Precise and Expressive Mode Systems for Typed Logic Programming Languages

David Overton

Submitted in total fulfilment of the requirements of
the degree of Doctor of Philosophy

December 2003

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria 3010, Australia

Abstract

In this thesis we look at mode analysis of logic programs. Being based on the mathematical formalism of predicate logic, logic programs have no *a priori* notion of data flow — a single logic program may run in multiple *modes* where each mode describes, or prescribes, a pattern of data flow.

A *mode system* provides an abstract domain for describing the flow of data in logic programs, and an algorithm for analysing programs to *infer* the modes of a program or to *check* the correctness of *mode declarations* given by the programmer. Such an analysis can provide much useful information to the compiler for optimising the program. In a prescriptive mode system, mode analysis is also an important part of the semantic analysis phase of compilation (much like type analysis) and can inform the programmer of many errors or potential errors in the program at compile time. We therefore believe it is an essential component of any industrial strength logic programming system.

Our aim is to develop a strong and prescriptive mode system that is both as precise and expressive as possible. We believe this requires a strongly typed and purely declarative language and so we focus on the language Mercury.

The first contribution of our work is to give a detailed description of Mercury's existing mode system, which is based on abstract interpretation. Although most of this system has been around for several years, this is the first time it has been described in this level of detail. This is also the first time the relationship of the mode system to the formalism of abstract interpretation has been made clear.

Following that, we look at ways of extending the mode system to provide further precision and expressiveness, and to overcome some of the limitations of the current system.

The first of these extensions is to support a form of constrained parametric polymorphism for modes. This is analogous to constrained parametric polymorphic type systems such as type classes, and adds a somewhat similar degree of expressiveness to the mode system.

Next we look at a method for increasing the precision of the mode analysis by keeping track of aliases between variables. The increased precision we gain from this allows an increase in expressiveness by allowing the use of partially instantiated data structures and more complex uniqueness annotations on modes.

The final area we look at is an alternative approach to mode analysis using Boolean constraints. This allows us to design a mode system that can capture complex mode constraints between variables and more clearly separates the various tasks required for mode analysis. We believe that this constraint-based system provides a good platform for further extension of the Mercury mode system.

The work we describe has all been implemented in the Melbourne Mercury compiler, although only constrained parametric polymorphism has so far become part of an official compiler release.

Declaration

This is to certify that

(i) the thesis comprises only my original work towards the PhD except where indicated in the Preface,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

David Overton
December 2003

Preface

This thesis comprises 7 chapters, including an introduction and conclusion. Following the introduction, Chapter 2 provides the background and notation necessary to understand the rest of the thesis. Chapter 3 presents the mode analysis system as it is currently implemented in the Melbourne Mercury compiler. Chapter 4 presents extensions to the mode system to allow mode polymorphism. Chapter 5 describes an extension to the mode system to keep track of definite aliases. Chapter 6 presents a new approach to mode analysis using Boolean constraints. Finally, Chapter 7 contains some concluding remarks.

The mode system described in Chapter 3 was designed and implemented by Fergus Henderson and others, however the notation for the formalisation of the system which is presented in this thesis is entirely my own work. Section 5.3 is based on research part of which was carried out jointly with Andrew Bromage. It has not previously been published. Section 5.4 is based on part of Ross, Overton, and Somogyi [117]. Chapter 6 is based on Overton, Somogyi, and Stuckey [111], however most of the material describing the implementation is new.

Acknowledgements

This research has been made possible by the financial support of the Commonwealth of Australia in the form of an Australian Postgraduate Award.

I would like to thank my supervisor, Zoltan Somogyi, and the other members of my advisory committee, Lee Naish and Harald Søndergaard, for their advice and support throughout my PhD candidature. Thank you also to Andrew Bromage, Peter Ross and Peter Stuckey with whom I have collaborated on various components of the research presented here. Thank you to Peter Schachte for providing the ROBDD package which I used for implementing the work of Chapter 6. Thank you to my very good friend Tom Conway, without whose encouragement I never would have got involved in the Mercury project or started a PhD (don't worry, Tom, I've forgiven you). Thank you to Fergus Henderson whose extremely thorough code reviews helped greatly to improve both this research and its implementation. To the rest of the Mercury team, Ralph Becket, Mark Brown, Simon Taylor, David Jeffery and Tyson Dowd, it has been great working with all of you. I have learnt a great deal about logic programming, language design and software engineering in my time in the Mercury office.

Much of the writing of this thesis was carried out while I was employed by the HAL project at Monash University. I would like to thank María García de la Banda and Kim Marriott at Monash, as well as Peter Stuckey at The University of Melbourne, for their generosity in giving me time to work on my thesis, without which it would never have been finished, and for providing such an enjoyable, stimulating and friendly work environment.

I would like to thank my family for their support. Thank you to my mother for providing a happy home environment, regular meals, and a roof over my head for a large proportion of my candidature. Most especially, I would like to thank my wife, Moana, for her constant love and encouragement, for believing in my ability to finish this thesis — even when I didn't believe it myself, for continuing to support me even when my finishing date kept moving, and for the sacrifices she has made to enable me to get the work done. I'm looking forward to spending many thesis-free weekends with her in the future.

DMO
Melbourne, June 2003

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The idea of using predicate logic as the basis for a programming methodology was introduced by Kowalski [79] in 1974. One of the major advantages he promoted for programming in logic was the ability clearly to separate the concept of *what* a program does from *how* it does it. This notion was captured in his now famous quote "algorithm = logic + control" [80]. The logic component determines the meaning of the algorithm whereas the control component determines the strategy used for solving the problem. The control component only affects the efficiency of the solution, not what solution is computed. He argued that a clear separation of these two components would lead to software that is more often correct, more reliable and more maintainable. In other words, logic programming should form an ideal programming paradigm for achieving the goals of software engineering.

The separation of logic and control also facilitates the possibility of the control component being automatically handled by the system. The system may modify the control component in order to improve efficiency while leaving the logic component unchanged, thus guaranteeing that the modified program still solves the same problem.

Another advantage of logic programming is that a single predicate may be used to solve more than one problem. For example, a predicate that concatenates two lists may also be used to split a list into two. The logic component of the predicate specifies the relationship between the arguments of the predicate while the control component determines which arguments are input and which are output and thus determines whether the predicate concatenates two lists or splits a list. Each of these different behaviours is called a *mode* of the predicate.

Unfortunately, traditional logic programming languages, such as Prolog, have often found it challenging living up to the ideals of programming in logic. Most versions of Prolog have a fixed control strategy (left-to-right selection of literals and depth-first search) which can make it hard to write programs in a purely logical way that can execute efficiently. It is particularly hard to write a program that will execute efficiently and be guaranteed to terminate if it is intended to be used in multiple modes. The depth-first search strategy can lead to incompleteness where the predicate fails to terminate when called in some modes and it may not be possible to write the predicate in a logical way that is guaranteed to terminate in all modes of interest. For this reason, Prolog has non-logical features, such as the cut and predicates for inspecting the instantiation states of variables. These features allow the programmer to alter some aspects of the control component of the program. However, such features can destroy the pure logical semantics of the program and therefore make it harder to prove its correctness, harder for the maintainer to understand, and harder for the compiler to analyse for the purpose of optimisation.

Most Prolog implementations take other shortcuts to gain acceptable efficiency. For example, they will usually omit the occur check from the unification procedure which can lead to unsoundness. They also do not check whether negation as failure is used only in ways where it is guaranteed to be sound.

Mode analysis systems analyse the modes of a logic program and the data flow within each mode. The information they produce can be used to alleviate many of these problems and enable logic programs to execute more efficiently without sacrificing their declarative semantics. For example, a mode system may be able to determine when it is safe to omit the occur check.

Mode systems fall into two broad categories. They are either *descriptive* or *prescriptive*.[1] Descriptive systems analyse the program as-is and usually operate over a small finite abstract domain approximating the possible instantiation states of variables. These domains usually include a "don't know" value in order to cope with cases where the mode system does not have enough precision to describe the instantiation state more accurately. Such mode systems do not remove any expressiveness from a program because they describe the program as-is and accept any valid Prolog program. However, because of their limited precision, they cannot always guarantee soundness and efficient execution.

A prescriptive mode system, on the other hand, will attempt to re-order the program to make it conform to the mode system's idea of *mode correctness*. It may also reject programs that it cannot prove to be mode correct. As a result, a prescriptive mode system must either sacrifice expressiveness of the language or else use a much more precise analysis domain than a descriptive system. Generally, absolute precision is not possible, and any particular prescriptive mode system will need to balance its requirements for expressiveness against the amount of precision it is able to provide while keeping the analysis time reasonable. Prescriptive mode systems usually require the programmer to provide *mode declarations* for some or all predicates which specify the modes in which the predicates are intended to run. The mode analyser will check that all mode declarations are correct.

Prescriptive mode systems can be further classified into *strong* mode systems and *weak* mode systems. Strong prescriptive mode systems generally cannot tolerate having a "don't know" value in the domain and will reject any program for which they cannot more precisely categorise its instantiation states. Weaker mode systems may be more tolerant of uncertainty in instantiation states, but will use the information they have to do re-ordering and will still reject programs that don't conform to their mode declarations.

Somogyi [128, 129][2] claimed that, in order to provide reliability, robustness and efficiency, a strong prescriptive mode system was essential for any "real world", industrial strength, logic programming language. Moreover, he argued that such a mode system can only attain the precision required to be sufficiently expressive "if it has precise information about the possible structures of terms, and that this information is exactly what is provided by a strong type system." [129, pp. 2–3]

[1] We discuss these categories in more detail and give examples in Section 2.4.
[2] See also Somogyi, Henderson, Conway, and O'Keefe [132].

Many of Somogyi's ideas have been realised in the strongly typed, strongly moded logic programming language Mercury [66, 131]. Mercury's mode system provides an extremely precise abstract domain for describing instantiation states of variables. However, the implementation of the mode analysis algorithm in the Melbourne Mercury compiler does not yet (as of version 0.10.1) allow the full potential of this precision to be utilised. The problem is that the mode system does not keep track of sufficient information about the relationships between the instantiation states of different variables. One consequence of this loss of precision is that it is not possible to make use of *partially instantiated data structures* (i.e. data structures with some "holes" left to be filled in later in the program) in any meaningful way. The expressiveness of Mercury's *unique modes* [65], which allow modelling of destructive update and provide hints for compile time garbage collection, also suffers from this lack of precision.

In this thesis, we propose a number of enhancements to the mode system in order to alleviate some of this lack of expressiveness by improving the precision of the analysis. The remainder of this thesis is organised as follows. In Chapter 2 we introduce the notations and concepts we will need throughout the rest of the thesis. This includes a more detailed introduction to mode systems and logic programming, and an overview of the Mercury language.

In Chapter 3 we present an in-depth description of the mode system of Mercury 0.10.1. This mode system was developed mostly by Fergus Henderson, with smaller contributions from other members of the Mercury team, including the author of this thesis. However, this is the first time it has been described in this level of detail and formality, aside from the implementation itself. This chapter provides essential information for understanding the enhancements proposed in the rest of the thesis. It also clarifies the relationship between the Mercury mode system and the formalism of abstract interpretation.

In Chapter 4 we present an extension of the mode system to provide a form of constrained parametric polymorphism in mode declarations. This allows, for example, for polymorphically typed predicates to have polymorphic instantiation states associated with each type variable. This is particularly useful when subtype information, which can be conveyed through the instantiation state, needs to be propagated from input arguments to output arguments. One important use of this is when the type variables are instantiated with higher-order types. These require higher-order mode information to be available in order for them to be useful (e.g. so that the higher-order object can be called). This extension has been implemented in the Melbourne Mercury compiler and has been part of the official release since version 0.11.0.

In Chapter 5 we describe another extension to the mode system to track aliases between variables (and subterms) within the body of a predicate. This provides an increase in the precision of the analysis which allows the use of partially instantiated data structures. It also improves the expressiveness of the unique modes system by allowing code where unique objects are nested inside other unique objects. This extension has been implemented in the Mercury compiler, but has not yet become part of an official Mercury release, mostly due to concerns over the added analysis time it requires.

In Chapter 6 we present an alternative approach to mode analysis. We use Boolean constraints to express the relationships between the instantiation states of variables in different parts of the predicate body. This approach makes it easier to separate the different conceptual phases of mode analysis. We believe that this provides a more appropriate platform for the further extension of the Mercury mode system. An experimental prototype of this analysis has been implemented within the Melbourne Mercury compiler.

Finally, in Chapter 7 we present some concluding remarks.

Chapter 2

Background

In this chapter, we cover the basic concepts that will be needed to understand the rest of the thesis, and also look at previous work on mode analysis in logic programming languages.

Section 2.1 briefly covers the notation we will use for the mathematical concepts we will require. Section 2.2 introduces logic programming. Section 2.3 introduces abstract interpretation. Section 2.4 introduces the concept of mode analysis in logic programming and also looks at previous work in that area. Section 2.5 gives an introduction to the Mercury programming language.

2.1 Fundamental Concepts

We first cover the notation we will use for the basic mathematical concepts we require throughout the rest of the thesis. For more information on these topics, there are many good text books, such as Arbib, Kfoury, and Moll [6], Davey and Priestley [46], Halmos [61]. Schachte [119] also has very clear and concise definitions of many of the concepts we need. Many of the definitions below are based on definitions found in that work.

We make use of the logical connectives $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies), $\Leftrightarrow$ (if and only if) and $\neg$ (not), and the quantifiers $\forall$ (for all) and $\exists$ (there exists). We define these more formally later.

2.1.1 Mathematical Preliminaries

Sets

A *set* is a (possibly infinite) collection of objects. We write $x \in S$ to denote that the object x is a *member* of the set S; similarly $x \notin S$ means that x is not a member of S (a slash through a symbol will generally indicate the negation of the meaning of that symbol). The symbol $\emptyset$ denotes the *empty set*.

A set can be defined by listing its members, enclosed in curly brackets: $S = \{ x_1, \ldots, x_n \}$, which defines S to be the set containing the elements $x_1, \ldots, x_n$; or by using a *set comprehension* of the form $S = \{ x \mid p(x) \}$ which defines S to be the set containing all elements x such that property $p(x)$ holds. We also write $\{ x \in S' \mid p(x) \}$ as a shorthand for $\{ x \mid x \in S' \wedge p(x) \}$.

The *cardinality* of a set S, denoted $|S|$, gives an indication of the size of the set. If S is finite, $|S|$ is the number of elements in S. In this thesis we do not need to deal with infinite sets and therefore we don't need to worry about their cardinality.

For two sets S_1 and S_2:

- $S_1 \cup S_2 = \{\, x \,|\, x \in S_1 \ \vee \ x \in S_2 \,\}$ is the *union* of S_1 and S_2;
- $S_1 \cap S_2 = \{\, x \,|\, x \in S_1 \ \wedge \ x \in S_2 \,\}$ is the *intersection* of S_1 and S_2; and
- $S_1 \setminus S_2 = \{\, x \,|\, x \in S_1 \ \wedge \ x \notin S_2 \,\}$; is the *set difference* of S_1 and S_2.

If every member of S_1 is also a member of S_2 we say that S_1 is a *subset* of S_2 and write $S_1 \subseteq S_2$. We write $\mathcal{P}\,S$ to denote the set of all possible subsets of S, that is, $\mathcal{P}\,S = \{\, S' \,|\, S' \subseteq S \,\}$. We call $\mathcal{P}\,S$ the *power set* of S.

If S is a set of sets, then $\bigcup S$ is the union of all the sets in S and $\bigcap S$ is the intersection of all the sets in S. We also use

$$\bigodot_{p(x)} x = \bigodot \{\, x \,|\, p(x) \,\}$$

and

$$\bigodot_{i=m}^{n} x = \bigodot_{i \in \{\, m, m+1, \ldots, n \,\}} x$$

where $\bigodot$ is any operator (such as $\bigcup$ or $\bigcap$).

Example 2.1.1. For any set S: $\bigcup \mathcal{P}\,S = S$ and $\bigcap \mathcal{P}\,S = \emptyset$. ◁

Tuples

A *tuple* is an ordered finite sequence of objects which we write enclosed in angle brackets: $\langle x_1, \ldots, x_n \rangle$. The number of elements n in a tuple is known as its *arity*. A tuple with n elements is an n-ary tuple, or n-tuple for short. A particularly important kind of tuple is the 2-tuple which we call a *binary* tuple or a *pair*. We use the notation $\overline{x}$ to refer to a tuple $\langle x_1, \ldots, x_n \rangle$ of arbitrary length n. We will also sometimes treat the tuple $\langle x_1, \ldots, x_n \rangle$ as though it were the set $\{\, x_1, \ldots, x_n \,\}$.

For sets S_1 and S_2, we define $S_1 \times S_2 = \{\, \langle x_1, x_2 \rangle \,|\, x_1 \in S_1 \ \wedge \ x_2 \in S_2 \,\}$ which we call the *Cartesian product* of S_1 and S_2.

Relations

A *relation* R is a set of tuples which all have the same arity. An n-ary relation is a set consisting of n-tuples. For an n-ary relation R, we use the notation $R(x_1, \ldots, x_n)$ as short-hand for $\langle x_1, \ldots, x_n \rangle \in R$. If R is a binary relation then we usually write this using infix notation: $x_1 \; R \; x_2$.

For an n-ary relation R, if $R \subseteq S_1 \times \cdots \times S_n$ then we say that $S_1 \times \cdots \times S_n$ is a *signature* for R. We will usually write this as $R : S_1 \times \cdots \times S_n$. If $S = S_1 = \cdots = S_n$ then we say that R is an n-ary relation *on* S.

Example 2.1.2. The binary relation $\leq$ on the natural numbers $\mathbb{N}$ has the signature $\leq : \mathbb{N} \times \mathbb{N}$. ◁

A binary relation R on S is

- *symmetric* iff $\forall x, y \in S.\ x\ R\ y \iff y\ R\ x$;
- *antisymmetric* iff $\forall x, y \in S.\ x\ R\ y\ \wedge\ y\ R\ x \iff x = y$;
- *reflexive* iff $\forall x \in S.\ x\ R\ x$;
- *transitive* iff $\forall x, y, z \in S.\ x\ R\ y\ \wedge y\ R\ z \iff x\ R\ z$.

(Here, and elsewhere throughout the thesis we use "iff" as an abbreviation for "if and only if".)

The *transitive closure* $\mathsf{trans}^*(R)$ of a binary relation R is the least set R' such that $R \subseteq R'$ and R' is transitive.

Partial Order Relations

A binary relation that is reflexive, antisymmetric, and transitive is called a *partial order* relation. We often use symbols such as $\leq$, $\preceq$ and $\sqsubseteq$ for partial order relations.

If $\sqsubseteq$ is a partial order relation on a set S then the pair $\langle S, \sqsubseteq \rangle$ is the set S *equipped with* $\sqsubseteq$. This is called a *partially ordered set* or *poset* for short.

If $x, y \in S$ and $\langle S, \sqsubseteq \rangle$ is a poset then if either $x \sqsubseteq y$ or $y \sqsubseteq x$ then we say that x and y are *comparable*; otherwise they are *incomparable*. If every pair of elements in S is comparable then we say that $\sqsubseteq$ is a *total order* relation on S.

If $\langle S, \sqsubseteq \rangle$ is a poset and $T \subseteq S$ then $x \in S$ is an *upper bound* of T if $\forall y \in T.\ y \sqsubseteq x$. If for every upper bound x' of T it holds that $x \sqsubseteq x'$ then we say that x is the *least upper bound* (*lub*) of T. Similarly, if $\forall y \in T.\ x \sqsubseteq y$ then x is a *lower bound* of T and if for every lower bound x' of T it holds that $x' \sqsubseteq x$ then x is the *greatest lower bound* (*glb*) of T. We write the lub and glb, respectively, of T as $\bigsqcup T$ and $\bigsqcap T$. If $T = \{ y_1, y_2 \}$ then we can write $y_1 \sqcup y_2 = \bigsqcup T$ and $y_1 \sqcap y_2 = \bigsqcap T$.

Lattices

If $\langle S, \sqsubseteq \rangle$ is a poset and for every pair of elements $x_1, x_2 \in S$ both $x_1 \sqcup x_2$ and $x_1 \sqcap x_2$ exist, then $\langle S, \sqsubseteq \rangle$ is a *lattice*. If $\bigsqcup T$ and $\bigsqcap T$ exist for every (possibly infinite) subset $T \subseteq S$, then $\langle S, \sqsubseteq \rangle$ is a *complete lattice*. By definition, for every complete lattice $\langle S, \sqsubseteq \rangle$ both $\bigsqcup S$ and $\bigsqcap S$ must exist. We denote them by $\top$ (pronounced *top*) and $\bot$ (pronounced *bottom*), respectively.

Example 2.1.3. The subset relation $\subseteq$ is a partial order relation, and for any set S, the poset $\langle \mathcal{P}\, S, \subseteq \rangle$ is a complete lattice with the least upper bound operator being $\bigcup$, the greatest lower bound operator being $\bigcap$, $\top = S$, and $\bot = \emptyset$. ◁

It is convenient to visualise posets and lattices using a *Hasse diagram*. In a Hasse diagram all the elements in the set to be represented are arranged as nodes of a graph such that for any pair of comparable elements, the greater element (in the partial order) is higher in the diagram than the lesser element, and there is a path in the graph between them.

Example 2.1.4. A Hasse diagram for the complete lattice $\langle \mathcal{P} \{ 0, 1, 2 \},\ \subseteq \rangle$ is shown in Figure 2.1 on the following page. Note that from the diagram it is clear that $\top = \{ 0, 1, 2 \}$ and $\bot = \emptyset$. ◁

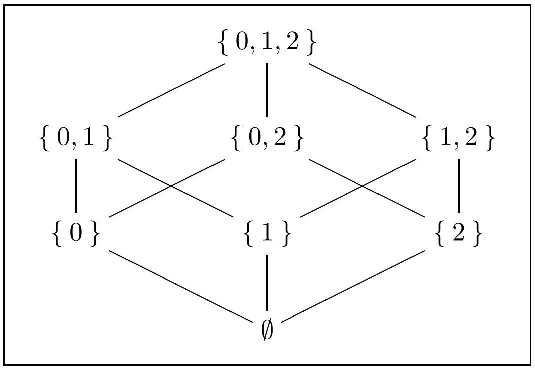

Figure 2.1: Example of a Hasse diagram

Functions

Another important kind of relation is the *function*. A relation $F : S_1 \times S_2$ is a *function* (or *mapping*) from S_1 to S_2 if $\forall x \in S_1.\ x\ F\ y_1\ \wedge\ x\ F\ y_2\ \Rightarrow\ y_1 = y_2$. To denote that F is a function we write the signature for F as $F : S_1 \rightarrow S_2$. We generally use the notation $x \mapsto y$ rather than $\langle x, y \rangle$ to denote a member of a function. The notation $y = F(x)$ is equivalent to $(x \mapsto y) \in F$ and we say that y is the result of the *application* of F to x.

For a function $F : S_1 \rightarrow S_2$ we say that the *domain* of F, written $\operatorname{dom} F$, is $\{\, x \mid \exists y.\ y = F(x) \,\}$. If $\operatorname{dom} F = S_1$ then we say that F is a *total function*; otherwise F is a *partial function* which is *undefined* for values in $S_1 \setminus \operatorname{dom} F$.

We will often define functions (and relations) using *pattern matching*. For example

$$\begin{aligned} \mathsf{fac}(0) &= 1 \\ \mathsf{fac}(n) &= n \cdot \mathsf{fac}(n-1) \end{aligned}$$

defines the factorial function and is equivalent to

$$\mathsf{fac}(n) = \text{if } (n = 0) \text{ then } 1 \text{ else } n \cdot \mathsf{fac}(n-1)$$

We will sometimes define functions using the notation of the *lambda calculus* [24, 25]: $F = \lambda x.\ e$ where x is a *lambda quantified* variable and e is an expression (usually containing x). This definition is equivalent to $F = \{\, y \mapsto z \mid z = e[x/y] \,\}$ where $e[x/y]$ means the expression e with x replaced by y anywhere it occurs. For example, an alternative definition of the factorial function might be

$$\mathsf{fac} = \lambda n.\ \text{if } (n = 0) \text{ then } 1 \text{ else } 1 \cdot \ldots \cdot n$$

A useful function is the *fixed-point combinator*:

$$\operatorname{fix} f = f(\operatorname{fix} f)$$

which takes a function f as its argument. The fixed-point combinator allows us to give yet another definition for factorial, one that does not require a recursive application of fac:

$$\mathsf{fac} = \mathrm{fix}(\lambda f.\ \lambda n.\ \text{if } (n = 0) \text{ then } 1 \text{ else } n \cdot f(n-1))$$

We will use the fixed-point combinator to allow us to define infinite terms. For example, if ':' is the list constructor then $\mathrm{fix}(\lambda f.\ 1 : f)$ is an infinite list of 1s.

2.1.2 Logic

Formal mathematical logic is the basis of logic programming and, indeed, can be used as a basis for all of mathematics.

Following Reeves and Clarke [114], we make a distinction between *object languages* and *meta languages*. An object language is a language we are studying such as the logic programming language Mercury, or the language of propositional calculus. A meta language is a language we use to describe the rules of the object language and the algorithms we use to analyse it.

We will use the language of mathematical logic for both our object languages and our meta language. To avoid confusion, we will often use different notation in the meta language to what we use in the object language. Such differences are noted in the following.

We give here a very brief overview of the concepts and notations of propositional and predicate logic and refer the reader to a text book, such as Reeves and Clarke [114], for further information.

Propositional Logic

The first, and simplest, type of logic we will look at is *propositional* or *Boolean* logic [14, 15]. Propositional logic is a mathematical system based on the set $\mathsf{Bool} = \{0, 1\}$ where we usually take 0 to mean *false* and 1 to mean *true*.

Sentences in propositional logic are constructed using the *logical connectives* $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ and $\neg$, which we have already been using informally.[1] We now define them more formally using the *truth table* in Table 2.1.

p	q	conjunction $p \wedge q$	disjunction $p \vee q$	implication $p \rightarrow q$	equivalence $p \leftrightarrow q$	negation $\neg p$
0	0	0	0	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	0	0
1	1	1	1	1	1	0

Table 2.1: Truth table for the connectives of propositional logic

Boolean Valuations and Constraints

We assume a set of *Boolean variables* BVar. A *Boolean valuation* is a mapping from Boolean variables to values in the domain Bool, i.e. $\mathcal{B} : \mathsf{BVal}$ where $\mathsf{BVal} = \mathsf{BVar} \rightarrow \mathsf{Bool}$. Given $\mathcal{B} \in \mathsf{BVal}$,

[1] Previously we have used $\Rightarrow$ and $\Leftrightarrow$ instead of $\rightarrow$ and $\leftrightarrow$. We will tend to use the former notation in our meta language and the latter in our object languages.

$x \in \mathsf{BVar}$ and $b \in \mathsf{Bool}$, we define

$$\mathcal{B}[b/x] = \lambda y.\ \text{if } (y = x) \text{ then } b \text{ else } \mathcal{B}(y)$$

A *Boolean constraint* (or *Boolean function*) $\mathcal{C} : \mathsf{BConstr}$ where $\mathsf{BConstr} = \mathsf{BVal} \to \mathsf{Bool}$ is a function which constrains the possible values of a set of Boolean variables $\mathsf{vars}(\mathcal{C}) \subseteq \mathsf{BVar}$. We require that $\forall \mathcal{B} \in \operatorname{dom} \mathcal{C}.\ \mathsf{vars}(\mathcal{C}) \subseteq \operatorname{dom} \mathcal{B}$. If $\mathcal{C}(\mathcal{B}) = 1$ for some $\mathcal{B} \in \mathsf{BVal}$ and $\mathcal{C} \in \mathsf{BConstr}$ then we say that $\mathcal{B}$ is a *model* of $\mathcal{C}$ which we write as $\mathcal{B} \models \mathcal{C}$.

If $\forall \mathcal{B} \in \mathsf{BVal}.\ \mathcal{B} \models \mathcal{C}$ then we say that $\mathcal{C}$ is *valid*. If $\forall \mathcal{B} \in \mathsf{BVal}.\ \mathcal{B} \not\models \mathcal{C}$ then we say that $\mathcal{C}$ is not *satisfiable*.

We overload the logical connectives by lifting them to the domain $\mathsf{BConstr}$ as defined below:

$$\begin{aligned}
\mathcal{C}_1 \wedge \mathcal{C}_2 &= \lambda \mathcal{B}.\ \mathcal{C}_1(\mathcal{B}) \wedge \mathcal{C}_2(\mathcal{B}) \\
\mathcal{C}_1 \vee \mathcal{C}_2 &= \lambda \mathcal{B}.\ \mathcal{C}_1(\mathcal{B}) \vee \mathcal{C}_2(\mathcal{B}) \\
\mathcal{C}_1 \to \mathcal{C}_2 &= \lambda \mathcal{B}.\ \mathcal{C}_1(\mathcal{B}) \to \mathcal{C}_2(\mathcal{B}) \\
\mathcal{C}_1 \leftrightarrow \mathcal{C}_2 &= \lambda \mathcal{B}.\ \mathcal{C}_1(\mathcal{B}) \leftrightarrow \mathcal{C}_2(\mathcal{B}) \\
\neg \mathcal{C} &= \lambda \mathcal{B}.\ \neg \mathcal{C}(\mathcal{B})
\end{aligned}$$

If a Boolean variable $x \in \mathsf{BVar}$ occurs in a context where we were expecting a Boolean constraint then we take it to mean the constraint $\lambda \mathcal{B}.\ \mathcal{B}(x)$. We also lift 0 and 1 to $\lambda \mathcal{B}.\ 0$ and $\lambda \mathcal{B}.\ 1$, respectively. That is, 0 represents the unsatisfiable constraint and 1 represents the valid constraint.

We define the *restriction* or "existential quantification" operation $\exists x.\ \mathcal{C}$ where $x \in \mathsf{BVar}$ and $\mathcal{C} \in \mathsf{BConstr}$ as $\exists x.\ \mathcal{C} = \lambda \mathcal{B}.\ \mathcal{C}(\mathcal{B}[0/x]) \ \vee\ \mathcal{C}(\mathcal{B}[1/x])$. Intuitively, we use the restriction $\exists x.\ \mathcal{C}$ when we don't care about what value of x is required to make $\mathcal{C}$ true. We also define restriction for a set of variables: $\exists \{ x_1, \ldots, x_n \}.\ \mathcal{C} = \exists x_1.\ \ldots \exists x_n.\ \mathcal{C}$.

Clauses and Resolution

A *Boolean formula* is an expression consisting of Boolean variables and the logical connectives. A Boolean formula can be used to define a Boolean function. Two Boolean formulas are *equivalent* iff they define the same Boolean function.

A *literal* is a Boolean formula which is either a single variable, e.g. x, or a negated variable, e.g. $\neg x$. We call x a *positive literal* whereas $\neg x$ is a *negative literal*.

A *clause* is a disjunction $\mathcal{L}_n \vee \cdots \vee \mathcal{L}_n$ where each $\mathcal{L}_i$ is a literal. Any Boolean formula can be rewritten as an equivalent formula which is a conjunction $\mathcal{K}_1 \wedge \cdots \wedge \mathcal{K}_n$ where each $\mathcal{K}_i$ is a clause. A Boolean formula in this form is said to be in *conjunctive normal form*.

A clause with at most one positive literal is called a *Horn clause* [70]. A Horn clause with exactly one positive literal is called a *definite clause*. A definite clause $x \vee \neg y_1 \vee \cdots \vee \neg y_n$ is often written in the equivalent form $x \leftarrow y_1 \wedge \cdots \wedge y_n$ where $\leftarrow$ is reverse implication (i.e. $x \leftarrow y \ \Leftrightarrow\ y \to x$). The literal x is known as the *head* of the clause and $y_1 \wedge \cdots \wedge y_n$ is the *body* of the clause. As an extension of this notation we will often write the clause x as $x \leftarrow$, and the clause $\neg y_1 \vee \cdots \vee \neg y_n$ as $\leftarrow y_1 \wedge \cdots \wedge y_n$. The *empty clause*, written $\leftarrow$, represents the Boolean function 0.

In our meta language, we will sometimes write the clause $x \Leftarrow y_1 \wedge \cdots \wedge y_n$ in the form

$$\frac{\begin{array}{c} y_1 \\ \vdots \\ y_n \end{array}}{x} \quad \text{or equivalently} \quad \frac{y_i\Big|_{i=1}^{n}}{x}$$

The problem of determining whether a given Boolean formula is satisfiable is known as the *propositional satisfiability problem*, or *SAT* for short. If the problem is restricted to formulas in clausal form where each clause can have at most two literals then we call the problem *2-SAT*. The general problem SAT is NP-complete, however the more restricted case 2-SAT can be solved in linear time.

One method of solving SAT is to do a proof by refutation, using the inference rule *resolution* [116]. The resolution rule says that if we have a set of clauses such that one clause contains a literal x and another contains a literal $\neg x$ then we can deduce a new clause which is the disjunction of the two clauses with the literals x and $\neg x$ removed. More formally:

$$\frac{\begin{array}{c} \mathcal{L}_1^1 \vee \cdots \vee \mathcal{L}_1^n \vee x \\ \mathcal{L}_2^1 \vee \cdots \vee \mathcal{L}_2^m \vee \neg x \end{array}}{\mathcal{L}_1^1 \vee \cdots \vee \mathcal{L}_1^n \vee \mathcal{L}_2^1 \vee \cdots \vee \mathcal{L}_2^m}$$

Proving that a Boolean formula $\mathcal{F}$ is satisfiable is equivalent to proving $\neg\mathcal{F}$ is not valid. We first convert $\neg\mathcal{F}$ into conjunctive normal form, and then, wherever possible, apply the resolution rule to add new clauses. If we add the empty clause $\leftarrow$ then we have proven that $\neg\mathcal{F}$ is not valid, and thus that $\mathcal{F}$ is satisfiable.

Predicate Logic

First order predicate logic is an extension of propositional logic where we use relations, or *predicates*, instead of propositions.

Assume we have a set of *logic variables* Var, a set of *predicate names* $\mathsf{PredName}$, and a set of *function symbols* (or *functors*) $\mathsf{FuncSym}$.

A *signature* Σ is a set of pairs f/n where $f \in \mathsf{FuncSym}$ and $n \geq 0$ is the integer *arity* of f. A function symbol with 0 arity is called a *constant*. Given a signature Σ, the set of all *ground terms* (also called the *Herbrand universe*), denoted $\tau(\Sigma)$, is defined as the least set satisfying:

$$\tau(\Sigma) = \bigcup_{f/n \in \Sigma} \{ f(t_1, \ldots, t_n) \mid \{ t_1, \ldots, t_n \} \subseteq \tau(\Sigma) \}.$$

For simplicity, we assume that Σ contains at least one constant.

Let $V \subseteq \mathsf{Var}$ be a set of variables. The set of all *terms* over Σ and V, denoted $\tau(\Sigma, V)$, is similarly defined as the least set satisfying:

$$\tau(\Sigma, V) = V \cup \bigcup_{f/n \in \Sigma} \{ f(t_1, \ldots, t_n) \mid \{ t_1, \ldots, t_n \} \subseteq \tau(\Sigma, V) \}$$

The set of *atomic formulas* or *atoms* over a function signature Σ, variable set V and predicate signature Π where each element of Π is a pair π/n, $\pi \in \mathsf{PredName}$ and $n \geq 0$, is defined by

$$\alpha(\Sigma, V, \Pi) = \{\, \pi(t_1, \ldots, t_n) \,|\, \pi/n \in \Pi \ \wedge \ \{\, t_1, \ldots, t_n \,\} \subseteq \tau(\Sigma, V) \,\}$$

In some of the following, we treat atoms as though they are terms.

A *substitution* over signature Σ and variable set V is a mapping from variables to terms in $\tau(\Sigma, V)$, written $\{\, x_1/t_1, \ldots, x_n/t_n \,\}$. We allow substitutions to be applied to terms as well as variables. If θ is a substitution and t is a term then $\theta(t)$ is the term such that any variable x occurring in t that is in $\operatorname{dom}\theta$ is replaced by $\theta(x)$.

A *unifier* for two terms t_1 and t_2 is a substitution θ such that $\theta(t_1)$ and $\theta(t_2)$ are syntactically identical. A *most general unifier* of two terms t_1 and t_2, denoted $\mathsf{mgu}(t_1, t_2)$, is a unifier θ which has the property that for every other unifier θ' of t_1 and t_2, there exists a substitution θ'' such that θ' is the composition of θ with θ''. A most general unifier of two terms can be computed using the *unification algorithm* which we do not give here. It is described in Lloyd [87] among other places.

Formulas in (first order) predicate logic are constructed from atoms, the logical connectives, and the universal and existential quantifiers $\forall$ and $\exists$:

For predicate logic, we define a literal to be either an atom or the negation of an atom. The clausal form we use is called *prenex normal* and is like conjunctive normal form except all quantifiers are at the front of the formula. The definitions of Horn clause and definite clause are then extended from their definitions in propositional logic in the obvious way.

We use a shorthand notation to avoid having to explicitly write quantifiers for Horn clauses. If the atoms of a Horn clause contain variables then we implicitly quantify the variables as follows. If $P \leftarrow Q_1 \wedge \cdots \wedge Q_n$ is a Horn clause in predicate logic then we say that it is implicitly equivalent to $\forall x_1.\ \ldots \forall x_n.\ P \leftarrow (\exists y_1.\ \ldots \exists y_m.\ Q_1 \wedge \cdots \wedge Q_n)$ where $x_1, \ldots, x_n$ are all the variables occurring in P and $y_1, \ldots, y_m$ are the variables that occur in $Q_1, \ldots, Q_n$ but not in P.

The resolution rule extended to predicate logic is

$$\frac{\begin{array}{c}\mathcal{L}_1 \vee \cdots \vee \mathcal{L}_n \vee \mathcal{A} \\ \mathcal{L}'_1 \vee \cdots \vee \mathcal{L}'_m \vee \neg\mathcal{A}' \\ \theta = \mathsf{mgu}(\mathcal{A}, \mathcal{A}')\end{array}}{\theta(\mathcal{L}'_1) \vee \cdots \vee \theta(\mathcal{L}'_n) \vee \theta(\mathcal{L}'_1) \vee \cdots \vee \theta(\mathcal{L}'_m)}$$

where $\mathcal{A}$ and $\mathcal{A}'$ are atoms and θ is their most general unifier (iff they are unifiable). This rule can be used for formulas in prenex normal form if we remove all existential quantifications using a process called *Skolemisation*. We are mainly interested in the specialised case of *SLD-resolution* which we will discuss below. The main thing to note here though is that resolution involves computing a substitution θ which we will find very useful when we look at using predicate logic as a programming language. We should also note that the satisfiability problem for predicate logic is, in general, undecidable.

2.2 Logic Programming

This section gives a very brief overview of logic programming. See Kowalski [79], Lloyd [87], van Emden and Kowalski [144] for more information.

2.2.1 Programming in Logic

Early research into unification and resolution in predicate logic [116] was mainly focused on automated theorem proving. Kowalski [79] realised that predicate logic could also be used for computation, that is, as the basis for a programming language. This involved using the subset of predicate logic consisting only of Horn clauses, plus a specialised resolution rule known as *SLD-resolution* [87].[2]

A *definite logic program* is a set of definite clauses, plus a clause consisting of only negative literals, known as the *query* or *goal* $\leftarrow Q_1 \wedge \cdots \wedge Q_n$. Execution of a logic program consists of applying the rule of SLD-resolution in order to attempt to refute the query. The result of a successful refutation is a substitution for the variables in the query for which its negation, i.e. $Q_1 \wedge \cdots \wedge Q_n$, is true. Thus, as well as *proving* a theorem, we have *computed* some useful information.

Logic programming gives us two different views of a clause $P \leftarrow Q_1 \wedge \cdots \wedge Q_n$.

1. P is true if $Q_1, \ldots, Q_n$ are true. This is the *declarative* view.
2. To execute P we must execute $Q_1, \ldots, Q_n$. This is the *operational* or *procedural* view.

The clause, then, resembles a procedure definition for P in a procedural programming language. However, a major advantage of logic programming is that the clause also has a well understood declarative semantics based on predicate logic.

2.2.2 Unification

Most logic programming languages contain a predicate $=/2$ which can be defined by the clause $x = x \leftarrow$ (where we use infix notation for the operator $=/2$). It can be seen that the effect of a body atom $t_1 = t_2$ is to unify the two terms t_1 and t_2. We generally refer to an atom $t_1 = t_2$ as a *unification* of t_1 and t_2 whereas an atom of the form $p(t_1, \ldots, t_n)$ is generally referred to as a *call* to the predicate p/n.

Unification is a fundamental part of logic programming and much effort has gone into optimising the unification algorithm. We note that the general unification algorithm can be quite expensive and that many of the logic programming analyses we will look at later try to find places in logic programs where the general algorithm can be replaced by a more specific algorithm for a particular subset of terms. A particularly expensive part of the algorithm is the *occur check* which involves checking that a variable to be unified with a term does not occur within that term (if it does, the unification should fail). This check is so expensive that many logic programming systems leave it out, for the pragmatic reason that it is virtually never needed. However leaving out the occur check can lead to unsoundness of the SLD-resolution so we would like to know when it is safe to leave it out and when it must be performed. This is one of the aims of mode analysis.

[2] SLD-resolution stands for SL-resolution for Definite clauses. SL stands for Linear resolution with Selection function.

2.2.3 Nondeterminism

Note that we can have multiple clauses with the same predicate symbol p/n in the head, i.e.

$$p(t_1, \ldots, t_n) \leftarrow Q_1 \wedge \cdots \wedge Q_i$$
$$p(t'_1, \ldots, t'_n) \leftarrow R_1 \wedge \cdots \wedge R_j$$

When trying to prove a goal $p(t''_1, \ldots, t''_n)$ the execution may try one clause first and, if it fails to prove the goal using that clause, may *backtrack* and try the other clause. A typical logic programming system will select the clauses in the order they appear in the program source code and use a *depth-first search strategy*. A predicate which has multiple clauses, or calls other predicates which have multiple clauses, may have more than one solution for any particular call. We say that such a predicate is *nondeterministic*.

2.2.4 Modes

Consider the clauses below which define a predicate append/3.

$$\text{append}(e,\ v,\ v) \leftarrow$$
$$\text{append}(w : x,\ y,\ w : z) \leftarrow \text{append}(x,\ y,\ z)$$

where e is a constant and : is a binary function symbol (for which we use infix notation) representing the list constructor. If we give a query $\leftarrow \text{append}(c : e,\ d : e,\ x)$ then we obtain the answer substitution $\{\, x/(c : (d : e)) \,\}$. We can see that the predicate append/3, when given two ground terms representing lists as its first two arguments, will "return" the concatenation of the two lists as its third argument. It is as though the first two arguments are "input" arguments and the third argument is "output". Now consider the query $\leftarrow \text{append}(x,\ y,\ c : (d : e))$. Due to the nondeterministic nature of this predicate definition, there are are several possible substitution sets that could be produced: $\{\, x/e,\ y/(c : (d : e)) \,\}$, $\{\, x/(c : e),\ y/(d : e) \,\}$, and $\{\, x/(c : (d : e)),\ y/e \,\}$. In this case the third argument is acting as an "input" and the first two arguments as "output". We say that append/3 can operate in different *modes*. In general, many more complex modes than just our "input" and "output" classifications are possible. The study of modes is the subject of this thesis.

2.2.5 Negation as Failure

Programming with definite clauses is not always convenient and we would like it to be possible for the body of the clause to contain more that just a conjunction of positive literals. In particular, we would like it to be possible for the body to contain negative literals. The most common way to achieve this is to use the concept of *negation as failure* [26] in which a negative literal $\neg P$ is considered true if it is not possible to prove P from the program. We use a modified resolution rule *SLDNF-resolution* (i.e. SLD-resolution with Negation as Failure). However, SLDNF-resolution is only sound if proving the negated literal does not cause any variables to be bound (i.e. does not cause any substitutions to be created) [103]. Many logic programming systems do not check this.

Negation as failure is not the only way of adding negation to logic programs. See Apt and Bol [5] for a survey of alternative approaches.

2.2.6 Prolog

The most widespread logic programming language is *Prolog* (*pro*gramming in *log*ic) for which there are now many implementations, text books [17, 29, 110, 134], and an ISO standard [51, 71]. Most modern versions of Prolog (and the ISO standard) use a syntax derived from DEC-10 (or Edinburgh) Prolog [154] and are implemented by compiling to some variant of an abstract machine known as the *Warren Abstract Machine* or *WAM* [2, 156].

Modern Prolog systems allow the body of a clause to be an arbitrary goal that can include disjunctions and if-the-else constructs as well as conjunctions and negations.

In the syntax of Prolog, the comma (‘`,`’) represents conjunction ($\wedge$), the semicolon (‘`;`’) represents disjunction ($\vee$), the operator ‘`:-`’ takes the place of $\leftarrow$ in separating the clause head from the body, each clause must be terminated with a full stop (‘`.`’), and variable names must start with a capital letter.

Example 2.2.1. The Prolog code for the predicate append/3, which we saw above, is

```
append([], V, V).
append([W | X], Y, [W | Z]) :- append(X, Y, Z).
```

Prolog uses the constant `[]` for the empty list and the binary function symbol `[ · | · ]` for list construction. Note how closely the Prolog code resembles the predicate logic clauses. $\triangleleft$

The language Mercury uses the syntax of Prolog with some extensions, e.g. to support functional and higher-order programming.

Prolog assumes a fixed execution order where conjunctions are executed from left to right and clauses are selected in the order they are given in the program source. Most modern Prolog implementations provide *first argument indexing*. This means that if the first argument in the head of each clause for a predicate has a different top-level function symbol then execution can jump immediately to the first matching clause when the predicate is called with the first argument bound to one of these function symbol. This can significantly improve execution times.

The Prolog language has some *nonlogical* features i.e. features for which there is no declarative semantics or where the operational semantics may be unsound with respect to the declarative semantics. Unfortunately, most programs it find necessary to use nonlogical features. For example, most programs need to use the *cut* for acceptable efficiency. Input/output (I/O) must also be done in a nonlogical way in Prolog.

2.3 Abstract Interpretation

Abstract interpretation [41, 42] is a formalised system providing a framework for the analysis of properties of programs. Abstract interpretation of logic programs has been studied in great depth, e.g. [18, 30, 34, 43, 74, 85, 89, 108, 119].

The idea behind abstract interpretation is to “mimic” the execution of a program using an abstraction of the semantics of the program. The abstraction of the semantics may involve a simple abstraction of the data values that variables may take, or it may be a more complex abstraction of the program state.

