439–449, 1981.
- [Wei82] M. Weiser. Programmers use slices when debugging. *CACM*, 25(7):446–452, 1982.
- [WH84] P. H. Winston and B. K. P. Horn. *LISP*. Addison-Wesley, 1984.
- [Wie86] Susan Wiedenbeck. Beacons in computer program comprehension. *IJMMS*, 25:697–709, 1986.
- [WmAL⁺87] Pierre Weis, María-Virginia Apoule, Alain Laville, Michel Mauny, and Ascánder Suárez. *The CAML Reference Manual version 2.5*. Project Formel, INRIA-ENS, 1987.
- [Zel73] M. V. Zelkowitz. Reversible execution. *CACM*, 16(9):566, 1973.

Index of Lemmas and Propositions

Lemma 5.1.....	81
Lemma 5.2.....	85
Lemma 6.1.....	100
Lemma 6.2.....	106
Lemma 6.3.....	107
Lemma 7.1 (The Trivial Lemma).....	118
Lemma 7.2.....	125
Lemma 7.3.....	131
Lemma 7.4.....	136
Proposition 7.5.....	139
Lemma 8.1.....	156
Lemma 8.2.....	161
Lemma 8.3.....	161
Lemma A.1.....	303
Lemma A.2.....	304
Lemma A.3.....	308
Lemma A.4.....	308
Lemma A.5.....	313
Lemma A.6.....	313
Proposition A.7.....	314

Index

- abstract syntax 52, 53, 110, 286
- advance organisers 12
- AIDS 207
- animation
 - augmented relational 215
 - complete relational 156
 - complete transitional 303
 - equivalence 310
 - initial relational 156
 - initial transitional 303
 - relational 151
 - transitional 302
- animation history
 - relational 151
 - transitional 302
- animation step 146, 151, 279
 - equivalence 313
 - relational 158
 - reverse 211
 - transitional 305
- APT 182
- attribute grammars 285, 287
- augmented relational animation 215
- automatic animation 197, 205, 208, 252, 261
- axiom 57
 - user-defined 215
- BALSA 17, 26
- basic step 197, 204, 258, 260
- boxes 168, 172, 248, 264, 274
- break
 - condition 206
 - level 208
 - point 40, 205, 265
- Brna, Paul 19, 211
- built-in operations 246, 264, 283
- C++ 243
- call 60
- call graph 288
 - dynamic 186, 198, 275
 - static 171
- CAML 18
- Centaur 30, 167, 285
- Cépage 167
- change
 - environment 282
 - memory 214, 261, 263

- result 214, 261, 263
- closed term 52, 53
- CODA 219, 222, 238, 268
- command buttons 250
- compilers 5, 287
- complete
 - relational animation 156
 - relational evaluation 100
 - transitional animation 303
 - transitional evaluation 81
- composition of substitutions 59
- computation step 33
 - equivalence 136
 - relational 87, 103
 - transitional 67, 83
- conclusion 45, 55, 56
- condition
 - break 206
 - side 57, 58, 62, 216
- configuration 54
- congruence 105, 106, 107, 160
- consistent
 - sequence 80
 - stack 79
- constructor 52
 - display rule 167, 178, 247
 - dynamic 199
 - hidden 199
 - static 199
 - type of 245
 - variable 57
- core animation functions 242, 255
- corresponding node 98
- current
 - environment 189
 - memory 190, 216
 - node 98, 257
 - sequent 96, 153, 257
- DBXTool 166
- debugging 15, 16, 19
- default
 - display attribute 247
 - display rule 171, 195, 247, 264, 278
 - environment 246
 - memory 246
- definitional interpreter 162, 273
- denotational semantics 39, 162, 273, 287
- describe 198
- determinacy restrictions 70, 75, 77, 90, 101
- Dewlap 18, 22, 187
- differing terms 301
- display attribute 94, 199, 202, 213, 222, 275
 - default 247
- display rule 167

- constructor 167, 178, 247
 - default 171, 195, 247, 264, 278
 - parameters of 170, 184
 - semantic 178, 247
 - value 248
- Dock 166
- dynamic
- call graph 186, 198, 275
 - constructor 199
 - rule set 213
 - semantics 5, 285
 - view 166, 178
- editors 242, 265, 283
- rule 288
 - structure 5, 287
- env 56, 57, 79, 100, 303
- environment 54
- changing 282
 - current 189
 - view 166, 176, 189, 247
- equivalent
- animation steps 313
 - animations 310
 - computation steps 136
 - evaluations 129
 - specifications 61, 110, 270, 298, 311
 - syntax 110
- evaluation 67, 87
- complete relational 100
 - complete transitional 81
 - equivalence 129
 - initial relational 100
 - initial transitional 81
 - model of 67, 87, 88, 89, 129, 280
 - relational 97
 - transitional 78, 81
- evaluation history
- relational 87, 95
 - transitional 67, 78
- evaluation state
- relational 55
 - transitional 55
- expansive 122, 125, 138
- expression 57
- extension
- of relational animation 157, 201, 210
 - of relational evaluation 102, 103
 - of transitional animation 305
 - of transitional evaluation 83
- focus
- of relational animation 155, 199, 202, 206, 216
 - of relational evaluation 147
 - of transitional animation 303
- Fortran 13
- free variables 58