To formalise the notion of abstraction, assume we have some concrete property C of programs which we are interested in, and some abstraction A which approximates that property. We call C the *concrete domain* and A the *abstract domain*.

Assume we have two relations $\sqsubseteq_C$ and $\sqsubseteq_A$ which are partial orders on C and A, respectively, that formalise the relative precision in each domain. E.g. if $a_1, a_2 \in A$ and $a_1 \sqsubseteq_A a_2$ then a_1 is a more precise description than a_2. The posets $\langle C, \sqsubseteq_C\rangle$ and $\langle A, \sqsubseteq_A\rangle$ are often complete lattices, although this is not necessary.

The abstraction is defined by an *abstraction function* $\alpha : C \to A$ which maps elements of C to their most precise counterparts in A, and a *concretisation function* $\gamma : A \to C$ which maps elements of A back into elements of C and defines the semantics of the abstract domain. If $\forall x \in C.\ \forall y \in A.\ \alpha(x) \sqsubseteq_A y \Leftrightarrow x \sqsubseteq_C \gamma(y)$, then we say that $\langle \alpha, \gamma\rangle$ is a *Galois connection*, which we write

$$\langle C, \sqsubseteq_C\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq_A\rangle$$

Having a Galois connection gives us the guarantees that $\forall x \in C.\ x \sqsubseteq_C \gamma(\alpha(x))$ and $\forall y \in A.\ \alpha(\gamma(y)) \sqsubseteq_A y$, i.e. that abstracting and then concretising a member of C doesn't give us a more precise member of C (which would be unsound), and that concretising and then abstracting a member of A won't lose any precision (so the analysis is as precise as possible given the abstract domain).

Example 2.3.1. Consider the case where $C = \mathcal{P}\tau(\Sigma, V)$, the powerset of all terms over signature Σ and variable set V; $\sqsubseteq_C\ =\ \subseteq$, the subset ordering; $A = \{\bot, \mathsf{ground}, \mathsf{free}, \top\}$; $\sqsubseteq_A\ = \{\langle y, y'\rangle \in A^2 \mid y = \bot \lor y' = \top\}$; and the concretisation and abstraction functions are defined as

$$\begin{aligned} \gamma(\bot) &= \emptyset \\ \gamma(\mathsf{ground}) &= \tau(\Sigma) \\ \gamma(\mathsf{free}) &= V \\ \gamma(\top) &= \tau(\Sigma, V) \end{aligned} \qquad \alpha(T) = \begin{cases} \bot & \text{if } T = \emptyset; \\ \mathsf{ground} & \text{if } T \subseteq \tau(\Sigma); \\ \mathsf{free} & \text{if } T \subseteq V; \\ \top & \text{otherwise.} \end{cases}$$

In the abstract domain, $\bot$ represents an undefined value, e.g. after an exception or infinite loop; ground represents ground terms; free represents variables; and $\top$ represents "don't know" and includes all other terms. We can see that $\langle \alpha, \gamma\rangle$ forms a Galois connection. This domain can be used as the basis for a very simple mode analysis system. We will discuss this further in the following section. ◁

In an analysis based on abstract interpretation, abstractions of the operators of the language to be studied must also be provided. For logic programs, these might include abstractions of unification, conjunction, disjunction, and so on. If $F_C : C \to C$ is an operation in the language and $F_A : A \to A$ is an abstraction of that operation, then, for the abstraction to be sound, we require

$$\forall c \in C.\ F_C(c) \sqsubseteq_C \gamma(F_A(\alpha(c)))$$

We want to ensure that the abstract interpretation terminates in a finite and reasonable time. In general, when abstractly interpreting a recursively defined procedure, to ensure termination we need to ensure that F_A reaches a *fixpoint*, that is a value $a \in A$ such that $F_A(a) = a$, in a finite number of applications. If A is a finite set, $\langle A, \sqsubseteq_A \rangle$ is a complete lattice, and F_A is *monotonic* (i.e. $\forall a \in A.\ a \sqsubseteq_A F_A(a)$), then this is easy to ensure since $\top$ will be a fixpoint of F_A that is reachable in a finite number of applications of F_A starting at any $a \in A$. However, if $\langle A, \sqsubseteq_A \rangle$ has no $\top$ element, or if A is not finite (or even very large) then other approaches may be needed to ensure termination in a reasonable time. One such approach is to reduce the precision of the analysis by using a *widening* operation [41, 44].

2.4 Mode Analysis

In Section 2.2 we noted that one of the features of logic programs is that predicates can execute in multiple different modes. This allows a form of code re-use that is not available in other kinds of programming languages. However, the mechanisms required to provide this feature can be hard to implement efficiently in a sound way. Even if a predicate is only intended to be used in one mode, the multi-moded nature of logic programming can make efficient implementation hard. We have already noted the efficiency issues associated with a general unification algorithm as one example. Another example is having to deal with the potential for nondeterminism by keeping around *choice points* [2] even where no further backtracking is eventually needed.

Mode analysis deals with analysing the possible modes in which a predicate may be run in order to obtain information that may be useful for specialising the predicate and thus helping the compiler to implement it more efficiently.

We are also interested in using mode analysis to detect and prevent potential errors in a program, such as the use of an unbound variable in a place where a bound variable is required, preventing unsound uses of negation as failure, and knowing when it is safe to leave out the occur check. We want to find as many such errors as possible at compile time to avoid them showing up unpredictably as bugs at run time.

Much research has gone into mode analysis systems (or "mode systems" for short) for logic programs. We present a survey of that work. Most work on mode analysis aims to categorise the run-time instantiation patterns of variables at different *computation points* in the execution of the program, and is thus inherently linked to the operational semantics of the program. The aim is usually to identify which parts of a program *produce* data (by instantiating variables) and which parts of the program *consume* that data. This makes mode analysis a form of *data flow analysis* [3, 77]. There has, however, been some work on a more declarative approach to modes [104, 106] which views modes as constraints on the success set of the program (i.e. the set of ground atoms which are solutions for the program).

It is useful to categorise different mode systems based on two criteria. The first is whether the mode system is *descriptive* or *prescriptive*, as defined below. The second is the degree of precision with which the mode system captures mode information about the program. We look at these two concepts below and then discuss how previous work on mode systems fits these criteria.

2.4.1 Descriptive versus Prescriptive Modes

Probably the most fundamental question to ask about a mode system is what purpose it is intended to serve. A mode system may aim to *describe* the execution of a program without imposing any constraints on what programs are allowed and without attempting to modify the program. Examples of these include [31, 33, 63, 64, 74, 84, 88, 108, 115, 119, 124, 127, 135, 146, 147].

The alternative to descriptive mode systems are mode systems which *prescribe* a particular pattern of data flow. Prescriptive systems may attempt to transform predicates (e.g. by re-ordering conjunctions) so that they conform to the required pattern of data flow, which is usually given by *mode declarations*. They may also reject programs which they cannot prove are *mode correct*. Examples of such systems are [23, 55, 65, 78, 124–126, 128, 129, 140, 141, 143, 157].

Prescriptive mode systems can be further classified into whether they are *strong* or *weak*. Strong mode systems [e.g. 128, 129] require exact information about the possible instantiation state of each variable. They must know, for each variable at each computation point, whether or not the variable is instantiated, and if yes, to what degree.[3] A weak prescriptive mode system [e.g. 55] will make use of information that is available to do re-ordering, and check that mode declarations are conformed to, but will not necessarily always know whether a particular variable is bound or unbound.

The difference between descriptive and prescriptive mode systems is largely a language design issue. For example, Mercury's mode system is prescriptive, but once the compiler has done all the re-ordering necessary to make the program mode correct, one could say that it is then a descriptive system — the modes describe how the modified program will behave.

2.4.2 Precision of Mode Analysis

The other criterion for categorising mode systems is the degree of precision, or granularity in their abstract domains. The simplest domain is that of the *groundness analyses* [31, 64, 84, 88] where the domain is $\{ \bot, \mathsf{ground}, \top \}$. The domain $\{ \bot, \mathsf{ground}, \mathsf{free}, \top \}$, which we saw in Example 2.3.1 on page 16, further distinguishes definitely free variables and is used by several analyses [47, 78, 124–126]. Some of these analyses take free to mean only uninitialised variables which don't have any aliases (aliased variables are mapped to $\top$). Others attempt to do a simple form of alias analysis [47].

Some analyses add the value nonvar to the domain where $\mathsf{ground} \sqsubseteq \mathsf{nonvar} \sqsubseteq \top$ [63, 95, 96, 146, 147]. The value nonvar represents the set of terms that are not variables.

All of the above schemes use small, finite domains for mode analysis and are what Zachary and Yelick [157] refer to as *fixed-value domains*. Later analyses have attempted to increase precision by further refining nonvar into multiple abstract values representing different states of "boundness" [54, 55, 74, 85, 100, 115, 124, 127–129, 135, 145, 157]. Some analyses even refine ground to a set of abstract values representing a kind of "subtyping" [55, 128, 129].

Most analyses that use these more precise abstract domains rely on getting information about the possible structure of terms from a type system [54, 55, 115, 124, 127–129, 145, 157]. However, others operate in an untyped language [74, 85, 100, 135]. The latter are generally less precise.

[3] If we allow mode polymorphism, which we will discuss in Chapter 4, an instantiation state may be represented by an instantiation variable which represents an unbounded number of instantiation states. However, the constraints that we require on instantiation variables mean that this is still a strong mode system.

In order to provide an expressive programming language, a prescriptive mode system will generally require a more precise domain than a descriptive mode system.

2.4.3 Previous Work on Mode Analysis

Early implementations of DEC-10 Prolog [154] introduced "mode declarations" which could be supplied by the programmer to annotate which arguments of a predicate were input and which were output. These annotations could then be used by the compiler for optimisation. However, the annotations were not checked by the compiler and unpredictable and erroneous results could occur if a predicate was used in a manner contrary to its mode declaration.

Several logic programming systems, including Epilog [113] and NU-Prolog [102, 139], have used mode annotations over fixed-value domains to control the order in which the literals of a query are selected for resolution. Similarly, the read-only variable annotations of Concurrent Prolog [122], and a similar concept in later versions of Parlog [28, 37], were used to control the parallel execution of goals that may share variables.

The first work on automatically deriving modes was done by Mellish [95, 96]. Debray and Warren [47] later improved on this work by explicitly considering variable aliasing to derive a more precise analysis, albeit with a simpler abstract domain.

Almost all work on mode analysis in logic programming has focused on untyped languages, mainly Prolog. As a consequence, most systems use very simple fixed-value analysis domains, such as $\{ \bot, \mathsf{ground}, \mathsf{nonvar}, \mathsf{free}, \top \}$. One can use patterns from the code to derive more detailed program-specific domains, as in e.g. Janssens and Bruynooghe [74], Le Charlier and Van Hentenryck [85], Mulkers et al. [100], Tan and Lin [135], but such analyses must sacrifice too much precision to achieve acceptable analysis times.

Somogyi [128, 129] proposed fixing this problem by requiring type information and using the types of variables as the domains of mode analysis. This made it possible to handle more complex instantiation patterns. Several papers since then e.g. [115, 127] have been based on similar ideas. Like other papers on mode inference, these also assume that the program is to be analysed as is, without reordering. They therefore use modes to describe program executions, whereas we are interested in using modes to prescribe program execution order, and insist that the compiler must have exact information about instantiation states.

Most other prescriptive mode analysis systems work with much simpler domains (for example, Ground Prolog [78] recognises only two instantiation states, free and ground).

Other related work has been on mode checking for concurrent logic programming languages and for logic programming languages with coroutining [16, 34, 53]: there the emphasis has been on detecting communication patterns and possible deadlocks. The modes in such languages are independent of any particular execution strategy. For example, in Parlog and the concurrent logic programming language Moded Flat GHC [23, 140, 141, 143][4] an argument declared as an "input" need not necessarily been instantiated at the start of the goal, and an argument declared as "output" need not necessarily be instantiated at the end of the goal. In other words, these languages allow predicates to "override" their declared modes. This is necessary when two or more coroutining predicates co-operate to construct a term. One of the predicates will be declared

[4]GHC here stands for Guarded Horn Clauses, not to be confused with the Glasgow Haskell Compiler.

as the "producer" of the term (i.e. the argument will be declared as "output") and the other will be declared the "consumer" (with the argument "input"). Generally, the "producer" will be responsible for binding the top level functor of the term, but the "consumer" will also bind parts of the term.

Moded Flat GHC uses a constraint-based approach to mode analysis. GHC and Moded Flat GHC rely on position in the clause (in the head or guard versus in the body) to determine if a unification is allowed to bind any variables, which significantly simplifies the problem of mode analysis. The constraints generated are equational, and rely on delaying the complex cases where there are three or more occurrences of a variable in a goal.

This simplified approach might be applied to Mercury by adding guards to clauses. However, this would be a significant change to the language and one that we consider to be undesirable for a number of reasons:

- it would make it much harder to write predicates which work in multiple modes;
- it would destroy the purity of the language by making it possible to write predicates whose operational semantics do not match their declarative semantics; and
- we feel it is not desirable from a software engineering point of view to require programmers to have to think about and write guards.

For Mercury we want a strong prescriptive mode system which is as precise as possible and allows an efficient implementation of Mercury programs without allowing unsoundness (e.g. through negation as failure or omitting the occur check). We also want to be able to handle higher-order programming constructs, which have largely been ignored in previous work, and uniqueness analysis as described by Henderson [65].

We look again at some of the above mode systems, and how they relate to Mercury, at relevant places later in this thesis.

2.4.4 Types and Modes

We made brief mention above about the importance of a type system to provide the information necessary for a precise and expressive strongly prescriptive mode system. It is worth making a few further observations about the relationship between types and modes since the two concepts are closely related.

In Mercury, we keep concepts of types and modes separate. The type of a variable refers to the set of possible ground values the variable is allowed to take, whereas the mode of a variable refers to how the instantiation state of that variable can change over the execution of a predicate and therefore describes the set of (possibly non-ground) terms that the variable can take. If an instantiation state for a variable represents a set of ground terms, then it effectively represents a sub-type of the type of that variable.

In other programming paradigms mode-like concepts are usually treated under the framework of type analysis. For example, the concept of *linear types* [153] in functional languages is closely related to Mercury's concept of *unique modes* which we will discuss in later chapters.

Even in logic programming, types and modes are sometimes combined. One example is the notion of *directional types* [16]. An example of a directional type for the predicate `append/3` would be append(list → list, list → list, free → list). This asserts that if `append/3` is called with the first and second arguments being lists then for any answer all arguments will be lists.

2.5 Mercury

We now describe the logic programming language Mercury which we use throughout the rest of this thesis. Our description will be brief and mainly highlight the aspects of Mercury we are interested in for the purpose of mode analysis. For further details of the language please refer to the language reference manual [66] or to the papers we cite below.

2.5.1 Logic Programming for the Real World

Mercury is a purely declarative logic programming language designed for the construction of large, reliable and efficient software systems by teams of programmers [130, 131]. Mercury's syntax is similar to the syntax of Prolog, but Mercury also has strong module, type, mode and determinism systems, which catch a large fraction of programmer errors and enable the compiler to generate fast code. Thus programming in Mercury feels very different from programming in Prolog, and much closer to programming in a strongly typed functional language such as Haskell or in a safety-oriented imperative language such as Ada or Eiffel. Somogyi, Henderson, Conway, and O'Keefe [132] argue that strong module, type, mode and determinism systems are essential for an industrial strength "real world" logic programming language.

The definition of a predicate in Mercury is a goal containing atoms, conjunctions, disjunctions, negations, if-then-elses and quantifications. Unlike Prolog, which requires predicates to be in conjunctive normal form (and transforms them to that form if they are not already in it), Mercury allows compound goals to be nested arbitrarily. To simplify its algorithms, the Mercury compiler converts the definition of each predicate into what we call *superhomogeneous normal form* [131]. In this form, each predicate is defined by one goal, all variables appearing in a given atom (including the clause head) are distinct, and all atoms are (ignoring higher-order constructs for now) in one of the following three forms:

$$\mathtt{p(X_1,\ \ldots,\ X_n)} \qquad\qquad \mathtt{Y\ =\ X} \qquad\qquad \mathtt{Y\ =\ f(X_1,\ \ldots,\ X_n)}$$

Example 2.5.1. The definition of predicate `append/3` in superhomogeneous normal form is

```
append(Xs, Ys, Zs) :-
        (
                Xs = [],
                Ys = Zs
        ;
                Xs = [X | Xs0],
                Zs = [X | Zs0],
                append(Xs0, Ys, Zs0)
        ).
```

2.5.2 Types

Mercury has a strong, static, parametric polymorphic type system based on the Hindley-Milner [69, 98] type system of ML and the Mycroft-O'Keefe [101] type system for Prolog.

A *type* defines a set of ground terms. Each type has a *type definition* which is of the form

:- type $f(v_1, \ldots, v_n)$ ---> $f_1(t_1^1, ..., t_{m_1}^1); \cdots; f_k(t_1^k, ..., t_{m_k}^k)$.

where f/n is a *type constructor*, $v_1, \ldots, v_n$ are *type parameters*, $f_1/m_1, \ldots, f_k/m_k$ are term constructors (i.e. members of our signature Σ for program terms) and $t_1^1, \ldots, t_{m_k}^k$ are types.

Example 2.5.2. Some examples of type declarations are:

```
:- type bool
        --->    no
        ;       yes.

:- type maybe(T)
        --->    no
        ;       yes(T).

:- type list(T)
        --->    []
        ;       [T | list(T)].
```

Note that two different types can share the same term constructor (the constant `no` in this example). That is, we allow *overloading* of constructors. Also note that a type definition may refer to itself, allowing us to define types for recursive data structures such as lists. It is useful to think of a type definition as defining a *type graph*, for example, the graph for `list/1` is shown in Figure 2.2. The nodes labelled with the types `list(T)` and `T` represent positions in terms and the sub-terms rooted at those positions, and give the types of those sub-terms. They are called *or-nodes* because each sub-term can, in general, be bound to any one of several function symbols. The nodes labelled `[]` and `[ · | · ]` represent function symbols (also called *term constructors*) and are called *and-nodes*.

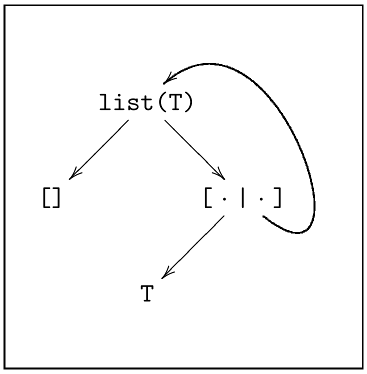

Figure 2.2: Type graph for `list/1`

The type of a predicate is declared using a '`:- pred`' declaration. For example, the declaration for `append/3` is

```
:- pred append(list(T), list(T), list(T)).
```

which declares that `append` is a predicate with three arguments, all of which are of type `list(T)`.

The Mercury run time system allows for information about types to be accessed by the program at run time [52]. The type system also supports Haskell-style *type classes* and *existential types* [75, 76]. These features are mostly unrelated to Mercury's mode system so we will not discuss them further here, except to note that we will need to take them into account in Section 4.4.

For more information on types in Mercury see Jeffery [75].

2.5.3 Modes

Mercury's mode system is based on the mode system of Somogyi [128, 129]. It is built on an abstract domain called the *instantiation state*, or *inst* as we will usually abbreviate it. An *inst* is an abstraction of the set of possible terms a variable may be bound to at a particular point during the execution of a program. (We refer to such a point as a *computation point*.) An inst attaches either `free` or `bound` to the or-nodes of the type tree. If an or-node is decorated with `free` then *all* sub-terms at the corresponding positions in the term described by the inst are free variables with no aliases; if an or-node is decorated with `bound` then *all* sub-terms at the corresponding positions in the term described by the inst are bound to function symbols.

The inst `ground` is a short-hand. It maps to `bound` not only the node to which it is attached, but also all the nodes reachable from it in the type graph.

The programmer can define insts through an *inst definition*. For example, the definition

```
:- inst list_skel == bound([] ; [free | list_skel]).
```

defines the inst `list_skel`. A variable with inst `list_skel` has its top-level function symbol bound to either the constant `[]` or the binary functor `[ · | · ]`, and, if it is bound to `[ · | · ]` then the first argument is a free variable and the second argument is bound to a `list_skel`. This definition gives us the *instantiation graph* shown in Figure 2.3. Note how the instantiation graph resembles the type graph for `list(T)` shown in Figure 2.2 on the preceding page, but with the or-nodes labelled with insts instead of types.

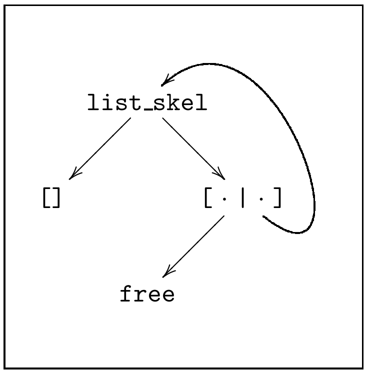

Figure 2.3: Instantiation graph for `list_skel`

A *mode* for a variable describes how that variable changes over the execution of a goal such as a predicate body. We write modes using the syntax ι `>>` ι' where ι is the inst of the variable at the start of the goal and ι' is the inst at the end of the goal. Modes can also be given names, for example the two most common modes, `in` and `out` are defined by

```
:- mode in == ground >> ground.
:- mode out == free >> ground.
```

and can be thought of as representing input and output arguments, respectively.

Inst and mode definitions may also take inst parameters. For example,

```
:- inst list_skel(I) == bound([] ; [I | list_skel(I)]).
:- mode in(I) == I >> I.
:- mode out(I) == free >> I.
```

A *mode declaration* for a predicate attaches modes to each of the predicate's arguments. A predicate may, in general, have multiple mode declarations. For example, two possible mode declarations for `append/3` are

```
:- mode append(in, in, out).
:- mode append(out, out, in).
```

Each mode of a predicate is called a *procedure*. The compiler generates separate code for each procedure.

In Mercury 0.10.1 mode declarations may not contain non-ground inst parameters. In Chapter 4 we look at how to extend the mode system to provide mode polymorphism. This extension is now part of Mercury 0.11.0.

If the predicate is not exported from the module in which it is defined then mode declarations are usually optional — modes can be *inferred* if no declaration is present. If a mode declaration is given for a predicate then the compiler will *check* that the declaration is valid. The compiler may re-order conjunctions if necessary to ensure that the mode declaration for a procedure is valid.

We define the mode system more formally, and give the rules and algorithms for mode inference and checking, in Chapter 3.

2.5.4 Determinism

Each procedure is categorised based on how many solutions it can produce and whether it can fail before producing a solution. This is known as its *determinism*. If we ignore committed choice contexts, which are of no concern in this thesis, there are six different categories, `det`, `semidet`, `multi`, `nondet`, `erroneous`, and `failure`. Their meanings are given in Table 2.2.

	Maximum number of solutions		
Can fail?	0	1	> 1
no	`erroneous`	`det`	`multi`
yes	`failure`	`semidet`	`nondet`

Table 2.2: Mercury's determinism categories

The determinism categories can also be arranged in a lattice representing how much information they contain, as shown in the Hasse diagram in Figure 2.4. Categories higher in the lattice contain less information than categories lower in the lattice. The more information the Mercury compiler has about the determinism of a procedure, the more efficient is the code it can generate for it.

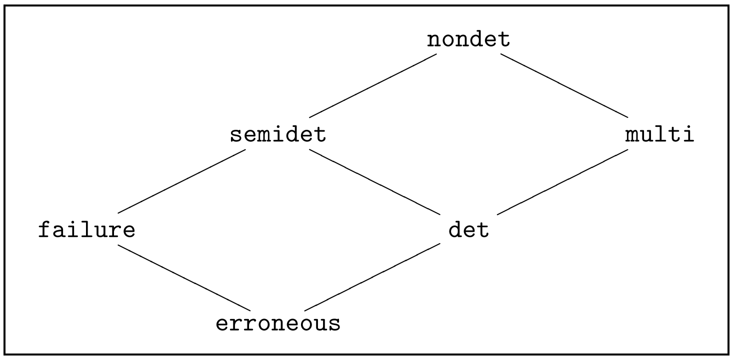

Figure 2.4: Mercury's determinism lattice

Determinism annotations can be added to mode declarations for the compiler to check. For example, we can annotate the mode declarations we gave above for `append/3`

```
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
```

to tell the compiler that calls to the procedure `append(in, in, out)` always have exactly one solution, and that calls to `append(out, out, in)` have at least one solution, and possibly more. The compiler can also infer determinism for predicates local to a module.

The determinism analysis system uses information provided by the mode system to check or infer the determinism for each procedure. It can then use this determinism information to generate very efficient code, specialised for each procedure. For more information on the determinism system see Henderson, Somogyi, and Conway [67]. See also Nethercote [108] which describes a determinism analysis system for Mercury in the context of a general abstract interpretation framework. (This work is based on the language HAL which uses the same determinism system as Mercury.)

2.5.5 Unique Modes

Unique modes are an extension to the Mercury mode system based on the work of Henderson [65] which in turn is based on the *linear types* of Wadler [153]. They allow the programmer to tell the compiler when a value is no longer needed so that the memory associated with it can be re-used. They also allow modelling of destructive update and input/output in logically sound ways. The system introduces new base instantiation states `unique` and `clobbered` which are the same as `ground` except that if a variable has inst `unique` there is only one reference to the corresponding value, and if a variable has inst `clobbered` there are no references to the corresponding value. A unique version of `bound` also exists. For example

```
:- inst unique_list_skel(I) == unique([] ; [I | unique_list_skel(I)]).
```

defines an inst `unique_list_skel/1` which is the same as `list_skel/1` except that the skeleton of the list must be uniquely referenced.

There are three common modes associated with uniqueness, `di` which stands for “destructive input”, `uo` which stands for “unique output” and `ui` which stands for “unique input”.

```
:- mode di == unique >> clobbered.
:- mode uo == free >> unique.
:- mode ui == unique >> unique.
```

Unique mode analysis ensures that there is only one reference to a `unique` value and that the program will never attempt to access a value that has been `clobbered`.

There are also variants of `unique` and `clobbered` called `mostly_unique` and `mostly_clobbered`. They allow the modelling of destructive update with trailing in a logically sound way. A value with inst `mostly_unique` has only one reference on forward execution, but may have more references on backtracking. A value with inst `mostly_clobbered` has no references on forward execution, but may be referenced on backtracking. There are also predefined modes `mdi`, `muo` and `mui` which are the same as `di`, `uo` and `ui`, except that they use `mostly_unique` and `mostly_clobbered` instead of `unique` and `clobbered`.

2.5.6 Higher-Order Programming

Higher-order programming allows predicates to be treated as first class data values and passed around in a program much like functions can be in functional languages.

A higher-order term can be created using a higher-order unification. For example

```
AddOne = (pred(X::in, Y::out) is det :- Y = X + 1)
```

gives the variable `AddOne` a value which is a higher-order term taking an input and returning its value incremented by one. Note that the modes and determinism of the higher-order term must always be supplied.

Such a term can be called with a goal such as

```
AddOne(2, A)
```

which would bind `A` to the value `3`. It may also be passed to another predicate. For example

```
map(AddOne, [1, 2, 3], B)
```

would bind `B` to the list `[2, 3, 4]`. The predicate `map/3` is a higher-order predicate which takes a higher-order term and a list, and applies the higher-order term to each element in the list. Its type and mode declarations are

```
:- pred map(pred(T, U), list(T), list(U)).
:- mode map(in(pred(in, out) is det), in, out) is det.
```

Note the use of the higher-order type `pred(T, U)` and the higher-order inst `pred(in, out) is det`.

Higher-order unification is, in general, undecidable so the Mercury mode system does not allow the general unification of two higher-order terms. The only unifications we allow involving higher-order terms are *assignments* (see 3.1). This means that Mercury's higher-order constructs can be integrated into its first-order semantics by a simple program transformation. Several methods for doing such a transformation have been proposed [e.g. 21, 22, 105, 155].

2.5.7 Modules

Mercury has a module system which allows separate compilation of large programs and also provides information hiding.

A Mercury module has an *interface* section and an *implementation* section. Any declarations which should be visible from outside the module are placed in the interface section. Internal declarations and all clauses are placed in the implementation section.

If a predicate is to be visible from outside the module in which it is defined, there must be type, mode and determinism declarations for it in the module interface.

Types can be exported *abstractly* from a module (that is, without exposing their implementation details) by giving an *abstract type* declaration in the module interface and giving the definition of the type in the implementation section. Abstract insts are not yet supported, although Section 4.5 discusses how they might be supported in future.

Chapter 3

The Current Mercury Implementation

In this chapter we will look at the mode analysis system implemented within the current Melbourne Mercury compiler and described in the Mercury reference manual [66].[1] This system is based on abstract interpretation [41–43] with the abstract domain being the *instantiation state* (or *inst* as we will usually abbreviate it).

In Section 3.1 we describe a simple mode system for a first-order subset of Mercury which does not include features such as unique modes, dynamic modes or higher-order modes. In Section 3.2 we describe the full Mercury mode system and in Section 3.3 we discuss some transformations that can turn a non-mode-correct program into a mode-correct program. In Section 3.4 we give the mode analysis algorithm and discuss some of its limitations. Finally, in Section 3.5 we look at how the Mercury mode system is related to other work, in particular the framework of abstract interpretation.

3.1 A Simple Mode System

We begin by describing a greatly simplified mode system for the first-order subset of Mercury. We look at what it means for a program to be *mode correct* in such a system and discuss some of the difficulties of checking mode correctness. In Section 3.2 we will build on this simple system in stages to eventually describe the full mode system for the Mercury language.

3.1.1 Abstract Syntax

To facilitate the discussion, we use the abstract syntax for first-order Mercury programs described in Figure 3.1 on the following page. The abstract syntax is based on the superhomogeneous normal form which was introduced in Section 2.5, but requires all the variables in the predicate body, except the head variables, to be explicitly existentially quantified. Any first order Mercury

[1] When we refer to the "current" implementation we are referring to version 0.10.1 released in April 2001. Version 0.11.0 was released on 24th December 2002 and, in addition to the mode system described in this chapter, implements the polymorphic mode system extensions described in Chapter 4.

program can be expressed in this abstract syntax through a straightforward transformation. In Section 3.2 we expand this into a full abstract syntax for all of Mercury including higher-order constructs.

Variable (Var)	v			
Function symbol (FuncSym)	f			
Predicate name (PredName)	π			
Flattened term (FTerm)	φ	$::=$	v	
		$\mid$	$f(\overline{v})$	
Goal (Goal)	G	$::=$	$\pi(\overline{v})$	(call)
		$\mid$	$v = \varphi$	(unification)
		$\mid$	$\exists\, \mathcal{P}\, v.G'$	(existential quantification)
		$\mid$	$\neg G'$	(negation)
		$\mid$	$\bigwedge \overline{G}$	(conjunction)
		$\mid$	$\bigvee \overline{G}$	(disjunction)
		$\mid$	$\mathsf{if}\ G_1\ \mathsf{then}\ G_2\ \mathsf{else}\ G_3$	(if-then-else)
Predicate (Pred)	C	$::=$	$\pi(\overline{v}) \leftarrow G$	
Program (Program)	P	$::=$	$\mathcal{P}\, C$	

Figure 3.1: Abstract syntax for first-order Mercury

The notation $\mathcal{P}\, x$ denotes a set whose elements are xs.

A *program* $P \in \mathsf{Program}$ is a set of predicates. A *predicate* $C \in \mathsf{Pred}$ has the form $\pi(\overline{v}) \leftarrow G$ where the atom $\pi(\overline{v})$ is the *head* of the predicate and the goal G is its *body*. The head consists of π, the name of the predicate, and $\overline{v}$, its argument vector. The arguments in $\overline{v}$ are all distinct variables.

A *goal* $G \in \mathsf{Goal}$ is either a call (where all the argument variables must be distinct), a unification, an existential quantification, a negation,[2] a conjunction, a disjunction, or an if-then-else. We refer to calls and unifications as *atomic* goals, and existential quantifications, negations, conjunctions, disjunctions and if-then-elses as *compound* goals. Head variables are implicitly universally quantified over the predicate body. All other variables in a goal must be existentially quantified: any non-head variable that is not explicitly quantified in the original program is implicitly existentially quantified to its closest enclosing scope in the transformation to the abstract syntax.

As in predicate logic, Mercury assumes we have a set of function symbols FuncSym, a signature Σ where $f/n \in \Sigma$ only if $f \in \mathsf{FuncSym}$, and a set of variables Var. This allows us to define the set of terms $\mathsf{Term} = \tau(\Sigma, \mathsf{Var})$. A *flattened term* $\varphi \in \mathsf{FTerm}$ (where $\mathsf{FTerm} \subseteq \mathsf{Term}$) is either a variable or a *functor* $f(\overline{v})$ applied to arguments that are distinct variables.

When writing goals, we will sometimes enclose them in corner brackets $\ulcorner \cdot \urcorner$ to distinguish them from the surrounding mathematics.

Example 3.1.1. The predicate `append/3` in our abstract syntax is shown in Figure 3.2 on the next page. $\triangleleft$

[2]It is not strictly necessary to have a negation goal type because it can be considered a special case of if-then-else, that is $\neg G$ is equivalent to $\mathsf{if}\ G\ \mathsf{then} \bigvee \langle\rangle\ \mathsf{else} \bigwedge \langle\rangle$ where the empty disjunction $\bigvee \langle\rangle$ is a goal that always fails, and the empty conjunction $\bigwedge \langle\rangle$ is a goal that always succeeds.

$$
\begin{array}{l}
\texttt{append}(\texttt{Xs},\texttt{Ys},\texttt{Zs}) \leftarrow \\
\quad \bigvee\langle\ \bigwedge\langle\ \texttt{Xs} = \texttt{[]}, \\
\quad\quad\quad \texttt{Ys} = \texttt{Zs} \\
\quad\quad \rangle, \\
\quad\quad \exists\,\{\,\texttt{Xs0},\texttt{Zs0},\texttt{X}\,\}.(\\
\quad\quad\quad \bigwedge\langle\ \texttt{Xs} = \texttt{[X | Xs0]}, \\
\quad\quad\quad\quad \texttt{Zs} = \texttt{[X | Zs0]}, \\
\quad\quad\quad\quad \texttt{append}(\texttt{Xs0},\texttt{Ys},\texttt{Zs0}) \\
\quad\quad\quad \rangle \\
\quad\quad) \\
\quad \rangle
\end{array}
$$

Figure 3.2: Abstract syntax for the predicate `append/3`

Definition 3.1.1 (unquantified variables) The function $\mathsf{uq} : \mathsf{Goal} \to \mathcal{P}\,\mathsf{Var}$ gives the set of *unquantified variables*[3] in a goal and is defined in Figure 3.3. ◁

$$
\mathsf{uq}(G) = \begin{cases}
\overline{v} & \text{if } G = \ulcorner \pi(\overline{v}) \urcorner, \\
\{\, v, v' \,\} & \text{if } G = \ulcorner v = v' \urcorner, \\
\{\, v \,\} \cup \overline{v} & \text{if } G = \ulcorner v = f(\overline{v}) \urcorner, \\
\mathsf{uq}(G') \setminus V & \text{if } G = \ulcorner \exists V.G' \urcorner, \\
\mathsf{uq}(G') & \text{if } G = \ulcorner \neg G' \urcorner, \\
\bigcup_{G' \in \overline{G}} \mathsf{uq}(G') & \text{if } G = \ulcorner \bigwedge \overline{G}\, \urcorner, \\
\bigcup_{G' \in \overline{G}} \mathsf{uq}(G') & \text{if } G = \ulcorner \bigvee \overline{G}\, \urcorner, \\
\bigcup_{G' \in \{\, G_1, G_2, G_3 \,\}} \mathsf{uq}(G') & \text{if } G = \ulcorner \mathsf{if}\ G_1\ \mathsf{then}\ G_2\ \mathsf{else}\ G_3 \urcorner.
\end{cases}
$$

Figure 3.3: Unquantified variables

3.1.2 Instantiation States

An *instantiation state* (often abbreviated *inst*) attaches instantiation information to the or-nodes of a type tree. This information describes whether the corresponding node is bound or free.[4] All children of a free node must be free.

Definition 3.1.2 (instantiation state) Figure 3.4 on the next page describes the form of our simplified instantiation states. An inst $\iota \in \mathsf{Inst}$ is either free, or bound to one of a set of possible functors whose argument insts are described recursively. Each function symbol must occur at most once in the set. ◁

[3]In predicate logic these are usually known as “free” variables, but we do not use that term here to avoid confusion with the alternative use of “free” in the Mercury mode system. Similarly, we refer to “quantified” variables rather than “bound” variables.

[4]In the full Mercury mode system, described later in this chapter, other information is also present, such as whether this node is a unique reference to the structure.

Instantiation state (Inst)	ι	$::=$	free
		$\mid$	$\mathsf{bound}(\mathcal{P}\, f(\overline{\iota}))$
			where each f must occur at most once

Figure 3.4: Simple instantiation state

Definition 3.1.3 (concretisation function γ) An inst approximates a set of terms given by the function $\gamma : \mathsf{Inst} \to \mathcal{P}\,\mathsf{Term}$:

$$\gamma(\mathsf{free}) = \{\ \_\ \}$$

$$\gamma(\mathsf{bound}(B)) = \bigcup_{f(\iota_1,\ldots,\iota_n)\in B} \{\ f(t_1,\ldots,t_n) \mid \forall j \in \{\ 1,\ldots,n\ \}\ .\ t_j \in \gamma(\iota_j)\ \}$$

We call γ the *concretisation* function for insts. ◁

We use an underscore ($\_ \in \mathsf{Var}$) to represent a fresh variable that does not occur elsewhere (i.e. is not *aliased* to any other part of the term or any other data structure). We will give the *abstraction function* $\alpha : \mathcal{P}\,\mathsf{Term} \to \mathsf{Inst}$, which does the reverse mapping, a little later on.

Note that in this representation, if we wish an inst to represent an infinite set of terms the inst will need to be infinitely large. For example if we wish to represent a set of terms that are lists of any finite length, $\{\ [], [\_], [\_,\_], \ldots\ \}$, the inst we need is $\mathsf{bound}(\{\ [], [\mathsf{free} \mid \mathsf{bound}(\{\ [], [\mathsf{free} \mid \mathsf{bound}(\ldots)]\ \})]\ \})$. The concept of infinite insts is a convenient mathematical abstraction which we will make use of to simplify the presentation which follows. In Section 3.2.5 we will discuss how these insts are actually implemented within the Mercury compiler. That section also gives a summary of the differences and correspondence between the abstract syntax for instantiation states presented in this chapter and the concrete Mercury syntax that was described in Chapter 2.

Also note that this definition does not allow us to represent terms containing variables that are aliased. We look at that issue in Chapter 5.

A major rôle of mode analysis is to determine how the instantiation states of the variables of a predicate change over the (forward) execution of the predicate body. There are certain instantiation state transitions which we must prohibit. In particular, a variable's instantiation state may not change from $\mathsf{bound}(B)$ (for some B) to free since once a variable is bound, it may not be unbound again on forward execution. The same restriction must apply to the argument insts of each functor of a bound inst. To describe the allowable transitions, we introduce a partial order relation.

Definition 3.1.4 (instantiatedness partial order for insts) We define a partial order $\preceq$ over instantiation states:

$$\iota \preceq \mathsf{free}$$

$$\begin{aligned} &\mathsf{bound}(B) \preceq \mathsf{bound}(B') \iff \\ &\qquad \forall \beta \in B.\ \exists \beta' \in B'.\ \beta = f(\iota_1,\ldots,\iota_n)\ \wedge\ \beta' = f(\iota'_1,\ldots,\iota'_n)\ \wedge \\ &\qquad \forall i \in \{\ 1,\ldots,n\ \}\ .\ \iota_i \preceq \iota'_i \end{aligned}$$

If $\iota \preceq \iota'$ then we say that ι is *at least as instantiated as* ι'. ◁

For any variable, a transition from instantiation state ι' to instantiation state ι over the forward execution of a goal is allowable if and only if $\iota \preceq \iota'$.

Definition 3.1.5 (not reached) We define the inst

$$\mathsf{not\_reached} = \mathsf{bound}(\emptyset).$$

This instantiation state indicates that we are at an unreachable computation point, such as after a computation that always fails, an infinite loop or a program abort. $\triangleleft$

We write $\iota \curlywedge \iota'$ to refer to the greatest lower bound, and $\iota \curlyvee \iota'$ to refer to the least upper bound of the insts ι and ι' under $\preceq$. Note that for all insts $\iota \in \mathsf{Inst}$,

$$\mathsf{not\_reached} \;\preceq\; \iota \;\preceq\; \mathsf{free}.$$

Also, every set of insts has a unique least upper bound and a unique greatest lower bound. Therefore $\langle \mathsf{Inst}, \preceq \rangle$ is a complete lattice with $\top = \mathsf{free}$ and $\bot = \mathsf{not\_reached}$.

Example 3.1.2. Figure 3.5 shows a Hasse diagram of the lattice $\langle \mathsf{Inst}, \preceq \rangle$ where the Herbrand universe (i.e. the set of ground terms) is restricted to $\{$ `g(c), h(d)` $\}$. $\triangleleft$

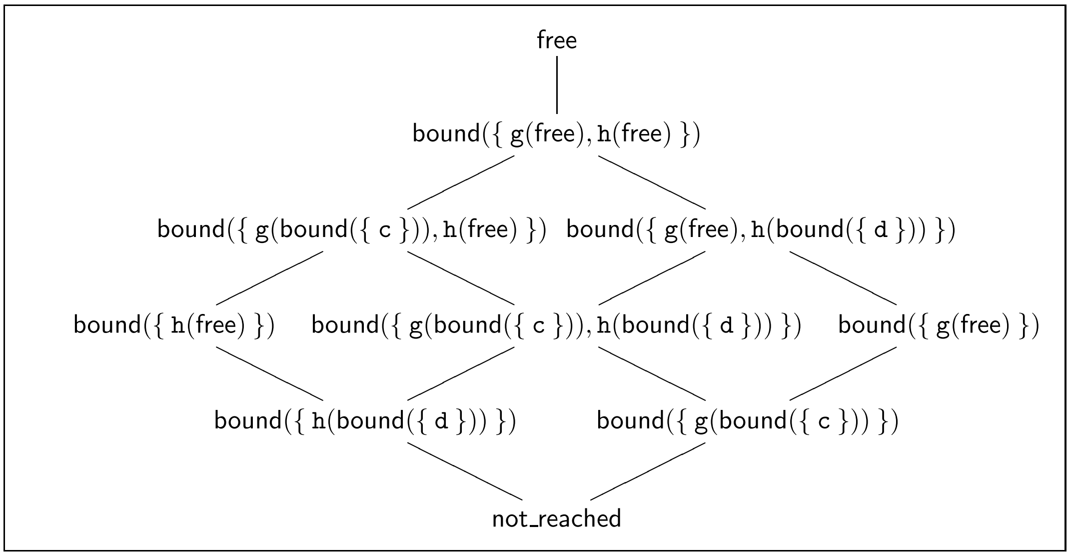

Figure 3.5: Hasse diagram for $\langle \mathsf{Inst}, \preceq \rangle$ and Herbrand universe $\{$ `g(c), h(d)` $\}$ (see Example 3.1.2)

Definition 3.1.6 (ground insts) It is convenient to define the inst

$$\mathsf{ground} = \mathsf{bound}\left(\left\{\, f(\underbrace{\mathsf{ground}, \ldots, \mathsf{ground}}_{n \text{ times}}) \;\middle|\; f/n \in \Sigma \,\right\}\right)$$

The concretisation $\gamma(\mathsf{ground})$ of ground is $\tau(\Sigma)$, the set of all ground terms.[5] If $\iota \preceq \mathsf{ground}$ then

[5] In practice, we have type information about every variable in the program. Type definitions define a subset of $\tau(\Sigma)$ so the ground inst can be thought of as approximating only the subset of $\tau(\Sigma)$ defined by the type of the particular variable to which it is associated.

we say that ι is a *ground* inst. $\triangleleft$

3.1.3 Instmaps, Modes and Procedures

Definition 3.1.7 (instmap) An *instmap* is a function

$$\mathsf{InstMap} \subseteq \mathsf{Var} \to \mathsf{Inst}$$

where

$$I \in \mathsf{InstMap} \iff (\exists v \in \operatorname{dom} I.\ I(v) = \mathsf{not\_reached}) \Rightarrow (\forall v \in \operatorname{dom} I.\ I(v) = \mathsf{not\_reached})$$

An instmap gives a mapping from program variables to instantiation states for a particular program point. If some variables in the instmap have inst $\mathsf{not\_reached}$ then all variables must have inst $\mathsf{not\_reached}$. The reason for this is that a variable will have inst $\mathsf{not\_reached}$ if and only if we are at an unreachable program point, and at such a point all variables will have inst $\mathsf{not\_reached}$. $\triangleleft$

Definition 3.1.8 (unreachable instmap) We define the set $\mathsf{unreachable} \subseteq \mathsf{InstMap}$ as

$$\mathsf{unreachable} = \{\, I \in \mathsf{InstMap} \mid \forall v \in \operatorname{dom} I.\ I(v) = \mathsf{not\_reached} \,\}$$

and refer to such instmaps as *unreachable*. $\triangleleft$

During forward execution, variables will become more instantiated. We provide an operation to allow an instmap to be updated when some subset of the variables in it become more instantiated.

Definition 3.1.9 (instmap update) The operator

$$\oplus : \mathsf{InstMap} \times \mathsf{InstMap} \to \mathsf{InstMap}$$

allows some values in an instmap to be updated:

$$(I \oplus I')(v) = \begin{cases} \mathsf{not\_reached} & \text{if } I \in \mathsf{unreachable} \lor I' \in \mathsf{unreachable}, \\ I'(v) & \text{if } v \in \operatorname{dom} I', \\ I(v) & \text{otherwise.} \end{cases}$$

We require that

$$\forall v \in \operatorname{dom} I'.\ v \in \operatorname{dom} I \land I'(v) \preceq I(v)$$

That is, any variable in I' must also be in I and must be at least as instantiated in I' as in I. $\triangleleft$

Definition 3.1.10 (mode) A *mode* is a pair

$$\mathsf{Mode} \subseteq \mathsf{InstMap} \times \mathsf{InstMap}$$

of instmaps and represents the initial and final instantiation states of a set of variables over a goal. The domains of the two instmaps must be the same set of variables and every variable must be at least as instantiated in the second instmap as in the first. That is,

$$\mathsf{Mode} = \left\{ \langle I, I' \rangle \;\middle|\; \begin{array}{l} I, I' \in \mathsf{InstMap} \;\wedge\; \operatorname{dom} I = \operatorname{dom} I' \;\wedge \\ \forall v \in \operatorname{dom} I.\ I'(v) \preceq I(v) \end{array} \right\}$$

We refer to the domain of a mode $M = \langle I, I' \rangle$ as $\operatorname{dom} M = \operatorname{dom} I = \operatorname{dom} I'$. The instmap I is the *initial instmap* which we write as M_{init}; I' is the *final instmap* which we write as M_{fin}. We also define $\oplus$ for modes such that $M \oplus I = \langle M_{\mathsf{init}}, M_{\mathsf{fin}} \oplus I \rangle$. ◁

At times during mode analysis, such as when analysing a procedure call or the end of a procedure body, we will need to ensure that a variable has a certain instantiation state. We may know that the variable in question actually has a more specific instantiation state which would do equally well. We would like to be able to express a relation between instantiation states to say when one inst "matches" another, i.e. is a more specific inst which may be used in place of the required (more general) inst.

Definition 3.1.11 (matches partial order for insts) We define a partial ordering $\sqsubseteq$ over insts

$$\begin{array}{l} \mathsf{free} \sqsubseteq \mathsf{free} \\ \mathsf{not\_reached} \sqsubseteq \mathsf{free} \\ \mathsf{bound}(B) \sqsubseteq \mathsf{bound}(B') \qquad \iff \\ \qquad \forall \beta \in B.\ \exists \beta' \in B'.\ \beta = f(\iota_1, \ldots, \iota_n) \;\wedge\; \beta' = f(\iota'_1, \ldots, \iota'_n) \;\wedge \\ \qquad \forall i \in \{\, 1, \ldots, n \,\}.\ \iota_i \sqsubseteq \iota'_i \end{array}$$

If $\iota \sqsubseteq \iota'$ then we say that ι *matches* ι'. ◁

Note that not_reached matches any inst, but we do not need a separate rule of the form $\mathsf{not\_reached} \sqsubseteq \mathsf{bound}(B)$ since not_reached is only syntactic sugar for $\mathsf{bound}(\emptyset)$ and the third case handles this.

Note also that the relation $\sqsubseteq$ is a subset of the relation $\preceq$, that is

$$\forall \iota, \iota' \in \mathsf{Inst}.\;\; \iota \sqsubseteq \iota' \implies \iota \preceq \iota',$$

the difference being that only free and not_reached match free. Also note that $\sqsubseteq$ is not a lattice (it has a bottom element $\bot = \mathsf{not\_reached}$, but no top element). We write $\iota \sqcap \iota'$ to refer to the greatest lower bound, and $\iota \sqcup \iota'$ to refer to the least upper bound (where it exists) of the insts ι and ι' under $\sqsubseteq$. The operators $\bigsqcap$ and $\bigsqcup$ give the glb and lub (where it is defined), respectively, for sets of insts.

Example 3.1.3. Figure 3.6 on the next page shows a Hasse diagram for the poset $\langle \mathsf{Inst}, \sqsubseteq \rangle$ and Herbrand universe $\{$ `g(c)`, `h(d)` $\}$. The diagram uses b and f instead of bound and free for readability. Edges that are in $\langle \mathsf{Inst}, \preceq \rangle$, but not $\langle \mathsf{Inst}, \sqsubseteq \rangle$ are shown dotted for comparison. For example,

$$\mathsf{bound}(\{\, \mathsf{g}(\mathsf{free}) \,\}) \quad \sqsubseteq \quad \mathsf{bound}(\{\, \mathtt{f}(\mathsf{bound}(\{\, \mathsf{c} \,\})), \mathsf{g}(\mathsf{free}) \,\})$$

and

$$\mathsf{bound}(\{\, \mathtt{f}(\mathsf{free}) \,\}) \sqcup \mathsf{bound}(\{\, \mathsf{g}(\mathsf{free}) \,\}) \quad = \quad \mathsf{bound}(\{\, \mathtt{f}(\mathsf{free}), \mathsf{g}(\mathsf{free}) \,\})$$ ◁

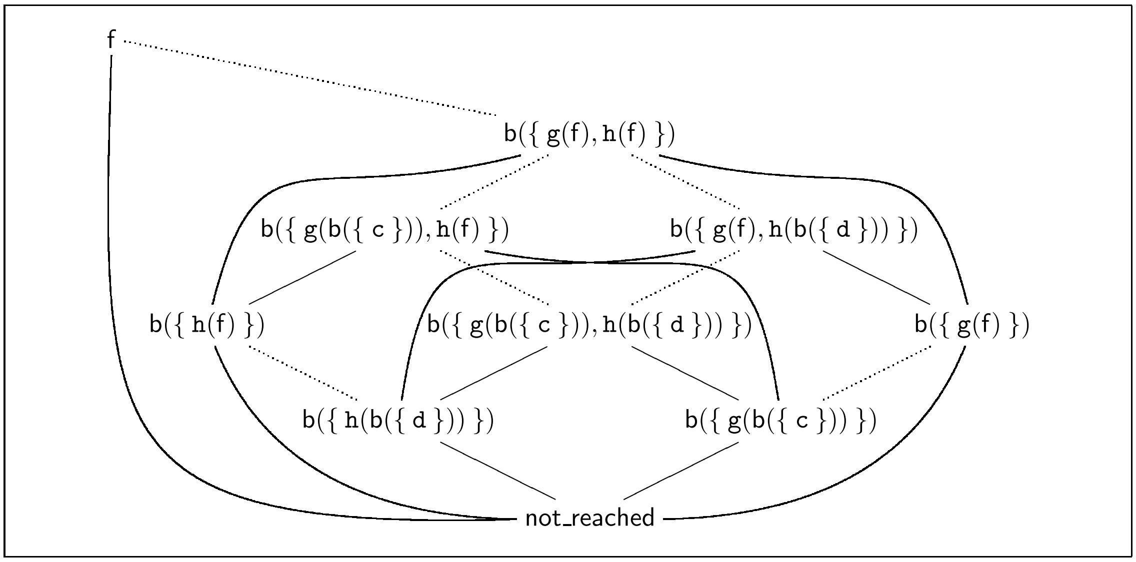

Figure 3.6: Hasse diagram for $\langle \mathsf{Inst}, \sqsubseteq \rangle$ and Herbrand universe { `g(c), h(d)` } (see Example 3.1.3)

Definition 3.1.12 (abstraction function) The partial order $\sqsubseteq$ allows us to define the abstraction function $\alpha : \mathcal{P}\,\mathsf{Term} \to \mathsf{Inst}$.

$$\alpha(T) = \bigsqcup_{t \in T} \alpha'(t)$$

where

$$\begin{aligned} \alpha'(\_) &= \mathsf{free} \\ \alpha'(f(t_1, \ldots, t_n)) &= \mathsf{bound}(\{\, f(\alpha'(t_1), \ldots, \alpha'(t_n)) \,\}) \end{aligned}$$

Note that α is a partial function. It is not defined for any set of terms for which the least upper bound of their abstractions is not defined (such as a set containing both variable and non-variable terms). It is also not defined for any term containing an aliased variable. ◁

The abstraction function gives us an alternative, and simpler, way of defining the inst ground:

$$\mathsf{ground} = \alpha(\tau(\Sigma))$$

In Section 3.5 we will look again at the concretisation and abstraction functions and how mode analysis fits into the framework of abstract interpretation.

Definition 3.1.13 (procedure) A *procedure* is a tuple

$$\mathsf{Proc} \subseteq \mathsf{Pred} \times \mathsf{Mode}$$

of a predicate and a mode which describes the initial and final instantiation states of the arguments of the predicate. The domain of the instmaps in the mode must be exactly the variables in the argument vector of the predicate (any other variables in the predicate body must be existentially quantified and must not appear in the procedure mode). That is,

$$\mathsf{Proc} = \{ \langle \pi(\overline{v}) \leftarrow G, M \rangle \in \mathsf{Pred} \times \mathsf{Mode} \,|\, \mathrm{dom}\, M = \overline{v} \} \qquad \triangleleft$$

3.1.4 Operations Used in Mode Analysis

We now define some operations which will be used in the mode rules for our simple mode system. Many of these operations are partial functions and we give definitions that state the conditions under which they are defined. A call to one of these functions with inputs for which it is undefined should be understood as corresponding to a mode error.

Definition 3.1.14 (mode sequence operation) The mode sequence operation $\triangleright$ is defined by the equation

$$\langle I, I' \rangle \triangleright \langle I', I'' \rangle = \langle I, I'' \rangle \,.$$

That is, the resulting mode has an initial instmap that is the same as the initial instmap of the first argument and a final instmap that is the same as the final instmap of the second argument. The final instmap of the first argument must be the same as the initial instmap of the second argument. $\triangleleft$

Definition 3.1.15 (restriction operation on modes) The operation

$$\ominus : \mathsf{Mode} \times \mathcal{P}\,\mathsf{Var} \rightarrow \mathsf{Mode}$$

restricts a mode by removing all the variables in a given set from the mode's domain:

$$M \ominus V = \Big\langle \big\{ v \mapsto M_{\mathsf{init}}(v) \;\big|\; v \in \mathrm{dom}\, M \setminus V \big\},\ \big\{ v \mapsto M_{\mathsf{fin}}(v) \;\big|\; v \in \mathrm{dom}\, M \setminus V \big\} \Big\rangle \qquad \triangleleft$$

Below we define operations for *merging* insts and modes. These are used in the mode rules to combine mode information from two branches of a *branched* goal, such as a disjunction or an if-then-else. A variable (or any node within the inst of a variable) may not be free at the end of one branch of a branched goal and bound to some set of functors in another branch. The set of functors that a variable may be bound to at the end of a branched goal is the union of the sets of functors it may be bound to at the end of each branch. The least upper bound operation under the matches partial order has the required properties:

Definition 3.1.16 (inst merge) The *merge* of two insts after a branched goal is defined as the least upper bound $\sqcup$ (if it exists) under the partial order $\sqsubseteq$. $\triangleleft$

Definition 3.1.17 (mode merge) We define the *merge* of two modes (if it exists) with the equation

$$\langle I, I' \rangle \bowtie \langle I, I'' \rangle = \big\langle I,\ \{ v \mapsto \iota \,|\, v \in \mathrm{dom}\, I \ \wedge\ \iota = I'(v) \sqcup I''(v) \} \big\rangle$$

The modes must have the same initial instmap, which is also the initial instmap of the merged mode. The final instmap of the merged mode is obtained by merging the insts of each variable in the final instmaps of the two modes. $\triangleleft$

Note that if one of the instmaps is unreachable (i.e. the corresponding branch cannot succeed—perhaps due to a call to the library predicate '`error`' which aborts the program) then the result of the merge will be the other instmap.

We now look at the operation of *abstract unification* of two instantiation states. Abstract unification is the abstract equivalent of the unification operation on terms.

The result of abstractly unifying two insts must be an inst that is at least as instantiated as each of the initial insts. Abstract unification then is the greatest lower bound under the partial order $\preceq$. However, because of our requirement in this simple system that variables may not be aliased, we cannot allow an abstract unification where any of the concrete unifications it represents might cause two variables to become aliased. A straightforward way to avoid this is to require that the resulting inst be ground — if there are no variables in the term resulting from the unification then it is obvious that the unification doesn't cause any variables to become aliased.[6] We therefore arrive at the following definition of abstract unification.

Definition 3.1.18 (abstract unification for insts) The abstract unification ι of two insts ι_1 and ι_2 is defined by

$$\mathsf{abstract\_unify\_inst}(\iota_1, \iota_2, \iota) \qquad \Longleftrightarrow \qquad \iota = \iota_1 \curlywedge \iota_2 \quad \wedge \quad \iota \preceq \mathsf{ground} \qquad \triangleleft$$

We can see that this definition explicitly requires the inst ι, resulting from taking the greatest lower bound of ι_1 and ι_2, to be ground, thus avoiding any problems with aliasing.

Example 3.1.4. The following examples can be clearly seen from the Hasse diagram in Figure 3.5 on page 33.

$$\begin{array}{l}
\mathsf{abstract_unify_inst}(\mathsf{free},\ \mathsf{bound}(\{\ \mathtt{f}(\mathsf{bound}(\{\ \mathsf{c}\ \}))\ \}),\ \iota) \\
\qquad\qquad \Longleftrightarrow \quad \iota = \mathsf{bound}(\{\ \mathtt{f}(\mathsf{bound}(\{\ \mathsf{c}\ \}))\ \}) \\
\mathsf{abstract_unify_inst}(\mathsf{bound}(\{\ \mathtt{f}(\mathsf{bound}(\{\ \mathsf{c}\ \})),\ \mathsf{g}(\mathsf{bound}(\{\ \mathtt{d}\ \})\ \}),\ \mathsf{bound}(\{\ \mathtt{f}(\mathsf{free})\ \}),\ \iota) \\
\qquad\qquad \Longleftrightarrow \quad \iota = \mathsf{bound}(\{\ \mathtt{f}(\mathsf{bound}(\{\ \mathsf{c}\ \}))\ \}) \\
\mathsf{abstract_unify_inst}(\mathsf{bound}(\{\ \mathsf{g}(\mathsf{free})\ \}),\ \mathsf{bound}(\{\ \mathtt{f}(\mathsf{free})\ \}),\ \iota) \\
\qquad\qquad \Longleftrightarrow \quad \iota = \mathsf{not_reached} \\
\mathsf{abstract_unify_inst}(\mathsf{free},\ \mathsf{bound}(\{\ \mathtt{f}(\mathsf{free})\ \}),\ \iota) \ \text{ is not defined.}
\end{array} \qquad \triangleleft$$

For unifications of the form $\ulcorner v = f(\overline{v}) \urcorner$ the abstract unification needs to compute the final inst of v and the final insts of each argument in $\overline{v}$.

Definition 3.1.19 (abstract unification with a functor) Abstract unification of an inst ι with a functor f with argument insts $\overline{\iota}$ is

$$\begin{array}{l}
\mathsf{abstract_unify_inst_functor}(\iota, f, \overline{\iota}, \iota', \overline{\iota'}) \iff \\
\quad \mathsf{abstract_unify_inst}(\iota, \mathsf{bound}(\{\ f(\overline{\iota})\ \}), \iota') \wedge \\
\quad \Big(\iota' = \mathsf{bound}(\{\ f(\overline{\iota'})\ \}) \ \vee\ \big(\iota' = \mathsf{not_reached} \ \wedge\ \overline{\iota'} = \langle \mathsf{not_reached}, \ldots, \mathsf{not_reached} \rangle\big)\Big)
\end{array}$$

where ι' is the inst resulting from the unification and $\overline{\iota'}$ are the resulting insts of the arguments. $\triangleleft$

[6] In Section 3.2 we look at how liveness information can be used to relax this restriction and in Chapter 5 we look at how alias tracking can relax it even further.

Note that in a variable-functor unification, the resulting inst must either be bound to the functor if the unification can succeed, or $\mathsf{not\_reached}$ if the unification always fails.

3.1.5 The Mode Analysis Rules

In this section we define, using a set of basic rules, what it means for a program to be *mode-correct*. We start by giving the rules for a mode-correct procedure. In a procedure, the order of execution of conjuncts in a conjunction is fixed. One of the tasks of mode analysis is to determine a mode-correct order for conjuncts. The rules given below do not do this ordering, they merely describe what it means for a procedure to be mode-correct. Section 3.3 discusses this further.

Intuitively, a procedure $R = \langle \pi(\overline{v}) \leftarrow G, \langle I, I' \rangle \rangle$ is mode-correct only if for any given initial variable substitution $\theta : \mathsf{Var} \rightarrow \mathsf{Term}$ where $\forall v \in \operatorname{dom} I.\ \theta(v) \in \gamma(I(v))$, the final substitution θ' after execution of the procedure body goal has the property $\forall v \in \operatorname{dom} I.\ \theta'(v) \in \gamma(I'(v))$.

The notation $\Gamma \overset{\mathsf{Proc}}{\vdash} R$ is a *mode judgement* and means that the procedure $R = \langle \pi(\overline{v}) \leftarrow G, M \rangle \in \mathsf{Proc}$ is mode-correct with respect to the *environment* Γ. The set $\Gamma \subseteq \mathsf{Proc}$ is the set of mode-correct procedures in the program.

Similarly, for a goal $G \in \mathsf{Goal}$, the notation $\langle \Gamma, \mathcal{N} \rangle \overset{\mathsf{Goal}}{\vdash} G : M$ means that G is a mode-correct goal with mode M, with respect to environment $\langle \Gamma, \mathcal{N} \rangle$. The environment for mode judgements for goals has an extra component $\mathcal{N} \subseteq \mathsf{Var}$ which is the set of non-local variables in the goal.

Figure 3.7 gives the relationship between mode judgements for goals and procedures: if a procedure R has goal G and mode M, and G is mode-correct with mode M' such that for each argument variable v $M_{\mathsf{init}}(v) \sqsubseteq M'_{\mathsf{init}}(v)$ and $M'_{\mathsf{fin}}(v) \sqsubseteq M_{\mathsf{fin}}(v)$ then R is mode-correct with mode M. The requirement $M_{\mathsf{init}}(v) \sqsubseteq M'_{\mathsf{init}}(v)$ means that the initial instantiation of the variable given by the mode declaration is at least as tight as what is required by the procedure body. For example, the procedure body may require the variable has inst $\mathsf{bound}(\{\, a, b, c \,\})$, but the mode declaration may be stricter and require, say, $\mathsf{bound}(\{\, a, b \,\})$. Conversely, $M'_{\mathsf{fin}}(v) \sqsubseteq M_{\mathsf{fin}}(v)$ means that the final inst of a variable guaranteed by the procedure body must be at least as tight as what is required by the mode declaration.

$$
\text{PROC}\ \frac{\begin{array}{c} R \in \Gamma \\ R = \langle \pi(\overline{v}) \leftarrow G, M \rangle \\ \langle \Gamma, \mathcal{N} \rangle \overset{\mathsf{Goal}}{\vdash} G : M' \\ \operatorname{dom} M = \operatorname{dom} M' = \overline{v} = \mathcal{N} = \mathsf{uq}(G) \\ \forall v \in \overline{v}.\ M_{\mathsf{init}}(v) \sqsubseteq M'_{\mathsf{init}}(v) \\ \forall v \in \overline{v}.\ M'_{\mathsf{fin}}(v) \sqsubseteq M_{\mathsf{fin}}(v) \end{array}}{\Gamma \overset{\mathsf{Proc}}{\vdash} R}
$$

Figure 3.7: Mode rule for a procedure

Figure 3.8 on the following page shows the mode rules for compound goals, namely conjunctions, disjunctions, existential quantifications, negations and if-then-elses. The relation $\mathsf{nobind}(M, \mathcal{N})$ is true if and only if the mode M does not cause any of the variables in the set $\mathcal{N}$

to become further instantiated, that is[7]

$$\mathsf{nobind}(M, \mathcal{N}) \iff \forall v \in \mathcal{N}.\ M_{\mathsf{init}}(v) \sqsubseteq M_{\mathsf{fin}}(v)$$

This test is required in the NOT and ITE rules to ensure soundness of negation as failure [103].

$$\text{CONJ}\ \frac{\begin{array}{c}\langle \Gamma, \mathsf{uq}(G_1)\rangle \stackrel{\mathsf{Goal}}{\vdash} G_1 : M_1 \\ \vdots \\ \langle \Gamma, \mathsf{uq}(G_n)\rangle \stackrel{\mathsf{Goal}}{\vdash} G_n : M_n \\ M = M_1 \triangleright \cdots \triangleright M_n\end{array}}{\langle \Gamma, \mathcal{N}\rangle \stackrel{\mathsf{Goal}}{\vdash} \ulcorner \bigwedge \langle G_1, \ldots, G_n\rangle \urcorner : M} \qquad \text{SOME}\ \frac{\begin{array}{c}\langle \Gamma, \mathcal{N} \setminus V\rangle \stackrel{\mathsf{Goal}}{\vdash} G : M' \\ M = M' \ominus V\end{array}}{\langle \Gamma, \mathcal{N}\rangle \stackrel{\mathsf{Goal}}{\vdash} \ulcorner \exists V.G \urcorner : M}$$

$$\text{DISJ}\ \frac{\begin{array}{c}\langle \Gamma, \mathsf{uq}(G_1)\rangle \stackrel{\mathsf{Goal}}{\vdash} G_1 : M_1 \\ \vdots \\ \langle \Gamma, \mathsf{uq}(G_n)\rangle \stackrel{\mathsf{Goal}}{\vdash} G_n : M_n \\ M = M_1 \bowtie \cdots \bowtie M_n\end{array}}{\langle \Gamma, \mathcal{N}\rangle \stackrel{\mathsf{Goal}}{\vdash} \ulcorner \bigvee \langle G_1, \ldots, G_n\rangle \urcorner : M} \qquad \text{NOT}\ \frac{\begin{array}{c}\langle \Gamma, \mathcal{N}\rangle \stackrel{\mathsf{Goal}}{\vdash} G : M \\ \mathsf{nobind}(M, \mathcal{N})\end{array}}{\langle \Gamma, \mathcal{N}\rangle \stackrel{\mathsf{Goal}}{\vdash} \ulcorner \neg G \urcorner : M}$$

$$\text{ITE}\ \frac{\begin{array}{c}\langle \Gamma, \mathsf{uq}(G_1)\rangle \stackrel{\mathsf{Goal}}{\vdash} G_1 : M_1 \\ \langle \Gamma, \mathsf{uq}(G_2)\rangle \stackrel{\mathsf{Goal}}{\vdash} G_2 : M_2 \\ \langle \Gamma, \mathsf{uq}(G_3)\rangle \stackrel{\mathsf{Goal}}{\vdash} G_3 : M_3 \\ \mathsf{nobind}(M_1, \mathcal{N}) \\ M = (M_1 \triangleright M_2) \bowtie M_3\end{array}}{\langle \Gamma, \mathcal{N}\rangle \stackrel{\mathsf{Goal}}{\vdash} \ulcorner \mathsf{if}\ G_1\ \mathsf{then}\ G_2\ \mathsf{else}\ G_3 \urcorner : M}$$

Figure 3.8: Mode rules for compound goals

A conjunction is mode-correct with mode M if each conjunct is mode-correct with mode M_i and M is the sequential combination of the modes M_i using $\triangleright$. A disjunction is mode-correct with mode M if each disjunct is mode-correct with mode M_i and M is the mode obtained by merging the modes M_i of the disjuncts using $\bowtie$. An existential quantification is mode-correct with mode M if the existentially quantified goal is mode-correct with mode M' and M is the mode obtained by removing the existentially quantified variables from M'. A negation is mode-correct with mode M if the negated goal is mode-correct with mode M and M does not further instantiate any variable that is non-local to the goal. An if-then-else goal is mode-correct with mode M if the condition, then and else goals are mode-correct with modes M_1, M_2 and M_3, respectively, M_1 does not further instantiate any variable that is nonlocal to the if-then-else, and $M = (M_1 \triangleright M_2) \bowtie M_3$.

[7] This definition could have used '$=$' instead of '$\sqsubseteq$' which would have been equivalent in this simple mode system since the definition of Mode already requires that $M_{\mathsf{fin}}(v) \preceq M_{\mathsf{init}}(v)$. However, the use of '$\sqsubseteq$' here will become important when we introduce unique modes (see Section 3.2.3) because it allows the inst of a variable to become less unique in a negated goal.

$$\text{CALL}\ \frac{\begin{array}{c}\langle \pi(v'_1,\ldots,v'_n) \leftarrow G, M' \rangle \in \Gamma \\ \forall i \in \{\,1,\ldots,n\,\}\,.\ M_{\mathsf{init}}(v_i) \sqsubseteq M'_{\mathsf{init}}(v'_i) \\ M_{\mathsf{fin}} = M_{\mathsf{init}} \oplus \{\, v_i \mapsto \iota_i \,|\, 1 \le i \le n \ \wedge\ \iota_i = M'_{\mathsf{fin}}(v'_i) \curlywedge M_{\mathsf{init}}(v_i) \,\}\end{array}}{\langle \Gamma, \mathcal{N} \rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner \pi(v_1,\ldots,v_n) \urcorner : M}$$

$$\text{UNIFY-VV}\ \frac{\begin{array}{c}\mathsf{abstract\_unify\_inst}(M_{\mathsf{init}}(v), M_{\mathsf{init}}(v'), \iota) \\ M_{\mathsf{fin}} = M_{\mathsf{init}} \oplus \{\, v \mapsto \iota, v' \mapsto \iota \,\}\end{array}}{\langle \Gamma, \mathcal{N} \rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner v = v' \urcorner : M}$$

$$\text{UNIFY-VF}\ \frac{\begin{array}{c}v \notin \{\, v_1,\ldots,v_n \,\} \\ \iota = M_{\mathsf{init}}(v) \\ \overline{\iota} = \langle M_{\mathsf{init}}(v_1),\ldots,M_{\mathsf{init}}(v_n) \rangle \\ \mathsf{abstract\_unify\_inst\_functor}(\iota, f, \overline{\iota}, \iota', \langle \iota'_1,\ldots,\iota'_n \rangle) \\ M_{\mathsf{fin}} = M_{\mathsf{init}} \oplus \{\, v \mapsto \iota', v_1 \mapsto \iota'_1,\ldots,v_n \mapsto \iota'_n \,\}\end{array}}{\langle \Gamma, \mathcal{N} \rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner v = f(v_1,\ldots,v_n) \urcorner : M}$$

Figure 3.9: Mode rules for atomic goals

Figure 3.9 shows the mode rules for atomic goals, that is unifications and calls. A call $\ulcorner \pi(\overline{v}) \urcorner$ is mode-correct with mode M if the predicate π has a mode-correct procedure with mode M' such that $M_{\mathsf{init}}(v_i)$ matches $M'_{\mathsf{init}}(v'_i)$ for each pair $\langle v_i, v'_i \rangle$ where v_i is an actual parameter to the call and v'_i is the corresponding formal parameter. For each actual parameter v_i, the final inst is the greatest lower bound $\curlywedge$ of the initial inst of v_i and the final inst of the corresponding formal parameter v'_i. The reason we take the greatest lower bound instead of just taking the final inst of v'_i is to avoid losing any information that may be present in the initial inst of v_i that is not in the final inst for v'_i. For example, the final inst of v'_i may not know that the initial inst of v_i prohibits it from being bound to a particular function symbol.

A variable-variable unification $\ulcorner v = v' \urcorner$ is mode-correct if the final instmap is the same as the initial instmap except that the inst of v and v' is the inst resulting from the abstract unification operation.

Similarly, a variable-functor unification $\ulcorner v = f(\overline{v}) \urcorner$ is mode-correct if the final instmap is the same as the initial instmap except that each variable involved in the unification has an inst as computed by abstract_unify_inst_functor. Note that if $v \in \overline{v}$ then we generate a mode error. This is effectively performing the occur check at compile time. We could allow such unifications and simply generate code for them that always fails. However, such a unification is not likely to be useful in practice and is very likely to be a programmer error so the most useful thing to do is to report a mode error to alert the programmer to the problem. This check, plus the fact that we don't allow the construction of terms containing free variables, means that we can safely avoid *all* run time occur checks.

We can categorise unifications according to their modes. A unification $\ulcorner v = v' \urcorner$ where v is free before the unification is called an *assignment* to v. If both variables are bound before the unification then it is a *complicated unification* which the Mercury compiler will replace with a call to a unification procedure specialised for the type and mode of its arguments. A unification $\ulcorner v = f(\overline{v}) \urcorner$ is a *construction* if v is free before the unification, and a *deconstruction* otherwise. Each of these different kinds of unification has an efficient specialised implementation in the Mercury abstract machine [39, 131]. Thus, mode analysis gives us the information required to avoid needing to use a general unification algorithm.

Example 3.1.5. As an example of our simple mode system, Figure 3.10 shows the abstract syntax for '`append/3`' (which we saw in Figure 3.2 on page 31) annotated with mode information for the mode '`append(in, in, out)`'. Given an initial instmap of $\{\,\mathtt{Xs} \mapsto \mathsf{ground}, \mathtt{Ys} \mapsto \mathsf{ground}, \mathtt{Zs} \mapsto \mathsf{free}\,\}$, we show, for each sub-goal, the variables whose insts have changed across that subgoal and the final insts for those variables. Note that the conjunction in the second disjunct is in a different order from that given in Figure 3.2. This re-ordering is necessary for '`append/3`' to be mode correct in this particular mode. We discuss this issue further in Section 3.3. ◁

$$
\begin{array}{ll}
\mathtt{append}(\mathtt{Xs}, \mathtt{Ys}, \mathtt{Zs}) \leftarrow & \\
\quad \bigvee\langle & \\
\quad\quad \bigwedge\langle & \\
\quad\quad\quad \mathtt{Xs} = \mathtt{[]}, & \{\,\mathtt{Xs} \mapsto \mathsf{bound}(\{\,\mathtt{[]}\,\})\,\} \\
\quad\quad\quad \mathtt{Ys} = \mathtt{Zs} & \{\,\mathtt{Zs} \mapsto \mathsf{ground}\,\} \\
\quad\quad \rangle, & \{\,\mathtt{Xs} \mapsto \mathsf{bound}(\{\,\mathtt{[]}\,\}), \mathtt{Zs} \mapsto \mathsf{ground}\,\} \\
\quad\quad \exists\{\,\mathtt{Xs0}, \mathtt{Zs0}, \mathtt{X}\,\}.(& \\
\quad\quad\quad \bigwedge\langle & \\
\quad\quad\quad\quad \mathtt{Xs} = \mathtt{[X\ |\ Xs0]}, & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\{\,\mathtt{[}\mathsf{ground}\,|\,\mathsf{ground}\mathtt{]}\,\}), \mathtt{X} \mapsto \mathsf{ground}, \\ \mathtt{Xs0} \mapsto \mathsf{ground} \end{array} \right\} \\
\quad\quad\quad\quad \mathtt{append}(\mathtt{Xs0}, \mathtt{Ys}, \mathtt{Zs0}), & \{\,\mathtt{Zs0} \mapsto \mathsf{ground}\,\} \\
\quad\quad\quad\quad \mathtt{Zs} = \mathtt{[X\ |\ Zs0]} & \{\,\mathtt{Zs} \mapsto \mathsf{bound}(\{\,\mathtt{[}\mathsf{ground}\,|\,\mathsf{ground}\mathtt{]}\,\})\,\} \\
\quad\quad\quad \rangle & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\{\,\mathtt{[}\mathsf{ground}\,|\,\mathsf{ground}\mathtt{]}\,\}), \mathtt{X} \mapsto \mathsf{ground}, \\ \mathtt{Xs0} \mapsto \mathsf{ground}, \mathtt{Zs0} \mapsto \mathsf{ground}, \\ \mathtt{Zs} \mapsto \mathsf{bound}(\{\,\mathtt{[}\mathsf{ground}\,|\,\mathsf{ground}\mathtt{]}\,\}) \end{array} \right\} \\
\quad\quad) & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\{\,\mathtt{[}\mathsf{ground}\,|\,\mathsf{ground}\mathtt{]}\,\}), \\ \mathtt{Zs} \mapsto \mathsf{bound}(\{\,\mathtt{[}\mathsf{ground}\,|\,\mathsf{ground}\mathtt{]}\,\}) \end{array} \right\} \\
\quad \rangle & \{\,\mathtt{Xs} \mapsto \mathsf{bound}(\{\,\mathtt{[]}, \mathtt{[}\mathsf{ground}\,|\,\mathsf{ground}\mathtt{]}\,\}), \mathtt{Zs} \mapsto \mathsf{ground}\,\}
\end{array}
$$

Figure 3.10: Abstract syntax for predicate '`append/3`' with mode annotations

We have now completely defined our simplified mode system. In the next section we will show how this can be expanded into the full Mercury mode system.

3.2 The Full Mercury Mode System

In this section we will gradually extend the simple mode system defined above until we are able to describe the full Mercury mode system. To do this, it will be necessary to redefine many of the types, relations, functions and rules defined above. When we refer to one of these in the text, we will be referring to the most recent version defined.

3.2.1 Using Liveness Information

In the simple mode system, it is not possible to have a unification where the resulting inst is not ground. This is because this system does not keep track of aliases between variables so if two variables are aliased by a unification and one later becomes further instantiated, the mode system will not know that the other variable has become more instantiated too. However, if we know that one of the variables involved in the unification will not be used again after the unification then the unification can be allowed.

Definition 3.2.1 (liveness) A variable is *live* at a given program point if it may occur at some later point reachable from that point by forward execution. $\triangleleft$

We will define the notion of liveness more precisely by extending the mode rules. We then use this to improve the abstract unification function. Liveness information will also be useful for the uniqueness analysis presented in Section 3.2.3.

We add a new component $\mathcal{L} \subseteq \mathsf{Var}$ to the environment. The set $\mathcal{L}$ is the set of variables that are live *after* success of the current goal. We also allow $\mathcal{L}$ to be used as a Boolean function $\mathcal{P}\,\mathsf{Var} \to \mathsf{Bool}$ which returns 1 iff all the variables in the argument set are live.

$$\mathcal{L}(V) = \begin{cases} 1 & \text{if } V \subseteq \mathcal{L}, \\ 0 & \text{otherwise} \end{cases}$$

We now redefine the mode rules to calculate the liveness set which is used in the rules for unification.

Figure 3.11 shows the mode rules for a procedure. The liveness set for the body of the procedure is the set of procedure arguments, which is equivalent to the set of unquantified variables in the body goal.

$$\text{PROC}\ \frac{\begin{array}{c} R = \langle \pi(\overline{v}) \leftarrow G, M \rangle \\ \langle \Gamma, \mathcal{N}, \mathcal{L} \rangle \overset{\mathsf{Goal}}{\vdash} G : M' \\ \forall v \in \overline{v}.\ M_{\mathsf{init}}(v) \sqsubseteq M'_{\mathsf{init}}(v) \\ \forall v \in \overline{v}.\ M'_{\mathsf{fin}}(v) \sqsubseteq M_{\mathsf{fin}}(v) \\ \operatorname{dom} M = \operatorname{dom} M' = \overline{v} = \mathcal{L} = \mathcal{N} = \mathsf{uq}(G) \\ R \in \Gamma \end{array}}{\Gamma \overset{\mathsf{Proc}}{\vdash} R}$$

Figure 3.11: Mode rule for a procedure with liveness information

Figure 3.12 on the next page shows the mode rules for compound goals. Most goals just pass on the liveness set to their sub-goals. The exception is for conjunctions and if-then-elses where the liveness set for a sub-goal also includes the unquantified variables of all other sub-goals which follow that sub-goal on forward execution.

Figure 3.13 on page 45 shows the mode rules for atomic goals. The rule for calls is unchanged from the previous section. The rules for unifications pass the required liveness information to abstract_unify_inst and abstract_unify_inst_functor.

$$\text{CONJ}\ \dfrac{\begin{array}{c}\left.\begin{array}{c}\langle\Gamma,\mathcal{N}_i,\mathcal{L}_i\rangle \overset{\mathsf{Goal}}{\vdash} G_i : M_i \\ \mathcal{N}_i = \mathsf{uq}(G_i) \\ \mathcal{L}_i = \mathcal{L} \cup \bigcup_{j=i+1}^{n} \mathsf{uq}(G_j)\end{array}\right|_{i=1}^{n} \\ M = M_1 \triangleright \cdots \triangleright M_n\end{array}}{\langle\Gamma,\mathcal{N},\mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner \bigwedge \langle G_1,\ldots,G_n\rangle \urcorner : M} \qquad \text{SOME}\ \dfrac{\begin{array}{c}\langle\Gamma,\mathcal{N}',\mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} G : M' \\ \mathcal{N}' = \mathcal{N} \setminus V \\ M = M' \ominus V\end{array}}{\langle\Gamma,\mathcal{N},\mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner \exists V.G \urcorner : M}$$

$$\text{DISJ}\ \dfrac{\begin{array}{c}\left.\begin{array}{c}\langle\Gamma,\mathcal{N}_i,\mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} G_i : M_i \\ \mathcal{N}_i = \mathsf{uq}(G_i)\end{array}\right|_{i=1}^{n} \\ M = M_1 \bowtie \cdots \bowtie M_n\end{array}}{\langle\Gamma,\mathcal{N},\mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner \bigvee \langle G_1,\ldots,G_n\rangle \urcorner : M} \qquad \text{NOT}\ \dfrac{\begin{array}{c}\langle\Gamma,\mathcal{N},\mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} G : M \\ \mathsf{nobind}(M,\mathcal{N})\end{array}}{\langle\Gamma,\mathcal{N},\mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner \neg G \urcorner : M}$$

$$\text{ITE}\ \dfrac{\begin{array}{c}\left.\begin{array}{c}\langle\Gamma,\mathcal{N}_i,\mathcal{L}_i\rangle \overset{\mathsf{Goal}}{\vdash} G_i : M_i \\ \mathcal{N}_i = \mathsf{uq}(G_i)\end{array}\right|_{i=1}^{3} \\ \mathcal{L}_1 = \mathcal{L} \cup \mathsf{uq}(G_2) \\ \mathcal{L}_2 = \mathcal{L}_3 = \mathcal{L} \\ \mathsf{nobind}(M_1,\mathcal{N}) \\ M = (M_1 \triangleright M_2) \bowtie M_3\end{array}}{\langle\Gamma,\mathcal{N},\mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner \mathsf{if}\ G_1\ \mathsf{then}\ G_2\ \mathsf{else}\ G_3 \urcorner : M}$$

Figure 3.12: Mode rules for compound goals with liveness information

Definition 3.2.2 (abstract inst unification with liveness)

$$\mathsf{abstract\_unify\_inst}(\ell,\iota_1,\iota_2,\iota) \iff \iota = \iota_1 \curlywedge \iota_2 \ \wedge\ (\ell = 1 \Rightarrow \iota \preceq \mathsf{ground})$$

The only change to $\mathsf{abstract\_unify\_inst}$ is that the resulting inst is now only required to be ground if both of the variables are live. ◁

Definition 3.2.3 (abstract inst-functor unification)

$$\begin{aligned}&\mathsf{abstract\_unify\_inst\_functor}(\ell,\langle\ell_1,\ldots,\ell_n\rangle,\iota,f,\langle\iota_1,\ldots,\iota_n\rangle,\iota',\langle\iota'_1,\ldots,\iota'_n\rangle) \iff \\ &\quad \iota'' = \iota \curlywedge \mathsf{bound}(\{\ f(\langle\iota_n,\ldots,\iota_n\rangle)\ \}) \wedge \\ &\quad \Big(\big(\iota'' = \mathsf{not\_reached} \ \wedge\ \iota' = \mathsf{not\_reached} \ \wedge\ \forall i \in \{\ 1,\ldots,n\ \}\ .\ \iota'_i = \mathsf{not\_reached}\big) \\ &\quad \vee \big(\iota'' = \mathsf{bound}(\{\ f(\langle\iota''_1,\ldots,\iota''_n\rangle)\ \}) \ \wedge\ \forall i \in \{\ 1,\ldots,n\ \}\ .\ \mathsf{abstract\_unify\_inst}(\ell \wedge \ell_i,\iota''_i,\iota_i,\iota'_i) \wedge \\ &\qquad \iota' = \mathsf{bound}(\{\ f(\langle\iota'_1,\ldots,\iota'_n\rangle)\ \})\big)\Big)\end{aligned}$$

First we calculate ι'' as an approximation to the final inst (which effectively assumes that $\ell = 0$). If $\iota'' = \mathsf{not\_reached}$ then the final inst and final argument insts are all $\mathsf{not\_reached}$. If $\iota'' = \mathsf{bound}(\{\ f(\overline{i''})\ \})$ then we need to compute the final argument insts by abstractly unifying the initial argument insts with the corresponding argument insts of ι''. ◁

$$\text{CALL}\ \frac{\begin{array}{c}\langle \pi(v'_1,\ldots,v'_n) \leftarrow G, M'\rangle \in \Gamma \\ \forall i \in \{\,1,\ldots,n\,\}\,.\ M_{\mathsf{init}}(v_i) \sqsubseteq M'_{\mathsf{init}}(v'_i) \\ M_{\mathsf{fin}} = M_{\mathsf{init}} \oplus \{\, v_i \mapsto \iota_i \,|\, 1 \le i \le n \ \wedge\ \iota_i = M'_{\mathsf{fin}}(v'_i) \curlywedge M_{\mathsf{init}}(v_i) \,\}\end{array}}{\langle \Gamma, \mathcal{N}, \mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner \pi(v_1,\ldots,v_n) \urcorner : M}$$

$$\text{UNIFY-VV}\ \frac{\begin{array}{c}\mathsf{abstract\_unify\_inst}(\mathcal{L}(\{\,v, v'\,\}), M_{\mathsf{init}}(v), M_{\mathsf{init}}(v'), \iota) \\ M_{\mathsf{fin}} = M_{\mathsf{init}} \oplus \{\, v \mapsto \iota, v' \mapsto \iota \,\}\end{array}}{\langle \Gamma, \mathcal{N}, \mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner v = v' \urcorner : M}$$

$$\text{UNIFY-VF}\ \frac{\begin{array}{c}v \notin \{\, v_1,\ldots,v_n \,\} \\ \iota = M_{\mathsf{init}}(v) \\ \overline{\iota} = \langle M_{\mathsf{init}}(v_1),\ldots,M_{\mathsf{init}}(v_n)\rangle \\ \ell = \mathcal{L}(\{\,v\,\}) \\ \overline{\ell} = \langle \mathcal{L}(\{\,v_1\,\}),\ldots,\mathcal{L}(\{\,v_n\,\})\rangle \\ \mathsf{abstract\_unify\_inst\_functor}(\ell, \overline{\ell}, \iota, f, \overline{\iota}, \iota', \langle \iota'_1,\ldots,\iota'_n\rangle) \\ M_{\mathsf{fin}} = M_{\mathsf{init}} \oplus \{\, v \mapsto \iota', v_1 \mapsto \iota'_1,\ldots,v_n \mapsto \iota'_n \,\}\end{array}}{\langle \Gamma, \mathcal{N}, \mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner v = f(v_1,\ldots,v_n) \urcorner : M}$$

Figure 3.13: Mode rules for atomic goals with liveness information

In our simple mode system we said that disallowing unifications of the form $v = f(\overline{v})$ where $v \in \overline{v}$ and not allowing unifications where the result is a non-ground term were sufficient conditions for us to safely omit the occur check at run time. We have now weakened the latter of these conditions by allowing the construction of a term containing a unbound variable if the variable is not live after the construction. This is still a sufficient condition to safely omit run time occur checks because if the unbound variable is not live then it cannot occur in any further unifications.

3.2.2 Dynamic Modes

Mercury is a strongly-moded logic programming language. However, sometimes we would like to trade off strong modes for more flexibility and/or expressiveness. For example, when doing constraint logic programming [73, 90] it is generally not possible for a mode analyser to know the exact instantiation state of constrained variables at every point in the program.

For such cases, Mercury allows a form of optional dynamic modes by introducing an instantiation state any. The definition of instantiation states is now as shown in Figure 3.14.

Instantiation state (Inst)	ι	$::=$	free
		$\mid$	any
		$\mid$	$\mathsf{bound}(\mathcal{P}\, f(\overline{\iota}))$

Figure 3.14: Instantiation state with any inst

With the addition of the any inst, the Mercury mode system resembles the mode system of the constraint logic programming language HAL [55] with any corresponding to HAL's old, and free corresponding to HAL's new. Indeed, Mercury's any inst is used by the HAL implementation, which targets Mercury. The any inst was introduced into Mercury because it was needed by Mercury clp(R) interface, which was a predecessor of the HAL project.

Definition 3.2.4 (γ function with any *insts)* The concretisation function γ is extended with a case for any:

$$\begin{aligned}\gamma(\mathsf{free}) &= \{\ \_\ \}\\ \gamma(\mathsf{any}) &= \mathsf{Term}\\ \gamma(\mathsf{bound}(B)) &= \bigcup_{f(\iota_1,\ldots,\iota_n)\in B} \{\, f(t_1,\ldots,t_n) \,|\, \forall j \in \{\,1,\ldots,n\,\}\,.\; t_j \in \gamma(\iota_j)\,\}\end{aligned}$$

The inst any approximates any term. ◁

Definition 3.2.5 (instantiatedness partial order with any *insts)* The $\preceq$ partial order is extended for any insts:

$$\begin{aligned}&\iota \preceq \mathsf{free}\\ &\mathsf{any} \preceq \mathsf{any}\\ &\mathsf{bound}(B) \preceq \mathsf{any} \quad\iff\quad \forall f(\iota_1,\ldots,\iota_n) \in B.\ \forall i \in \{\,1,\ldots,n\,\}\,.\ \iota_i \preceq \mathsf{any}\\ &\mathsf{bound}(B) \preceq \mathsf{bound}(B') \quad\iff\\ &\qquad \forall \beta \in B.\ \exists \beta' \in B'.\ \beta = f(\iota_1,\ldots,\iota_n)\ \wedge\ \beta' = f(\iota'_1,\ldots,\iota'_n)\ \wedge\\ &\qquad \forall i \in \{\,1,\ldots,n\,\}\,.\ \iota_i \preceq \iota'_i\end{aligned}$$

An inst that contains no free components is at least as instantiated as any. ◁

We allow $\mathsf{any} \preceq \mathsf{any}$ because we only use the partial order for describing how the inst of a variable changes on forward execution. The rule $\mathsf{any} \preceq \mathsf{any}$ is safe in this context, although it is not safe for arbitrary pairs of insts described by any.