- Garden 17, 278
- generality 59
- granularity 22, 188
- Graphtrace 17, 191, 278
- hidden
 - constructor 199
 - rule set 213
- highlighting 3, 178, 180, 183, 187, 202, 265
- HLP84 285
- identifier 55
- implementer 9
- Incense 192
- inference tree 31, 34, 35, 95, 152, 192
- initial
 - relational animation 156
 - relational evaluation 100
 - transitional animation 303
 - transitional evaluation 81
- instance 58, 59, 121, 122
- InterLisp 171, 205
- interrupt 208, 251, 261
- InterViews 255
- I/O 283
- keywords 246
- language
 - definer 9, 283
 - layout 172, 192, 274
 - object 32, 272
 - specification 32, 272
- layout language 172, 192, 274
- lazy evaluation 268
- learning 12, 24, 30, 211
- lexical
 - analysis 246
 - proximity 21, 167, 169
 - view 167, 179, 188
- lhm 56, 57, 79, 100, 303
- lhs 56, 57, 79, 100, 303
- Lieberman, Henry 23, 209, 211
- Lisp 171, 205
- local premise 61
- Logo 13
- lst 56, 57, 79, 100, 303
- Macintosh Pascal 166
- marks 206, 211, 251, 252, 261, 265
- Masterscope 205
- matched premise 98, 153
- matching
 - strong 60, 69, 79, 99, 104, 120, 125, 131, 137, 154, 304, 309
 - weak 60, 70, 123
- memory 54
 - changing 214, 261, 263
 - current 190, 216
 - view 166, 190, 247
- mental model 4, 12, 15, 281

- menu buttons 250
- metavariable 57
- Milner, Robin 47, 74, 76
- minimal substitution 59
- minimal extension
 - of relational animation 201, 210
 - of relational animations 157
 - of transitional animation 305
- ML 93, 220, 269, 272, 280
- model
 - mental 4, 12, 15, 281
 - of evaluation 67, 87, 88, 89, 129, 280
- multiple views 16, 202
- natural semantics 93
- non-determinism 270
- non-expansive 122, 125, 137
- notional machine 12
- object 53
- object language 32, 272
- operational semantics 31
- optimisation 286
- order
 - of evaluation 46, 90, 96, 268
 - of sequents 98
- outline view 169, 171, 198, 264
- parse rules 245
- partially consistent
 - sequence 121
 - stack 120
- Pascal 170, 174, 184, 189, 191
- phrase 53
- phrase evaluation 209, 210, 212, 251, 260
- PIC 172, 274
- PL/1 13
- Plotkin, Gordon 33, 47, 49, 73, 162
- premise 45, 55, 56, 70
 - local 61
 - matched 98, 153
 - remote 61
 - trivial 98
 - unmatched 98, 153
- Proc 3, 9, 31, 37, 151, 170, 193, 221, 244, 246, 256, 268, 278, 281, 324
- program 9
 - relational 55
 - schemas 20, 25
 - transitional 55
- program-specific view 17, 191, 278
- Prolog 13, 16, 18, 19, 38, 39, 47, 187, 205, 211, 213, 214, 219, 238, 268, 273
- Provide 192
- proximity 20
 - lexical 21, 167, 169