Definition 3.2.6 (abstract inst unification with any*)*

$$\mathsf{abstract\_unify\_inst}(\ell, \iota_1, \iota_2, \iota) \iff \iota = \iota_1 \curlywedge \iota_2\ \wedge\ (\ell = 1 \Rightarrow \iota \preceq \mathsf{any})$$

Instead of restricting the result of a live unification to be $\preceq \mathsf{ground}$, we now restrict it to be $\preceq \mathsf{any}$. It is still not possible to have any free components in the result of a unification between two live variables. ◁

Definition 3.2.7 (matches partial order with any *insts)* The $\sqsubseteq$ partial order is extended for any insts:

$$\begin{aligned}&\mathsf{free} \sqsubseteq \mathsf{free}\\ &\mathsf{not\_reached} \sqsubseteq \mathsf{free}\\ &\mathsf{any} \sqsubseteq \mathsf{any}\\ &\mathsf{bound}(B) \sqsubseteq \mathsf{any} \quad\iff\quad \forall f(\iota_1,\ldots,\iota_n) \in B.\ \forall i \in \{\,1,\ldots,n\,\}\,.\ \iota_i \sqsubseteq \mathsf{any}\end{aligned}$$

$$\begin{aligned}\mathsf{bound}(B) \sqsubseteq \mathsf{bound}(B') \quad &\iff \\ &\forall \beta \in B.\ \exists \beta' \in B'.\ \beta = f(\iota_1, \ldots, \iota_n) \ \wedge\ \beta' = f(\iota'_1, \ldots, \iota'_n) \ \wedge \\ &\forall i \in \{\, 1, \ldots, n \,\}.\ \iota_i \sqsubseteq \iota'_i \end{aligned}$$ ◁

The only insts that do not match any are free and any bound inst where one of the functors has a free argument. We could potentially allow free $\sqsubseteq$ any, which would make $\langle \mathsf{Inst}, \sqsubseteq \rangle$ a complete lattice. However, variables with inst any will generally require some sort of representation at run time and thus need to be initialised, whereas free variables are represented by uninitialised memory. The mode system could potentially introduce a call to an initialisation predicate when a predicate is called with a free variable in an argument that was expecting an any inst. The HAL mode system does, in fact, introduce such initialisation predicates, however the Mercury mode system does not. More significantly though, allowing free $\sqsubseteq$ any would seriously weaken the mode system because it would allow the merge of any two insts to succeed, possibly producing an any inst (e.g. we would have free $\sqcup$ ground = any). This would mean that dynamic modes could be introduced in places in the code where the programmer does not intend it, rather than only where predicates with dynamic mode declarations are explicitly called. This would prevent the compiler from reporting many common programming errors so we do not believe it is a good idea.

The definition of the abstraction function α does not change, except that it uses the new definition of $\sqsubseteq$.

*Definition 3.2.8 (*nobind *with* any*)* An extra condition must be added to the nobind predicate for any insts:

$$\mathsf{nobind}(M, \mathcal{N}) \quad \iff \quad \forall v \in \mathcal{N}.\ M_{\mathsf{init}}(v) \sqsubseteq M_{\mathsf{fin}}(v) \ \wedge \\ (M_{\mathsf{init}}(v) \text{ does not contain } \mathsf{any})$$

A variable in the set $\mathcal{N}$ may not have an initial inst with any any components. ◁

Since we can't test whether a variable with inst any has become more bound across a goal, we must disallow such variables in negations and the conditions of if-then-elses to ensure soundness. This may be overly restrictive in some cases (and indeed has found to be so in typical HAL programs where any is used quite extensively) so some further analysis may be warranted to allow simple cases where we are able to prove that the variable has not become further instantiated, such as where it is simply unified with a free variable.

It is important to note that although the any inst can be used to subvert Mercury's strong mode system, its use is entirely optional and does not affect variables for which it is not used. That being said, it must also be noted that the programmer is entirely responsible for ensuring that any insts are not used in a way that would cause the program to be unsound with respect to its declarative semantics. Therefore, the use of any will always be a trade-off between expressiveness and the safety of static checking.

3.2.3 Unique Modes

Unique modes keep track of the number of references to an object. This allows the possibility for destructive update if it is known that a variable is the only reference to an object (i.e. it is a *unique* reference). Mercury's unique modes are based on Henderson [65] and are similar to "linear types" [153] or "uniqueness types" in some other languages, such as Concurrent Clean [9, 62, 112].

Uniqueness Annotations

To describe unique modes we introduce the concept of *uniqueness* annotations and attach a uniqueness annotation to each bound inst.

Definition 3.2.9 (uniqueness annotation) Figure 3.15 gives the definition of the set Uniq of uniqueness annotations. $\triangleleft$

Uniqueness annotation (Uniq)	u	$::=$	unique
		$\mid$	$\mathsf{mostly\_unique}$
		$\mid$	shared
		$\mid$	$\mathsf{mostly\_clobbered}$
		$\mid$	$\mathsf{clobbered}$

Figure 3.15: Uniqueness annotations

The annotation unique means that this is the only reference to the corresponding memory cell; $\mathsf{mostly\_unique}$ means that this is the only reference on forward execution, but there may be more references on backtracking; shared means the number of references is unknown, but there is at least one; $\mathsf{clobbered}$ means that the corresponding data may not be accessed (for example, it may have been destructively updated); $\mathsf{mostly\_clobbered}$ means that the corresponding data may not be accessed on forward execution, but may be accessed on backtracking.

Definition 3.2.10 (uniqueness total order) We define a total order $\trianglelefteq$ on uniqueness annotations such that that is

$$\mathsf{unique} \trianglelefteq \mathsf{mostly\_unique} \trianglelefteq \mathsf{shared} \trianglelefteq \mathsf{mostly\_clobbered} \trianglelefteq \mathsf{clobbered}$$

If $u \trianglelefteq u'$ then we say that u is *at least as unique as* u' or that u' is *no more unique than* u. The operators $\vartriangle$ and $\triangledown$ (for pairs) and $\bigtriangleup$ and $\bigtriangledown$ (for sets) represent the greatest lower bound and least upper bound, respectively, under $\trianglelefteq$. $\triangleleft$

The intuition behind this definition is that a value with a uniqueness annotation that is higher in the total order is more restricted than a value with a uniqueness annotation that is lower in the total order, in terms of what can be done with that value. For example, a value with a unique annotation can be used anywhere (including places that will destructively update the value); a value with annotation shared can only be used where a unique reference is not required; and a value with annotation $\mathsf{clobbered}$ may not be used where the value is required to be examined.

Reference Counts

We can define Uniq more formally as an abstraction of a concrete property of terms at runtime — the number of references there are to the memory cell representing each function symbol in a term. We define the concrete domain $\mathsf{RefCount} = \mathcal{P}\,\mathbb{N} \times \mathcal{P}\,\mathbb{N}$ where $\mathbb{N}$ is the set of natural numbers.

We can imagine each function symbol f in a term being annotated with a tuple $\langle F, B\rangle \in \mathsf{RefCount}$ where F represents the number of possible references to the memory cell representing f on forward execution, and B represents the number of possible references when considering both forward and backward execution. We require $|F| \geq 1$ and $|B| \geq 1$. In general, the sets F and B may have more than one element because the number of references will depend on what computation path was taken to reach a particular program point. We also require $\forall n \in F.\ \exists m \in B.\ n \leq m$ and $\neg\exists m \in B.\ m < \min(F)$ because there must be at least as many references on forward execution and backward execution combined as there are on forward execution alone.

We define a concretisation function $\gamma_{\mathsf{u}} : \mathsf{Uniq} \to \mathsf{RefCount}$ and an abstraction function $\alpha_{\mathsf{u}} : \mathsf{RefCount} \to \mathsf{Uniq}$ which give the semantics of uniqueness annotations based on reference counts. These functions are given in Figure 3.16. We use $\mathbb{P} = \{ 1, 2, \dots \}$, the set of positive integers.

$$
\begin{aligned}
\gamma_{\mathsf{u}}(\mathsf{clobbered}) &= \langle \mathbb{N}, \mathbb{N} \rangle \\
\gamma_{\mathsf{u}}(\mathsf{mostly\_clobbered}) &= \langle \mathbb{N}, \mathbb{P} \rangle \\
\gamma_{\mathsf{u}}(\mathsf{shared}) &= \langle \mathbb{P}, \mathbb{P} \rangle \\
\gamma_{\mathsf{u}}(\mathsf{mostly\_unique}) &= \langle \{ 1 \}, \mathbb{P} \rangle \\
\gamma_{\mathsf{u}}(\mathsf{unique}) &= \langle \{ 1 \}, \{ 1 \} \rangle
\end{aligned}
\qquad
\alpha_{\mathsf{u}}(\langle F, B\rangle) = \begin{cases}
\mathsf{clobbered} & \text{if } 0 \in F \wedge 0 \in B \\
\mathsf{mostly\_clobbered} & \text{if } 0 \in F \wedge 0 \notin B \\
\mathsf{unique} & \text{if } F = \{ 1 \} \wedge B = \{ 1 \} \\
\mathsf{mostly\_unique} & \text{if } F = \{ 1 \} \\
\mathsf{shared} & \text{if } F \subseteq \mathbb{P} \wedge B \subseteq \mathbb{P}
\end{cases}
$$

Figure 3.16: Concretisation and abstractions functions for uniqueness annotations

We define a partial order $\leq_{\mathsf{R}}$ on reference counts

$$\langle F, B\rangle \leq_{\mathsf{R}} \langle F', B'\rangle \iff F \subseteq F' \ \wedge\ B \subseteq B'$$

To annotate a function symbol f with a reference count $R \in \mathsf{RefCount}$ we write f^R. We use the notation t^R where $t \in \mathsf{Term}$ and $R \in \mathsf{RefCount}$ to mean the set of terms obtained by annotating every function symbol in t with some reference count, as follows:

$$
\begin{aligned}
(v)^R &= \{ v \} \\
\big(f(t_1, \dots, t_n)\big)^R &= \left\{ f^{R'}(r_1, \dots, r_n) \,\middle|\, R \leq_{\mathsf{R}} R' \ \wedge\ \forall i \in \{ 1, \dots, n \} \,.\, r_i \in t_i^{R'} \right\}
\end{aligned}
$$

we require the condition $R' \leq_{\mathsf{R}} R$ because a sub-term must have at least as many references as its parent term. Note that we do not record reference count annotations on variables. Variables may not be aliased so we implicitly assume that each variable must have a reference count of $\langle \{ 1 \}, \{ 1 \} \rangle$.

If $T \subseteq \mathsf{Term}$ then we define

$$T^R = \bigcup_{t \in T} t^R$$

We define the set RTerm of annotated terms by

$$\mathsf{RTerm} = \bigcup_{R \in \mathsf{RefCount}} \mathsf{Term}^R$$

Given the uniqueness concretisation function γ_{u} and the partial order $\leq_{\mathsf{R}}$ on reference counts, an alternative definition of the partial order $\trianglelefteq$ on uniqueness annotations is

$$u \trianglelefteq u' \iff \gamma_{\mathsf{u}}(u) \leq_{\mathsf{R}} \gamma_{\mathsf{u}}(u')$$

We can see that a uniqueness annotation lower in the total order has more information about the possible number of references to the value. For a value with a uniqueness annotation of shared, we know that there is at least one reference to the value (i.e. the current reference) and so we may access the value, but we may not use it in a way that assumes that this is the only reference, e.g. we may not re-use its memory. For a value with a uniqueness annotation of unique, we know that this is the only reference to the value. If we have a unique reference to a variable at the start of an atomic goal and the variable is not live after that goal, then its forward execution reference count goes from $\{ 1 \}$ to $\{ 0 \}$ and we may re-use the memory holding that value. If the memory of a variable has been re-used on any computation path then the rest of the system must assume that its original content has been clobbered. This is why we map any set of reference counts including 0 to the uniqueness $\mathsf{clobbered}$ which means we must avoid using the variable in any situation where its value may be required.

Instantiation States

Definition 3.2.11 (instantiation state) Figure 3.17 gives the new definition of instantiation states which includes uniqueness annotations. The possible uniqueness annotations are constrained so that no bound inst is more unique than any of its parents; this is necessary because a subterm must have at least as many references to it as the term in which it is contained. More precisely,

$$\mathsf{Inst} = \mathsf{Inst}(\mathsf{unique})$$

where

$$\begin{aligned} \mathsf{Inst}(u) = {} & \{ \mathsf{free} \} \cup \{ \mathsf{any}(u') \mid u \trianglelefteq u' \} \cup \\ & \{ \mathsf{bound}(u', B) \mid u \trianglelefteq u' \ \wedge\ \forall f(\iota_1, \ldots, \iota_n) \in B.\ \{ \iota_1, \ldots, \iota_n \} \subseteq \mathsf{Inst}(u') \} \end{aligned} \qquad \triangleleft$$

Instantiation state (Inst)	ι	$::=$	free
		$\mid$	$\mathsf{any}(u)$
		$\mid$	$\mathsf{bound}(u, \mathcal{P}\, f(\iota_1, \ldots, \iota_n))$
			where each f must occur at most once,
			and the arguments ι_i must be no more
			unique than u.

Figure 3.17: Instantiation states with uniqueness annotations

Note that there is no uniqueness annotation on the free inst. This is because free insts have an implicit annotation of unique — a free variable may not have any other references to it (i.e. it may not be aliased).

Definition 3.2.12 (concretisation function γ) With our new definition of insts with uniqueness annotations, the definition of the concretisation function γ must be modified to return a set of terms annotated with reference counts: $\gamma : \mathsf{Inst} \to \mathcal{P}\,\mathsf{RTerm}$.

$$\begin{aligned}
\gamma(\mathsf{free}) &= \{\ _\ \} \\
\gamma(\mathsf{any}(u)) &= \mathsf{Term}^{\gamma_{\mathsf{u}}(u)} \\
\gamma(\mathsf{bound}(u, B)) &= \bigcup_{f(\iota_1,\ldots,\iota_n)\in B} \left\{\ \big(f(t_1,\ldots,t_n)\big)\ \big|\ \forall j \in \{\,1,\ldots,n\,\}\,.\ t_j \in \gamma(\iota_j)\ \right\}^{\gamma_{\mathsf{u}}(u)}
\end{aligned} \qquad \triangleleft$$

Definition 3.2.13 (uniqueness function) We define a partial function $\mathsf{uniqueness} : \mathsf{Inst} \to \mathsf{Uniq}$ which returns the uniqueness annotation of the top-level functor of an instantiation state.

$$\begin{aligned}
\mathsf{uniqueness}(\mathsf{any}(u)) &= u \\
\mathsf{uniqueness}(\mathsf{bound}(u, B)) &= u
\end{aligned}$$

The function is undefined for the inst free. $\triangleleft$

Definition 3.2.14 (instantiatedness partial order with uniqueness) The partial order $\preceq$ over instantiation states is modified to the following:

$$\begin{aligned}
&\iota \preceq \mathsf{free} \\
&\mathsf{any}(u) \preceq \mathsf{any}(u') \iff u' \trianglelefteq u \\
&\mathsf{bound}(u, B) \preceq \mathsf{any}(u') \iff \\
&\qquad \forall f(\iota_1,\ldots,\iota_n) \in B.\ \forall i \in \{\,1,\ldots,n\,\}\,.\ \iota_i \preceq \mathsf{any}(u') \ \wedge\ u' \trianglelefteq u \\
&\mathsf{bound}(u, B) \preceq \mathsf{bound}(u', B') \iff \\
&\qquad \forall \beta \in B.\ \exists \beta' \in B'.\ \beta = f(\iota_1,\ldots,\iota_n) \ \wedge\ \beta' = f(\iota'_1,\ldots,\iota'_n) \ \wedge \\
&\qquad \forall i \in \{\,1,\ldots,n\,\}\,.\ \iota_i \preceq \iota'_i \ \wedge\ u' \trianglelefteq u
\end{aligned}$$

That is, the definition from Section 3.1 is augmented with the extra requirement that if neither inst is free then the first must be no more unique than the second. This is because, in this mode system, a variable may not become more unique on forward execution. $\triangleleft$

Definition 3.2.15 (matches partial order with uniqueness) The partial order $\sqsubseteq$ over instantiation states is modified as follows:

$$\begin{aligned}
&\mathsf{free} \sqsubseteq \mathsf{free} \\
&\mathsf{not_reached}(u) \sqsubseteq \mathsf{free} \\
&\mathsf{any}(u) \sqsubseteq \mathsf{any}(u') \iff u \trianglelefteq u' \\
&\mathsf{bound}(u, B) \sqsubseteq \mathsf{any}(u') \iff \\
&\qquad \forall f(\iota_1,\ldots,\iota_n) \in B.\ \forall i \in \{\,1,\ldots,n\,\}\,.\ \iota_i \sqsubseteq \mathsf{any}(u') \ \wedge\ u \trianglelefteq u' \\
&\mathsf{bound}(u, B) \sqsubseteq \mathsf{bound}(u', B') \iff \\
&\qquad \forall \beta \in B.\ \exists \beta' \in B'.\ \beta = f(\iota_1,\ldots,\iota_n) \ \wedge\ \beta' = f(\iota'_1,\ldots,\iota'_n) \ \wedge \\
&\qquad \forall i \in \{\,1,\ldots,n\,\}\,.\ \iota_i \sqsubseteq \iota'_i \ \wedge\ u \trianglelefteq u'
\end{aligned}$$

That is, the definition from Section 3.1 is augmented with the extra requirement that if neither inst is free then the first must be at least as unique than the second. This is because it is okay for a variable to be more unique in the initial inst of the mode declaration than the initial inst of the procedure body requires, and it is okay for a variable to be more unique in the final inst of the procedure body than the final inst of the mode declaration requires. Similar considerations apply for calls. ◁

Note that the uniqueness ordering on $\sqsubseteq$ is the reverse of that for $\preceq$ so the relation $\sqsubseteq$ is no longer a subset of $\preceq$.

Definition 3.2.16 (abstraction function α) The abstraction function must also operate on sets of terms annotated with reference counts, i.e. $\alpha : \mathcal{P}\,\mathsf{RTerm} \rightarrow \mathsf{Inst}$.

$$\alpha(T) = \bigsqcup_{t \in T} \alpha'(t)$$

where

$$\begin{aligned} \alpha'(\_) &= \mathsf{free} \\ \alpha'(f^R(t_1, \ldots, t_n)) &= \mathsf{bound}(\alpha_{\mathsf{u}}(R), \{\, f(\alpha'(t_1), \ldots, \alpha'(t_n)) \,\}) \end{aligned}$$ ◁

Definition 3.2.17 (ground) We define an inst ground parameterised on uniqueness:

$$\mathsf{ground}(u) = \mathsf{bound}\left(u, \left\{ f(\underbrace{\mathsf{ground}(u), \ldots, \mathsf{ground}(u)}_{n \text{ times}}) \;\middle|\; f/n \in \Sigma \right\}\right)$$

By defining $\mathsf{ground} = \mathsf{ground}(\mathsf{unique})$ we can say that an inst ι is ground if and only if $\iota \preceq \mathsf{ground}$. ◁

Definition 3.2.18 (not reached) Likewise, we define

$$\begin{aligned} \mathsf{not\_reached}(u) &= \mathsf{bound}(u, \emptyset) \\ \mathsf{not\_reached} &= \mathsf{not\_reached}(\mathsf{unique}) \end{aligned}$$

An inst ι is not reached if and only if $\iota \preceq \mathsf{not\_reached}$. ◁

Mode Rules

The mode rules for goals are unchanged apart from being based on the updated definitions of abstract_unify_inst and abstract_unify_inst_functor given below. However the initial liveness set for the procedure body can be more precise, since any variable that has a uniqueness annotation of clobbered or mostly_clobbered at its top level must not be live after the procedure succeeds. The changed procedure rule is shown in Figure 3.18 on the next page.

The predicates abstract_unify_inst and abstract_unify_inst_functor are modified as shown below.

$$\text{PROC}\ \frac{\begin{array}{c} R = \langle \pi(\overline{v}) \leftarrow G, M \rangle \\ \langle \Gamma, \mathcal{N}, \mathcal{L} \rangle \overset{\mathsf{Goal}}{\vdash} G : M' \\ \forall v \in \overline{v}.\ M_{\mathsf{init}}(v) \sqsubseteq M'_{\mathsf{init}}(v) \\ \forall v \in \overline{v}.\ M'_{\mathsf{fin}}(v) \sqsubseteq M_{\mathsf{fin}}(v) \\ \operatorname{dom} M = \operatorname{dom} M' = \overline{v} = \mathcal{N} = \mathsf{uq}(G) \\ \mathcal{L} = \{\, v \in \overline{v} \mid \neg\, \mathsf{mostly\_clobbered} \trianglelefteq \mathsf{uniqueness}(M_{\mathsf{fin}}(v)) \,\} \\ R \in \Gamma \end{array}}{\Gamma \overset{\mathsf{Proc}}{\vdash} R}$$

Figure 3.18: Mode rule for a procedure with unique modes

Definition 3.2.19 (abstract inst unification with uniqueness)

$$\begin{aligned} \mathsf{abstract\_unify\_inst}(1, \iota_1, \iota_2, \iota \curlywedge \mathsf{any}(\mathsf{shared})) &\iff \iota = \iota_1 \curlywedge \iota_2 \ \wedge\ \iota \preceq \mathsf{any}(\mathsf{unique}) \\ \mathsf{abstract\_unify\_inst}(0, \iota_1, \iota_2, \iota) &\iff \iota = \iota_1 \curlywedge \iota_2 \ \wedge\ \neg\ \mathsf{semidet\_clobbered\_unify}(\iota_1, \iota_2) \end{aligned}$$

where

$$\begin{aligned} &\mathsf{semidet\_clobbered\_unify}(\iota, \iota') \iff \\ &\quad \iota \neq \mathsf{free} \ \wedge\ \iota' \neq \mathsf{free} \ \wedge\ (|\gamma(\iota)| > 1 \ \vee\ |\gamma(\iota')| > 1) \ \wedge \\ &\quad (\mathsf{mostly\_clobbered} \trianglelefteq \mathsf{uniqueness}(\iota) \ \vee\ \mathsf{mostly\_clobbered} \trianglelefteq \mathsf{uniqueness}(\iota')) \end{aligned}$$ ◁

If the unification is between two live variables then the result must be shared since there are now multiple live references to the value. We achieve this by taking the glb of the two input insts (ι_1 and ι_2) and $\mathsf{any}(\mathsf{shared})$ after first checking that $\iota_1 \curlywedge \iota_2$ contains no free components.

If at least one of the variables is not live the result is just the greatest lower bound of the two insts, with no loss of uniqueness since we are not creating any extra live references to the value. However, we must make the restriction that if any part of the insts is clobbered, then the unification must not be able to fail (i.e. is not semidet). The reason for this is that the values of variables with clobbered insts are not available, but if the unification can possibly fail then it must examine the values of the variables.

Definition 3.2.20 (abstract inst-functor unification with uniqueness)

$$\begin{aligned} &\mathsf{abstract\_unify\_inst\_functor}(\ell, \langle \ell_1, \ldots, \ell_n \rangle, \iota, f, \langle \iota_1, \ldots, \iota_n \rangle, \iota', \langle \iota'_1, \ldots, \iota'_n \rangle) \iff \\ &\quad \iota'' = \iota \curlywedge \mathsf{bound}(\mathsf{unique}, \{\, f(\langle \iota_n, \ldots, \iota_n \rangle) \,\}) \ \wedge \\ &\quad \Big(\big(\iota'' = \mathsf{not\_reached} \ \wedge\ \iota' = \mathsf{not\_reached} \ \wedge\ \forall i \in \{\, 1, \ldots, n \,\} \,.\, \iota'_i = \mathsf{not\_reached}\big) \\ &\quad \vee\ \big(\iota'' = \mathsf{bound}(u, \{\, f(\langle \iota''_1, \ldots, \iota''_n \rangle) \,\}) \ \wedge \\ &\qquad \forall i \in \{\, 1, \ldots, n \,\} \,.\, \mathsf{abstract\_unify\_inst}(\ell \wedge \ell_i, \iota''_i, \iota_i, \iota'_i) \ \wedge \\ &\qquad \iota' = \mathsf{bound}(u, \{\, f(\langle \iota'_1, \ldots, \iota'_n \rangle) \,\})\big)\Big) \end{aligned}$$

The reason for the annotation of unique in $\iota \curlywedge \mathsf{bound}(\mathsf{unique}, \{\, f(\langle \iota_n, \ldots, \iota_n \rangle) \,\})$ is that unifications can construct unique terms. If ι is not free then the $\curlywedge$ operation ensures that the resulting inst is no more unique than ι. ◁

Example 3.2.1. Figure 3.19 shows the abstract syntax for '`append/3`' annotated with mode information for the mode '`append(di, di, uo)`'. Given an initial instmap of $\{\, \mathtt{Xs} \mapsto \mathsf{ground}(\mathsf{unique}), \mathtt{Ys} \mapsto \mathsf{ground}(\mathsf{unique}), \mathtt{Zs} \mapsto \mathsf{free} \,\}$, we show, for each sub-goal, the variables whose insts have changed across that subgoal and the final insts for those variables. The mode declaration gives final inst for `Xs` and `Ys` of $\mathsf{ground}(\mathsf{clobbered})$. This gives the mode checker the information that these two variables are not live after the unifications in which they appear. For example, after the `Ys = Zs` unification, we know that `Ys` is dead. This means that the resulting inst is $\mathsf{ground}(\mathsf{unique})$ rather than $\mathsf{ground}(\mathsf{shared})$ which it would be if `Ys` was still live. Note that in the final inferred instmap `Xs` and `Ys` still have inst $\mathsf{ground}(\mathsf{unique})$. This is okay because $\mathsf{ground}(\mathsf{unique}) \sqsubseteq \mathsf{ground}(\mathsf{clobbered})$ so the inferred final insts match the declared final insts. ◁

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{append}(\mathtt{Xs}, \mathtt{Ys}, \mathtt{Zs}) \leftarrow} \\
\quad \bigvee\langle & \\
\qquad \bigwedge\langle & \\
\qquad\quad \mathtt{Xs} = \mathtt{[]}, & \{\, \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{unique}, \{\, \mathtt{[]} \,\}) \,\} \\
\qquad\quad \mathtt{Ys} = \mathtt{Zs} & \{\, \mathtt{Zs} \mapsto \mathsf{ground}(\mathsf{unique}) \,\} \\
\qquad \rangle, & \{\, \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{unique}, \{\, \mathtt{[]} \,\}), \mathtt{Zs} \mapsto \mathsf{ground}(\mathsf{unique}) \,\} \\
\qquad \exists \{\, \mathtt{Xs0}, \mathtt{Zs0}, \mathtt{X} \,\}.(& \\
\qquad\quad \bigwedge\langle & \\
\qquad\qquad \mathtt{Xs = [X \mid Xs0]}, & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{unique}, \{\, \mathtt{[}\mathsf{ground}(\mathsf{unique}) \mid \mathsf{ground}(\mathsf{unique})\mathtt{]} \,\}), \\ \mathtt{X} \mapsto \mathsf{ground}(\mathsf{unique}), \mathtt{Xs0} \mapsto \mathsf{ground}(\mathsf{unique}) \end{array} \right\} \\
\qquad\qquad \mathsf{append}(\mathtt{Xs0}, \mathtt{Ys}, \mathtt{Zs0}), & \{\, \mathtt{Zs0} \mapsto \mathsf{ground}(\mathsf{unique}) \,\} \\
\qquad\qquad \mathtt{Zs = [X \mid Zs0]} & \{\, \mathtt{Zs} \mapsto \mathsf{bound}(\mathsf{unique}, \{\, \mathtt{[}\mathsf{ground}(\mathsf{unique}) \mid \mathsf{ground}(\mathsf{unique})\mathtt{]} \,\}) \,\} \\
\qquad\quad \rangle & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{unique}, \{\, \mathtt{[}\mathsf{ground}(\mathsf{unique}) \mid \mathsf{ground}(\mathsf{unique})\mathtt{]} \,\}), \\ \mathtt{X} \mapsto \mathsf{ground}(\mathsf{unique}), \\ \mathtt{Xs0} \mapsto \mathsf{ground}(\mathsf{unique}), \mathtt{Zs0} \mapsto \mathsf{ground}(\mathsf{unique}), \\ \mathtt{Zs} \mapsto \mathsf{bound}(\mathsf{unique}, \{\, \mathtt{[}\mathsf{ground}(\mathsf{unique}) \mid \mathsf{ground}(\mathsf{unique})\mathtt{]} \,\}) \end{array} \right\} \\
\qquad) & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{unique}, \{\, \mathtt{[}\mathsf{ground}(\mathsf{unique}) \mid \mathsf{ground}(\mathsf{unique})\mathtt{]} \,\}), \\ \mathtt{Zs} \mapsto \mathsf{bound}(\mathsf{unique}, \{\, \mathtt{[}\mathsf{ground}(\mathsf{unique}) \mid \mathsf{ground}(\mathsf{unique})\mathtt{]} \,\}) \end{array} \right\} \\
\quad \rangle & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{unique}, \{\, \mathtt{[]}, \mathtt{[}\mathsf{ground}(\mathsf{unique}) \mid \mathsf{ground}(\mathsf{unique})\mathtt{]} \,\}), \\ \mathtt{Zs} \mapsto \mathsf{ground}(\mathsf{unique}) \end{array} \right\}
\end{array}
$$

Figure 3.19: Abstract syntax for predicate '`append/3`' with unique mode annotations

When mode checking uniqueness annotations, it is also necessary to ensure that the program does not allow variables with a clobbered inst to be referenced on backtracking. This check must be done after determinism analysis [67], which in turn must be done after the main phase of mode analysis which we describe here. (We don't know where backtracking might occur until after determinism analysis and the determinism of a procedure may depend on the modes of the calls it makes.) Determinism analysis and the second phase of unique mode analysis are beyond the scope of this thesis.

Uniqueness annotations declared by the programmer and inferred by mode analysis can be used to implement automatic structure re-use and compile time garbage collection. For a discussion of this see Taylor [138]. Adding explicit uniqueness annotations to mode declarations can be cumbersome. Some work has been done on compile time garbage collection without the need for the programmer to manually add such annotations [91, 92], although this work can make use of such annotations if they exist.

3.2.4 Higher-Order Modes

Figure 3.20 shows the changes required to the abstract syntax and to instantiation states to support Mercury's higher-order programming features. For convenience, we introduce the notion of an *argument mode*, that is a "mode" for a single procedure argument or variable. If $\mu = \iota \rightarrowtail \iota' \in \mathsf{ArgMode}$ then $\iota' \preceq \iota$, $\mu_{\mathsf{init}} = \iota$, and $\mu_{\mathsf{fin}} = \iota'$.

Flattened term (FTerm)	φ	$::=$	v	
		$\mid$	$f(\overline{v})$	
		$\mid$	$\lambda(\overline{v :: \mu}) \leftarrow G$	(higher-order term)
Goal (Goal)	G	$::=$	$\pi(\overline{v})$	(call)
		$\mid$	$v(\overline{v})$	(higher-order call)
		$\mid$	$v = \varphi$	(unification)
		$\mid$	$\exists \mathcal{P}\, v.G'$	(existential quantification)
		$\mid$	$\neg G'$	(negation)
		$\mid$	$\bigwedge \overline{G}$	(conjunction)
		$\mid$	$\bigvee \overline{G}$	(disjunction)
		$\mid$	$\mathsf{if}\ G_1\ \mathsf{then}\ G_2\ \mathsf{else}\ G_3$	(if-then-else)
Instantiation state (Inst)	ι	$::=$	free	
		$\mid$	$\mathsf{any}(u)$	
		$\mid$	$\mathsf{bound}(u, \mathcal{P}\, f(\overline{\iota}))$	
		$\mid$	$\mathsf{higher\_order}(u, \overline{\mu})$	(higher-order inst)
Argument mode (ArgMode)	μ	$::=$	$\iota \rightarrowtail \iota'$	

Figure 3.20: Higher-order Mercury

A higher-order term may be constructed through a unification of the form $v = \lambda(\overline{v :: \mu}) \leftarrow G$. The arguments $v_i :: \mu_i$ are pairs consisting of a *lambda-quantified* variable v_i and an argument mode μ_i. The lambda-quantified variables must be distinct from each other and from the variables outside the higher-order term. The goal G is the body of the higher-order term.

A higher-order call is an atomic goal of the form $v(\overline{v})$ where v is a variable (that must have previously been bound to a higher-order term) and $\overline{v}$ is a sequence of distinct variables.

A higher-order instantiation state has the form $\mathsf{higher\_order}(u, \overline{\mu})$ where u is the uniqueness of the inst and $\overline{\mu}$ is the vector of argument modes.[8]

Definition 3.2.21 (Concretisation with higher-order) The γ function is extended with a case for

[8] Higher-order insts also have an associated determinism annotation. However it is only required for determinism analysis so we omit it for simplicity.

higher-order insts:

$$
\begin{aligned}
&\gamma(\mathsf{free}) = \{\,\_\,\}\\
&\gamma(\mathsf{any}(u)) = \mathsf{Term}^{\gamma_{\mathsf{u}}(u)}\\
&\gamma(\mathsf{bound}(u,B)) = \bigcup_{f(\iota_1,\ldots,\iota_n)\in B} \{\, f(t_1,\ldots,t_n) \,|\, \forall j \in \{\,1,\ldots,n\,\}\,.\; t_j \in \gamma(\iota_j)\,\}^{\gamma_{\mathsf{u}}(u)}\\
&\gamma(\mathsf{higher\_order}(u, \langle \mu_1,\ldots,\mu_n\rangle)) =\\
&\qquad \left\{ \big(\lambda(v_1 :: \mu_1,\ldots,v_n :: \mu_n) \leftarrow G\big) \;\middle|\; \begin{array}{l} \forall j \in \{\,1,\ldots,n\,\}\,.\; v_j \in \mathsf{Var} \,\wedge \\ \mu_j \in \mathsf{ArgMode} \,\wedge\, G \in \mathsf{Goal} \end{array} \right\}^{\gamma_{\mathsf{u}}(u)}
\end{aligned}
$$
◁

Definition 3.2.22 (uniqueness function with higher-order) A case is added to the $\mathsf{uniqueness}$ function for obtaining the uniqueness annotation of a higher-order inst.

$$
\begin{aligned}
\mathsf{uniqueness}(\mathsf{any}(u)) &= u\\
\mathsf{uniqueness}(\mathsf{bound}(u,B)) &= u\\
\mathsf{uniqueness}(\mathsf{higher\_order}(u,\overline{\mu})) &= u
\end{aligned}
$$
◁

Definition 3.2.23 (instantiatedness partial order with higher-order) The instantiatedness partial order $\preceq$ is extended to incorporate higher-order insts:

$$
\begin{aligned}
&\iota \preceq \mathsf{free}\\
&\mathsf{any}(u) \preceq \mathsf{any}(u') \iff u' \trianglelefteq u\\
&\mathsf{bound}(u,B) \preceq \mathsf{any}(u') \iff\\
&\qquad \forall f(\iota_1,\ldots,\iota_n) \in B.\; \forall i \in \{\,1,\ldots,n\,\}\,.\; \iota_i \preceq \mathsf{any}(u') \,\wedge\, u' \trianglelefteq u\\
&\mathsf{higher\_order}(u,\overline{\mu}) \preceq \mathsf{any}(u') \iff u' \trianglelefteq u\\
&\mathsf{bound}(u,B) \preceq \mathsf{bound}(u',B') \iff\\
&\qquad \forall \beta \in B.\; \exists \beta' \in B'.\; \beta = f(\iota_1,\ldots,\iota_n) \,\wedge\, \beta' = f(\iota'_1,\ldots,\iota'_n) \,\wedge\\
&\qquad \forall i \in \{\,1,\ldots,n\,\}\,.\; \iota_i \preceq \iota'_i \,\wedge\, u' \trianglelefteq u\\
&\mathsf{not\_reached}(u) \preceq \mathsf{higher\_order}(u',\overline{\mu}) \iff u' \trianglelefteq u\\
&\mathsf{higher\_order}(u,\overline{\mu}) \preceq \mathsf{ground}(u') \iff u' \trianglelefteq u\\
&\mathsf{higher\_order}(u,\langle \mu_1,\ldots \mu_n\rangle) \preceq \mathsf{higher\_order}(u',\langle \mu'_1,\ldots,\mu'_n\rangle) \iff\\
&\qquad \forall i \in \{\,1,\ldots,n\,\}\,.\; (\mu'_i)_{\mathsf{init}} \preceq (\mu_i)_{\mathsf{init}} \,\wedge\, (\mu_i)_{\mathsf{fin}} \preceq (\mu'_i)_{\mathsf{fin}} \,\wedge\, u' \trianglelefteq u
\end{aligned}
$$
◁

We see from this definition that higher-order insts are ground insts and that two higher-order insts are comparable only if all their corresponding initial and final insts are comparable. When analysing a call to a predicate which has a higher-order argument, the caller may have more specific information about the initial inst of the higher-order argument than the callee requires. We want the $\curlywedge$ operation to preserve this extra information. By defining comparability of higher-order insts in this way we ensure this. Example 3.2.2 on the facing page shows how the greatest lower bound $\curlywedge$ of two higher-order insts is used in the CALL rule.

Comparing two higher-order values for equality is, in general, undecidable (it requires proving that two different formulas in predicate logic are equivalent) so, ideally, we would like the Mercury compiler to generate an error if a program attempts to unify two variables with higher-order insts. The Melbourne Mercury compiler adds this check to its implementation of $\mathsf{abstract\_unify\_inst}$.

Unfortunately, this cannot catch all occurrences of attempting to unify two higher-order values — a variable with a higher-order value does not necessarily have a higher-order inst, e.g. it may have the inst ground. Any undecidable higher-order unifications which are not detected at compile time will be caught at run time.

Definition 3.2.24 (matches partial order with higher-order) We also extend the matches partial order $\sqsubseteq$ to incorporate higher-order insts:

$$
\begin{array}{l}
\mathsf{free} \sqsubseteq \mathsf{free} \\
\mathsf{not\_reached}(u) \sqsubseteq \mathsf{free} \\
\mathsf{any}(u) \sqsubseteq \mathsf{any}(u') \iff u \trianglelefteq u' \\
\mathsf{bound}(u, B) \sqsubseteq \mathsf{any}(u') \iff \\
\qquad \forall f(\iota_1, \ldots, \iota_n) \in B.\ \forall i \in \{\,1, \ldots, n\,\}\ .\ \iota_i \sqsubseteq \mathsf{any}(u') \ \wedge\ u \trianglelefteq u' \\
\mathsf{higher\_order}(u, \overline{\mu}) \sqsubseteq \mathsf{any}(u') \iff u \trianglelefteq u' \\
\mathsf{bound}(u, B) \sqsubseteq \mathsf{bound}(u', B') \iff \\
\qquad \forall \beta \in B.\ \exists \beta' \in B'.\ \beta = f(\iota_1, \ldots, \iota_n) \ \wedge\ \beta' = f(\iota'_1, \ldots, \iota'_n) \ \wedge \\
\qquad \forall i \in \{\,1, \ldots, n\,\}\ .\ \iota_i \sqsubseteq \iota'_i \ \wedge\ u \trianglelefteq u' \\
\mathsf{not\_reached}(u) \sqsubseteq \mathsf{higher\_order}(u', \overline{\mu}) \iff u \trianglelefteq u' \\
\mathsf{higher\_order}(u, \overline{\mu}) \sqsubseteq \mathsf{ground}(u') \iff u \trianglelefteq u' \\
\mathsf{higher\_order}(u, \langle \mu_1, \ldots \mu_n \rangle) \sqsubseteq \mathsf{higher\_order}(u', \langle \mu'_1, \ldots, \mu'_n \rangle) \iff \\
\qquad \forall i \in \{\,1, \ldots, n\,\}\ .\ (\mu'_i)_{\mathsf{init}} \sqsubseteq (\mu_i)_{\mathsf{init}} \ \wedge\ (\mu_i)_{\mathsf{fin}} \curlywedge (\mu'_i)_{\mathsf{init}} \sqsubseteq (\mu'_i)_{\mathsf{fin}} \ \wedge\ u \trianglelefteq u'
\end{array}
$$

◁

Under the partial order $\sqsubseteq$ higher-order insts are comparable if and only if the corresponding insts of their argument modes are comparable. If a higher-order inst ι matches another higher-order inst ι' then a higher-order value with inst ι can be used anywhere an value with inst ι' is required. As with ordinary calls, it is okay for the caller to provide a tighter initial inst than the callee requires and for the callee to provide a tighter final inst than the caller requires.

Example 3.2.2. Let

$$
\begin{aligned}
\iota_0 &= \mathsf{higher\_order}(\mathsf{shared}, \langle \mathsf{ground}(\mathsf{shared}) \rightarrowtail \mathsf{ground}(\mathsf{shared}) \rangle) \\
\iota_1 &= \mathsf{higher\_order}(\mathsf{shared}, \langle \mathsf{bound}(\mathsf{shared}, \{\, c \,\}) \rangle \rightarrowtail \mathsf{bound}(\mathsf{shared}, \{\, c \,\})) \\
\iota_2 &= \mathsf{higher\_order}(\mathsf{shared}, \langle \mathsf{ground}(\mathsf{shared}) \rightarrowtail \mathsf{bound}(\mathsf{shared}, \{\, c \,\}) \rangle),
\end{aligned}
$$

then $\iota_2 = \iota_0 \curlywedge \iota_1$ and $\iota_0 \sqsubseteq \iota_1$. Refer to the CALL rule in Figure 3.13 on page 45. If we have a call where the mode of the formal parameter is $\iota_1 \rightarrowtail \iota_1$ and the initial inst of the actual parameter is ι_0 then the final inst of the formal parameter is ι_2. ◁

Definition 3.2.25 (abstraction function α) We extend the abstraction function α to abstract higher-order terms.

$$\alpha(T) = \bigsqcup_{t \in T} \alpha'(t)$$

where

$$\alpha'(\_) = \mathsf{free}$$

$$\alpha'(f^R(t_1,\ldots,t_n)) = \mathsf{bound}(\alpha_\mathsf{u}(R), \{\, f(\alpha'(t_1),\ldots,\alpha'(t_n)) \,\})$$
$$\alpha'((\lambda(v_1 :: \mu_1,\ldots,v_n :: \mu_n) \leftarrow G)^R) = \mathsf{higher\_order}(\alpha_\mathsf{u}(R), \langle \mu_1,\ldots,\mu_n \rangle) \qquad \triangleleft$$

Figure 3.21 shows the mode rule for higher-order calls. The rule is similar to the CALL rule (see Figure 3.13 on page 45) except it uses the insts from the higher-order inst argmodes rather than from the mode of the procedure being called.

$$\text{HO-CALL}\frac{\begin{array}{c} M_{\mathsf{init}}(v) = \mathsf{higher\_order}(u, \langle \mu_1,\ldots,\mu_n \rangle) \\ \forall i \in \{\, 1,\ldots,n \,\} \,.\, M_{\mathsf{init}}(v_i) \sqsubseteq (\mu_i)_{\mathsf{init}} \\ M_{\mathsf{fin}} = M_{\mathsf{init}} \oplus \{\, v_i \mapsto \iota_i \,|\, 1 \le i \le n \;\wedge\; \iota_i = (\mu_i)_{\mathsf{fin}} \curlywedge M_{\mathsf{init}}(v_i) \,\} \end{array}}{\langle \Gamma, \mathcal{N}, \mathcal{L} \rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner v(v_1,\ldots,v_n) \urcorner : M}$$

Figure 3.21: Mode rule for higher-order calls

Figure 3.22 on the facing page shows the mode rule for unification between a variable v and a higher-order term $\ulcorner \lambda(\overline{v :: \mu}) \leftarrow G \urcorner$. The first six lines describe how the instmap of the procedure is changed by the unification: v is bound to a higher-order inst and all live variables that occur both inside and outside the higher-order term (denoted by L) become shared; such variables must also have inst $\preceq$ any(unique) (i.e. they must not have any free components) because otherwise we would end up with unwanted aliasing between the higher-order term and those variables. The remaining lines describe how the higher-order term itself is mode checked. The mode M' of the higher-order term contains all the unquantified variables in the goal G. The lambda-quantified variables $\overline{v}$ have modes given by the argmodes. Any other variables in M' have the same initial insts as in M (except that live variables become shared) and may not be further instantiated within the lambda goal. Finally, the goal G must be mode correct with mode M' with respect to the environment $\langle \Gamma, \mathsf{uq}(G), \mathcal{L}' \rangle$ where $\mathcal{L}'$ is the set of unquantified variables in G which are not clobbered at the end of G.

3.2.5 Concrete Syntax and Recursive Insts

For convenience and uniformity in the presentation, the abstract syntax we have used in this chapter for insts differs from the concrete syntax used in the Mercury language. Table 3.1 on the next page gives the correspondence between the two syntaxes which is fairly straightforward.

We remind the reader that in our exposition in this chapter we have, for simplicity, ignored the determinism annotations on higher-order insts.

Also note that there is no concrete syntax for some insts, for example, higher-order insts with a uniqueness annotation other than shared. Syntax for these insts could be added by a simple modification to Mercury's parser if it was ever deemed to be a useful feature.

As described in Section 2.5, the language allows for the programmer to define named insts which may possibly be recursive, that is, the definition of a named inst may refer to itself. In the above, we have assumed that all named insts have been expanded out. We use the fixed-point combinator where necessary to convert recursive definitions into infinite insts. Such insts are examples of *regular trees* [1, 36, 56, 57]. We require that all insts in a Mercury program be regular trees so that they can be expressed finitely.

$$\dfrac{\begin{array}{c}\mathsf{abstract\_unify\_inst}(\mathcal{L}(\{\ v\ \}), M_{\mathsf{init}}(v), \mathsf{higher\_order}(\mathsf{unique}, \overline{\mu}), \iota) \\ V = \mathsf{uq}(G) \setminus \overline{v} \\ L = V \cap \mathcal{L} \\ \forall v' \in L.\ M_{\mathsf{init}}(v') \preceq \mathsf{any}(\mathsf{unique}) \\ I = \{\ v' \mapsto M_{\mathsf{init}}(v') \curlywedge \mathsf{any}(\mathsf{shared}) \mid v' \in L\ \} \\ M_{\mathsf{fin}} = M_{\mathsf{init}} \oplus \{\ v \mapsto \iota\ \} \oplus I \\ \operatorname{dom} M' = \mathsf{uq}(G) \\ \langle v_i, \ldots, v_n \rangle = \overline{v} \\ \langle \mu_i, \ldots, \mu_n \rangle = \overline{\mu} \\ \forall i \in \{\ 1, \ldots, n\ \}\,.M'_{\mathsf{init}}(v_i) = (\mu_i)_{\mathsf{init}} \ \wedge\ M'_{\mathsf{fin}}(v_i) = (\mu_i)_{\mathsf{fin}} \\ \forall v' \in V.M'_{\mathsf{init}}(v') = (M_{\mathsf{init}} \oplus I)(v') \\ \mathsf{nobind}(M', V) \\ \mathcal{L}' = \{\ v' \in \mathsf{uq}(G) \mid \neg\, \mathsf{mostly\_clobbered} \trianglelefteq \mathsf{uniqueness}(M'_{\mathsf{fin}}(v'))\ \} \\ \langle \Gamma, \mathsf{uq}(G), \mathcal{L}' \rangle \overset{\mathsf{Goal}}{\vdash} G : M' \end{array}}{\langle \Gamma, \mathcal{N}, \mathcal{L} \rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner v = \lambda(\overline{v :: \mu}) \leftarrow G \urcorner : M}\ \text{UNIFY-V}\lambda$$

Figure 3.22: Mode rule for higher-order unifications

Mercury concrete syntax	Abstract syntax
`free`	free
`any`	any(shared)
`unique_any`	any(unique)
`mostly_unique_any`	any(mostly_unique)
`clobbered_any`	any(clobbered)
`mostly_clobbered_any`	any(mostly_clobbered)
`ground`	ground(shared)
`unique`	ground(unique)
`mostly_unique`	ground(mostly_unique)
`clobbered`	ground(clobbered)
`mostly_clobbered`	ground(mostly_clobbered)
`not_reached`	not_reached
`bound(`β_1 `; ... ;` β_n`)`	bound(shared, $\{\ \beta_1, \ldots, \beta_n\ \}$)
`unique(`β_1 `; ... ;` β_n`)`	bound(unique, $\{\ \beta_1, \ldots, \beta_n\ \}$)
`mostly_unique(`β_1 `; ... ;` β_n`)`	bound(mostly_unique, $\{\ \beta_1, \ldots, \beta_n\ \}$)
`pred(`μ_1`, ...,` μ_n`) is` *det*	higher_order(shared, $\langle \mu_1, \ldots, \mu_n \rangle$)

Table 3.1: Comparison of Mercury concrete and abstract syntax for insts

Example 3.2.3. For example, given the Mercury inst definition

```
:- inst list_skel == bound([] ; [free | list_skel]).
```

any occurrence of '`list_skel`' is expanded to

$$\text{fix}\,\lambda\iota.\ \mathsf{bound}(\mathsf{shared}, \{\,[\,], [\mathsf{free}\,|\,\iota]\,\})$$

in the abstract syntax. ◁

Obviously, it is not possible to represent recursive insts as infinite data structures in the compiler. The actual implementation of the Inst type contains an extra alternative defined_inst(inst_name) where inst_name is a pointer into a lookup table (known as the inst_table) which contains the definition of the inst. Any operation on insts in the compiler must be passed this inst_table and must also keep track of which defined_insts it has seen to ensure that it does not keep expanding recursive insts indefinitely. Operations that create new insts, such as $\curlywedge$ and $\sqcup$, may also need to add new entries to the inst_table.

All this extra book-keeping adds considerably to the complexity of the mode system. In order to maximise the comprehensibility of the exposition, we have chosen to leave it out of this presentation, and instead use the concept of infinite insts.

For more information on regular trees and their implementation techniques see Aiken and Murphy [1], Comon et al. [36], Gécseg and Steinby [56, 57].

3.3 Modifying Goals During Mode Analysis

The mode rules given in Sections 3.1 and 3.2, above, assume that the procedure body is fixed. However, if a particular procedure is not mode-correct the mode system is able to perform some limited modifications to it which may possibly transform it into a mode-correct form whilst preserving its declarative semantics. The two modifications that the system may perform are

1. re-ordering the conjuncts in a conjunction; and
2. introducing extra variables and unifications to allow a call to an *implied mode* of a predicate.

The mode system must also decide which procedure of a predicate will be used for each call to that predicate, since this information is not specified in the source code. These issues are discussed below.

3.3.1 Conjunct Re-ordering

If a conjunction is not mode-correct, some permutation of it may be. The mode system is free to select any permutation which is mode-correct. Section 3.4 discusses the algorithm used to make this selection.

Conjunct re-ordering is particularly important for predicates with multiple modes. It may not be possible to write clauses for the predicate which are valid in all the required modes.

Example 3.3.1. For example, the predicate '`append/3`', after conversion to superhomogeneous form looks like this:

```
:- pred append(list(T), list(T), list(T)).
append(Xs, Ys, Zs) :-
        (
                Xs = [],
                Ys = Zs
        ;
                Xs = [X | Xs0],
                Zs = [X | Zs0],
                append(Xs0, Ys, Zs0)
        ).
```

After mode analysis the procedure with mode '`append(in, in, out)`' has had the conjunction in the second disjunct re-ordered. We have annotated each unification to indicate whether it is an assignment (`:=`), a construction (`<=`), or a deconstruction (`=>`).

```
:- mode append(in, in, out) is det.
append(Xs, Ys, Zs) :-
        (
                Xs => [],
                Zs := Ys
        ;
                Xs => [X | Xs0],
                append(Xs0, Ys, Zs0),
                Zs <= [X | Zs0]
        ).
```

For the procedure with mode '`append(out, out, in)`' to be mode-correct, a different order is required for the conjunction in the second disjunct:

```
:- mode append(out, out, in) is multi.
append(Xs, Ys, Zs) :-
        (
                Xs <= [],
                Ys := Zs
        ;
                Zs => [X | Zs0],
                append(Xs0, Ys, Zs0),
                Xs <= [X | Xs0]
        ).
```

◁

3.3.2 Implied Modes and Selecting Procedures

Definition 3.3.1 (Implied mode) If we know that mode M is a valid mode for a predicate then mode M' is also a valid mode for that predicate if

$$
\begin{aligned}
&\operatorname{dom} M' = \operatorname{dom} M \;\wedge \\
&\forall v \in \operatorname{dom} M.\;\; \Big(\big(M_{\mathsf{init}}(v) = \mathsf{free} \;\wedge\; M'_{\mathsf{init}}(v) \preceq \mathsf{free} \big) \;\vee\; M'_{\mathsf{init}}(v) = M_{\mathsf{init}}(v) \Big) \;\wedge \\
&\phantom{\forall v \in \operatorname{dom} M.\;\;} M'_{\mathsf{fin}}(v) = M_{\mathsf{fin}}(v)
\end{aligned}
$$

M' is known as an *implied mode* of M which we write as $M \Rrightarrow M'$. ◁

Example 3.3.2. For example, '`append(in, in, in)`' is an implied mode of both '`append(in, in, out)`' and '`append(out, out, in)`'. ◁

When analysing a call to a predicate, if there is no mode available for the predicate which matches the mode we require, we may be able to use a mode for which that mode is an implied mode. In particular, if we have a call $\pi(v_1, \ldots, v_n)$ where, for some variable v_k $(1 \leq k \leq n)$, $M_{\mathsf{init}}(v_k) \neq \mathsf{free}$, and the procedure has a mode M' such that $M' \Rrightarrow M$, but in M', the corresponding argument requires an initial inst of free then we can replace v_k by a new free variable v'_k in the call and add a unification between v_k and v'_k after the call. The goal then becomes the existentially quantified conjunction

$$\ulcorner \exists\, \{\, v'_k \,\}\,.\, \bigwedge \langle \pi(v_1, \ldots, v_{k-1}, v'_k, v_{k+1}, \ldots, v_n),\; v'_k = v_k \rangle \urcorner$$

The extension of the transformation to multiple arguments is obvious.

The other task of the mode system when analysing calls is to work out, for a call to a predicate with multiple procedures, which procedure should be selected. In general, there may be several procedures which may be valid for a particular call. These may include procedures which can be called using implied modes, as described above, and matching procedures which have a more general mode than required. Mode analysis should try to select the mode which is likely to be the most efficient. We should always prefer an exact match over an implied mode, and we should always prefer a more specific matching mode over a more general matching mode. An exact match will in general be more efficient than an implied mode because it can avoid unnecessary construction and comparison of data structures. A more specific matching mode may be more efficient than a more general matching mode because the extra information it has about the input arguments may allow it to avoid some tests of the values of those arguments.

3.4 Mode Analysis Algorithms

Previously in this chapter, we have given the mode rules that must be obeyed by a mode-correct Mercury procedure, and described how procedures may be transformed to allow them to obey these rules. We now describe the actual algorithms used to do the mode analysis and transformation. Since we have already presented the bulk of the information needed for understanding mode analysis, we here only give an informal presentation of the algorithms, filling in some important information about the implementation which has not been presented above.

These algorithms take as input a Mercury module consisting of a set of predicates plus a set of mode declarations. Mode declarations are required for any predicate exported from the module. They are optional for other predicates.

For each predicate, the output will be either a set of procedures which are mode-correct according to the rules given previously in this chapter, or a set of mode errors. If a predicate has a set of mode declarations given for it, a procedure will be output corresponding to each mode declaration, provided it can be proved mode-correct. If there are no mode declarations for a predicate, the modes for the procedures are inferred based on the modes of the predicate's callers (see Section 3.4.2 on page 65).

The algorithm is free to modify the body of any procedure as described in Section 3.3. Considerable effort has been expended in the implementation to make the error messages as informative and helpful as possible. However, we omit discussion of this here.

We annotate each subgoal in the body of each procedure with a map that describes how the instmap changes across the execution of that goal. We refer to this as an *instmap delta*. For efficiency, we treat `ground` as a primitive inst rather than defining it in terms of `bound`. This allows ground insts for any type to be represented using only a single word rather than as a complex data structure describing the structure of the inst. Since most Mercury programs only rarely use insts other than `free` and `ground`, the efficiency benefits of doing this are considerable.

3.4.1 Mode Checking Algorithm

Mode Checking a Procedure

To mode check a procedure:

1. Initialise the insts of the head variables using the initial insts of the mode declaration.

2. Mode check the body goal (see below for mode checking goals).

3. If there are no mode errors for the goal analysis, check that the final insts computed by the analysis match the final insts of the mode declaration.

Scheduling Sub-Goals

Mode checking a conjunction or if-then-else requires the notion of *scheduling* the sub-goals of the goal. To attempt to *schedule* a goal, first mode check the goal as described below with respect to the current instmap. If mode checking succeeds, then scheduling succeeds and we commit to executing that goal immediately after the previously scheduled goals for the current conjunction. If mode checking reports an error due to an insufficiently instantiated non-local variable, then scheduling fails in the current instmap and we may try again with another instmap.

Mode Checking a Goal

To mode check a goal, if the goal is

a conjunction Attempt to schedule each sub-goal. If a sub-goal can be scheduled, then schedule it, otherwise delay it. Continue with the remaining sub-goals until there are no goals left.

Every time a variable gets bound, see whether we should wake up a delayed goal (i.e. attempt to schedule it with the new current instmap), next time we get back to the conjunction. If there are still delayed goals when we reach the end of the conjunction, report a mode error. Every time we try a new ordering of goals we must recompute the liveness sets for the goals to comply with the CONJ rule.

a disjunction Mode check the sub-goals; check that the final insts of all the non-local variables are comparable (using $\sqsubseteq$) for all the sub-goals; compute the final instmap using the $\bowtie$ operator (see the DISJ rule).

an existential quantification Mode check the sub-goal. Remove the existentially quantified variables from the instmap (see the SOME rule).

a negation Mode check the sub-goal. Check that the sub-goal does not further instantiate any non-local variables (see the NOT rule).

an if-then-else Attempt to schedule the condition. If successful, then check that it doesn't further instantiate any non-local variables, mode check the 'then' part and the 'else' part, and then check that the final insts match; compute the final instmap using the $\bowtie$ operator (see the ITE rule).

a unification Perform abstract unification in accordance with one of the rules UNIFY-VV, UNIFY-VF and UNIFY-Vλ, depending on the form of the unification.

a predicate call Check that there is a mode declaration for the predicate which matches the current instantiation of the arguments (see the CALL rule). If the predicate has no mode declarations, we need to do mode inference (see Section 3.4.2). Also handle calls to implied modes (see Section 3.3.2).

a higher-order call Check that the higher-order mode matches the current instantiation of the arguments (see the HO-CALL rule).

Limitations of Mode Checking

The main limitation of mode checking is that we do not keep track of aliases. Chapter 5 discusses the impact of this limitation on the precision and expressiveness of the mode system, and offers a solution to the problem.

Algorithmic Complexity

We have not carried out a formal analysis of the computational complexity of the mode checking algorithm. However, we will we make a few observations from practical experience.

First, we note that the mode checking phase in the Mercury compiler takes up a significant proportion of the overall compile time — profiling indicates that it is typically 40% or more.

Mode checking time for a module is linear in the number of predicates. However, within a predicate, the time is potentially super-linear in the size of the goal. A particularly bad case occurs in a unification which constructs a deeply nested term. For example, the unification '`A = [B, C, D, E]`', where `A` is `free` and the other variables are `ground`, when expanded to superhomogeneous normal form, becomes the conjunction '`A = [B | V1], V1 = [C | V2], V2 = [D | V3], V3 = [E | V4], V4 = []`'. The compiler cannot know, before doing the expansion, what the final order of the conjunction will be (it is the job of mode analysis to determine this). The above order would be correct if `A` was ground before the unification. It also happens to be the most efficient order for type analysis so it is the order that the compiler initially chooses. However, the mode analyser must reverse the order of the conjuncts in order to make it mode correct for the case where `A` is free before the unification. The order in which it tries to schedule the unifications (which we denote by their left-hand-side variables) is `A` (fail), `V1` (fail), `V2` (fail), `V3` (fail), `V4` (succeed), `A` (fail), `V1` (fail), `V2` (fail), `V3` (succeed), `A` (fail), `V1` (fail), `V2` (succeed), `A` (fail), `V1` (succeed), `A` (succeed). The reversal takes $O(n^2)$ time, where n is the nesting depth of the original term.

Given the significant cost of mode analysis, even with the limitations of the current system, it is important that we make any extensions as efficient as possible.

3.4.2 Mode Inference Algorithm

The Mercury mode system allows (basic) modes to be inferred for predicates which are not exported from the module in which they are defined.

If, during mode checking, we come across a call to a predicate for which there are no mode declarations, we attempt to infer a mode for the predicate based on the initial insts of the arguments with which it is called. If the predicate is recursive (either directly or indirectly through an SCC where no predicate has declared modes) then it may be necessary to repeat the inference process until a fixpoint is reached.

Normalisation

To prevent an excessive number of procedures from being produced (each call site for a predicate could potentially result in a new procedure being created), and to ensure termination of the mode inference algorithm, we *normalise* the insts of the procedure by replacing each ground inst with an inst $\mathsf{ground}(u)$ which it matches.

Definition 3.4.1 (normalise) The function normalise *normalises* an inst ι:

$$\mathsf{normalise}(\iota) = \begin{cases} \mathsf{ground}(u) & \text{if } u = \bigtriangledown \{\, u' \mid \iota \preceq \mathsf{ground}(u') \,\}; \\ \mathsf{bound}(u, \mathsf{normalise}'(B)) & \text{if } \iota = \mathsf{bound}(u, B) \ \wedge \ \iota \not\preceq \mathsf{ground}; \\ \iota & \text{otherwise.} \end{cases}$$

where

$$\mathsf{normalise}'(B) = \{\, f(\mathsf{normalise}(\iota_1), \ldots, \mathsf{normalise}(\iota_n)) \mid f(\iota_1, \ldots, \iota_n) \in B \,\}$$

If ι is ground then the normalised inst is a ground inst with uniqueness being the least upper bound of the set of uniquenesses u' for which $\iota \preceq \mathsf{ground}(u')$. If ι is a non-ground bound inst then the function attempts to normalise each argument of the functors in ι.[9] $\triangleleft$

[9] The actual implementation does not do the recursive step. In theory, this means that the inference algorithm may not terminate with non-ground insts, however, this has not so far been a problem in practice — probably

Fixpoint Algorithm

During mode analysis, when we see a call to a predicate for which the modes weren't declared, we first check whether the call matches any of the modes we've already inferred. If not, we

1. Create a new procedure for the predicate.
2. Set the initial insts to a normalised version of the insts of the call arguments
3. Set the final insts to not_reached.
4. Mode check the body goal.
5. Normalise the computed final insts.
6. If the final insts have changed, go to step 4.
7. After reaching a fixpoint, return to the calling procedure, set the insts of the arguments to the normalised final insts computed for the call, and continue the analysis.

If the procedure for which we are inferring modes is directly recursive then we may need multiple iterations of the loop in steps 4 to 6 to reach a fixpoint. The first iteration will typically derive insts for the final arguments that are appropriate for the non-recursive clauses only; since the final insts in the recursive call will be not_reached, the insts of all the variables at the end of a recursive clause will be not_reached. In the second iteration the recursive clauses may be able to contribute reachable insts. In general, if ι_n is the inst computed for an argument in iteration n then $\iota_n \sqsubseteq \iota_{n+1}$. If an iteration of the algorithm infers a final inst which is ground, the normalise function cause the result to quickly converge to a fixpoint $\mathsf{ground}(u)$ for some u.

Limitations of Mode Inference

There are a number of limitations to this implementation of mode inference:

- It can only infer modes for predicates that are not exported from a module.
- It can only infer "normalised" modes.
- It cannot re-order conjunctions which contain calls to predicates whose modes need to be inferred. Doing such re-ordering may require the mode system to attempt to analyse many different modes for each such predicate which would cause a combinatorial explosion.
- It cannot infer modes where some of the arguments become clobbered.
- It cannot infer higher-order modes — all higher-order variables/arguments must have a mode declared either in a predicate declaration or a higher-order unification.

In Chapter 6 we look at an alternative approach to mode analysis which can overcome the first three of these limitations.

because neither mode inference nor partially-instantiated insts are used very much at the moment. The implementation should also ensure that all inferred partially instantiated insts are regular trees so that they can be described finitely. Again, the implementation does not do this.

3.5 Related Work

3.5.1 Relationship to Abstract Interpretation

We now take a further look at the relationship of mode analysis in Mercury to abstract interpretation. First let us look at the mode system without unique modes. We have already given definitions for the abstraction function $\alpha : \mathcal{P}\,\mathsf{Term} \to \mathsf{Inst}$ and the concretisation function $\gamma : \mathsf{Inst} \to \mathcal{P}\,\mathsf{Term}$. Using the "matches" partial order $\sqsubseteq$ we can form a Galois connection

$$\langle \mathcal{P}\,\mathsf{Term}, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathsf{Inst}, \sqsubseteq \rangle$$

Note that α is not a total function on $\mathcal{P}\,\mathsf{Term}$, and likewise, $\langle \mathsf{Inst}, \sqsubseteq \rangle$ is not a complete lattice. The sets of terms $T \subseteq \mathsf{Term}$ for which α is not defined represent sets of possible values that may be produced by programs that are not mode correct, such as sets containing both variable and non-variable terms (other than Term itself, which is the concretisation of any).

If we were interested in a descriptive mode system or a weak prescriptive one, it would be possible to make α a total function by adding $\mathsf{free} \sqsubseteq \mathsf{any}$ to the definition of $\sqsubseteq$ so that any is an upper bound for any pair of insts. However, as previously discussed, we believe that this would make the mode system too weak to catch many of the programmer errors we would like it to catch and so is not suitable for a strong prescriptive system.

For uniqueness annotations, we have a Galois connection

$$\langle \mathsf{RefCount}, \leq_{\mathsf{R}} \rangle \underset{\alpha_{\mathsf{u}}}{\overset{\gamma_{\mathsf{u}}}{\leftrightarrows}} \langle \mathsf{Uniq}, \trianglelefteq \rangle$$

To obtain a Galois connection for Inst with uniqueness annotations we need to define an appropriate partial order $\leq_{\mathsf{RT}}$ on $\mathcal{P}\,\mathsf{RTerm}$:

$$\begin{aligned} T \leq_{\mathsf{RT}} T' \iff & \forall f^R(t_1, \ldots, t_n) \in T.\ \exists f^{R'}(t'_1, \ldots, t'_n) \in T'.\ R \leq_{\mathsf{R}} R' \wedge \\ & \forall i \in \{\, 1, \ldots, n \,\}.\ t_i \leq_{\mathsf{RT}} t'_i \end{aligned}$$

We then have a Galois connection

$$\langle \mathcal{P}\,\mathsf{RTerm}, \leq_{\mathsf{RT}} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathsf{Inst}, \sqsubseteq \rangle$$

When doing mode checking, we avoid the need for a fixpoint computation by using the declared modes for the predicate. When doing mode inference, we need to do a widening step [41, 44] in the form of the $\mathsf{normalise}$ function, in order to ensure we reach a fixpoint in a finite and reasonable time.

Abstract interpretation does not provide for modifying the program during the analysis, which we do for re-ordering conjunctions and introducing unifications for implied modes. It should be possible to formalise the transformation under the program transformation framework of Cousot and Cousot [45], which is based on abstract interpretation, but we have not looked at doing so.

3.5.2 Other Related Work

As already noted, the Mercury mode system is based on the work of Somogyi [128, 129] and Henderson [65]. The features which set Mercury's mode system apart from other mode systems for logic programming languages are that it is a strong prescriptive system, it allows an extremely high level of precision in describing modes, it performs re-ordering of conjunctions where necessary, it supports higher-order programming, and it contains a uniqueness analysis. We believe that no other mode system provides all of these features.

The most closely related mode system to that of Mercury is the mode system of HAL. There are some differences, however. First, HAL uses the names new and old instead of free and any. HAL also allows new $\sqsubseteq$ old, which makes the mode system much weaker than Mercury's for detecting mode errors at compile time, but is more convenient for doing constraint logic programming, where the inst of a constraint variable will usually be old and it is not generally possible to know a more precise instantiation state. The HAL compiler inserts a call to an `init` predicate when a new variable needs to be initialised. After initialisation the variable becomes old.

HAL does not allow new to occur inside an argument of a bound inst — one must use old instead. It does not support Mercury's unique modes features. HAL also has facilities for making an inst look different inside a module than it looks outside that module. This is convenient for writing constraint solvers where one would typically want constraint variables to have inst old outside the module in which the solver is defined, and some ground inst inside the module. See García de la Banda et al. [55] for more information on HAL's mode system, and Demoen et al. [49, 50] for more information on the HAL language in general.

In terms of precision and expressiveness, probably the next most closely related mode system to Mercury's is that of Smaus et al. [124, 127]. They describe mode analysis domains for the statically typed logic programming language Gödel [68]. In their mode system, instantiation states are based on types. In some ways their system is more expressive than Mercury's, for example they can describe "open" (or "non-terminated") structures such as difference lists [110, 134]. However, their instantiation states must reflect the structure of the type very closely so there is no way to describe subtypes, such as a list with an even number of elements, or a non-empty list. That is, there is no equivalent of the Mercury insts given in the following inst definitions.

```
:- inst even_list(I) == bound([] ; [I, I | even_list(I)]).
:- inst non_empty_list(I) == bound([I, list_skel(I)]).
```

They also appear to ignore higher-order constructs; and even though their system is built on top of a prescriptive type system, the mode system itself is descriptive.

Their implementation works by the technique of *abstract compilation* where the abstraction of the program is compiled into a Prolog program and then executed. This means that difference lists and other complex variable aliasing patterns may be expressed by variable aliasing in the abstract program. However, their system constructs a single Prolog module to abstract an entire Gödel program. This prohibits separate compilation of a multi-module program which we believe would make this technique impractical for large real-world programs.

As we mentioned earlier, Mercury's unique modes are based on the work of Henderson [65] which in turn is based on the work of Wadler [153]. The functional language Concurrent Clean [9, 62, 112] uses a similar idea where uniqueness annotations are placed on types.

Ueda [142] presents a "linearity analysis" for the concurrent logic language Moded Flat GHC [143]. In this work, a constraint-based approach is used to determine, for each data structure, whether it is used in a *linear* or *non-linear* way. Each data structure must have exactly one producer; a data structure is linear if it also has exactly one consumer. If a consumer of a data structure knows that it is the only consumer of that data stucture then it may free or re-use the memory associated with that data structure. Thus, this concept is very similar to Mercury's destructive input mode '`di == unique >> clobbered`'. However, this system is, again, a descriptive system rather than a prescriptive one and, as such, is closer in spirit to the work of Mazur et al. [91, 92] which aims to find opportunities for compile time garbage collection and structure re-use in Mercury without any prescribed uniqueness conditions. In general, such an automated system is likely to be preferred by the programmer because it allows him or her to avoid explicitly annotating mode declarations with uniqueness. However, for some uses of unique modes, such as for modelling input/output or arrays, it is necessary to prescribe unique modes in order to ensure that the data structures are used only in ways that ensure the soundness of the operational semantics (e.g. to ensure that the program does not try to backtrack and undo I/O operations, or access a value in an array after it has been destructively updated).

Chapter 4

Mode Polymorphism

4.1 The Problem with General Mode Polymorphism

Mercury's parametric polymorphic type system is an important feature for facilitating code reuse. For example, the predicate '`append/3`', when declared with

```
:- pred append(list(T), list(T), list(T)).
```

is able to operate on lists containing elements of any type. That is, it is a *polymorphically typed* predicate. However, in order to use this predicate polymorphically, we must also give it a mode that will work with any type '`T`'. The insts '`ground`' and '`free`', and the modes '`in == ground >> ground`' and '`out == free >> ground`', are useful in such situations because they can be used with any type. For example, a mode declaration of

```
:- mode append(in, in, out).
```

will work equally well with lists of any type '`T`'.

However, it is possible in some situations that using '`ground`' in a mode of a polymorphically typed predicate will lead to an unacceptable loss of precision and expressiveness.

The above mode of '`append/3`' may be called with its input arguments having instantiation states which match '`ground`', but are more specific. For example, the elements of the lists may be bound to higher-order terms and have corresponding higher-order insts. The predicate builds the output list out of the elements of the two input lists so we know that the elements of the output list ought also to have a higher-order inst. Unfortunately, using the mode declared above, this higher-order information is lost from the output argument. This means that an element of this list may not be called as a higher-order term later in the program.

Clearly, we want to be able to use some sort of parametric polymorphism in the mode declaration to associate the insts of the input arguments with the final inst of the output argument. This would then need to be checked when analysing the mode of the predicate.

Given a defined inst of

```
:- inst list_skel(I) == bound([] ; [I | list_skel(I)]).
```

we might first think to specify an inst variable '`I`' analogous to the type variable '`T`' from the type declaration above:

```
:- mode append(in(list_skel(I)), in(list_skel(I)), out(list_skel(I))).
```

where '`I`' can stand for any inst.

Unfortunately, this will not work. Since '`I`' can be any inst, it could be the inst '`free`'. Given the usual definition of '`append/3`', this would lead to the predicate creating aliasing between free variables in the elements of the lists, which Mercury does not allow. We can observe, however, that if we knew that the inst represented by '`I`' did not contain any '`free`' components then there would be no problem with such a mode. Thus, to allow mode polymorphism, we need to *constrain* the possible insts that the inst variable '`I`' is allowed to represent.

In the rest of this chapter, we present an extension to the Mercury mode system which allows a form of constrained polymorphism on mode declarations. Section 4.2 presents the syntax for the system and gives the mode checking rules for it. Section 4.3 presents a further extension which gives more flexibility with handling uniqueness within the system. In Section 4.4 we look at how polymorphic mode information may be derived from polymorphic type declarations. In Section 4.5 we discuss the issues surrounding abstract types and abstract instantiation states which are closely related to polymorphism. Finally, in Section 4.6 we look at related work.

4.2 Constrained Mode Polymorphism

4.2.1 Syntax

Inst variable constraints are of the form ω `=<` ι which states that the inst variable ω represents any inst which is a *sub-inst* of the inst ι. (The definition of "sub-inst" will be made precise in Section 4.2.2.)

Such constraints may only appear on the mode declaration for a predicate, preceded by the operator '`<=`' which is analogous to the syntax used for type class constraints. Each inst variable may appear in at most one constraint.

For example, in the mode declaration

```
:- mode append(in(list_skel(I)), in(list_skel(I)), out(list_skel(I)))
                <= I =< ground.
```

the inst variable '`I`' is constrained to represent a sub-inst of '`ground`'.

Any inst variable appearing in a mode declaration must occur in the initial inst of at least one argument mode. The reason for this is that the parameters are designed to relate the initial insts of some arguments to the final insts of some (possibly different) arguments. When a predicate with a constrained polymorphic mode is called, the actual inst represented by the inst variable will be calculated from these initial insts and then used to determine the final insts of the arguments after the call.

As a syntactic convenience, we assume that any unconstrained inst variable in a mode declaration is constrained to be a sub-inst of '`ground`'. For example,

```
:- mode append(in(list_skel(I)), in(list_skel(I)), out(list_skel(I))).
```

is equivalent to the declaration given in the previous example.

Another valid mode declaration for '`append/3`', one that is more general than the above, is

```
:- mode append(in(list_skel(I)), in(list_skel(I)), out(list_skel(I)))
                <= I =< any.
```

4.2.2 Sub-insts

We now formally define the notion of a "sub-inst". We want to use this to describe an inst which matches a given inst and is no more unique than it.

Definition 4.2.1 (sub-inst) An inst ι is a *sub-inst* of an inst ι' if and only if $\iota \sqsubseteq \iota'$ and $\iota \preceq \iota'$. We write $\iota \boxminus \iota'$. ◁

The constraint ω =< ι' means that ω represents some inst ι such that $\iota \boxminus \iota'$. Note that if $\iota \boxminus \iota'$ then ι and ι' must have exactly the same "level" of uniqueness. For example, if $\iota' = \mathsf{ground}(u)$ then every component of ι must have a uniqueness annotation of u. This constraint is imposed by the fact that the definition of $\sqsubseteq$ uses the uniqueness partial order in the reverse of the way it is used in the definition of $\preceq$. Section 4.3 looks at how we might allow an inst variable to represent insts with a range of uniqueness annotations.

Example 4.2.1. For example, $\mathsf{bound}(\mathsf{shared}, \{ c \}) \boxminus \mathsf{ground}(\mathsf{shared})$, but $\mathsf{bound}(\mathsf{unique}, \{ c \}) \not\boxminus \mathsf{ground}(\mathsf{shared})$ and $\mathsf{bound}(\mathsf{clobbered}, \{ c \}) \not\boxminus \mathsf{ground}(\mathsf{shared})$. ◁

4.2.3 Constrained Inst Variables

We implement constrained inst variables by adding another alternative, $\mathsf{constrained\_inst}(\Omega, \iota)$, where Ω is a set of inst variables and ι is an inst, to the definition of our abstract domain Inst, as shown in Figure 4.1. The inst ι may not contain any further occurrences of $\mathsf{constrained\_inst}$. The meaning of this is that if $\iota' = \mathsf{constrained\_inst}(\Omega, \iota)$ then ι' represents some inst that is a sub-inst of ι and also a sub-inst of all of the insts represented by the inst variables $\omega \in \Omega$. The inst $\mathsf{constrained\_inst}(\emptyset, \iota)$ is equivalent to ι.

Instantiation variable ($\mathsf{InstVar}$)	ω		
Instantiation variable set ($\mathsf{InstVarSet}$)	Ω	::=	$\mathcal{P}\,\omega$
Instantiation state (Inst)	ι	::=	free
		|	$\mathsf{any}(u)$
		|	$\mathsf{bound}(u, \mathcal{P}\, f(\overline{\iota}))$
		|	$\mathsf{higher\_order}(u, \overline{\mu})$
		|	$\mathsf{constrained\_inst}(\Omega, \iota')$

Figure 4.1: Instantiation states with constrained polymorphism

When creating a procedure $R = \langle \pi(\overline{v}) \leftarrow G, M \rangle$ the mode M is derived from the mode declaration by replacing each occurrence of a constrained inst variable ω, where there is a constraint ω =< ι, in the mode declaration, by $\mathsf{constrained\_inst}(\{ \omega \}, \iota)$ in the abstract domain. This ensures two properties that must hold for any mode M belonging to a procedure $R = \langle \pi(\overline{v}) \leftarrow G, M \rangle$:

Property 4.2.1 *For all Ω such that $\mathsf{constrained\_inst}(\Omega, \iota)$ occurs in M, $\Omega = \{ \omega \}$ for some ω. That is, Ω is a singleton set.* □

Property 4.2.2 *For all* Ω*, if* $\mathsf{constrained\_inst}(\Omega, \iota)$ *occurs in* M *and* $\mathsf{constrained\_inst}(\Omega, \iota')$ *also occurs in* M *then* $\iota = \iota'$. □

These properties need not necessarily hold for modes of goals within a procedure body.

We must again extend the partial orders $\preceq$ and $\sqsubseteq$. The treatment of the sets, Ω, of inst variables in the definitions below is designed to give these partial orders the required properties for the operations of abstract unification, inst merge, and matching. They are explained further below.

Definition 4.2.2 (Instantiatedness partial order with constrained_inst*)* The $\preceq$ partial order is defined as:

$$
\begin{array}{l}
\iota \preceq \mathsf{free} \\
\mathsf{any}(u) \preceq \mathsf{any}(u') \iff u' \trianglelefteq u \\
\mathsf{bound}(u, B) \preceq \mathsf{any}(u') \iff \\
\qquad \forall f(\iota_1, \ldots, \iota_n) \in B.\ \forall i \in \{\,1, \ldots, n\,\}.\ \iota_i \preceq \mathsf{any}(u') \ \wedge\ u' \trianglelefteq u \\
\mathsf{higher\_order}(u, \overline{\mu}) \preceq \mathsf{any}(u') \iff u' \trianglelefteq u \\
\mathsf{bound}(u, B) \preceq \mathsf{bound}(u', B') \iff \\
\qquad \forall \beta \in B.\ \exists \beta' \in B'.\ \beta = f(\iota_1, \ldots, \iota_n) \ \wedge\ \beta' = f(\iota'_1, \ldots, \iota'_n) \ \wedge \\
\qquad \forall i \in \{\,1, \ldots, n\,\}.\ \iota_i \preceq \iota'_i \ \wedge\ u' \trianglelefteq u \\
\mathsf{not\_reached}(u) \preceq \mathsf{higher\_order}(u', \overline{\mu}) \iff u' \trianglelefteq u \\
\mathsf{higher\_order}(u, \overline{\mu}) \preceq \mathsf{ground}(u') \iff u' \trianglelefteq u \\
\mathsf{higher\_order}(u, \langle \mu_1, \ldots \mu_n \rangle) \preceq \mathsf{higher\_order}(u', \langle \mu'_1, \ldots, \mu'_n \rangle) \iff \\
\qquad \forall i \in \{\,1, \ldots, n\,\}.\ (\mu_i)'_{\mathsf{init}} \preceq (\mu_i)_{\mathsf{init}} \ \wedge\ (\mu_i)_{\mathsf{fin}} \preceq (\mu'_i)_{\mathsf{fin}} \ \wedge\ u' \trianglelefteq u \\
\mathsf{constrained\_inst}(\Omega, \iota) \preceq \mathsf{constrained\_inst}(\Omega', \iota') \iff \\
\qquad \iota \preceq \iota' \ \wedge\ (\iota \sqsubseteq \iota' \Rightarrow \Omega' \subseteq \Omega) \ \wedge\ (\neg(\iota \sqsubseteq \iota') \Rightarrow \Omega \cap \Omega' = \emptyset) \\
\iota \preceq \mathsf{constrained\_inst}(\Omega', \iota') \ \wedge\ \iota \neq \mathsf{constrained\_inst}(\Omega'', \iota'') \iff \\
\qquad \iota \preceq \iota' \ \wedge\ (\iota \sqsubseteq \iota' \Rightarrow \Omega' = \emptyset) \\
\mathsf{constrained\_inst}(\Omega, \iota) \preceq \iota' \ \wedge\ \iota' \neq \mathsf{constrained\_inst}(\Omega'', \iota'') \iff \iota \preceq \iota'
\end{array}
$$
◁

Recall that the partial order $\preceq$ defines which transitions between instantiation states are valid for a variable over the forward execution of a goal (refer to Section 3.1.2 on page 31). An inst transition from an inst ι' to an inst ι is allowed only if $\iota \preceq \iota'$. The definition for the case $\mathsf{constrained\_inst}(\Omega, \iota) \preceq \mathsf{constrained\_inst}(\Omega', \iota')$, as well as requiring that $\iota \preceq \iota'$, also constrains the sets Ω and Ω': if ι is a sub-inst of ι' then Ω must contain all the inst variables that are in Ω'; if ι is not a sub-inst of ι' then Ω may not contain any of the inst variables that are in Ω'. The reason we define it like this is that if ι is a sub-inst of ι' then we know that the new inst is still a sub-inst of all the insts represented by the inst variables $\omega' \in \Omega'$. However, if ι is not a sub-inst of ι' then we do not know that the new inst is still a sub-inst of all the insts represented by the inst variables $\omega' \in \Omega'$ so these inst variables must not be in Ω.

We can see from the definition that an inst $\iota \neq \mathsf{constrained\_inst}(\Omega', \iota')$ is treated the same as $\mathsf{constrained\_inst}(\emptyset, \iota)$.

The abstract unification operation is defined in terms of $\curlywedge$, the greatest lower bound under the partial order $\preceq$. From Definition 4.2.2, we see that

$$\mathsf{constrained\_inst}(\Omega, \iota) \curlywedge \mathsf{constrained\_inst}(\Omega', \iota') = \mathsf{constrained\_inst}(\Omega'', \iota'')$$

where

$$\iota'' = \iota \curlywedge \iota'$$

$$\Omega'' = \begin{cases} \emptyset & \text{if } \neg(\iota'' \leqq \iota) \ \wedge \ \neg(\iota'' \leqq \iota'); \\ \Omega & \text{if } \iota'' \leqq \iota \ \wedge \ \neg(\iota'' \leqq \iota'); \\ \Omega' & \text{if } \neg(\iota'' \leqq \iota) \ \wedge \ \iota'' \leqq \iota'; \\ \Omega \cup \Omega' & \text{if } \iota'' \leqq \iota \ \wedge \ \iota'' \leqq \iota'. \end{cases}$$

This means that when doing an abstract unification with a constrained_inst, the resulting inst will be a constrained_inst which contains the inst variables in one of the original insts if and only if the resulting inst is a sub-inst of that original inst.

Example 4.2.2. Let

$$\begin{aligned} \iota_1 &= \mathsf{constrained\_inst}(\{\, \omega_1 \,\}, \mathsf{bound}(\mathsf{shared}, \{\, a, b \,\})) \\ \iota_2 &= \mathsf{constrained\_inst}(\{\, \omega_2 \,\}, \mathsf{bound}(\mathsf{shared}, \{\, b, c \,\})) \\ \iota_3 &= \mathsf{constrained\_inst}(\{\, \omega_3 \,\}, \mathsf{bound}(\mathsf{unique}, \{\, b, c \,\})), \end{aligned}$$

then

$$\begin{aligned} \iota_1 \curlywedge \iota_2 &= \mathsf{constrained\_inst}(\{\, \omega_1, \omega_2 \,\}, \mathsf{bound}(\mathsf{shared}, \{\, b \,\})) \\ \iota_1 \curlywedge \iota_3 &= \mathsf{constrained\_inst}(\{\, \omega_1 \,\}, \mathsf{bound}(\mathsf{shared}, \{\, b \,\})) \end{aligned}$$

Note that $\iota_1 \curlywedge \iota_2$ includes both inst variables ω_1 and ω_2 because $\mathsf{bound}(\mathsf{shared}, \{\, b \,\}) \leqq \mathsf{bound}(\mathsf{shared}, \{\, a, b \,\})$ and $\mathsf{bound}(\mathsf{shared}, \{\, b \,\}) \leqq \mathsf{bound}(\mathsf{shared}, \{\, b, c \,\})$. However, $\iota_1 \curlywedge \iota_3$ does not include ω_3 because $\mathsf{bound}(\mathsf{shared}, \{\, b \,\}) \not\leqq \mathsf{bound}(\mathsf{unique}, \{\, bc \,\})$ because the uniqueness annotation has changed from unique to shared. ◁

Definition 4.2.3 (Matches partial order with constrained_inst*)* The $\sqsubseteq$ partial order is defined as:

$$\begin{aligned}
&\mathsf{free} \sqsubseteq \mathsf{free} \\
&\mathsf{not_reached}(u) \sqsubseteq \mathsf{free} \\
&\mathsf{any}(u) \sqsubseteq \mathsf{any}(u') \iff u \trianglelefteq u' \\
&\mathsf{bound}(u, B) \sqsubseteq \mathsf{bound}(u', B') \iff \\
&\qquad \forall \beta \in B.\ \exists \beta' \in B'.\ \beta = f(\iota_1, \ldots, \iota_n) \ \wedge \ \beta' = f(\iota'_1, \ldots, \iota'_n) \ \wedge \\
&\qquad \forall i \in \{\, 1, \ldots, n \,\}.\ \iota_i \sqsubseteq \iota'_i \ \wedge \ u \trianglelefteq u' \\
&\mathsf{not_reached}(u) \sqsubseteq \mathsf{higher_order}(u', \overline{\mu}) \iff u \trianglelefteq u' \\
&\mathsf{higher_order}(u, \overline{\mu}) \sqsubseteq \mathsf{ground}(u') \iff u \trianglelefteq u' \\
&\mathsf{higher_order}(u, \langle \mu_1, \ldots \mu_n \rangle) \sqsubseteq \mathsf{higher_order}(u', \langle \mu'_1, \ldots, \mu'_n \rangle) \iff \\
&\qquad \forall i \in \{\, 1, \ldots, n \,\}.\ (\mu'_i)_{\mathsf{init}} \sqsubseteq (\mu_i)_{\mathsf{init}} \ \wedge \ (\mu_i)_{\mathsf{fin}} \sqsubseteq (\mu'_i)_{\mathsf{fin}} \ \wedge \ u \trianglelefteq u' \\
&\mathsf{constrained_inst}(\Omega, \iota) \sqsubseteq \mathsf{constrained_inst}(\Omega', \iota') \iff \iota \sqsubseteq \iota' \ \wedge \ \Omega' \subseteq \Omega \\
&\iota \sqsubseteq \mathsf{constrained_inst}(\Omega', \iota') \ \wedge \ \iota \neq \mathsf{constrained_inst}(\Omega'', \iota'') \iff \\
&\qquad \iota \sqsubseteq \iota' \ \wedge \ \Omega' = \emptyset \\
&\mathsf{constrained_inst}(\Omega, \iota) \sqsubseteq \iota' \ \wedge \ \iota' \neq \mathsf{constrained_inst}(\Omega'', \iota'') \iff \iota \sqsubseteq \iota'
\end{aligned}$$

◁

This definition of $\sqsubseteq$ requires that, to match an inst $\mathsf{constrained\_inst}(\Omega', \iota')$ (with non-empty inst variable set Ω'), requires an inst $\mathsf{constrained\_inst}(\Omega, \iota)$ where ι matches ι' and Ω contains all the inst variables in Ω'.

The partial order $\sqsubseteq$ is used to check whether one inst matches another and is used in two places: (1) when checking whether the insts of the argument variables for a call are correctly instantiated for the procedure being called; and (2) when checking that the insts inferred for the end of a procedure body match the final insts from the procedure's mode declaration. For the latter, we can see that the above definition requires that if an inst variable occurs in a final inst in the mode declaration then the same inst variable must occur in the corresponding final inferred inst. However, for the former case, this definition does not directly work. It is first necessary to apply an *inst substitution* on the inst variables in the procedure's mode. This is described in Section 4.2.4, below.

Example 4.2.3. Let

$$\begin{aligned}\iota_1 &= \mathsf{constrained\_inst}(\{\,\omega_1, \omega_2\,\}, \mathsf{ground}(\mathsf{shared}))\\ \iota_2 &= \mathsf{constrained\_inst}(\{\,\omega_2\,\}, \mathsf{ground}(\mathsf{shared}))\end{aligned}$$

then $\iota_1 \sqsubseteq \iota_2$, but $\iota_2 \not\sqsubseteq \iota_1$. ◁

The inst merge operation is defined as the least upper bound $\sqcup$ under $\sqsubseteq$. The definition of $\sqsubseteq$ above leads to

$$\mathsf{constrained\_inst}(\Omega, \iota) \sqcup \mathsf{constrained\_inst}(\Omega', \iota') = \mathsf{constrained\_inst}(\Omega \cap \Omega', \iota \sqcup \iota')$$

Therefore, merging two $\mathsf{constrained_inst}$s requires taking the intersection of their inst variable sets.

Example 4.2.4. Let

$$\begin{aligned}\iota_1 &= \mathsf{constrained\_inst}(\{\,\omega_1, \omega_2\,\}, \mathsf{bound}(\mathsf{shared}, \{\,a, b\,\}))\\ \iota_2 &= \mathsf{constrained\_inst}(\{\,\omega_2, \omega_3\,\}, \mathsf{bound}(\mathsf{shared}, \{\,b, c\,\}))\end{aligned}$$

then

$$\iota_1 \sqcup \iota_2 = \mathsf{constrained\_inst}(\{\,\omega_2\,\}, \mathsf{bound}(\mathsf{shared}, \{\,a, b, c\,\}))$$

◁

4.2.4 Inst Substitutions

When analysing a call to a procedure that has constrained polymorphic insts in its mode it is necessary to substitute the inst variables in the mode with the corresponding actual insts of the arguments to the call.

To do this we need the notion of an *inst substitution*. This is analogous to the usual variable substitution, but works with inst variables rather than program variables.

Definition 4.2.4 (Inst substitution) An *inst substitution* is a function mapping inst variables to insts:

$$\mathsf{InstSub} = \mathsf{InstVar} \to \mathsf{Inst} \qquad \lhd$$

In order to cause the inst variables in an inst ι to be substituted in accordance with a substitution σ, we *apply* σ to ι. We will write this as $\iota\,\sigma$. The substitution will only be used where ι comes from a mode declaration so we can assume that Property 4.2.1 on page 73 holds (that is, that in any occurrence of $\mathsf{constrained\_inst}(\Omega, \iota')$ in ι, Ω is a singleton set).

Definition 4.2.5 (Apply substitution) For inst $\iota \in \mathsf{Inst}$ and inst substitution $\sigma \in \mathsf{InstSub}$, $\iota\,\sigma$ is the inst obtained by replacing any occurrence of $\mathsf{constrained\_inst}(\{\,\omega\,\}, \iota')$ in ι where $\omega \in \operatorname{dom}\sigma$ with $\sigma(\omega)$. The substitution fails unless $\omega \in \operatorname{dom}\sigma$ and $\sigma(\omega) \sqsubseteq \iota'$:

$$\iota\,\sigma = \begin{cases} \sigma(\omega) & \text{if } \iota = \mathsf{constrained\_inst}(\{\,\omega\,\}, \iota') \\ & \quad \wedge\ \omega \in \operatorname{dom}\sigma\ \wedge\ \sigma(\omega) \sqsubseteq \iota'; \\ \mathsf{bound}(u, B\,\sigma) & \text{if } \iota = \mathsf{bound}(u, B); \\ \mathsf{higher\_order}(u, \langle \mu_1\,\sigma, \ldots, \mu_n\,\sigma \rangle) & \text{if } \iota = \mathsf{higher\_order}(u, \langle \mu_1, \ldots, \mu_n \rangle); \\ \iota & \text{if } \iota = \mathsf{free}\ \vee\ \iota = \mathsf{any}(u). \end{cases}$$

where

$$B\,\sigma = \{\, f(\iota_1\,\sigma, \ldots, \iota_n\,\sigma) \mid f(\iota_1, \ldots, \iota_n) \in B \,\}$$

and

$$\mu\,\sigma = (\mu_{\mathsf{init}}\,\sigma \rightarrowtail \mu_{\mathsf{fin}}\,\sigma) \qquad \lhd$$

To analyse a call to a procedure with constrained_inst insts in its mode the first thing we need to do is work out what substitution needs to be applied to the insts in the procedure's mode, based on the insts of the argument variables in the caller. The function get_subst, defined below, will help with this. Again, we will only be using this function on insts from a mode declaration so we can assume that Property 4.2.1 on page 73 holds.