- referential 173, 185
 - structural 21, 186, 187
 - temporal 21, 185
- psychology 11, 12, 14, 19, 25, 281
- PV 17, 189, 192, 279
- realisation 110, 113, 126
- recursive 90
- referential proximity 21, 173, 185
- relational
- animation 151
 - animation history 151
 - animation step 158
 - computation step 87, 103
 - evaluation 97
 - evaluation history 87, 95
 - evaluation state 55
 - program 55
 - rule 88
 - semantics 47, 61, 87
 - sequent 55
 - specification 61
- remote
- premise 61
 - rule set 60
- result 56, 57, 79, 100, 303
- changing 214, 261, 263
- reverse
- animation step 211, 260, 283
 - evaluation 211, 251, 260, 283
 - phrase evaluation 212, 251, 260
 - total evaluation 213, 251, 261
 - view-specific step 212, 251, 260
- rhm 56, 57, 79, 100, 303
- rhs 56, 57, 79, 100, 303
- rst 56, 57, 79, 100, 303
- rule
- editors 288
 - form 57, 67
 - relational 88
 - semantic 33, 45
 - transitional 67
- rule set 60, 76
- dynamic 213
 - graph 68, 89, 131, 139, 280
 - hidden 213
 - static 213
- schemas 20, 25
- semantic
- display rule 178, 247
 - rule 33, 45
 - view 166, 192
- semantics 25, 30, 268, 281
- denotational 39, 162, 273, 287
 - dynamic 5, 285
 - natural 93
 - operational 31
 - relational 47, 61, 87
 - static 5, 285

- structured operational 33, 43
- transitional 48, 61, 66
- sequence (consistent) 80
 - partially consistent 121
- sequent
 - current 96, 257
 - relational 55
 - transitional 56
- Shapiro, Ehud 283
- side condition 57, 58, 62, 216
- single-threaded 54
- slice 16, 173, 185, 264, 288
- SMC machine 47, 49, 162
- source view 166, 247
- specification
 - equivalence 61, 110, 270, 298, 311
 - language 32, 272
 - relational 61
 - transitional 61
- SPY 205
- stack 184, 187, 188, 262
 - consistent 79
 - partially consistent 120
 - view 187, 247, 262
- Standard ML 93, 220, 269, 272, 280
- state
 - left 56
 - relational evaluation 55
 - right 56
 - transitional evaluation 55
- static
 - call graph 171
 - constructor 199
 - rule set 213
 - semantics 5, 285
 - view 166
- step
 - animation 146, 151, 279
 - basic 197, 204, 258, 260
 - computation 33, 136
 - equivalence 313
 - relational animation 158
 - relational computation 87, 103
 - reverse animation 211
 - transitional animation 305
 - transitional computation 67, 83
 - view-specific 198, 210, 212, 251, 260
- strong matching 60, 69, 79, 99, 104, 120, 125, 131, 137, 154, 304, 309
- structural proximity 21, 186, 187
- structure editors 5, 287
- structured operational semantics 33, 43
- sub-animation 156
- sub-evaluation 100

- subject 56, 57, 79, 100, 303
- substitution 58, 110, 113
 - composition of 59
 - minimal 59
- SUCH-THAT clause 269
- symbolic
 - debuggers 189, 215
 - evaluation 284
- syntax
 - constraints 285
 - map 110
- Synthesizer Generator 167
- system 9
- temporal
 - proximity 21, 185
 - view 185, 188, 286
- term 52
 - closed 52, 53
 - user-defined 215
- T_EX 168
- theorem provers 288
- tool 9
- topstack view 247, 280
- total evaluation 210, 213, 251, 261
- trace 40, 185
- transitional
 - animation 302
 - animation step 305
 - computation step 67, 83
 - evaluation 78, 81
 - evaluation state 55
 - evaluation history 67, 78
 - program 55
 - rule 67
 - semantics 48, 61, 66
 - sequent 56
 - specification 61
 - tree 119, 123
- tree path 123, 304
- trivial premise 98
- type
 - of constructor 245
 - of variable 55, 57, 245
 - of view 247, 286
- type-specific view 264
- Typol 30, 39
- undo 214, 251, 261, 263
- unification 59, 272
- UNIX 255
- unmatched premise 98, 153
- user 9
- user-defined
 - axioms 215
 - terms 215
- user interface 242, 265
- validity 97, 98, 103, 147, 154, 302
- value 53
 - display rule 248

- environment 54
- phrase 53, 94
- variable 52
 - free 58
 - meta 57
 - type of 55, 57, 245
- view 261, 264
 - dynamic 166, 178
 - environment 166, 176, 189, 247
 - lexical 167, 179, 188
 - memory 166, 190, 247
 - multiple 16, 202
 - of procedures 182
 - outline 169, 171, 198, 264
 - program-specific 17, 191, 278
 - semantic 166, 192
 - slice 16, 173, 185, 264, 288
 - source 166, 247
 - stack 187, 247, 262
 - static 166
 - temporal 185, 188, 286
 - topstack 247, 280
 - type of 247, 286
 - type-specific 264
- view set 200, 202
- view-specific step 198, 210, 212, 251, 260
- WHERE clause 195, 247, 264, 277
- while loops 71, 74
- X window system 255
- YACC 29, 243, 331
- ZSTEP 148, 151, 166, 182
- Ward, Robert 15, 20
- weak matching 60, 70, 123