Definition 4.2.6 (Get substitution) The function $\mathsf{get\_subst} : \mathcal{P}(\mathsf{Inst} \times \mathsf{Inst}) \to \mathsf{InstSub}$, defined in Figure 4.2 on the following page, calculates the substitution set for a call given a set of pairs where the first member of each pair is the inst of an argument variable in the calling procedure and the second member is the initial inst of the called procedure's corresponding argument.

The function get_subst calls get_subst_inst for each pair of insts and then calls merge_substs to merge the results into one substitution. For a pair of insts ι and ι', if $\iota' = \mathsf{constrained\_inst}(\{\,\omega\,\}, \iota'')$, $\mathsf{get\_subst\_inst}(\iota, \iota'')$ returns the substitution $\{\,\omega \mapsto \iota \curlywedge \iota''\,\}$. The reason for taking the greatest lower bound of ι and ι'' instead of just using ι is to ensure that the inst to be substituted is not more unique than than ι''. This is a necessary condition for it to be a sub-inst of ι''. The next three cases in the definition of get_subst_inst search for inst substitutions recursively within compound insts, using $\mathsf{get\_subst\_bound}$ and $\mathsf{get\_subst\_ho}$, respectively, to find corresponding pairs of insts within the functors of bound insts and within the argument modes of higher-order insts. The final case returns the empty set if no substitutions are found.

The function $\mathsf{merge\_substs}$ is responsible for combining a set of substitutions into a single substitution. If an inst variable ω occurs in multiple substitutions σ then we take the least upper bound $\sqcup$ of the insts $\sigma(\omega)$. ◁

$$\mathsf{get\_subst}(\mathcal{I}) = \mathsf{merge\_substs}\left(\bigcup_{\langle \iota, \iota' \rangle \in \mathcal{I}} \mathsf{get\_subst\_inst}(\iota, \iota')\right)$$

$$\mathsf{get\_subst\_inst}(\iota, \iota') = \begin{cases} \{\, \omega \mapsto \iota \curlywedge \iota'' \,\} & \text{if } \iota' = \mathsf{constrained\_inst}(\{\, \omega \,\}, \iota''); \\ \mathsf{get\_subst\_inst}(\iota'', \iota') & \text{if } \iota = \mathsf{constrained\_inst}(\Omega, \iota''); \\ \mathsf{get\_subst\_bound}(B, B') & \text{if } \iota = \mathsf{bound}(u, B) \ \wedge \ \iota' = \mathsf{bound}(u', B'); \\ \mathsf{get\_subst\_ho}(\overline{\mu}, \overline{\mu'}) & \text{if } \iota = \mathsf{higher\_order}(u, \overline{\mu}) \ \wedge \\ & \quad \iota' = \mathsf{higher\_order}(u', \overline{\mu'}); \\ \emptyset & \text{otherwise.} \end{cases}$$

$$\mathsf{get\_subst\_bound}(B, B') = \mathsf{get\_subst}\left(\left\{ \langle \iota_i, \iota'_i \rangle \;\middle|\; \begin{array}{l} f(\iota_1, \ldots, \iota_n) \in B \quad \wedge \\ f(\iota'_1, \ldots, \iota'_n) \in B' \quad \wedge \\ i \in \{\, 1, \ldots, n \,\} \end{array} \right\}\right)$$

$$\mathsf{get\_subst\_ho}(\langle \mu_1, \ldots, \mu_n \rangle, \langle \mu'_1, \ldots, \mu'_n \rangle) = \mathsf{get\_subst}\left(\bigcup_{i=1}^{n} \{\, \langle (\mu_i)_{\mathsf{init}}, (\mu'_i)_{\mathsf{init}} \rangle, \langle (\mu_i)_{\mathsf{fin}}, (\mu'_i)_{\mathsf{fin}} \rangle \,\}\right)$$

$$\mathsf{merge\_substs}(\mathcal{S}) = \left\{ \omega \mapsto \iota \;\middle|\; \omega \in \bigcup_{\sigma \in \mathcal{S}} \operatorname{dom} \sigma \ \wedge \ \iota = \bigsqcup_{\sigma \in \mathcal{S} \ \wedge \ \omega \in \operatorname{dom} \sigma} \sigma(\omega) \right\}$$

Figure 4.2: The $\mathsf{get\_subst}$ function

Example 4.2.5. Assume we have a mode declaration

```
:- mode append(in(list_skel(I)), in(list_skel(I)), out(list_skel(I))).
```

for the predicate `append/3`, and a call to `append/3` where the initial insts are `list_skel(bound(c))`, `list_skel(bound(d))` and `free`. In the abstract syntax for insts, $\mathsf{list\_skel}/1$ is defined as $\mathsf{list\_skel}(\iota) = \operatorname{fix} \lambda \iota'.\ \mathsf{bound}(\mathsf{shared}, \{\, \texttt{[]}, \texttt{[}\iota \,|\, \iota'\texttt{]} \,\})$. We call

$$\mathsf{get\_subst}(\{\langle \mathsf{list\_skel}(\mathsf{bound}(\mathsf{shared}, \{\, \mathtt{c} \,\})), \mathsf{list\_skel}(\omega) \rangle, \\ \langle \mathsf{list\_skel}(\mathsf{bound}(\mathsf{shared}, \{\, \mathtt{d} \,\})), \mathsf{list\_skel}(\omega) \rangle, \langle \mathsf{free}, \mathsf{free} \rangle\})$$

which computes the substitution $\{\, \omega \mapsto \mathsf{bound}(\mathsf{shared}, \{\, \mathtt{c}, \mathtt{d} \,\}) \,\}$. ◁

4.2.5 Mode Checking with Constrained Inst Variables

To mode check calls to procedures with $\mathsf{constrained\_insts}$ in their modes we must modify the CALL rule from its previous definition (see Figure 3.13 on page 45). The new definition is shown in Figure 4.3. The difference from the previous definition is that we use $\mathsf{get\_subst}$ to calculate a substitution σ from the initial insts of the caller and callee. We then apply this substitution to all insts from the callee before using them. A similar modification is required for the HO-CALL rule.

$$\text{CALL}\ \frac{\begin{array}{c}\langle \pi(v'_1,\ldots,v'_n) \leftarrow G, M'\rangle \in \Gamma \\ \mathcal{I} = \bigcup_{i=1}^{n} \{\, \langle M_{\mathsf{init}}(v_i), M'_{\mathsf{init}}(v'_i)\rangle \,\} \\ \sigma = \mathsf{get\_subst}(\mathcal{I}) \\ \forall i \in \{1,\ldots,n\}.\ M_{\mathsf{init}}(v_i) \sqsubseteq M'_{\mathsf{init}}(v'_i)\,\sigma \\ M_{\mathsf{fin}} = M_{\mathsf{init}} \oplus \{\, v_i \mapsto \iota_i \,|\, 1 \le i \le n \ \wedge\ \iota_i = M'_{\mathsf{fin}}(v'_i)\,\sigma \curlywedge M_{\mathsf{init}}(v_i) \,\}\end{array}}{\langle \Gamma, \mathcal{N}, \mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner \pi(v_1,\ldots,v_n) \urcorner : M}$$

$$\text{HO-CALL}\ \frac{\begin{array}{c}M_{\mathsf{init}}(v) = \mathsf{higher\_order}(u, \langle \mu_1,\ldots,\mu_n\rangle) \\ \mathcal{I} = \bigcup_{i=1}^{n} \{\, \langle M_{\mathsf{init}}(v_i), (\mu_i)_{\mathsf{init}}\rangle \,\} \\ \sigma = \mathsf{get\_subst}(\mathcal{I}) \\ \forall i \in \{1,\ldots,n\}.\ M_{\mathsf{init}}(v_i) \sqsubseteq (\mu_i)_{\mathsf{init}}\,\sigma \\ M_{\mathsf{fin}} = M_{\mathsf{init}} \oplus \{\, v_i \mapsto \iota_i \,|\, 1 \le i \le n \ \wedge\ \iota_i = (\mu_i)_{\mathsf{fin}}\,\sigma \curlywedge M_{\mathsf{init}}(v_i) \,\}\end{array}}{\langle \Gamma, \mathcal{N}, \mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} \ulcorner v(v_1,\ldots,v_n) \urcorner : M}$$

Figure 4.3: Mode rules for calls with constrained polymorphic modes

The other mode rules remain unchanged and simply use the new definitions of $\preceq$ and $\sqsubseteq$ to correctly calculate the inst variable sets within $\mathsf{constrained\_insts}$.

Example 4.2.6. Figure 4.4 on the following page shows the abstract syntax for '`append/3`' annotated with mode information for the mode

```
:- mode append(in(list_skel(I)), in(list_skel(I)), out(list_skel(I)))
                <= I =< ground.
```

Given an initial instmap of

$$\left\{\begin{array}{l}\mathtt{Xs} \mapsto \mathsf{list\_skel}(\mathsf{constrained\_inst}(\{\,\omega\,\}, \mathsf{ground}(\mathsf{shared}))), \\ \mathtt{Ys} \mapsto \mathsf{list\_skel}(\mathsf{constrained\_inst}(\{\,\omega\,\}, \mathsf{ground}(\mathsf{shared}))), \\ \mathtt{Zs} \mapsto \mathsf{free}\end{array}\right\}$$

we show, for each sub-goal, the variables whose insts have changed across that subgoal and the final insts for those variables. For space reasons, we use the symbol ι to denote the inst $\mathsf{constrained\_inst}(\{\,\omega\,\}, \mathsf{ground}(\mathsf{shared}))$. Note that the final inst inferred for `Zs` matches the final inst required by the mode declaration. In particular, it includes the constrained inst variable ω. $\triangleleft$

$$
\begin{array}{ll}
\mathtt{append(Xs, Ys, Zs)} \leftarrow & \\
\quad \bigvee\langle & \\
\qquad \bigwedge\langle & \\
\qquad\quad \mathtt{Xs} = \mathtt{[]}, & \{\, \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{shared}, \{\ \mathtt{[]}\ \})\,\} \\
\qquad\quad \mathtt{Ys} = \mathtt{Zs} & \{\, \mathtt{Zs} \mapsto \mathsf{list\_skel}(\iota)\,\} \\
\qquad \rangle, & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{unique}, \{\ \mathtt{[]}\ \}), \\ \mathtt{Zs} \mapsto \mathsf{list\_skel}(\iota) \end{array} \right\} \\
\qquad \exists\,\{\, \mathtt{Xs0}, \mathtt{Zs0}, \mathtt{X} \,\}\,.(& \\
\qquad\quad \bigwedge\langle & \\
\qquad\qquad \mathtt{Xs} = \mathtt{[X\ |\ Xs0]}, & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{shared}, \{\ \mathtt{[}\iota \,|\, \mathsf{list\_skel}(\iota)\mathtt{]}\ \}), \\ \mathtt{X} \mapsto \iota, \mathtt{Xs0} \mapsto \mathsf{list\_skel}(\iota) \end{array} \right\} \\
\qquad\qquad \mathtt{append(Xs0, Ys, Zs0)}, & \{\, \mathtt{Zs0} \mapsto \mathsf{list\_skel}(\iota)\,\} \\
\qquad\qquad \mathtt{Zs} = \mathtt{[X\ |\ Zs0]} & \{\, \mathtt{Zs} \mapsto \mathsf{bound}(\mathsf{unique}, \{\ \mathtt{[}\iota \,|\, \mathsf{list\_skel}(\iota)\mathtt{]}\ \})\,\} \\
\qquad\quad \rangle & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{shared}, \{\ \mathtt{[}\iota \,|\, \mathsf{list\_skel}(\iota)\mathtt{]}\ \}), \\ \mathtt{X} \mapsto \iota, \\ \mathtt{Xs0} \mapsto \mathsf{list\_skel}(\iota), \mathtt{Zs0} \mapsto \mathsf{list\_skel}(\iota), \\ \mathtt{Zs} \mapsto \mathsf{bound}(\mathsf{unique}, \{\ \mathtt{[}\iota \,|\, \mathsf{list\_skel}(\iota)\mathtt{]}\ \}) \end{array} \right\} \\
\qquad) & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{shared}, \{\ \mathtt{[}\iota \,|\, \mathsf{list\_skel}(\iota)\mathtt{]}\ \}), \\ \mathtt{Zs} \mapsto \mathsf{bound}(\mathsf{unique}, \{\ \mathtt{[}\iota \,|\, \mathsf{list\_skel}(\iota)\mathtt{]}\ \}) \end{array} \right\} \\
\quad \rangle & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{list\_skel}(\iota), \\ \mathtt{Zs} \mapsto \mathsf{list\_skel}(\iota) \end{array} \right\}
\end{array}
$$

Figure 4.4: Abstract syntax for predicate '`append/3`' with polymorphic modes

4.3 Uniqueness Ranges

One problem with the scheme for mode polymorphism presented in the previous sections is that the set of insts that a constrained inst variable may represent must all have the same uniqueness. It would be useful to be able instead to specify an upper and lower bound on the uniqueness of insts that may be represented by the inst variable. For example, consider a procedure `p` with mode

```
:- mode p(I >> I, ...) <= I =< unique.
```

If the procedure is mode correct, it will ensure that the uniqueness of the argument is maintained. However, if the argument is not used within the procedure in a way that requires it to be unique, there is no reason why `I` could not be `ground`.

We want to be able to specify that our inst variables are constrained to within some uniqueness range, e.g. from shared to unique. We will do this using a pair of insts which must differ only in their uniqueness, with the first being no more unique than the second. The following partial order relation gives the definition we need.

Definition 4.3.1 (Uniqueness partial order for insts) We define the partial order $\overset{\trianglelefteq}{=}$ on insts such that

$$\iota \overset{\trianglelefteq}{=} \iota' \qquad \Longleftrightarrow \qquad \iota \preceq \iota' \ \wedge\ \iota' \sqsubseteq \iota$$

The insts ι and ι' must be identical apart from their level of uniqueness with ι being no more unique than ι'. $\triangleleft$

We extend the inst $\mathsf{constrained\_inst}(\Omega, \iota)$ by replacing ι with a pair of insts ι_1 and ι_2 and require that $\iota_1 \overset{\trianglelefteq}{=} \iota_2$. The new definition of Inst is shown in Figure 4.5. The meaning of $\mathsf{constrained\_inst}(\Omega, \iota_1, \iota_2)$ is that $\omega \in \Omega$ represents some inst ι' such that

$$\iota' \sqsubseteq \iota_1 \qquad \wedge \qquad \iota' \preceq \iota_2.$$

We can compare this with the previous definition of $\mathsf{constrained\_inst}(\Omega, \iota)$ where the constraint was $\iota' \leqq \iota$, that is,

$$\iota' \sqsubseteq \iota \qquad \wedge \qquad \iota' \preceq \iota.$$

Note that if $\iota_1 = \iota_2 = \iota$ then the old and new definitions are equivalent.

$$\begin{array}{lcll}
\text{Instantiation state } (\mathsf{Inst}) & \iota & ::= & \mathsf{free} \\
 & & | & \mathsf{any}(u) \\
 & & | & \mathsf{bound}(u, \mathcal{P}\, f(\overline{\iota})) \\
 & & | & \mathsf{higher_order}(u, \overline{\mu}) \\
 & & | & \mathsf{constrained_inst}(\Omega, \iota_1, \iota_2)
\end{array}$$

Figure 4.5: Instantiation states with constrained polymorphism and uniqueness ranges

In $\mathsf{constrained\_inst}(\Omega, \iota_1, \iota_2)$ we have two insts ι_1 and ι_2 which approximate the actual inst ι' represented by the inst variables $\omega \in \Omega$. In some places during the mode analysis, ι_1 will be a safe approximation of ι', whereas in other places ι_2 will be a safe approximation. We formalise when each of ι_1 and ι_2 are safe approximations in our new definitions of $\preceq$ and $\sqsubseteq$, below.

Definition 4.3.2 (Instantiatedness partial order with $\mathsf{constrained\_inst}/3$*)* The $\preceq$ partial order is defined as:

$$\begin{array}{l}
\iota \preceq \mathsf{free} \\
\mathsf{any}(u) \preceq \mathsf{any}(u') \quad \Longleftrightarrow \quad u' \trianglelefteq u \\
\mathsf{bound}(u, B) \preceq \mathsf{any}(u') \quad \Longleftrightarrow \\
\qquad \forall f(\iota_1, \ldots, \iota_n) \in B.\ \forall i \in \{\,1, \ldots, n\,\}\,.\ \iota_i \preceq \mathsf{any}(u') \ \wedge\ u' \trianglelefteq u \\
\mathsf{higher_order}(u, \overline{\mu}) \preceq \mathsf{any}(u') \quad \Longleftrightarrow \quad u' \trianglelefteq u \\
\mathsf{bound}(u, B) \preceq \mathsf{bound}(u', B') \quad \Longleftrightarrow \\
\qquad \forall \beta \in B.\ \exists \beta' \in B'.\ \beta = f(\iota_1, \ldots, \iota_n) \ \wedge\ \beta' = f(\iota'_1, \ldots, \iota'_n) \ \wedge \\
\qquad \forall i \in \{\,1, \ldots, n\,\}\,.\ \iota_i \preceq \iota'_i \ \wedge\ u' \trianglelefteq u \\
\mathsf{not_reached}(u) \preceq \mathsf{higher_order}(u', \overline{\mu}) \quad \Longleftrightarrow \quad u' \trianglelefteq u \\
\mathsf{higher_order}(u, \overline{\mu}) \preceq \mathsf{ground}(u') \quad \Longleftrightarrow \quad u' \trianglelefteq u \\
\mathsf{higher_order}(u, \langle \mu_1, \ldots \mu_n \rangle) \preceq \mathsf{higher_order}(u', \langle \mu'_1, \ldots, \mu'_n \rangle) \quad \Longleftrightarrow \\
\qquad \forall i \in \{\,1, \ldots, n\,\}\,.\ (\mu'_i)_{\mathsf{init}} \preceq (\mu_i)_{\mathsf{init}} \ \wedge\ (\mu_i)_{\mathsf{fin}} \preceq (\mu'_i)_{\mathsf{fin}} \ \wedge\ u' \trianglelefteq u \\
\mathsf{constrained_inst}(\Omega, \iota_1, \iota_2) \preceq \mathsf{constrained_inst}(\Omega', \iota'_1, \iota'_2) \quad \Longleftrightarrow \\
\qquad \iota_1 \preceq \iota'_1 \ \wedge\ \iota_2 \preceq \iota'_2 \ \wedge \\
\qquad (\iota_1 \leqq \iota'_1 \Rightarrow \Omega' \subseteq \Omega) \ \wedge\ (\neg(\iota_1 \leqq \iota'_1) \Rightarrow \Omega \cap \Omega' = \emptyset) \\
\iota \preceq \mathsf{constrained_inst}(\Omega', \iota'_1, \iota'_2) \ \wedge\ \iota \neq \mathsf{constrained_inst}(\Omega'', \iota''_1, \iota''_2) \quad \Longleftrightarrow \\
\qquad \iota \preceq \iota'_1 \ \wedge\ (\iota \leqq \iota'_1 \Rightarrow \Omega' = \emptyset)
\end{array}$$

$$\mathsf{constrained\_inst}(\Omega, \iota_1, \iota_2) \preceq \iota' \ \wedge \ \iota' \neq \mathsf{constrained\_inst}(\Omega'', \iota_1'', \iota_2'') \qquad \Longleftrightarrow \qquad \iota_2 \preceq \iota'$$

For the case $\mathsf{constrained\_inst}(\Omega, \iota_1, \iota_2) \preceq \mathsf{constrained\_inst}(\Omega', \iota_1', \iota_2')$ we require both $\iota_1 \preceq \iota_1'$ and $\iota_2 \preceq \iota_2'$ because this ensures that for every inst in the range of the smaller inst there is an inst at least as large in the range of the larger inst. To determine the constraints on Ω and Ω' we must compare ι_1 and ι_1'. If $\iota_1 \sqsubseteq\!\!\!\preceq \iota_1'$ then $\mathsf{constrained\_inst}(\Omega, \iota_1, \iota_2)$ is a sub-inst of the inst represented by each $\omega' \in \Omega'$ so Ω should include all the inst variables in Ω'. For the cases where a constrained_inst is compared with an inst ι which is not a constrained_inst, we treat ι as though it were $\mathsf{constrained\_inst}(\emptyset, \iota, \iota)$. ◁

Definition 4.3.3 (Matches partial order with constrained_inst/3*)* The $\sqsubseteq$ partial order is defined as:

$$
\begin{aligned}
&\mathsf{free} \sqsubseteq \mathsf{free}\\
&\mathsf{not\_reached}(u) \sqsubseteq \mathsf{free}\\
&\mathsf{any}(u) \sqsubseteq \mathsf{any}(u') \qquad \Longleftrightarrow \qquad u \trianglelefteq u'\\
&\mathsf{bound}(u, B) \sqsubseteq \mathsf{bound}(u', B') \qquad \Longleftrightarrow\\
&\qquad \forall \beta \in B.\ \exists \beta' \in B'.\ \beta = f(\iota_1, \ldots, \iota_n) \ \wedge \ \beta' = f(\iota_1', \ldots, \iota_n') \ \wedge\\
&\qquad \forall i \in \{\, 1, \ldots, n \,\}\,.\ \iota_i \sqsubseteq \iota_i' \ \wedge \ u \trianglelefteq u'\\
&\mathsf{not\_reached}(u) \sqsubseteq \mathsf{higher\_order}(u', \overline{\mu}) \qquad \Longleftrightarrow \qquad u \trianglelefteq u'\\
&\mathsf{higher\_order}(u, \overline{\mu}) \sqsubseteq \mathsf{ground}(u') \qquad \Longleftrightarrow \qquad u \trianglelefteq u'\\
&\mathsf{higher\_order}(u, \langle \mu_1, \ldots \mu_n \rangle) \sqsubseteq \mathsf{higher\_order}(u', \langle \mu_1', \ldots, \mu_n' \rangle) \qquad \Longleftrightarrow\\
&\qquad \forall i \in \{\, 1, \ldots, n \,\}\,.\ (\mu_i')_{\mathsf{init}} \sqsubseteq (\mu_i)_{\mathsf{init}} \ \wedge \ (\mu_i)_{\mathsf{fin}} \sqsubseteq (\mu_i')_{\mathsf{fin}} \ \wedge \ u \trianglelefteq u'\\
&\mathsf{constrained\_inst}(\Omega, \iota_1, \iota_2) \sqsubseteq \mathsf{constrained\_inst}(\Omega', \iota_1', \iota_2') \qquad \Longleftrightarrow\\
&\qquad \iota_1 \sqsubseteq \iota_1' \ \wedge \ \iota_2 \sqsubseteq \iota_2' \ \wedge \ \Omega' \subseteq \Omega\\
&\iota \sqsubseteq \mathsf{constrained\_inst}(\Omega', \iota_1', \iota_2') \ \wedge \ \iota \neq \mathsf{constrained\_inst}(\Omega'', \iota_1'', \iota_2'') \qquad \Longleftrightarrow\\
&\qquad \iota \sqsubseteq \iota_2' \ \wedge \ \Omega' = \emptyset\\
&\mathsf{constrained\_inst}(\Omega, \iota_1, \iota_2) \sqsubseteq \iota' \ \wedge \ \iota' \neq \mathsf{constrained\_inst}(\Omega'', \iota_1'', \iota_2'') \qquad \Longleftrightarrow \qquad \iota_1 \sqsubseteq \iota'
\end{aligned}
$$

For the case $\mathsf{constrained\_inst}(\Omega, \iota_1, \iota_2) \sqsubseteq \mathsf{constrained\_inst}(\Omega', \iota_1', \iota_2')$ we must ensure that $\iota_1 \sqsubseteq \iota_1'$ and $\iota_2 \sqsubseteq \iota_2'$. As with the definition of $\preceq$, when comparing a constrained_inst to a non-constrained_inst ι, we treat ι as though it were $\mathsf{constrained\_inst}(\emptyset, \iota, \iota)$. ◁

We must also redefine application of inst substitutions and the get_subst_inst function.

Definition 4.3.4 (Apply substitution with constrained_inst/3*)* For applications of inst substitutions, we merely need to modify the check of the substituted inst so that instead of checking $\sigma(\omega) \sqsubseteq\!\!\!\preceq \iota'$ we check that $\sigma(\omega) \sqsubseteq \iota_1'$ and $\sigma(\omega) \preceq \iota_2'$. Thus,

$$
\iota\,\sigma = \begin{cases}
\sigma(\omega) & \text{if } \iota = \mathsf{constrained\_inst}(\{\,\omega\,\}, \iota_1', \iota_2') \ \wedge \\
& \quad \omega \in \operatorname{dom}\sigma \ \wedge \ \sigma(\omega) \sqsubseteq \iota_1' \ \wedge \ \sigma(\omega) \preceq \iota_2';\\
\mathsf{bound}(u, B\,\sigma) & \text{if } \iota = \mathsf{bound}(u, B);\\
\mathsf{higher\_order}(u, \langle \mu_1\,\sigma, \ldots, \mu_n\,\sigma \rangle) & \text{if } \iota = \mathsf{higher\_order}(u, \langle \mu_1, \ldots, \mu_n \rangle);\\
\iota & \text{if } \iota = \mathsf{free} \ \vee \ \iota = \mathsf{any}(u).
\end{cases}
$$

◁

For $\mathsf{get\_subst\_inst}$, when we see $\mathsf{constrained\_inst}(\Omega, \iota''_1, \iota''_2)$ instead of $\mathsf{constrained\_inst}(\Omega, \iota'')$ from the previous definition, we must decide whether to use ι''_1 or ι''_2 in place of ι''.

Definition 4.3.5 (Get substitution with $\mathsf{constrained\_inst}/3$*)* The new definition of $\mathsf{get\_subst\_inst}(\iota, \iota')$ is shown in Figure 4.6. The auxiliary functions $\mathsf{get\_subst\_bound}$ and $\mathsf{get\_subst\_ho}$ are unchanged, although they call the new version of $\mathsf{get\_subst\_inst}$.

If $\iota' = \mathsf{constrained\_inst}(\{\,\omega\,\}, \iota''_1, \iota''_2)$ then we return the substitution $\{\,\omega \mapsto \iota \curlywedge \iota''_2\,\}$. We take the glb of ι and ι''_2 to ensure that the inst represented by ω is no more unique than ι''_2. The mode rule for calls ensures that $\iota \sqsubseteq \iota''_1$, therefore the uniqueness of ι falls within the range of uniquenesses specified by ι''_1 and ι''_2.

If $\iota = \mathsf{constrained\_inst}(\Omega, \iota''_1, \iota''_2)$ then we call $\mathsf{get\_subst\_inst}$ recursively with ι''_1 which is a safe approximation of the inst ι_ω represented by each inst variable $\omega \in \Omega$ because we know that $\iota_\omega \sqsubseteq \iota''_1$. ◁

$$\mathsf{get\_subst\_inst}(\iota, \iota') = \begin{cases} \{\,\omega \mapsto \iota \curlywedge \iota''_2\,\} & \text{if } \iota' = \mathsf{constrained\_inst}(\{\,\omega\,\}, \iota''_1, \iota''_2); \\ \mathsf{get\_subst\_inst}(\iota''_1, \iota') & \text{if } \iota = \mathsf{constrained\_inst}(\Omega, \iota''_1, \iota''_2); \\ \mathsf{get\_subst\_bound}(B, B') & \text{if } \iota = \mathsf{bound}(u, B) \ \wedge\ \iota' = \mathsf{bound}(u', B'); \\ \mathsf{get\_subst\_ho}(\overline{\mu}, \overline{\mu'}) & \text{if } \iota = \mathsf{higher\_order}(u, \overline{\mu}) \ \wedge \\ & \quad \iota' = \mathsf{higher\_order}(u', \overline{\mu'}); \\ \emptyset & \text{otherwise.} \end{cases}$$

Figure 4.6: The $\mathsf{get\_subst\_inst}$ function with $\mathsf{constrained\_inst}/3$

Example 4.3.1. Figure 4.7 on the following page gives an example of using constrained polymorphic insts with uniqueness ranges. We define the recursive inst $\mathsf{unique\_list\_skel}(\iota) = \mathrm{fix}\,\lambda\iota'.\ \mathsf{bound}(\mathsf{unique}, \{\ \texttt{[]}, \texttt{[}\iota\,\texttt{|}\,\iota'\texttt{]}\ \})$, and assume a mode declaration for `map/3` in which the initial instmap is

$$I = \left\{ \begin{array}{l} \texttt{P} \mapsto \mathsf{higher\_order}(\mathsf{shared}, \langle \mathsf{ground}(\mathsf{shared}) \rightarrowtail \mathsf{ground}(\mathsf{shared}), \\ \qquad \mathsf{free} \rightarrowtail \mathsf{constrained\_inst}(\{\,\omega\,\}, \mathsf{ground}(\mathsf{shared}), \mathsf{ground}(\mathsf{unique}))\rangle) \\ \texttt{Xs} \mapsto \mathsf{ground}(\mathsf{shared}) \\ \texttt{Ys} \mapsto \mathsf{free} \end{array} \right\}$$

and the final instmap is

$$I \oplus \{\ \texttt{Ys} \mapsto \mathsf{unique\_list\_skel}(\mathsf{constrained\_inst}(\{\,\omega\,\}, \mathsf{ground}(\mathsf{shared}), \mathsf{ground}(\mathsf{unique})))\ \}$$

The inst parameter ω is constrained to some inst ι such that $\iota \sqsubseteq \mathsf{ground}(\mathsf{shared})$ and $\iota \preceq \mathsf{ground}(\mathsf{unique})$ and is used to relate the higher-order inst of `P` to the final inst of `Ys`. If the caller of `map/3` instantiates `P` to a predicate where the final inst of the second argument is $\mathsf{ground}(\mathsf{unique})$ then the final inst of `Ys` will be a list with a unique skeleton and where each of the elements has inst $\mathsf{ground}(\mathsf{unique})$. If, on the other hand, `P` is instantiated to a predicate where the final inst of the second argument is $\mathsf{ground}(\mathsf{shared})$, then the final inst of the elements of `Ys` will be ground(shared). In the mode annotations in Figure 4.7 we use the symbol ι in place of the inst constrained_inst($\{\,\omega\,\}$, ground(shared), ground(unique)). ◁

$$
\begin{array}{ll}
\mathtt{map(P,Xs,Ys)} \leftarrow & \\
\quad \bigvee\langle & \\
\qquad \bigwedge\langle & \\
\qquad\quad \mathtt{Xs} = \mathtt{[]}, & \{\, \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{shared}, \{\ \mathtt{[]}\ \})\,\} \\
\qquad\quad \mathtt{Ys} = \mathtt{[]} & \{\, \mathtt{Ys} \mapsto \mathsf{bound}(\mathsf{unique}, \{\ \mathtt{[]}\ \})\,\} \\
\qquad \rangle, & \{\, \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{shared}, \{\ \mathtt{[]}\ \}), \mathtt{Ys} \mapsto \mathsf{bound}(\mathsf{unique}, \{\ \mathtt{[]}\ \})\,\} \\
\qquad \exists\,\{\, \mathtt{Xs0}, \mathtt{Ys0}, \mathtt{X}, \mathtt{Y}\,\}.(& \\
\qquad\quad \bigwedge\langle & \\
\qquad\qquad \mathtt{Xs = [X\ |\ Xs0]}, & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{shared}, \{\ \mathtt{[}\mathsf{ground}(\mathsf{shared})\,|\,\mathsf{ground}(\mathsf{shared})\mathtt{]}\ \}), \\ \mathtt{X} \mapsto \mathsf{ground}(\mathsf{shared}), \mathtt{Xs0} \mapsto \mathsf{ground}(\mathsf{shared}) \end{array} \right\} \\
\qquad\qquad \mathtt{P(X,Y)} & \{\, \mathtt{Y} \mapsto \iota\,\} \\
\qquad\qquad \mathtt{map(P,Xs0,Ys0)}, & \{\, \mathtt{Ys0} \mapsto \mathsf{unique\_list\_skel}(\iota)\,\} \\
\qquad\qquad \mathtt{Ys = [Y\ |\ Ys0]} & \{\, \mathtt{Ys} \mapsto \mathsf{bound}(\mathsf{unique}, \{\ \mathtt{[}\iota\,|\,\mathsf{unique\_list\_skel}(\iota)\mathtt{]}\ \})\,\} \\
\qquad\quad \rangle & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{shared}, \{\ \mathtt{[}\mathsf{ground}(\mathsf{shared})\,|\,\mathsf{ground}(\mathsf{shared})\mathtt{]}\ \}), \\ \mathtt{X} \mapsto \mathsf{ground}(\mathsf{shared}), \mathtt{Y} \mapsto \iota, \\ \mathtt{Xs0} \mapsto \mathsf{ground}(\mathsf{shared}), \mathtt{Ys0} \mapsto \mathsf{unique\_list\_skel}(\iota), \\ \mathtt{Ys} \mapsto \mathsf{bound}(\mathsf{unique}, \{\ \mathtt{[}\iota\,|\,\mathsf{unique\_list\_skel}(\iota)\mathtt{]}\ \}) \end{array} \right\} \\
\qquad) & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{shared}, \{\ \mathtt{[}\mathsf{ground}(\mathsf{shared})\,|\,\mathsf{ground}(\mathsf{shared})\mathtt{]}\ \}), \\ \mathtt{Ys} \mapsto \mathsf{bound}(\mathsf{unique}, \{\ \mathtt{[}\iota\,|\,\mathsf{unique\_list\_skel}(\iota)\mathtt{]}\ \}) \end{array} \right\} \\
\quad \rangle & \left\{ \begin{array}{l} \mathtt{Xs} \mapsto \mathsf{bound}(\mathsf{shared}, \{\ \mathtt{[]}, \mathtt{[}\mathsf{ground}(\mathsf{shared})\,|\,\mathsf{ground}(\mathsf{shared})\mathtt{]}\ \}), \\ \mathtt{Ys} \mapsto \mathsf{unique\_list\_skel}(\iota), \end{array} \right\}
\end{array}
$$

Figure 4.7: Abstract syntax for predicate '`map/3`' with polymorphic modes

4.4 Theorems for Free

Adding constrained inst parameters to the mode declaration for every polymorphically-typed predicate has a big impact on both the syntactic bulkiness of the declaration and on the time required for mode analysis For example, using an inst '`list_skel(I)`' instead of '`ground`' adds extra complexity because the compiler must traverse the structure of the inst every time it processes it during the analysis. There is also extra overhead from the book-keeping required for recursive insts.

It would be nice if the compiler could infer situations where constrained mode polymorphism could be used so that the programmer can avoid complicated mode declarations and the compiler does not have to worry about the added complexity during mode analysis of the (potentially) polymorphically moded predicate.

When a predicate has a polymorphically-typed argument, the operations possible on that argument are restricted. For variable-typed values, the predicate does not know the type of the value so all it can do to values of that type is either discard them or pass them to the output arguments. (This is not strictly true in Mercury, due to the presence of run time type information, however a way around this problem is discussed below.) This allows us to infer relationships between arguments that share polymorphic type variables based solely on the predicate's type declaration. This concept is generally know as "theorems for free" [151].

The idea of using "theorems for free" to obtain implicit mode polymorphism is mentioned in García de la Banda et al. [55] (with respect to HAL). It is achieved by propagating type information into the insts in the mode declaration for a procedure and, where there is a polymorphic type variable in the type declaration, inserting a corresponding polymorphic inst variable in the mode declaration. For example, given a pred and mode declaration such as

```
:- pred member(T, list(T)).
:- mode member(out, in).
```

we could infer a more precise mode declaration of

```
:- mode member(out(I), in(list_skel(I))) <= I =< ground.
```

This means that the programmer does not have to worry about using a polymorphic mode declaration for `member/2`. It also means that the mode analyser can use the simpler declaration during mode analysis of `member/2`, thus saving considerable analysis time. The analyser must still use the inferred polymorphic declaration when analysing calls to `member/2`, but we have found that this does not add significant time to the mode analysis.

Theorems for free will only work in the case of unconstrained type polymorphism. If any of the type variables in the predicate declaration have type class constraints [76] on them then we are unable to infer any theorems for free which involve those type variables since methods of the instances of the type class will have information about the type in question and are able to construct new values of the type.

Mercury programs have access to run time type information (RTTI) [52] about every type in the program. This allows predicates to perform certain operations on objects of any type, even if the type is not known at compile time. This includes operations that allow construction of new values of an arbitrary type and conversion to and from the universal type '`univ`'. Such operations invalidate the use of "theorems for free" in much the same way as type class constraints. It is as though there is an implicit type class constraint on every type variable in every predicate declaration. For the "theorems for free" approach to work, we must modify the Mercury language to require this implicit constraint to be made explicit. That is, we must create a new type class and require a type to have an instance for this class if we wish to perform operations which manipulate values of the type using run time type information. Any polymorphically-typed predicate which wished to use these operations would then be required to have a type class constraint on its declaration so we would know that we cannot apply "theorems for free" to it.

4.5 Abstract Insts

As we mentioned in Section 2.5.7, Mercury supports abstract types. Consider a module M that has an abstract type declaration

```
:- type t.
```

in its interface section and a type definition

```
:- type t == pred(int, int).
```

in its implementation which defines `t` to be a higher-order type. Values of type `t` can be passed around outside module M, but because the type is abstract, only predicates within module M know how it is implemented and can actually make use of values of type `t`, e.g. by calling them.

As we have seen, for a value of a higher-order type, such as `t`, to be callable it needs to have a higher-order instantiation state which provides the mode declaration for the predicate which it represents. We can define such an instantiation state to be used with `t`, e.g.

```
:- inst i == (pred(in, out) is det).
```

however, in keeping with the principle of information hiding, if the type `t` is abstract, then we would like `i` to be an abstract inst also, otherwise we have revealed information about `t` outside the module M, namely, we know from the definition of `i` that `t` must be a higher-order type taking two arguments.

The problem with providing abstract insts is that an abstract inst may contain `free` components and therefore, we do not know where it is safe to use it without creating aliases between `free` variables. This is very similar to the problem of providing general mode polymorphism so we can provide a similar solution — provide a constraint on the abstract inst declaration to give the mode system the information it needs about the inst without exposing any of its implementation details.

Mitchell and Plotkin [99] showed how abstract types can be thought of as existentially quantified types. We can treat an abstract inst in a similar way, as a kind of existentially quantified inst.

The idea is that when we give an abstract inst declaration, we also need to give a constraint on that inst. For example

```
:- inst i =< ground.
```

declares `i` to be an abstract inst which is a sub-inst of ground or, in other words, the declaration says "there exists some inst `i` such that `i` is a sub-inst of `ground`". When mode checking module M, the mode system can compare the abstract inst declaration to the definition of `i` in the module implementation to ensure that the constraint is valid.

Whenever `i` appears in a predicate mode declaration which is visible outside module M, we replace it with $\mathsf{constrained\_inst}(\{\,\mathtt{i}\,\}, \mathsf{ground}(\mathsf{shared}))$. Inside module M, we use the implementation of `i` instead of the constrained inst.

The only other change we need to make to our mode checking algorithm is that in $\mathsf{get\_subst\_inst}$, if we see $\mathsf{constrained\_inst}(\{\,\mathtt{i}\,\}, \mathsf{ground}(\mathsf{shared}))$ we do not create a substitution for `i` like we would if it was an inst variable ω. This means that if a predicate requires inst `i` in an initial inst of an argument, then only a variable with inst `i` will match it.

4.6 Related Work

As we saw in Section 4.4, the HAL mode system [55] is able to propagate polymorphic type information into the modes of a predicate using the "theorems for free" principle. However, it is only able to do this where the original inst corresponding to the type variable is `ground`. HAL also does not provide support for the programmer to give arbitrary constrained inst parameters in mode declarations, so our system is more general. The HAL mode system also does not support uniqueness annotations so it does not have to deal with the issues surrounding mode polymorphism and uniqueness.

Mode polymorphism over uniqueness ranges provides similar expressiveness to the *attribute variables* of Concurrent Clean [9, 62, 112] which allow polymorphism over uniqueness type attributes. Attribute variables differ from our mode polymorphism in that they parameterise the uniqueness attributes themselves rather than being parameterised insts or types.

Chapter 5

Alias Tracking

In this chapter we look at a significant shortcoming in the mode system described in Chapter 3 — the inability of that analysis to keep track of aliases between variables. In Section 5.1 we look at why we want to keep track of aliases. In Section 5.2 we explain why we choose to track *definite aliases* rather than *possible aliases*. In Section 5.3 we present an extension to the mode system of Chapter 3 which adds definite aliasing analysis, and describe the benefits that this addition entails for the precision of the mode system. In Section 5.4 we look at a further extension to the mode system, and an extension to the Mercury abstract machine, to allow a limited form of partially instantiated data structures (without the loss of precision caused by the use of `any` insts). We also give an example of using these extensions to provide opportunities for turning a call into a tail call by moving construction unifications from after the call to before it. In Section 5.5 we look at some related work. Finally, in Section 5.6, we present some limitations of the alias tracking system and discuss possible future work.

To simplify the discussion, we will base our alias tracking system on the mode system described in Chapter 3; we omit the polymorphic mode extension described in Chapter 4. However we note that the extensions described in the current chapter and in Chapter 4 are orthogonal and there should be no problem implementing them in the same mode system.

5.1 The Need for Alias Tracking

We motivate the need for alias tracking by presenting some examples where its presence improves the precision of the analysis and allows it to admit programs that it would not otherwise be able to prove mode correct.

5.1.1 Aliases and Precision

The need for alias tracking to improve the precision of mode analysis generally arises when two variables have been aliased (usually through a unification) and the inst of one of those variables later changes. For ultimate precision and sometimes even for correctness, we would like that change to be reflected in the inst of the other variable.

Example 5.1.1. A simple example is the conjunction,

```
X = Y,
Y = c,
p(X)
```

where `Y` is `ground` at the start of the conjunction and the mode of `p/1` is `p(in(bound(c)))`.

Without alias tracking information the call to `p/1` would result in a mode error because the mode system does not know that `X` is bound to the functor `c/0`. If we kept track of the fact that `X` is aliased to `Y` then we would know that when `Y` is bound to `c/0` then `X` is also bound to `c/0` and we can confirm that the call to `p/1` is mode correct. ◁

Example 5.1.2. A variation on this is

```
X = f(Y),
Y = c,
q(X)
```

where the mode of `q/1` is `q(in(bound(f(bound(c)))))` and `Y` is again `ground` at the start of the conjunction. Here, instead of `Y` being directly aliased to `X`, it is aliased to an argument of the functor to which `X` is bound. Here too, we can see that keeping track of the alias can give us more precise information about the inst of `X` when the inst of `Y` changes. ◁

In the above two examples alias tracking allowed us to gain more precise information about the inst of the variable `X`. Alias information can also allow us to avoid throwing away information which may be useful for determining mode correctness later in the analysis.

Consider Example 5.1.1 again but assume the initial inst of `Y` is `unique` rather than `ground`. After the first unification `X` and `Y` are still live so the inst of each of the them would need to become `ground` (i.e. `ground(shared)` — we need to throw away the `unique` annotation). If the mode of `p/1` requires `X` to be `unique` then we will again run into a mode error. However, if we record that after the first unification `X` is aliased to `Y` and nothing else, then we can say that `X` and `Y` are still `unique`, as long as we remember that any change to the inst of one will also affect the inst of the other.[1] This is a subtle change in what we mean by `unique`. Previously, if a variable had inst `unique` then it meant that it only had a single reference. Now it means that there may be multiple references, but we know where they all are.

Now consider Example 5.1.2 again, but assume that the initial inst of `Y` is `free`. Also assume, for the purpose of the discussion, that the conjunction cannot be re-ordered.[2] In the mode system of Chapter 3, the first unification would result in a mode error because `X` and `Y` are both live. There was no way of tracking how a later change in the inst of one variable would affect the inst of the other. When dealing with ground insts, the lack of alias tracking could lead to a loss of precision, which may be acceptable in some circumstances. However, for non-ground insts, (i.e. *partially instantiated* insts) allowing two live free variables to be unified wouldn't just lose precision, it would also introduce the possibility that the inst that the mode system would compute might no longer be a correct approximation of the actual term to which the variable is bound. Therefore,

[1] For example, if the call to `p(X)` causes `X` to become `clobbered` then we must also record that `Y` has become `clobbered`.

[2] Actually, in this particular example the mode system would re-order the conjunction so that the unification '`X = f(Y)`' is delayed until after the unification '`Y = c`'. The re-ordered conjunction would then be mode correct. In Section 5.1.3 we will see an example where such re-ordering is not possible.

the mode system has no option but to report a mode error for the unification. Now, if we record the alias between `X` and `Y`, then we know that when `Y` becomes bound, the argument to the functor of `X` also becomes bound. This means that we can allow such partially instantiated insts to exist, provided we know all their aliases.

We have seen that alias tracking can be useful for increasing the overall precision of the mode analysis, but its potential to make unique modes more useful and allow (a limited use of) partially instantiated insts are by far the most useful gains. We expand on these two uses of alias tracking below with some more examples.

5.1.2 Aliases and Unique Modes

The mode system described in Chapter 3 handles simple unique modes, such as `di` and `uo`, adequately for most purposes. However, the lack of alias tracking means that it cannot correctly analyse nested unique modes, that is, when a unique object is placed inside another unique data structure.

Example 5.1.3. Consider the code in Figure 5.1 on the next page. The code defines a type `record` which consists of a single functor `r` with two arguments of types `io__state` and `int`. It defines an inst `record` where the functor `r` and its first argument are `unique` while the second argument is `ground`.[3] Another inst, `no_io`, is the same as `record` except that the first argument is `clobbered`.

The predicate `get_io/2` extracts the unique `io__state` from inside a `record`. When `R` is deconstructed, there are two live references to the `io__state` in its first field. Without alias tracking, the analysis will give `IO` the inst ground(shared), which does not match the required inst of ground(unique). With alias tracking we would know that only one of these references is live after `get_io` exits because the first argument of `R` becomes clobbered, so we could infer the correct inst and thus allow this procedure to be admitted as mode correct.

5.1.3 Aliases and Partially Instantiated Modes

Partially instantiated data structures are data structures that contain free variables. They are useful when one wants different parts of a data structure to be filled in by different parts of a program. Partially instantiated insts approximate partially instantiated data structures.

Example 5.1.4. Consider the code in Figure 5.2 on page 93. In the goal '`length(L, 10), iota(L, 3)`', `length/2` constructs the skeleton of a list with a specified length and `iota/2` fills in the elements of the list. Without alias tracking the system is unable to verify the mode correctness of the second disjunct of `iota/2`. The deconstruction unification '`L = [H | T]`' unifies `T` with the tail of `L`, which is itself partially instantiated. The recursive call then instantiates `T` to `ground`, but without the information that the tail of `L` is aliased to `T`, the mode checker is unable to determine that the recursive call causes the tail of `L` to become ground, which is required by the mode declaration of `iota/2`. For this reason, it will generate a mode error for the deconstruction unification.

If we could re-order the conjunction so that `T` was `ground` before the deconstruction unification then we would not need to generate a mode error. However, this is not possible because the mode

[3] In our abstract syntax the inst `record` is bound(unique, { r(ground(unique), ground(shared)) }) (see Table 3.1 on page 59).

```
:- type record ---> r(io__state, int).

:- inst record == unique(r(unique, ground)).
:- inst no_io == unique(r(clobbered, ground)).

:- pred hello(record, record).
:- mode hello(record >> clobbered, free >> record) is det.
hello(R0, R) :-
        get_io(R0, IO0),
        io__write_string("Hello World\n", IO0, IO),
        set_io(R0, IO, R).

:- pred get_io(record, io__state).
:- mode get_io(record >> no_io, free >> unique) is det.
get_io(R, IO) :-
        R = r(IO, _).

:- pred set_io(record, io__state, record).
:- mode set_io(no_io >> clobbered, unique >> clobbered,
              free >> record) is det.
set_io(R0, IO, R) :-
        R0 = r(_, N),
        R = r(IO, N).
```

Figure 5.1: Nested unique modes example

of `iota/2` requires `T` to be partially instantiated before the recursive call and the deconstruction unification is the only conjunct that can partially instantiate `T`. ◁

5.2 Definite versus Possible Aliasing

Before we design our alias tracking system, there is one more important question that we need to answer: what is the most appropriate way to handle indefinite aliasing information?

When considering two variables there are three possible cases:

1. the two variables are definitely not aliased;
2. the two variables may possibly be aliased, but we don't know for certain; and
3. the two variables are definitely aliased.

Example 5.2.1. Consider the following code fragment.

```
(
        A = B,
        A = D
;
        A = C,
        A = D
)
```

```
:- pred length(list(int), int).
:- mode length(out(list_skel(free)), in) is det.

length(L, N) :-
    (
            L = [],
            N = 0
    ;
            L = [_ | K],
            M = N - 1,
            length(K, M)
    ).

:- pred iota(list(int), int).
:- mode iota(list_skel(free) >> ground, in) is det.

iota(L, X) :-
    (
            L = []
    ;
            L = [H | T],
            H = X,
            Y = X + 1,
            iota(T, Y)
    ).
```

Figure 5.2: Partial instantiation example.

At the end of the first disjunct, `A` is definitely aliased to `B` and `D` whereas at the end of the second disjunct `A` is definitely aliased to `C` and `D`. At the end of the disjunction itself, we know that `A` is definitely aliased to `D`, possibly aliased to `B` and possibly aliased to `C` (depending on which branch of the disjunct was taken). ◁

For partially instantiated data structures, variables must be either definitely aliased or definitely not aliased. The case of a possible alias needs to be a mode error because we have insufficient information to be able to say what happens to one variable when another variable to which it is possibly aliased becomes further instantiated. Therefore, for partially instantiated data structures we need to keep track of definite aliases.

For unique modes, on the other hand, we want to know whether or not a variable is definitely unique, that is, not possibly aliased. Therefore, ideally, for unique modes we should keep track of possible aliases. However keeping track of both definite and possible aliases would be quite expensive so we make a conservative approximation that allows us also to use just definite aliasing information for unique mode analysis. The approximation is to use the shared inst annotation to indicate that a variable is possibly aliased, without specifying what variable(s) it may be aliased to.

5.3 Extending the Mode System

We now describe how to extend the mode system of Chapter 3 with an alias tracking system that will allow us to write programs such as the examples shown above.

5.3.1 Alias Insts

We extend Inst with a new alternative $\mathsf{alias}(\kappa)$, where $\kappa \in \mathsf{InstKey}$ is an *inst key*, as shown in Figure 5.3. $\mathsf{InstKey}$ is a set of symbols that is distinct from Var. The alias inst may not appear in the mode of a procedure or in the argument modes of a higher-order term.

Inst key ($\mathsf{InstKey}$)	κ		
Instantiation state (Inst)	ι	$::=$	free
		$\mid$	$\mathsf{any}(u)$
		$\mid$	$\mathsf{bound}(u, \mathcal{P}\, f(\overline{\iota}))$
		$\mid$	$\mathsf{higher\_order}(u, \overline{\mu})$
		$\mid$	$\mathsf{alias}(\kappa)$

Figure 5.3: Instantiation states with aliases

We extend $\mathsf{InstMap}$ to allow mapping of inst keys to insts, i.e.

$$\mathsf{InstMap} : (\mathsf{Var} \cup \mathsf{InstKey}) \to \mathsf{Inst}.$$

We assume the existence of a function

$$\mathsf{new\_inst\_key} : \mathsf{InstMap} \to \mathsf{InstKey}$$

which, given an instmap, returns an inst key which does not occur in that instmap.[4]

The meaning of $\mathsf{alias}(\kappa)$ is that two variables (or parts of variables) with the same inst $\mathsf{alias}(\kappa)$ (in a given instmap I) are *definitely aliased*. The actual underlying inst can be obtained by looking up the inst key κ in the instmap I.[5]

The addition of $\mathsf{alias}(\kappa)$ means that insts are now only meaningful in reference to an instmap. Every program point has an associated instmap which we will refer to as the *current instmap* for that point. If we are about to execute a goal, the current instmap is the initial instmap of the mode of that goal. If we have just executed a goal, the current instmap is the final instmap of the mode of that goal.

When we abstractly unify two insts $\mathsf{alias}(\kappa)$ and $\mathsf{alias}(\kappa')$ we will need to record that the two inst keys κ and κ' are now aliased. We do this by adding $\kappa \mapsto \mathsf{alias}(\kappa')$ to the instmap. Thus, we will end up with "chains" of inst keys which are analogous to the pointer chains for aliased variables in the WAM [2, 156]. We will need to be able to determine, given an inst key κ and an

[4] It is not necessary for the new inst key to be unique across all instmaps in the procedure because, as described below, we rename all inst keys when we merge two instmaps.

[5] In the actual implementation, the inst key to inst mapping is kept in a separate data structure called the *inst key table*, rather than in the instmap. This simplifies the types and is more convenient for the implementation, but is less convenient for describing the system here as an extension to the mode system of previous chapters.

instmap I, what is the last key in the chain described by I which contains κ. For this we use the *last inst key* function.

Definition 5.3.1 (last inst key) We define the function $\star : \mathsf{InstMap} \times \mathsf{InstKey} \to \mathsf{InstKey}$ as

$$I \star \kappa = \begin{cases} I \star \kappa' & \text{if } I(\kappa) = \mathsf{alias}(\kappa'); \\ \kappa & \text{otherwise.} \end{cases}$$

We say that $I \star \kappa$ is the *last* inst key in I which corresponds to κ. ◁

Definition 5.3.2 (inst keys in inst) It is also useful to be able to obtain the set of inst keys that an inst ι refers to with respect to an instmap I:

$$\mathsf{inst\_keys}(\iota, I) = \begin{cases} \{\,\kappa\,\} \cup \mathsf{inst\_keys}(I(\kappa), I) & \text{if } \iota = \mathsf{alias}(\kappa); \\ \displaystyle\bigcup_{i=1}^{m} \bigcup_{j=1}^{n_m} \mathsf{inst\_keys}(\iota_{i,j}, I) & \text{if } \iota = \mathsf{bound}(u, \{f_1(\iota_{1,1}, \ldots, \iota_{1,n_1}), \ldots, \\ & \qquad\qquad f_m(\iota_{m,1}, \ldots, \iota_{m,n_m})\}); \\ \emptyset & \text{otherwise.} \end{cases}$$

◁

Operations which modify insts will need access to, and may need to modify, the current instmap. We give the new definitions of these operations below.

Comparisons between two insts, such as the partial orders $\preceq$ and $\sqsubseteq$, can usually remain unchanged, as long as we first *expand* the insts being compared by substituting $\mathsf{alias}(\kappa)$ for $I(\kappa)$ for each inst key κ, where I is the current instmap, wherever it appears in the insts being compared.

Definition 5.3.3 (alias expansion) The *expansion* of an inst ι with respect to a current instmap I is given by

$$\mathsf{expand}(\iota, I) = \begin{cases} \mathsf{expand}(I(\kappa), I) & \text{if } \iota = \mathsf{alias}(\kappa); \\ \mathsf{bound}(u, \mathsf{expand}'(B, I)) & \text{if } \iota = \mathsf{bound}(u, B); \\ \iota & \text{otherwise.} \end{cases}$$

where

$$\mathsf{expand}'(B, I) = \{\, f(\mathsf{expand}(\iota_1, I), \ldots, \mathsf{expand}(\iota_n, I)) \mid f(\iota_1, \ldots, \iota_n) \in B\,\}$$ ◁

Definition 5.3.4 (alias expansion) We also define a variation of expansion which, if it removes an alias inst, will make the resulting inst shared. This function is not defined if the inst is partially instantiated.

$$
\begin{array}{l}
\mathsf{expand\_shared}(\iota_0, I) = \iota \qquad \iff \\
\quad \text{if } \iota_0 = \mathsf{alias}(\kappa) \text{ then} \\
\qquad \iota_1 = \mathsf{expand\_shared}(I(\kappa), I) \quad \wedge \\
\qquad \iota_1 \preceq \mathsf{any}(\mathsf{unique}) \quad \wedge \\
\qquad \iota = \iota_1 \curlywedge \mathsf{any}(\mathsf{shared}) \\
\quad \text{else if } \iota_0 = \mathsf{bound}(u, B) \text{ then} \\
\qquad \iota = \mathsf{bound}(u, \mathsf{expand\_shared}'(B, I)) \\
\quad \text{else} \\
\qquad \iota = \iota_0
\end{array}
$$

where

$$
\mathsf{expand\_shared}'(B, I) = \{\, f(\mathsf{expand\_shared}(\iota_1, I), \ldots, \mathsf{expand\_shared}(\iota_n, I)) \mid f(\iota_1, \ldots, \iota_n) \in B \,\}
$$

◁

If it happens that a particular inst key κ appears only once in an instmap I then that inst key provides no useful information and should be removed from the instmap (by replacing the single occurrence of $\mathsf{alias}(\kappa)$ with $I(\kappa)$) to make it clear that the variable in whose inst it appears does not really have an alias on it due to the inst key. For this purpose, we assume the existence of a function $\mathsf{remove\_singleton\_inst\_keys} : \mathsf{InstMap} \to \mathsf{InstMap}$.

Definition 5.3.5 (restriction operation on modes) Singleton inst keys are likely to occur after a mode restriction operation (see Definition 3.1.15 on page 37). We redefine this operation to ensure that such singleton inst keys are removed:

$$
M \ominus V = \langle M_{\mathsf{init}} \ominus' V, M_{\mathsf{fin}} \ominus' V \rangle
$$

where

$$
I \ominus' V = \mathsf{remove\_singleton\_inst\_keys}(\{\, v \mapsto I(v) \mid v \in \operatorname{dom} I \setminus V \,\})
$$

◁

5.3.2 Abstract Unification

Recall that abstract unification is defined in terms of the greatest lower bound $\curlywedge$ under the partial order $\preceq$. For alias tracking, abstract unification needs access to the current instmap in order to lookup inst keys that may occur within the insts to be abstractly unified. It may also need to update the value of an inst key in the instmap if the inst to which it corresponds becomes more instantiated. Therefore, we define parameterised operators $\preceq_M$ and $\curlywedge_M$ which are analogous to $\preceq$ and $\curlywedge$, respectively, but are parameterised by a mode M. The current instmap is passed to the operator as M_{init}. The operator returns M_{fin} which will be M_{init} updated with any changes to the inst key mapping that are made by the operation.

Definition 5.3.6 (instantiatedness "partial order" with aliasing) The new parameterised "partial order" $\preceq_M$ is mostly the same as the previous definition of $\preceq$ except that it has a mode parameter.

Also, another case is added for comparing an inst $\mathsf{alias}(\kappa)$ with another inst ι'. In this case, we lookup the value of κ in the current instmap and find the "greatest lower bound" ι'' of that inst and ι'. We then update the instmap entry for κ with the new value ι''. We also need to ensure that κ does not occur in ι''. This is necessary in order to avoid getting a cycle in the instmap. We need the instmap to be acyclic to allow us to safely leave out the occur check at run time.

There is no rule for $\iota \preceq_M \mathsf{alias}(\kappa')$ where $\iota \neq \mathsf{alias}(\kappa)$ because we do not want to lose aliasing information from an inst on forward execution.

$$
\begin{array}{l}
\iota \preceq_M \mathsf{free} \iff M_{\mathsf{fin}} = M_{\mathsf{init}} \\
\mathsf{any}(u) \preceq_M \mathsf{any}(u') \iff u' \trianglelefteq u \ \wedge\ M_{\mathsf{fin}} = M_{\mathsf{init}} \\
\mathsf{bound}(u, B) \preceq_M \mathsf{any}(u') \iff \\
\qquad B = \{\, f_1(\iota_{1,1}, \ldots, \iota_{1,n_1}), \ldots, f_m(\iota_{m,1}, \ldots, \iota_{m,n_m}) \,\} \ \wedge \\
\qquad \forall i \in \{\, 1, \ldots, m \,\} .\ (\forall j \in \{\, 1, \ldots, n_i \,\} .\ (\iota_{i,j} \preceq_{M_{i,j}} \mathsf{any}(u')) \ \wedge \\
\qquad\qquad M_i = M_{i,1} \rhd \cdots \rhd M_{i,n_i}) \ \wedge \\
\qquad M = M_1 \rhd \cdots \rhd M_m \ \wedge\ u' \trianglelefteq u \\
\mathsf{higher\_order}(u, \overline{\mu}) \preceq_M \mathsf{any}(u') \iff u' \trianglelefteq u \ \wedge\ M_{\mathsf{fin}} = M_{\mathsf{init}} \\
\mathsf{bound}(u, B) \preceq_M \mathsf{bound}(u', B') \iff \\
\qquad B = \{\, f_1(\iota_{1,1}, \ldots, \iota_{1,n_1}), \ldots, f_m(\iota_{m,1}, \ldots, \iota_{m,n_m}) \,\} \ \wedge \\
\qquad \forall i \in \{\, 1, \ldots, m \,\} .\ (\exists \beta' \in B' .\ (\beta' = f_i(\iota'_{i,1}, \ldots, \iota'_{i,n_i}) \ \wedge \\
\qquad\qquad\qquad \forall j \in \{\, 1, \ldots, n_i \,\} .\ (\iota_{i,j} \preceq_{M_{i,j}} \iota'_{i,j})) \ \wedge \\
\qquad\qquad M_i = M_{i,1} \rhd \cdots \rhd M_{i,n_i}) \ \wedge \\
\qquad M = M_1 \rhd \cdots \rhd M_m \ \wedge\ u' \trianglelefteq u \\
\mathsf{not\_reached}(u) \preceq_M \mathsf{higher\_order}(u', \overline{\mu}) \iff u' \trianglelefteq u \ \wedge\ M_{\mathsf{fin}} = M_{\mathsf{init}} \\
\mathsf{higher\_order}(u, \overline{\mu}) \preceq_M \mathsf{ground}(u') \iff u' \trianglelefteq u \ \wedge\ M_{\mathsf{fin}} = M_{\mathsf{init}} \\
\mathsf{higher\_order}(u, \langle \mu_1, \ldots \mu_n \rangle) \preceq_M \mathsf{higher\_order}(u', \langle \mu'_1, \ldots, \mu'_n \rangle) \iff \\
\qquad \forall i \in \{\, 1, \ldots, n \,\} .\ (\mu'_i)_{\mathsf{init}} \preceq (\mu_i)_{\mathsf{init}} \ \wedge\ (\mu_i)_{\mathsf{fin}} \preceq (\mu'_i)_{\mathsf{fin}} \ \wedge \\
\qquad u' \trianglelefteq u \ \wedge\ M_{\mathsf{fin}} = M_{\mathsf{init}} \\
\mathsf{alias}(\kappa) \preceq_{\langle I_0, I \rangle} \iota' \iff \\
\qquad \iota'' = I_0(\kappa) \curlywedge_{\langle I_0, I_1 \rangle} \iota' \ \wedge\ \kappa \notin \mathsf{inst\_keys}(\iota'', I_1) \ \wedge\ I = I_1 \oplus \{\, \kappa \mapsto \iota'' \,\}
\end{array}
\quad \triangleleft
$$

Definition 5.3.7 (instantiatedness "greatest lower bound" with aliasing) The new parameterised "greatest lower bound" operation $\curlywedge_M$ is defined in terms of the new parameterised "partial order" $\preceq_M$.

$$
\begin{array}{l}
\iota_0 \curlywedge_{\langle I_0, I \rangle} \iota_1 = \iota \iff \\
\quad \iota \preceq_{\langle I_0, I_1 \rangle} \iota_0 \ \wedge\ \iota \preceq_{\langle I_1, I \rangle} \iota_1 \ \wedge \\
\quad \forall \iota' \exists I'_1 \exists I'_2 \exists I' .\ (\iota' \preceq_{\langle I_0, I'_1 \rangle} \iota_0 \ \wedge\ \iota' \preceq_{\langle I'_1, I'_2 \rangle} \iota_1) \Rightarrow \iota' \preceq_{\langle I'_2, I' \rangle} \iota
\end{array}
\quad \triangleleft
$$

Example 5.3.1. Assume $\{\, \kappa_1 \mapsto \iota_1, \kappa_2 \mapsto \iota_2 \,\} \subseteq I_0$ and that ι_1 and ι_2 do not contain any alias insts, then

$$
\mathsf{alias}(\kappa_1) \curlywedge_{\langle I_0, I \rangle} \mathsf{alias}(\kappa_2) = \mathsf{alias}(\kappa_1)
$$

where

$$I = I_0 \oplus \{\, \kappa_1 \mapsto \mathsf{alias}(\kappa_2), \kappa_2 \mapsto \iota_1 \curlywedge \iota_2 \,\}$$

The instmap is updated to record that κ_1 and κ_2 are now aliased to each other and represent the inst $\iota_1 \curlywedge \iota_2$. $\triangleleft$

Unification between two variables, or between a variable and a functor, creates aliases. When we perform an abstract unification between two insts we represent the newly created alias by a new inst key, κ. We record in the instmap that κ maps to the greatest lower bound of the two insts we are abstractly unifying, and then return $\mathsf{alias}(\kappa)$ as the result of the abstract unification.

If the unification is between two live variables we no longer need to worry about making sure that the result is ground and shared. We can instead delay this checking to places where we may need to remove definite aliases — the ends of disjuncts, the ends of the arms of if-then-elses, and the end of the predicate body.

Definition 5.3.8 (abstract inst unification with alias tracking)

$$\begin{aligned}
&\mathsf{abstract_unify_inst}(\ell, \iota_1, \iota_2, \mathsf{alias}(\kappa), \langle I_0, I\rangle) \iff \\
&\quad \iota = \iota_1 \curlywedge_{\langle I_0, I_1\rangle} \iota_2 \quad \wedge \\
&\quad \kappa = \mathsf{new_inst_key}(I_1) \quad \wedge \\
&\quad I = I_1 \oplus \{\, \kappa \to \iota \,\} \quad \wedge \\
&\quad \big(\ell = 0 \Rightarrow \neg\, \mathsf{semidet_clobbered_unify}(\mathsf{expand}(\iota_1, I_0), \mathsf{expand}(\iota_2, I_0))\big)
\end{aligned}$$

$\triangleleft$

Definition 5.3.9 (abstract inst-functor unification with alias tracking) We add a mode parameter to $\mathsf{abstract\_unify\_inst\_functor}$ for it to pass to $\curlywedge_M$ and $\mathsf{abstract\_unify\_inst}$.

$$\begin{aligned}
&\mathsf{abstract_unify_inst_functor}(\ell, \langle \ell_1, \ldots, \ell_n\rangle, \iota, f, \langle \iota_1, \ldots, \iota_n\rangle, \iota', \langle \iota'_1, \ldots, \iota'_n\rangle, \langle I_0, I\rangle) \iff \\
&\quad \iota'' = \iota \curlywedge_{\langle I_0, I_1\rangle} \mathsf{bound}(\mathsf{unique}, \{\, f(\langle \iota_n, \ldots, \iota_n\rangle) \,\}) \wedge \\
&\quad \Big(\big(\mathsf{expand}(\iota'', I_1) = \mathsf{not_reached} \ \wedge\ \iota' = \iota'' \ \wedge \\
&\qquad \forall i \in \{\, 1, \ldots, n \,\} \,.\, \iota'_i = \mathsf{not_reached} \ \wedge\ I = I_1\big) \\
&\quad \vee \big(\mathsf{expand}(\iota'', I_1) = \mathsf{bound}(u, \{\, f(\langle \iota''_1, \ldots, \iota''_n\rangle) \,\}) \ \wedge \\
&\qquad \forall i \in \{\, 1, \ldots, n \,\} \,.\, \mathsf{abstract_unify_inst}(\ell \wedge \ell_i, \iota''_i, \iota_i, \iota'_i, \langle I_i, I_{i+1}\rangle) \wedge \\
&\qquad \iota' = \iota'' \curlywedge_{\langle I_{n+1}, I\rangle} \mathsf{bound}(u, \{\, f(\langle \iota'_1, \ldots, \iota'_n\rangle) \,\})\big)\Big)
\end{aligned}$$

$\triangleleft$

5.3.3 Merging

Because we are only interested in definite aliases, when merging two instmaps we need to ensure that we keep any aliasing information that is common to both instmaps and throw away any information that is in only one instmap. When throwing away aliasing information, we may need to make unique insts become shared and we we may need to generate mode errors if an inst is partially instantiated.

Definition 5.3.10 (inst merge with aliasing) Instead of using $\sqcup$ to merge two insts, we use the predicate $\mathsf{merge\_insts}$ which is defined in Figure 5.4 on the facing page. This predicate takes two insts ι' and ι'' which are the insts of a particular variable at the end of the two branches of a branched goal. We also pass in I' and I'' which are the instmaps at the end of each branch, and ℓ, the liveness of the variable. The outputs are the merged inst ι, an instmap I which contains mappings for any inst keys needed to describe ι, and L and R which are two sets of inst keys. L contains any inst keys from I' which we needed to remove during the merge. Likewise, R contains any inst keys from I'' which we needed to remove. We collect these inst keys when merging the insts for each variable. They are then used to adjust the inst key mappings in the final instmap to ensure that any uses of these keys are made shared.

If the variable whose insts we are merging is not live then we can safely expand the insts and then merge them by taking the least upper bound $\sqcup$. This ensure that we catch situations where different arms of the disjunction disagree on whether the variable is instantiated or not, but throws away any information about aliasing of this variable. Alias information is only needed for live variables, which is why, for live variables, we call the auxiliary predicate $\mathsf{merge\_insts}'$, whose job is to preserve this information.

In $\mathsf{merge\_insts}'$, if both insts are alias insts then we first obtain their last corresponding inst keys κ' and κ'' from the appropriate instmaps. We then lookup the insts corresponding to these inst keys and call $\mathsf{merge\_insts}'$ recursively with these insts. We then create a new inst key κ for the resulting inst. Previously, we have left the structure of inst keys unspecified. However, when creating the new inst key κ we will later on need be able to associate it back to the inst keys κ' and κ'' so we make κ a pair containing these two inst keys.[6] We add κ to the instmap and return $\iota = \mathsf{alias}(\kappa)$ as the resulting inst.

If both insts are bound we use the auxiliary predicate $\mathsf{merge\_bound\_insts}$, defined in Figure 5.5 on page 101. Otherwise, we expand the insts and make all the inst keys they contain shared. The inst keys we have made shared are recorded in L and R. The resulting inst is the least upper bound $\sqcup$ of the two expanded insts.

$$
\begin{array}{l}
\mathsf{merge\_insts}(\iota', \iota'', I', I'', \ell, L, R, I, \iota) \qquad \Longleftrightarrow \\
\qquad \big(\ell = 0 \;\wedge\; \iota = \mathsf{expand}(\iota', I') \sqcup \mathsf{expand}(\iota'', I'') \;\wedge\; L = R = \emptyset \;\wedge\; I = \emptyset\big) \;\vee \\
\qquad \big(\ell = 1 \;\wedge\; \mathsf{merge\_insts}'(\iota', \iota'', I', I'', L, R, I, \iota)\big) \\
\\
\mathsf{merge\_insts}'(\iota', \iota'', I', I'', L, R, I, \iota) \qquad \Longleftrightarrow \\
\qquad \text{if } \iota' = \mathsf{alias}(\kappa'_0) \;\wedge\; \iota'' = \mathsf{alias}(\kappa''_0) \text{ then} \\
\qquad\qquad \kappa' = I' \star \kappa'_0 \;\wedge\; \kappa'' = I'' \star \kappa''_0 \quad \wedge \\
\qquad\qquad \mathsf{merge\_insts}'(I'(\kappa'), I''(\kappa''), I', I'', L, R, I_0, \iota_1) \quad \wedge \\
\qquad\qquad \kappa = \langle \kappa', \kappa'' \rangle \;\wedge\; I = I_0 \oplus \{\, \kappa \mapsto \iota_1 \,\} \;\wedge\; \iota = \mathsf{alias}(\kappa) \\
\qquad \text{else if } \iota' = \mathsf{bound}(u', B') \;\wedge\; \iota'' = \mathsf{bound}(u'', B'') \text{ then} \\
\qquad\qquad \mathsf{merge\_bound\_insts}(B', B'', I', I'', L, R, I, B) \quad \wedge \\
\qquad\qquad \iota = \mathsf{bound}(u' \triangledown u'', B) \\
\qquad \text{else} \\
\qquad\qquad \iota'_0 = \mathsf{expand\_shared}(\iota', I') \quad \wedge \\
\qquad\qquad \iota''_0 = \mathsf{expand\_shared}(\iota'', I'') \quad \wedge \\
\qquad\qquad L = \mathsf{inst\_keys}(\iota', I') \;\wedge\; R = \mathsf{inst\_keys}(\iota'', I'') \quad \wedge \\
\qquad\qquad I = \emptyset \;\wedge\; \iota = \iota'_1 \sqcup \iota''_1
\end{array}
$$

Figure 5.4: Merging insts with alias tracking

[6] In the actual implementation, inst keys are represented by integers and we have a separate data structure for recording the $\langle \kappa', \kappa'' \rangle \mapsto \kappa$ mapping. This separate data structure can then be discarded after we have finished merging the instmaps.

The predicate $\mathsf{merge\_bound\_insts}$ takes two sets of bound inst functors, B' and B'', and merges them. It first splits the sets into three parts. The set B_{p} contains the corresponding pairs of functors and arguments for functors which occur in both B' and B''. The sets B'_0 and B''_0 contain functors and arguments for functors which occur in only B' and B'', respectively. We pass B_{p} to $\mathsf{merge\_bound\_insts}'$ which merges each pair of functors and arguments and returns the result in B_0.

Any inst key κ that occurs only in B'_0 can be retained in the final instmap and we do not need to make it shared here. This is because we know that the variable cannot be bound to the function symbol containing κ in the other branch; if it could be then that function symbol would also be in B''. If the variable is bound to the function symbol containing κ, we know that it is still definitely aliased. However, we need to record which branch κ came from (in case we discover elsewhere that we need to make it shared) so we replace it with $\langle \kappa, 0 \rangle$ in the inst and insert $\langle \kappa, 0 \rangle \mapsto I'(\kappa)$ in the new instmap. The inst key 0 is a dummy inst key that is guaranteed not to occur by itself in any instmap. We do the same with B''_0 except any inst key κ is substituted with $\langle 0, \kappa \rangle$ to record that it came from the second branch. The above operations are done by the calls to $\mathsf{add\_aliases}$. Finally, we combine the results from $\mathsf{merge\_bound\_insts}'$ and $\mathsf{add\_aliases}$ to obtain B and I.

The predicate $\mathsf{merge\_bound\_insts}'$ passes each corresponding pair of insts in the functor arguments to $\mathsf{merge\_insts}'$ to be merged. It then combines the results to produce L, R, I and B.

The predicate $\mathsf{add\_aliases}$ takes a function θ, a set of bound insts B_0 and an instmap I_0. It calls the auxiliary predicate $\mathsf{add\_aliases}'$ for each inst in B_0 and then combines the results to produce B and I.

The predicate $\mathsf{add\_aliases}'$ takes a function $\theta : \mathsf{InstKey} \to \mathsf{InstKey}$ for renaming inst keys, an inst ι_0, and instmap I_0. If ι_0 is an alias inst $\mathsf{alias}(\kappa_0)$ then we obtain the last inst key $\kappa_1 = I_0 \star \kappa_0$ and recursively call $\mathsf{add\_aliases}'$ with the inst $I_0(\kappa_1)$ to obtain I_1 and ι_1. We then obtain a new inst key κ by applying θ to κ_1, add $\kappa \mapsto \iota_1$ to I_1 to obtain I, and return $\iota = \mathsf{alias}(\kappa)$. $\triangleleft$

Now that we have defined how to merge two insts, we will show how to merge two modes. This involves two main steps: the first is to call $\mathsf{merge\_insts}$ for each corresponding pair of insts from the final instmaps of the two modes. The second step is to use the additional information obtained while merging the insts to determine which inst keys we need to make shared. We need to know which variables are live so we pass the liveness set $\mathcal{L}$ as a parameter to the mode merge operator.

Definition 5.3.11 (mode merge with aliasing) The operation $M' \bowtie_{\mathcal{L}} M'' = M$, defined in Figure 5.6 on page 102, merges two modes M' and M'' using liveness set $\mathcal{L}$ to produce a mode M. The initial instmaps of M' and M'' must be identical; the initial instmap of M is also the same instmap. If one of the final instmaps is unreachable then the final instmap of the result is the other final instmap. Otherwise, we merge the insts for each variable in the final instmaps and combine the results of these merges to produce the sets L and R of instkeys, and the instmap I which contains the mapping from instkey to inst for any new inst keys created during the merges.

Next we obtain K, the set of inst keys in $\operatorname{dom} I$ that need to be made shared. Recall that each inst key in $\operatorname{dom} I$ is a pair $\langle \kappa, \kappa' \rangle$ where $\kappa \in \operatorname{dom} M'_{\mathsf{fin}} \cup \{ 0 \}$ and $\kappa' \in \operatorname{dom} M''_{\mathsf{fin}} \cup \{ 0 \}$. If either κ or κ' occurs in one of the other inst keys in $\operatorname{dom} I$, or in L or R, then $\langle \kappa, \kappa' \rangle$ must become shared. To do this, we first ensure that the insts corresponding to inst keys in K are not partially

$$\begin{array}{l}
\mathsf{merge_bound_insts}(B', B'', I', I'', L, R, I, B) \quad \iff \\
\quad B_{\mathsf{p}} = \left\{ \langle \beta', \beta'' \rangle \;\middle|\; \begin{array}{l} \beta' \in B' \;\wedge\; \beta'' \in B'' \;\wedge \\ \beta' = f(\iota'_1, \ldots, \iota'_n) \;\wedge\; \beta'' = f(\iota''_1, \ldots, \iota''_n) \end{array} \right\} \quad \wedge \\
\quad B'_0 = \left\{ \beta' \;\middle|\; \begin{array}{l} \beta' \in B' \;\wedge \beta' = f(\iota'_1, \ldots, \iota'_n) \;\wedge \\ \neg\exists \beta''.\ \beta'' \in B'' \;\wedge\; \beta'' = f(\iota''_1, \ldots, \iota''_n) \end{array} \right\} \quad \wedge \\
\quad B''_0 = \left\{ \beta'' \;\middle|\; \begin{array}{l} \beta'' \in B'' \;\wedge \beta'' = f(\iota''_1, \ldots, \iota''_n) \;\wedge \\ \neg\exists \beta'.\ \beta' \in B' \;\wedge\; \beta' = f(\iota'_1, \ldots, \iota'_n) \end{array} \right\} \quad \wedge \\
\quad \mathsf{merge_bound_insts}'(B_{\mathsf{p}}, I', I'', L, R, I_0, B_0) \quad \wedge \\
\quad \mathsf{add_aliases}((\lambda\kappa.\, \langle \kappa, 0 \rangle), B'_0, I', B_1, I_1) \quad \wedge \\
\quad \mathsf{add_aliases}((\lambda\kappa.\, \langle 0, \kappa \rangle), B''_0, I'', B_2, I_2) \quad \wedge \\
\quad B = B_0 \cup B_1 \cup B_2 \;\wedge\; I = I_0 \cup I_1 \cup I_2
\end{array}$$

$$\begin{array}{l}
\mathsf{merge_bound_insts}'(B_{\mathsf{p}}, I', I'', L, R, I, B) \quad \iff \\
\quad B_{\mathsf{p}} = \{ \langle \beta'_1, \beta''_1 \rangle, \ldots, \langle \beta'_m, \beta''_m \rangle \} \quad \wedge \\
\quad \forall i \in \{ 1, \ldots, m \}.\ \big(\beta'_i = f_i(\iota'_{i,1}, \ldots, \iota'_{i,n_i}) \;\wedge\; \beta''_i = f_i(\iota''_{i,1}, \ldots, \iota''_{i,n_i}) \quad \wedge \\
\qquad \forall j \in \{ 1, \ldots, n_i \}.\ \mathsf{merge_insts}'(\iota'_{i,j}, \iota''_{i,j}, I', I'', L_{i,j}, R_{i,j}, I_{i,j}, \iota_{i,j}) \quad \wedge \\
\qquad \beta_i = f_i(\iota_{i,1}, \ldots, \iota_{i,n_i}) \big) \quad \wedge \\
\quad L = \bigcup_{i=1}^{m} \bigcup_{j=1}^{n_i} L_{i,j} \;\wedge\; R = \bigcup_{i=1}^{m} \bigcup_{j=1}^{n_i} R_{i,j} \;\wedge\; I = \bigcup_{i=1}^{m} \bigcup_{j=1}^{n_i} I_{i,j} \quad \wedge \\
\quad B = \{ \beta_1, \ldots, \beta_m \}
\end{array}$$

$$\begin{array}{l}
\mathsf{add_aliases}(\theta, B_0, I_0, B, I) \quad \iff \\
\quad B_0 = \{ f_1(\iota_{1,1}, \ldots, \iota_{1,n_1}), \ldots, f_m(\iota_{m,1}, \ldots, \iota_{m,n_m}) \} \quad \wedge \\
\quad \forall i \in \{ 1, \ldots, m \}.\ \forall j \in \{ 1, \ldots, n_i \}.\ \mathsf{add_aliases}'(\theta, \iota_{i,j}, I_0, \iota'_{i,j}, I_{i,j}) \quad \wedge \\
\quad B = \{ f_1(\iota'_{1,1}, \ldots, \iota'_{1,n_1}), \ldots, f_m(\iota'_{m,1}, \ldots, \iota'_{m,n_m}) \} \quad \wedge \\
\quad I = \bigcup_{i=1}^{m} \bigcup_{j=1}^{n_i} I_{i,j}
\end{array}$$

$$\begin{array}{l}
\mathsf{add_aliases}'(\theta, \iota_0, I_0, \iota, I) \quad \iff \\
\quad \text{if } \iota_0 = \mathsf{alias}(\kappa_0) \text{ then} \\
\qquad \kappa_1 = I_0 \star \kappa_0 \;\wedge\; \mathsf{add_aliases}'(\theta, I_0(\kappa_1), I_0, \iota_1, I_1) \quad \wedge \\
\qquad \kappa = \theta(\kappa_1) \;\wedge\; \iota = \mathsf{alias}(\kappa) \;\wedge\; I = I_1 \oplus \{ \kappa \mapsto \iota_1 \} \\
\quad \text{else if } \iota_0 = \mathsf{bound}(u, B_0) \text{ then} \\
\qquad \mathsf{add_aliases}(\theta, B_0, I_0, B, I) \;\wedge\; \iota = \mathsf{bound}(u, B) \\
\quad \text{else} \\
\qquad \iota = \iota_0 \;\wedge\; I = \emptyset
\end{array}$$

Figure 5.5: Merging bound insts with alias tracking

instantiated. We then obtain the set K', which is the set of all inst keys referred to from the set K, and use the $\mathsf{expand\_shared}$ function to make the corresponding insts shared.

The final stage is to add the computed variable-inst mappings into I' and then remove any singleton inst key. ◁

$$
\begin{array}{l}
M' \bowtie_{\mathcal{L}} M'' = M \qquad \iff \\
\quad M'_{\mathsf{init}} = M''_{\mathsf{init}} = M_{\mathsf{init}} \quad \wedge \\
\quad \text{if } M'_{\mathsf{fin}} \in \mathsf{unreachable} \text{ then} \\
\qquad M_{\mathsf{fin}} = M''_{\mathsf{fin}} \\
\quad \text{else if } M''_{\mathsf{fin}} \in \mathsf{unreachable} \text{ then} \\
\qquad M_{\mathsf{fin}} = M'_{\mathsf{fin}} \\
\quad \text{else} \\
\qquad \{\, v_1 \mapsto \iota_1, \ldots, v_n \mapsto \iota_n \,\} = \{\, v \mapsto M'_{\mathsf{fin}}(v) \mid v \in \mathsf{Var} \cap \operatorname{dom} M'_{\mathsf{fin}} \,\} \quad \wedge \\
\qquad \{\, v_1 \mapsto \iota'_1, \ldots, v_n \mapsto \iota'_n \,\} = \{\, v \mapsto M''_{\mathsf{fin}}(v) \mid v \in \mathsf{Var} \cap \operatorname{dom} M''_{\mathsf{fin}} \,\} \quad \wedge \\
\qquad \forall i \in \{\, 1, \ldots, n \,\}.\ \mathsf{merge\_insts}(\iota_i, \iota'_i, M'_{\mathsf{fin}}, M''_{\mathsf{fin}}, \mathcal{L}(v_i), L_i, R_i, I_i, \iota''_i) \quad \wedge \\
\qquad L = \bigcup_{i=1}^{n} L_i \ \wedge\ R = \bigcup_{i=1}^{n} R_i \ \wedge\ I = \bigcup_{i=1}^{n} I_i \qquad \wedge \\
\qquad K = \{\, \langle \kappa, \kappa' \rangle \in \operatorname{dom} I \mid \exists \kappa''.\ \langle \kappa, \kappa'' \rangle \in \operatorname{dom} I \ \wedge\ \kappa' \neq \kappa'' \,\} \quad \cup \\
\qquad\qquad \{\, \langle \kappa, \kappa' \rangle \in \operatorname{dom} I \mid \exists \kappa''.\ \langle \kappa'', \kappa' \rangle \in \operatorname{dom} I \ \wedge\ \kappa \neq \kappa'' \,\} \quad \cup \\
\qquad\qquad \{\, \langle \kappa, \kappa' \rangle \in \operatorname{dom} I \mid \kappa \in L \ \vee\ \kappa' \in R \,\} \quad \wedge \\
\qquad \forall \kappa \in K.\ \mathsf{expand}(I(\kappa), I) \preceq \mathsf{any}(\mathsf{unique}) \quad \wedge \\
\qquad K' = K \cup \bigcup_{\kappa \in K} \mathsf{inst\_keys}(I(\kappa), I) \quad \wedge \\
\qquad I' = I \oplus \{\, \kappa \mapsto \mathsf{expand\_shared}(\mathsf{alias}(\kappa), I) \mid \kappa \in K' \,\} \quad \wedge \\
\qquad M_{\mathsf{fin}} = \mathsf{remove\_singleton\_inst\_keys}(I' \oplus \{\, v_1 \mapsto \iota''_1, \ldots, v_n \mapsto \iota''_n \,\})
\end{array}
$$

Figure 5.6: Merging modes with alias tracking

Example 5.3.2. Assume we have the code shown in Example 5.2.1 on page 92 and that the instmap at the start of the disjunction is

$$I_0 = \{\, v \mapsto \mathsf{ground}(\mathsf{unique}) \mid v \in \{\, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D} \,\} \,\}$$

At the end of the first disjunct the instmap is

$$I_1 = \left\{ \begin{array}{l} \mathtt{A} \mapsto \mathsf{alias}(\kappa_2), \mathtt{B} \mapsto \mathsf{alias}(\kappa_1), \mathtt{C} \mapsto \mathsf{ground}(\mathsf{unique}), \mathtt{D} \mapsto \mathsf{alias}(\kappa_2), \\ \kappa_2 \mapsto \mathsf{alias}(\kappa_1), \kappa_1 \mapsto \mathsf{ground}(\mathsf{unique}) \end{array} \right\}$$

At the end of the second disjunct the instmap is

$$I_2 = \left\{ \begin{array}{l} \mathtt{A} \mapsto \mathsf{alias}(\kappa_4), \mathtt{B} \mapsto \mathsf{ground}(\mathsf{unique}), \mathtt{C} \mapsto \mathsf{alias}(\kappa_3), \mathtt{D} \mapsto \mathsf{alias}(\kappa_4), \\ \kappa_4 \mapsto \mathsf{alias}(\kappa_3), \kappa_3 \mapsto \mathsf{ground}(\mathsf{unique}) \end{array} \right\}$$

Assuming $\mathcal{L} = \{\, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D} \,\}$ we apply the mode merge operator

$$\langle I_0, I_1 \rangle \bowtie_{\mathcal{L}} \langle I_0, I_2 \rangle = \langle I_0, I \rangle$$

where

$$I = \left\{ \begin{array}{l} \mathtt{A} \mapsto \mathsf{alias}(\langle\kappa_2,\kappa_4\rangle), \mathtt{B} \mapsto \mathsf{ground}(\mathsf{shared}), \mathtt{C} \mapsto \mathsf{ground}(\mathsf{shared}), \\ \mathtt{D} \mapsto \mathsf{alias}(\langle\kappa_2,\kappa_4\rangle), \langle\kappa_2,\kappa_4\rangle \mapsto \mathsf{ground}(\mathsf{shared}) \end{array} \right\}$$

The final instmap I records that A and D are still definitely aliased (via the inst key $\langle\kappa_2,\kappa_4\rangle$). The alias information has been discarded from B and C so the insts must become ground(shared). ◁

5.3.4 Mode Rules

The mode rules for compound goals remain unchanged from Figure 3.12 on page 44, except that the DISJ and ITE rules need to pass the liveness set $\mathcal{L}$ to $\bowtie$. The updated mode rules for atomic goals are shown in Figure 5.7 on the following page. These have been modified to pass mode parameters to abstract_unify_inst, abstract_unify_inst_functor and $\curlywedge_M$, and to remove aliases before doing comparisons using $\preceq$ and $\sqsubseteq$.

The updated mode rule for a procedure is shown in Figure 5.8 on page 105. The mode of the procedure is not allowed to contain any alias insts. We must expand out any alias insts in the final instmap of the goal before comparing it with the final instmap of the procedure. We do this in several stages. First we expand out the insts of any non-live variables, then we remove any singleton inst keys. Finally, the insts are expanded with the expand_shared function, which will cause the insts corresponding to any remaining inst keys to become shared and will fail if any such insts are partially instantiated.

5.4 Implementing Partially Instantiated Data Structures

We have seen above how to modify the mode system to track aliases and how this allows a limited use of partially instantiated data structures with precise mode information (as opposed to the loss of precision inherent in the use of the any inst). However, in order to actually implement partially instantiated data structures, the Mercury code generator requires extra information about free insts which we implement with another change to the mode system. A corresponding change to the abstract machine is also necessary. We describe these changes below.

5.4.1 Annotating free Insts

At the Mercury abstract machine level [131], a variable whose instantiation after expansion of aliases is bound is represented by a word containing its value, while a variable whose inst is free is represented by an uninitialised word. If a free variable is inside a partially instantiated data structure we need to represent it as a reference to the location where the value should go once the variable becomes further instantiated. For this, we need to differentiate between two types of free — free(not_aliased) to indicate that the free variable is not aliased to any live data structure, and free(aliased) to indicate that the free variable is aliased to exactly one live data structure. The new definition of Inst is shown in Figure 5.9 on page 105. A variable with inst free(not_aliased) is treated the same as we previously treated variables with inst free. A variable with inst free(aliased) is represented by a word that contains the address where its value will be placed when the variable is instantiated.

$$\text{UNIFY-VV}\ \dfrac{\begin{array}{c}\mathsf{abstract\_unify\_inst}(\mathcal{L}(\{\, v, v' \,\}), M_{\mathsf{init}}(v), M_{\mathsf{init}}(v'), \iota, M') \\ M = M' \oplus \{\, v \mapsto \iota, v' \mapsto \iota \,\}\end{array}}{\langle \Gamma, \mathcal{N}, \mathcal{L} \rangle \stackrel{\mathsf{Goal}}{\vdash} \ulcorner v = v' \urcorner : M}$$

$$\text{UNIFY-VF}\ \dfrac{\begin{array}{c} v \notin \{\, v_1, \ldots, v_n \,\} \\ \iota = M_{\mathsf{init}}(v) \\ \overline{\iota} = \langle M_{\mathsf{init}}(v_1), \ldots, M_{\mathsf{init}}(v_n) \rangle \\ \ell = \mathcal{L}(\{\, v \,\}) \\ \overline{\ell} = \langle \mathcal{L}(\{\, v_1 \,\}), \ldots, \mathcal{L}(\{\, v_n \,\}) \rangle \\ \mathsf{abstract\_unify\_inst\_functor}(\ell, \overline{\ell}, \iota, f, \overline{\iota}, \iota', \langle \iota'_1, \ldots, \iota'_n \rangle, M') \\ M = M' \oplus \{\, v \mapsto \iota', v_1 \mapsto \iota'_1, \ldots, v_n \mapsto \iota'_n \,\}\end{array}}{\langle \Gamma, \mathcal{N}, \mathcal{L} \rangle \stackrel{\mathsf{Goal}}{\vdash} \ulcorner v = f(v_1, \ldots, v_n) \urcorner : M}$$

$$\text{UNIFY-V}\lambda\ \dfrac{\begin{array}{c}\mathsf{abstract\_unify\_inst}(\mathcal{L}(\{\, v \,\}), M_{\mathsf{init}}(v), \mathsf{higher\_order}(\mathsf{unique}, \overline{\mu}), \iota, M_0) \\ V = \mathsf{uq}(G) \setminus \overline{v} \\ V \cap \mathcal{L} = \{\, v'_1, \ldots, v'_m \,\} \\ \forall j \in \{\, 1, \ldots, m \,\}.\ \mathsf{expand}(M_{\mathsf{init}}(v'_j), M_{\mathsf{init}}) \preceq \mathsf{any}(\mathsf{unique}) \\ \wedge\ \iota_j = (M_j)_{\mathsf{init}}(v'_j) \curlywedge_{M_j} \mathsf{any}(\mathsf{shared}) \\ M = M_0 \triangleright \cdots \triangleright M_m \oplus \{\, v \mapsto \iota, v'_1 \mapsto \iota_1, \ldots, v'_m \mapsto \iota_m \,\} \\ \operatorname{dom} M' = \mathsf{uq}(G) \\ \langle v_i, \ldots, v_n \rangle = \overline{v} \\ \langle \mu_i, \ldots, \mu_n \rangle = \overline{\mu} \\ \forall i \in \{\, 1, \ldots, n \,\}.M'_{\mathsf{init}}(v_i) = (\mu_i)_{\mathsf{init}}\ \wedge\ M'_{\mathsf{fin}}(v_i) = (\mu_i)_{\mathsf{fin}} \\ \forall v' \in V.M'_{\mathsf{init}}(v') = M_{\mathsf{fin}}(v') \\ \mathsf{nobind}(M', V) \\ \mathcal{L}' = \{\, v' \in \mathsf{uq}(G) \,|\, \neg\, \mathsf{mostly\_clobbered} \trianglelefteq \mathsf{uniqueness}(M'_{\mathsf{fin}}(v')) \,\} \\ \langle \Gamma, \mathsf{uq}(G), \mathcal{L}' \rangle \stackrel{\mathsf{Goal}}{\vdash} G : M' \end{array}}{\langle \Gamma, \mathcal{N}, \mathcal{L} \rangle \stackrel{\mathsf{Goal}}{\vdash} \ulcorner v = \lambda(\overline{v :: \mu}) \leftarrow G \urcorner : M}$$

$$\text{CALL}\ \dfrac{\begin{array}{c}\langle \pi(v'_1, \ldots, v'_n) \leftarrow G, M' \rangle \in \Gamma \\ \forall i \in \{1, \ldots, n\}.\ \mathsf{expand}(M_{\mathsf{init}}(v_i), M_{\mathsf{init}}) \sqsubseteq M'_{\mathsf{init}}(v'_i)\ \wedge\ \iota_i = M'_{\mathsf{fin}}(v'_i) \curlywedge_{M_i} M_{\mathsf{init}}(v) \\ M = (M_1 \triangleright \cdots \triangleright M_n) \oplus \{\, v_i \mapsto \iota_i \,|\, 1 \leq i \leq n \,\}\end{array}}{\langle \Gamma, \mathcal{N}, \mathcal{L} \rangle \stackrel{\mathsf{Goal}}{\vdash} \ulcorner \pi(v_1, \ldots, v_n) \urcorner : M}$$

$$\text{HO-CALL}\ \dfrac{\begin{array}{c}\mathsf{expand}(M_{\mathsf{init}}(v), M_{\mathsf{init}}) = \mathsf{higher\_order}(u, \langle \mu_1, \ldots, \mu_n \rangle) \\ \forall i \in \{1, \ldots, n\}.\ \mathsf{expand}(M_{\mathsf{init}}(v_i), M_{\mathsf{init}}) \sqsubseteq (\mu_i)_{\mathsf{init}}\ \wedge\ \iota_i = (\mu_i)_{\mathsf{fin}} \curlywedge_{M_i} M_{\mathsf{init}}(v) \\ M = (M_1 \triangleright \cdots \triangleright M_n) \oplus \{\, v_i \mapsto \iota_i \,|\, 1 \leq i \leq n \,\}\end{array}}{\langle \Gamma, \mathcal{N}, \mathcal{L} \rangle \stackrel{\mathsf{Goal}}{\vdash} \ulcorner v(v_1, \ldots, v_n) \urcorner : M}$$

Figure 5.7: Mode rules for atomic goals with alias tracking

$$
\text{PROC}\;\frac{\begin{array}{c}
R = \langle \pi(\overline{v}) \leftarrow G, M\rangle \\
\langle \Gamma, \mathcal{N}, \mathcal{L}\rangle \overset{\mathsf{Goal}}{\vdash} G : M' \\
\forall v \in \overline{v}.\ M_{\mathsf{init}}(v) \sqsubseteq M'_{\mathsf{init}}(v) \\
\mathcal{L} = \{\, v \in \overline{v} \mid \neg\, \mathsf{mostly\_clobbered} \trianglelefteq \mathsf{uniqueness}(M_{\mathsf{fin}}(v)) \,\} \\
I = M'_{\mathsf{fin}} \oplus \{\, v \mapsto \mathsf{expand}(M'_{\mathsf{fin}}(v), M'_{\mathsf{fin}}) \mid v \in \overline{v} \setminus \mathcal{L} \,\} \\
I' = \mathsf{remove\_singleton\_inst\_keys}(I) \\
\forall v \in \overline{v}.\ \mathsf{expand\_shared}(I'(v), I') \sqsubseteq M_{\mathsf{fin}}(v) \\
\operatorname{dom} M = \operatorname{dom} M' = \overline{v} = \mathcal{N} = \mathsf{uq}(G) \\
R \in \Gamma
\end{array}}{\Gamma \overset{\mathsf{Proc}}{\vdash} R}
$$

Figure 5.8: Mode rule for a procedure with alias tracking

Inst key (InstKey)	κ		
Instantiation state (Inst)	ι	::=	$\mathsf{free}(r)$
		|	$\mathsf{any}(u)$
		|	$\mathsf{bound}(u, \mathcal{P}\, f(\overline{\iota}))$
		|	$\mathsf{higher\_order}(u, \overline{\mu})$
		|	$\mathsf{alias}(\kappa)$
Alias count	r	::=	not_aliased
		|	aliased

Figure 5.9: Instantiation states with annotations on free

During mode analysis, uninstantiated parts of a partially instantiated data structure have the inst free(aliased) rather than free(not_aliased). In most places during mode analysis, free(not_aliased) and free(aliased) insts are treated the same. The exception to this is in the unification of two free variables. While it is still a mode error to attempt to unify two live free(not_aliased) variables, two variables whose insts are free(not_aliased) and free(aliased) may now be unified; after the unification, both will have the inst free(aliased). We do not allow the unification of two variables whose insts are both free(aliased). This is because this would require a free(aliased) variable to record the address of more than one memory location to be filled in when the variable is instantiated. While this could be implemented, it would require significant overhead. As long as a free(aliased) variable can be aliased to only one slot in a data structure, we could leave that slot uninitialised and store its address in the variable's slot in the stack frame. When binding the variable, the code generator could preserve the address in the stack frame avoiding the need for trailing. If we were to allow aliases between two or more slots in data structures, we would need to record the set of addresses of the slots in those data structures themselves, using either terminated pointer chains as in the WAM [2, 156], or circular pointer chains as in PARMA [86, 136, 137]. In that case, instantiating the aliased variable would overwrite the addresses in the data structures, requiring the instantiation to be trailed. This would require every place in the program where backtracking may occur to check whether untrailing is needed. This overhead would be required even in parts of the program that do not use aliases. Unless the program uses aliases heavily, the overhead is likely to outweigh the benefits gained. The PARMA scheme has actually been implemented for Mercury by the HAL project [49] where it is used in conjunction with the imprecise any inst to provide Prolog-style "logical" variables. HAL uses a trail for other types of constraint solvers so there is no extra cost in also using it for Herbrand constraints. Schrijvers and Demoen [120] have also achieved some success in reducing the size of the trail and thus the cost of trailing in this scheme.

5.4.2 Extending the Mercury Abstract Machine

We modify the Mercury abstract machine so that an argument of a procedure whose initial inst is free(aliased) will have its required location passed to the procedure. The code generator reserves an input register for this argument, and it places the address of the memory location to be filled in in this register before calling the procedure. When a variable whose inst is free(aliased) is further instantiated, the code generator emits code to place its value in the memory location that the representation of the free(aliased) variable points to.

5.4.3 Tail Call Optimisation

We now look at how we can use partially instantiated data structures to enable an optimisation that can transform calls into tail calls. A tail call is a call which is the last action to be executed for some execution path through the body of a procedure. Tail calls allow a saving in stack usage because they allow the stack space of the caller to be re-used by the callee. A tail call can be implemented by an unconditional jump. In the case that the tail call is a recursive call, this means that the recursive procedure can be implemented with a simple loop (possibly after low-level transformations to optimise away the decrements and increments of the stack pointer). Tail calls are important because they can allow recursive procedures (or SCCs) to execute in constant stack space, rather than requiring space proportional to the depth of recursion. They can also improve execution time by avoiding the cost of setting up and tearing down stack frames, and, more importantly on modern architectures, by improving locality and thus avoiding cache misses.[7]

If a predicate that would be tail recursive in Prolog is not tail recursive in Mercury, the reason is usually that the standard Mercury implementation returns all output arguments of a call in registers. If an output argument needs to be placed somewhere in memory, this must be done after returning from the call.

For example, consider the recursive clause of '`append(in, in, out)`':

```
append(Xs, Ys, Zs) :-
        Xs = [X | Xs1],
        Zs = [X | Zs0],
        append(Xs1, Ys, Zs0).
```

In WAM-based implementations of Prolog, *last call optimisation* [2] would ensure that this clause is tail recursive. In Mercury, however, the recursive call to `append` would return `Zs0` in a register which then needs to be copied into the correct memory location (i.e. the second argument

[7] Nethercote and Mycroft [109] show that cache misses are a major performance bottle-neck for the lazy functional language Haskell running on a modern architecture. We believe that similar results are likely for Mercury and that cache-related effects are likely to be the main cause of performance improvements due to tail call introduction.

of the cons cell). Actually, without support for partially instantiated data structures the situation is even worse — the mode checker will delay the entire construction unification '`Zs = [X | Zs0]`' until after the recursive call.

Somogyi et al. [131] briefly outlines a solution to these problems: reorder the code so that the construction unifications come before the call, then create a new version of the called predicate which returns the relevant arguments in memory and pass it the addresses in which to place the results.

We have taken this outline and used it as the basis for a new optimisation pass in the Mercury compiler which we call the *last call modulo constructors* optimisation (LCMC).[8] The optimisation relies on the changes to the mode system and abstract machine that we have described earlier in this chapter.

This optimisation has a similar aim to the work of Bigot and Debray [11] which describes an algorithm for deciding when to return output values in registers and when to return them in memory. Their technique requires profiling information (the number of times each procedure is called from each one of its call sites) which could potentially be supplied by the Mercury deep profiler [38, 40]. However, their algorithm would require significant extensions to handle separate compilation. Our implementation of LCMC probably captures most of the benefit with much less implementation complexity.

The LCMC optimisation involves a very simple transformation to move a call to the end of a procedure body. The back end of the compiler can then generate a tail call for this call. The LCMC pass occurs after other passes that may increase the chances of being able to apply LCMC. These include simplification, inlining and accumulator introduction. At this stage, mode analysis has selected an initial scheduling order for each procedure which avoids the construction of partially instantiated terms.

LCMC traverses the goal for each procedure in turn. A procedure is determined to be a candidate for LCMC if any path through the goal contains a final conjunction where the last call in the conjunction is followed only by construction unifications. The algorithm used is described more formally in Figure 5.10 on the next page.

For all conjunctions where LCMC can be applied, the conjunction is re-ordered so that the constructions appear before the call. Now the required mode for the called predicate will be different from the previously-determined mode. We need to find a procedure for the predicate π where any output arguments that appear in the right hand side of one of the construction unifications become pass-by-reference. These arguments will have an initial inst of free(aliased) rather than free(not_aliased). If such a procedure does not already exist, we create it and then run the lcmc transformation on it. If the called predicate is the same as the calling predicate, this will result in a tail recursive procedure.

After running LCMC on a procedure we run a subset of the mode checker on the new body goal. This subset does not do re-ordering and believes the existing mode annotations on calls, but recomputes the annotations on unifications and compound goals and in the process checks that the new definition is mode correct. If the transformation has created a unification between two free(aliased) variables this will manifest itself as a mode error and we abandon the transformation for this procedure.

[8] This term comes from a similar idea described by Wadler [150] which he calls *tail recursion modulo cons*.

Procedure: $\mathsf{lcmc}()$
Input: G_{in}, the goal of the procedure to be transformed
Output: G_{out}, the transformed goal

if
  $G_{in} = \ulcorner \bigvee \langle G_1, \ldots, G_n \rangle \urcorner$
then
  $G_{out} = \ulcorner \bigvee \langle \mathsf{lcmc}(G_1), \ldots, \mathsf{lcmc}(G_n) \rangle \urcorner$
else if
  $G_{in} = \ulcorner$ $\mathsf{if}\ G_1\ \mathsf{then}\ G_2\ \mathsf{else}\ G_3 \urcorner$,
then
  $G_{out} = \ulcorner$ $\mathsf{if}\ G_1\ \mathsf{then}\ \mathsf{lcmc}(G_2)\ \mathsf{else}\ \mathsf{lcmc}(G_3) \urcorner$
else if
  $G_{in} = \ulcorner \bigwedge \langle G_1, \ldots, G_n \rangle \urcorner$
then
  if
    $\exists k \in \{\, 1, \ldots, n-1 \,\} .\ G_k = \ulcorner \pi(\overline{v}) \urcorner$
    $\wedge\ \ \forall i \in \{\, k+1, \ldots, n \,\} .\ G_i = \ulcorner v_i = f(\overline{v}'_i) \urcorner\ \wedge\ G_i : M_i\ \wedge\ (M_i)_{\mathsf{init}}(v_i) = \mathsf{free}$
  then
    $G_{out} = \ulcorner \bigwedge \langle G_1, \ldots, G_{k-1}, G_{k+1}, \ldots, G_n, G_k \rangle \urcorner$
  else
    $G_{out} = \ulcorner \bigwedge \langle G_1, \ldots, G_{n-1}, \mathsf{lcmc}(G_n) \rangle \urcorner$
else
  $G_{out} = G_{in}$

Figure 5.10: The LCMC transformation

Let us look at how the transformation is applied to the procedure `append(in, in, out)` which we will refer to as mode 0 of the predicate `append/3`. The code for this procedure, before LCMC, is shown in Figure 5.11 on the facing page. The comments show the change in instantiation state of the variables after each sub-goal (uniqueness annotations are omitted since they are not relevant to LCMC).

LCMC discovers that in the second arm of the disjunction, construction of `Zs` can be moved before the recursive call. `Zs0` now has an inst of free(aliased) before the call so LCMC tries to find a mode of `append/3` that matches this inst. The required mode is not found so it is created (mode 1) by copying mode 0 and changing the initial inst of `Zs`. The code for mode 0 at this stage is shown in Figure 5.12 on the next page.

LCMC is now run on the new procedure, mode 1. When it re-orders the call and construction it finds that an appropriate mode of `append/3` already exists (i.e. mode 1 itself). The code for mode 1 looks the same as mode 0 except that the mode of the third argument is free(aliased) $\rightarrowtail$ ground rather than `out` (free(not_aliased) $\rightarrowtail$ ground).

Finally, the C code generated by the compiler for the tail recursive mode 1 of `append` is shown in Figure 5.13 on page 110. (We have made the code more readable by e.g. shortening label names and removing type casts.) The arguments `Xs`, `Ys` and `Zs` are passed in registers `r1`, `r2` and `r3`, respectively.

After creating a new stack frame and storing the return address (`succip`) on it, the code checks whether `Xs` (in `r1`) is the empty list (i.e. '`r1 == mkword(mktag(0), mkbody(0))`'), and branches

```
:- mode append(in, in, out).

append(Xs, Ys, Zs) :-
    % Initial insts: Xs -> ground, Ys -> ground, Zs -> free(not_aliased)
    (
        Xs => [],
        Zs := Ys                        % Zs -> ground
    ;
        Xs => [X | Xs1],                % X -> ground, Xs1 -> ground
        append(Xs1, Ys, Zs0),           % call mode 0 of append/3
                                        % Zs0 -> ground
        Zs <= [X | Zs0]                 % Zs -> bound([ground | ground])
    ).                          % Zs -> ground
```

Figure 5.11: Mode 0 of `append/3` before transformation

```
:- mode append(in, in, out).

append(Xs, Ys, Zs) :-
    % Initial insts: Xs -> ground, Ys -> ground, Zs -> free(not_aliased)
    (
        Xs => [],
        Zs := Ys                        % Zs -> ground
    ;
        Xs => [X | Xs1]                 % X -> ground,  Xs1 -> ground
        Zs <= [X | Zs0]                 % Zs -> bound([ground | alias(IK0)])
                                        % Zs0 -> alias(IK0)
                                        % IK0 -> free(aliased)
        append(Xs1, Ys, Zs0)            % call mode 1 of append/3
                                        % IK0 -> ground
    ).                          % Zs -> ground
```

Figure 5.12: Mode 0 of `append/3` after transformation

to label `append_3_1_i3` if it is. If it is not, it falls through to the code for the recursive case. This allocates a new two-word cell on the heap, and places a pointer to this cell tagged with the tag value for cons (1), which represents the value of `Zs`, in `r5`. It places the value of the `Zs` where the caller expects it, which is the memory location whose address the caller passed in `r3`. It fills in the first word of the new cell with `X`, and puts the address of the second into `r4`, which is eventually copied into `r3` to tell the recursive call where to put the value of `Zs0`. Extracting `Xs1` from the second field of the cons cell of `Xs` and putting it into `r1` completes the setup for the recursive call, since `Ys` is already in `r2`. The recursive call is an unconditional jump to the start of the procedure. The code of the base case implements the assignment '`Zs := Ys`' by copying `Ys` to where the caller expects `Zs`. The low-level optimisation passes in the compiler can improve this code further by eliminating unnecessary jumps, making more efficient use of registers and by removing the stack frame allocation altogether (`append/3` does not call any other predicates so it is not necessary to save `succip` on the stack).

The LCMC optimisation does not guarantee a tail call in every procedure it transforms. The

```
Define_entry(append_3_1);
        incr_sp_push(1);
        stackvar(1) = succip;
Define_label(append_3_1_i1);
        if (r1 ==  mkword(mktag(0), mkbody(0)))
                GOTO_LABEL(append_3_1_i3);
        tag_incr_hp(r5, mktag(1), 2);
        field(mktag(1), r5,  0) = field(mktag(1), r1,  0);
        *r3 = r5;
        field(mktag(1), r5, 1) = 0;
        r4 = &field(mktag(1), r5, 1);
        r5 = r1;
        r1 = field(mktag(1), r5,  1);
        r3 = r4;
        GOTO_LABEL(append_3_1_i1);
Define_label(append_3_1_i3);
        *r3 = r2;
        succip = stackvar(1);
        decr_sp_pop(1);
        proceed();
```

Figure 5.13: Generated C code for mode 1 of `append/3` after transformation

transformed call will be a tail call if and only if it returns exactly the same sequence of outputs in exactly the same locations as its containing procedure. If it does not, the call must be followed by code that places the output arguments of the containing procedure in their proper places. In such cases, the optimisation may still be useful as it may allow a tail call somewhere in the new pass-by-reference procedure by reducing the number of output registers of the procedure. This is seen in the `append` example where the call from mode 0 to mode 1 is not a tail call because mode 0 needs to save the value of `Zs` on the stack across the call and then place it in a register before it returns. However, the recursive call from mode 1 to itself is tail recursive because it can place `Zs` in the required location in memory as soon as it is constructed, so there is nothing that needs to be done after the recursive call.

Since the compiler runs LCMC recursively on each new procedure at the point where it is created, we can check whether the optimisation is actually going to give us a tail call somewhere in the new procedure and, if not, abandon the optimisation for this procedure.

The overall effect is that although our optimisation does not depend on the last call being a recursive call, the last call will be a recursive call in most of the situations in which our optimisation can be usefully applied.

In Ross, Overton, and Somogyi [117] we show how this optimisation can be combined with an accumulator introduction optimisation, based on information about the associativity of predicates, to allow further opportunities for tail recursion optimisation. In Table 5.1 on the next page we present some benchmark results showing how the two optimisations can combine synergistically to improve the execution times of programs. We benchmarked a set of small programs on a PC with a 450MHz Intel Pentium III CPU, 512KB of L2 cache, 256MB of memory, running Linux 2.2.12. The benchmark programs all process lists, and they were tested with their input being a list of

30 integers. Their executions were repeated several thousand times to eliminate the uncertainties involved in measuring small time intervals. The results are presented in the form of normalised run times; the raw data is available from the Mercury web site.[9]

The table shows two sets of results: without and with garbage collection. Within each set, the columns show performance without either of the optimisations, with just LCMC, with just accumulator introduction, and with both. In all four cases, all the other optimisations of the Mercury compiler were turned on with the exception of unused argument elimination, because unused argument elimination makes relating a call back to a promise using that call difficult. (We intend to move unused argument elimination after the tail recursion optimisations, but this requires first modifying its implementation to accept the modes produced by LCMC.)

	without gc				with gc			
	None	LCMC	Acc	Both	None	LCMC	Acc	Both
append	100.0	90.4	100.0	90.7	100.0	97.6	99.4	97.7
sumlist	100.0	99.9	80.3	80.2	100.0	99.9	79.2	79.2
nrev	100.0	74.1	13.1	13.9	100.0	96.2	12.5	12.5
sumdbl	100.0	99.9	99.9	90.0	100.0	99.2	99.2	98.6
mirror	100.0	72.9	100.0	10.3	100.0	95.7	99.2	9.5

Table 5.1: Normalised benchmark results for tail call optimisation

The `append` benchmark is the `append/3` predicate we have seen in previous examples. The `sumlist` predicate sums a list of integers. These two benchmarks only achieve a speed increase when the relevant optimisation is enabled.

Naïve reverse (`nrev`) reverses a list using a naïve and inefficient algorithm involving repeated calls to `append/3`. It benefits from both optimisations. LCMC makes the call to `append` tail recursive, which improves the performance of its caller `nrev`. Accumulator introduction allows a speed increase by reducing the big-O complexity of the algorithm (effectively turning it into non-naïve reverse) and by making the `nrev` predicate tail recursive. However turning on LCMC after turning on accumulator introduction on slows the code down slightly because in the accumulator version of nrev, the first argument of append is always a list of length one, and the fixed overhead of the LCMC transformation thus cannot be compensated for by LCMC making `append` tail recursive. The predicate `sumdbl`, which in one traversal both sums a list of integers and returns another list which has each element doubled is a case where we don't get any significant speedups unless both optimisations are turned on. (Such predicates can be expected to be produced by deforestation [138, 152].) The predicate `mirror` also requires both optimisations to turned on for it to become tail recursive. However, mirror can also benefit from LCMC alone because of its use of `append`.

At the moment, Mercury uses the Boehm conservative garbage collector for C [12]. Using this collector requires a function call on every memory allocation, whereas in the no-gc case, Mercury allocates memory by simply incrementing the heap pointer. (The Boehm collector also lacks provision for efficiently recovering memory without garbage collection, e.g. between different repetitions of the same test.) Since memory allocation accounts for a bigger fraction of the runtime of the program with gc than without gc, we get smaller relative speedups with gc than with without

[9] `http://www.mercury.cs.mu.oz.au/`

gc for optimisations that do not eliminate memory allocations themselves. LCMC cannot eliminate memory allocations. Accumulator introduction can eliminate memory allocations, and will do so whenever it reduces the big-O complexity of a predicate that allocates memory every iteration. The `nrev` and `mirror` predicates both benefit from this effect.

5.5 Related Work

Early work on mode analysis [e.g. 95, 96] did not attempt to do alias tracking. Instead, terms containing aliased free variables were mapped to $\top$. Debray and Warren [47] introduced a simple alias analysis which operated locally within each predicate. Each predicate was annotated with an *alias bit* to indicate whether it could create aliasing between any of its arguments. This was an inexpensive way of improving on the status quo at the time, without the need for global analysis, however this approach does not provide the kind of precision we require for a prescriptive mode system.

Analysis of possible aliases between non-ground variables has been studied extensively. This work includes the set sharing domain, Sharing, of Jacobs and Langen [72] (which Codish et al. [32, 33] have since shown to be isomorphic to the domain Pos [88]), and pair sharing domains such as ASub of Søndergaard [133] and the recent work of Lagoon and Stuckey [81]. Such work has somewhat different aims from ours. It is generally designed as a descriptive system to provide information for optimisations, such as occur check reduction [133], where possible rather than definite aliases are required. These analyses also only deal with non-ground aliasing whereas we need to know, particularly for uniqueness analysis, about aliasing between ground terms as well. Lagoon has suggested[10] that his work with Stuckey [81] could be adapted to analysis of ground terms by adding an extra variable for each functor in each term. We do not require definite aliases for uniqueness analysis so it would be interesting to try using this approach (in fact, we believe this could be a good way to add uniqueness analysis to the mode analysis system we will describe in Chapter 6).

The work of Mazur et al. [91, 92], which we have already mentioned in Chapter 3, looks for possible aliasing between ground terms in order to determine when the memory occupied by a term can be re-used. Their system can operate on individual modules rather than whole programs and could be used to automatically infer (or check) uniqueness annotations for modes. However, it does not deal with partially instantiated modes.

The approach to alias tracking that we have taken is based on that suggested by Henderson [65]. However, Henderson does not provide a detailed description of the abstract operations or algorithms required for such an analysis. In particular, he does not mention how to merge insts containing aliases at the end of a branched goal. Also, the mode system described in that work is a descriptive mode system ("free" in his system means only "possibly free") so it can tolerate imprecision.

[10] In personal communication with the author.

5.6 Limitations and Possible Future Work

We have presented an alias tracking system which enhances the Mercury mode system by allowing more programs to be admitted as mode correct. The system does however have two main limitations, namely that it is not expressive enough to do everything that we might want a mode system to do, and that it is quite inefficient. We discuss these two limitations in more detail below.

5.6.1 Limitations on Expressiveness

The alias tracking system we have presented is designed for dealing only with aliasing between variables within a procedure. As such, it cannot track aliases that "escape" from a single procedure — if two live variables are aliased at the end of a procedure, we have not provided a mechanism for tracking that alias after the procedure succeeds. When dealing with unique modes this is not too serious — we can simply make the insts of the the two variables become `shared`. However, if the variables are partially instantiated then we must report a mode error. There are programs we would like to handle which we cannot because of this limitation.

One example is the well-known logic programming benchmark program 'serialise', shown in Figure 5.14.

```
:- pred serialise(list(int), list(int)).

serialise(L, R) :-
        pairlists(L, R, A),
        arrange(A, T),
        numbered(T, 1, N).

:- pred pairlists(list(int), list(int), list(pair(int, int))).

pairlists([X | L], [Y | R], [pair(X, Y) | A]) :-
        pairlists(L, R, A).
pairlists([], [], []).

:- pred arrange(list(pair(int, int)), tree(pair(int, int))).
:- pred numbered(tree(pair(int, int)), int, int).
```

Figure 5.14: Serialise program

The intended use of `serialise/2` is that `L` should be input and `R` should be output. The call to `pairlists/3` traverses `L` and instantiates `R` to a list skeleton of the same length as `L` and instantiates `A` to a list of pairs where the first element is from `L` and the second element is aliased to the corresponding free variable from `R`. The call to `arrange/2` arranges `A` into a binary tree `T` ordered by the first element in each pair. Finally, the call to `numbered/3` instantiates the second element of each pair in the tree, thus also instantiating the elements of `R`.

A set of Mercury mode declarations for this might look like

```
:- mode serialise(in, out).
:- mode pairlists(in, out(listskel(free)), out(listskel(pair(ground, free)))).
:- mode arrange(in(listskel(pair(ground, free))),
```

```
                out(treeskel(pair(ground, free)))).
:- mode numbered(treeskel(pair(ground, free)) >> treeskel(pair(ground, ground)),
                in, out).
```

However, `pairlists/3` and `arrange/2` create aliases between the `free` components of the data structures which are not reflected in these modes so there is no way for the mode system to know that `R` eventually becomes `ground`.

If a loss of mode precision were acceptable, we could use the HAL approach of replacing each occurrence of `free` in the mode declarations with `any`. However, this only allows the mode system to conclude a final inst for `R` of `listskel(any)` rather than `ground`. We could add a "promise" that `R` is ground, but that could not be checked by the compiler, leading to potential undefined behaviour at run time.

Another type of program we would like to be able to handle, but cannot currently, is a form of co-routining where two or more predicates co-operate to instantiate a data structure. This can be used to model client-server processes which in turn can model I/O (where the program acts as a client to an "I/O server") and database access.

One way of modelling a client-server relationship is via *synchronised streams*, that is, by using a list whose elements represent requests from the client to the server and another list whose elements represent the corresponding responses from the server to the client. This technique was first suggested by Landin [82, 83] and has been used to model I/O in functional languages [60]. One problem with this is that the programmer must manually ensure that the two streams remain synchronised. This can get quite difficult in a complex program. Logic programming languages can solve this problem by combining the two streams into one [27, 121, 129]. The client makes a request by partially instantiating an element of the list. The server then carries out the request and instantiates the rest of the element with the result. This is known as the *paradigm of incomplete messages*. For example, a simple case where there are two kinds of request, one for reading from a file and one for writing to a file, might have type, inst and mode declarations like those shown in Figure 5.15 on the next page.

This code would be called with a goal like '`io_client(Actions), io_server(Actions)`'. The client and the server run as co-routines with the server blocking until it receives the next request from the client and the client then blocking until the server instantiates the result. Unfortunately, the mode declarations given above are not quite right because they would require the entire stream of requests to be produced by the client before they are passed to the server. We need something like the *iterated mode segments* of Somogyi [129] or the *recursive modes* suggested by Conway [38, chapter 6], neither of which have been implemented, partly due to the problem of keeping track of aliases in the mode analysis. In Chapter 6 we will look at a constraint-based approach to mode analysis which should be easier to extend to handle this sort of program.

5.6.2 Performance Limitations

The alias tracking system we have presented in this chapter requires a lot of book-keeping which adds a very large overhead to the cost of mode analysis, both in terms of memory use and time. The cost increase is present even when analysing procedures for which the alias tracking provides no extra useful precision to the mode analysis. For example, Table 5.2 on the facing page shows

```
:- type io_action
        --->    read(
                        string, % filename
                        char    % char read
                )
        ;       write(
                        string, % filename
                        char    % char to write
                ).

:- inst request
        --->    read(ground, free)
        ;       write(ground, ground).

:- pred io_client(list(io_action)).
:- mode io_client(out(listskel(request))).

:- pred io_server(list(io_action)).
:- mode in_server(listskel(request) >> ground).
```

Figure 5.15: Declarations for a client/server system using streams

mode analysis times for some common logic programming benchmark programs. Times are in milliseconds. We measured the time for 100 repeats of mode checking each benchmark and divided the result by 100. The tests were run on a Dell Optiplex GX240 PC with a 1600MHz Intel Pentium IV, 256KB of L2 cache and 512MB of memory running Linux kernel 2.4.16. The column labelled "no alias" contains times for the mode system without the alias tracking extension. The column labelled "alias" contains times for the mode system with alias tracking. None of these programs require the extra precision provided by alias tracking, however it can be seen that adding the alias tracking analysis usually slows down the mode analysis time by around 40 to 75 per cent.

	no alias	alias	alias / no alias
cqueens	9.5	13.3	1.40
crypt	36.9	53.3	1.44
deriv	42.1	73.2	1.73
nrev	31.0	27.5	0.88
poly	39.3	69.4	1.76
primes	4.4	7.5	1.70
qsort	84.7	63.2	0.74
queens	8.0	11.0	1.37
query	14.0	23.1	1.65
tak	3.2	5.0	1.56

Table 5.2: The effect of alias tracking on mode analysis times

It is interesting to note that in two of the benchmarks, nrev and qsort, the mode checker is actually faster with alias tracking than without. The mode checking time for these two benchmarks is dominated by analysing code that builds large static lists which the benchmarks use as input data. A unification of the form $\mathtt{V}_n$ `=` `[`$\mathtt{X}_1$`, ..., `$\mathtt{X}_n$`]` is converted into a conjunction of unifications `V`$_0$ `= [],` `V`$_1$ `= [X`$_n$ `| V`$_0$`], ..., V`$_n$ `= [X`$_1$ `| V`$_{n-1}$`]` in superhomogeneous normal form (for nrev $n = 30$ and for qsort $n = 100$). The mode checker calls abstract_unify_inst_functor for each of these unifications (refer to Definition 5.3.9 on page 98) which in turn calls abstract_unify_inst for each argument. In this case, the second argument of each functor will already have an inst of alias(κ) for some κ. This means that the call to abstract_unify_inst for that inst will simply be comparing alias(κ) with alias(κ) which it can do immediately rather than having to take the glb of two more complicated insts, thus expediting the process. This sort of improvement is only likely to outweigh the costs of alias tracking in extreme cases such as this.

In order to improve analysis times with alias tracking, Becket [10] has recently suggested an alternative, demand driven approach where we keep only minimal book-keeping information about aliases and do the full alias tracking analysis for a variable only if we discover that we need to (e.g. if we need to prove that a variable is `unique` but are unable to show that without alias tracking).

The constraint-based approach of Chapter 6 offers another alternative approach to alias tracking. However, as we shall see in that chapter, it too has performance problems that we still need to overcome.

Chapter 6

A Constraint-Based Approach to Mode Analysis

As well as the limitations already noted, the mode analysis algorithm presented in the preceding chapters is very complicated. It combines several conceptually distinct stages of mode analysis into a single pass. This makes modifications to the algorithm (e.g. to include new functionality) quite difficult. The algorithm is also quite inefficient when analysing code involving complex modes.

In this chapter, we present a new approach to mode analysis of Mercury programs which attempts to solve some of the problems of the systems we have presented in previous chapters. We separate mode checking into several distinct stages and use a constraint-based approach to naturally express the constraints that arise during mode analysis. We believe that this approach makes it easier to implement an extensible mode analysis system for Mercury that can overcome the limitations of the current system.

We associate with each program variable a set of *positions*, which correspond to nodes in its type graph. The key idea of the new mode system is to identify, for each position in the type of a variable, where that position is produced, i.e. which goal binds that part of the variable to a function symbol. We associate to each node in the type graph, and each goal in which the program variable occurs, a Boolean variable that indicates whether the node in the graph of that program variable is bound within the goal. Given these Boolean variables, we can describe the constraints that arise from correct moding in terms of Boolean constraints. We can then represent and manipulate these descriptions using standard data structures such as reduced ordered binary decision diagrams (ROBDDs) [19, 20]. By allowing constraints between individual positions in different data structures, we obtain a much more precise analysis than the mode system of Chapter 3, while avoiding the complications of the alias tracking system of Chapter 5.

In Section 6.1 we give the background information the rest of the chapter depends on. In Section 6.2 we give a simplified example of our constraint-based system before presenting the full system in Section 6.3. In Section 6.4 we show how the results of the analysis are used to select an execution order for goals in a conjunction. In Section 6.5 we look at some of the issues with implementing the system. In Section 6.6 we give some experimental results of the performance of the constraint-based system. In Section 6.7 we look at some related work. In Section 6.8 we look at some of the limitations of the new system and possible future work.

6.1 Background

6.1.1 Programs

In this chapter, we assume that in all unifications (after transformation to superhomogeneous normal form), neither side is a variable that does not appear outside the unification itself. (Unifications that do not meet this requirement cannot influence the execution of the program and can thus be deleted.) For simplicity, we also assume that all negations have been replaced by if-then-elses (one can replace $\ulcorner \neg G \urcorner$ with $\ulcorner$ $\mathsf{if}\, G\, \mathsf{then} \bigvee \langle\rangle\, \mathsf{else} \bigwedge \langle\rangle$ $\urcorner$[1]). Rather than using explicit existential quantification goals to define the scope of variables, we assume that each goal has been annotated with the set of variables which are *nonlocal* to that goal (see Definition 6.1.2, below). For now, we also leave out discussion of higher-order code. Apart from the omission of negation and existential quantification goals, we use the abstract syntax given in Figure 3.1 on page 30. We use v, w, x, y and z, with and without subscripts and primes, to represent program variables.

In order to describe where a variable becomes bound, our algorithms need to be able to uniquely identify each subgoal of a predicate body. The code of a subgoal itself cannot serve as its identifier, since a piece of code may appear more than once in a predicate definition. We therefore use *goal paths* for this purpose. A goal path consists of a sequence of *path components*. We use ϵ to represent the path with zero components, which denotes the entire procedure body.

- If the goal at path p is a conjunction, then the goal path $p.\mathsf{c}_n$ denotes its nth conjunct.
- If the goal at path p is a disjunction, then the goal path $p.\mathsf{d}_n$ denotes its nth disjunct.
- If the goal at path p is an if-then-else, then the goal path $p.\mathsf{c}$ denotes its condition, $p.\mathsf{t}$ denotes its then-part, and $p.\mathsf{e}$ denotes its else-part.

Definition 6.1.1 The *parent* of goal path $p.\mathsf{c}_n$, $p.\mathsf{d}_n$, $p.\mathsf{c}$, $p.\mathsf{t}$, or $p.\mathsf{e}$, is goal path p. The function $\mathsf{parent}(p)$ maps a goal path to its parent. $\triangleleft$

Definition 6.1.2 Let $\nu(G)$ be the set of variables that occur within a goal, G. Let $\eta(G) \subseteq \nu(G)$ be the set of variables that are *nonlocal* to goal G, i.e. occur both inside and outside G. For convenience, we also define ν and η for a set of goals, S: $\nu(S) = \bigcup \{\, \nu(G) \mid G \in S \,\}$, and $\eta(S) = \bigcup \{\, \eta(G) \mid G \in S \,\}$. $\triangleleft$

Since each goal path uniquely identifies a goal within a predicate body, we sometimes apply operations on goals to goal paths.

6.1.2 Deterministic Regular Tree Grammars

In order to be able to express all the different useful modes on a program variable, we must be able to talk about each of the individual parts of the terms which that program variable will be able to take as values. To do so in a finite manner, we use *regular trees*, expressed as *tree grammars* [36, 56, 57].

[1]The empty disjunction $\ulcorner \bigvee \langle\rangle \urcorner$ is a goal that always fails, whereas the empty conjunction $\ulcorner \bigwedge \langle\rangle \urcorner$ always succeeds.

A *tree* over a signature Σ is a ground term in the set $\tau(\Sigma)$, as defined in Section 2.1.2.

A *tree grammar* r over signature Σ and *non-terminal set* NT is a finite sequence of *production rules* of the form $x \rightarrow t$ where $x \in NT$ and t is of the form $f(x_1, \ldots, x_n)$ where $f/n \in \Sigma$ and $\{ x_1, \ldots, x_n \} \subseteq NT$. A tree grammar is *deterministic regular* if for each $x \in NT$ and $f/n \in \Sigma$, there can be at most one rule of the form $x \rightarrow f(x_1, \ldots, x_n)$.

For brevity we shall often write tree grammars in a more compressed form. We use $x \rightarrow t_1; t_2; \cdots; t_n$ as shorthand for the sequence of production rules: $x \rightarrow t_1$, $x \rightarrow t_2$, $\ldots$, $x \rightarrow t_n$.

6.1.3 Types

Recall that types in Mercury are polymorphic Hindley-Milner types [69, 98]. *Type expressions* (or *types*) are terms in the language $\tau(\Sigma_{type}, V_{type})$ where Σ_{type} are *type constructors* and variables V_{type} are *type parameters*. Each type constructor $f/n \in \Sigma_{type}$ must have a definition.

Definition 6.1.3 A *type definition* for f/n is of the form

$$\texttt{:- type}\ \ f(v_1, \ldots, v_n)\ \ \texttt{--->}\ \ f_1(t^1_1, ..., t^1_{m_1}); \cdots; f_k(t^k_1, ..., t^k_{m_k}).$$

where $v_1, \ldots, v_n$ are distinct type parameters, $\{ f_1/m_1, \ldots, f_k/m_k \} \subseteq \Sigma_{tree}$ are distinct tree constructor/arity pairs, and $t^1_1, \ldots, t^k_{m_k}$ are type expressions involving at most parameters $v_1, \ldots, v_n$. If all occurrences of f/n on the right hand side of the definition are in the form $f(v_1, \ldots, v_n)$ then we say that the type definition is *regular* [101]. $\triangleleft$

Clearly, we can view the type definition for f as simply a sequence of production rules over signature Σ_{tree} and non-terminal set $\tau(\Sigma_{type}, V_{type})$.

In order to avoid type expressions that depend on an infinite number of types, we restrict the type definitions to be regular. (Regularity ensures that for any type t, $\mathsf{grammar}(t)$, defined below, is finite.)

We can associate with each (non-parameter) type expression the production rules that define the topmost symbol of the type. Let t be a type expression of the form $f(t_1, \ldots, t_n)$ and let f/n have type definition of the form in Definition 6.1.3. We define $\mathsf{rules}(t)$ to be the production rules:

$$\begin{array}{rcl}
\theta(f(v_1, \ldots, v_n)) & \rightarrow & f_1(\theta(t^1_1), \ldots, \theta(t^1_{m_1})) \\
& \vdots & \\
\theta(f(v_1, \ldots, v_n)) & \rightarrow & f_k(\theta(t^k_1), \ldots, \theta(t^k_{m_k})))
\end{array}$$

where $\theta = \{ v_1/t_1, \ldots, v_n/t_n \}$ substitutes t_i for v_i. If $t \in V_{type}$ we define $\mathsf{rules}(t)$ to be the empty sequence.

We can extend this notation to associate a tree grammar with a type expression. Let $\mathsf{grammar}(t)$ be the sequence of production rules recursively defined by

$$\mathsf{rules}(t) \mathbin{++} \mathsf{grammar}(\theta(t^1_1)) \mathbin{++} \cdots \mathbin{++} \mathsf{grammar}(\theta(t^k_{m_k}))$$

where the ++ operation concatenates sequences of production rules, removing second and later occurrences of duplicate production rules.

We call each nonterminal in a set of production rules a *position*, since we will use them to describe positions in terms; each position is the root of one of the term's subterms. We also sometimes call positions *nodes*, since they correspond to nodes in type trees.

Example 6.1.1. Consider the type definitions

```
:- type list(T) ---> [] ; [T | list(T)].
:- type abc ---> a ; b ; c.
```

The grammar for the type `list(abc)` is

$$\begin{array}{rcl} \mathsf{list(abc)} & \rightarrow & [\,] \ ; \ [\mathsf{abc}|\mathsf{list(abc)}] \\ \mathsf{abc} & \rightarrow & a \ ; \ b \ ; \ c \end{array}$$

There are two nonterminals and thus two positions in the grammar: $\mathsf{list(abc)}$ and abc which correspond to the or-nodes in Figure 2.2 on page 22. $\triangleleft$

Mode inference and checking takes place after type checking, so we assume that we know the type of every variable appearing in the program.

6.1.4 Instantiations and Modes

Insts are also defined using tree grammars. The differences are that no inst associated with a predicate involves instantiation parameters (i.e. no polymorphic modes — although these are a possible extension, as we have seen in Chapter 4), and there are two base insts: free and ground representing completely uninitialised variables, and completely bound terms. Instantiation expressions are terms in $\tau(\Sigma_{inst}, V_{inst})$.

Definition 6.1.4 An *instantiation definition* for $g \in \Sigma_{inst}$ is of the form:

`:- inst` $g(\omega_1, ..., \omega_n)$ `==` `bound`$(f_1(\iota^1_1, ..., \iota^1_{m_1}); \cdots; f_k(\iota^k_1, ..., \iota^k_{m_k}))$.

where $\omega_1, \ldots, \omega_n$ are distinct instantiation parameters, $\{ f_1/m_1, \ldots, f_k/m_k \} \subseteq \Sigma_{tree}$ are distinct function symbols, and $\iota^1_1, \ldots, \iota^k_{m_k}$ are instantiation expressions in $\tau(\Sigma_{inst} \cup \{ \mathsf{free}, \mathsf{ground} \}, V_{inst})$. $\triangleleft$

We can associate a set of production rules $\mathsf{rules}(\iota)$ with an instantiation expression ι just as we do for type expressions. For the base instantiations we define $\mathsf{rules(free)} = \mathsf{rules(ground)} = \emptyset$.

Definition 6.1.5 A goal that changes the instantiation state of a position from free to bound is said to *produce* or *bind* that position; a goal that requires the initial instantiation state of a position to be bound is said to *consume* that position. $\triangleleft$

6.2 Simplified Example

The motivation for our constraint-based mode analysis system is to avoid the problems in the current system. In order to do, so we break the mode analysis problem into phases. The first phase determines which subgoals produce which variables, while the second uses that information to determine an execution order for the procedure. For now, we will focus on the first task; we will return to the second in Section 6.4.

For ease of explanation, we will first show a simplified form of our approach. This simplified form requires variables to be instantiated all at once (i.e. the only two instantiation states it recognises are `free` and `ground`), and requires all variables to eventually reach the ground state. This avoids the complexities that arise when different parts of variables are bound at different times, or some parts are left unbound. We will address those complexities when we give the full algorithm in Section 6.3.

6.2.1 Constraint Generation

The algorithm associates several constraint variables with each program variable. With every program variable v, we associate a family of constraint variables of the form v_p; v_p is true iff v is produced by the goal at path p in the predicate body.

We explain the algorithm using `append/3`. The code below is shown after transformation by the compiler into *hyperhomogeneous form.* Hyperhomogeneous form is superhomogeneous form with the additional restriction that each variable appears in at most one argument of one functor. The compiler does this by adding extra unifications if necessary.

```
:- pred append(list(T), list(T), list(T)).

append(A, B, C) :-
    (
        A = [],
        B = C
    ;
        A = [AH | AT],
        C = [CH | CT],
        AH = CH,
        append(AT, B, CT)
    ).
```

We examine the body and generate constraints from it. The body is a disjunction, so the constraints we get simply specify, for each variable nonlocal to the disjunction, that if the disjunction produces a variable, then all disjuncts must produce that variable, while if the disjunction does not produce a variable, then no disjunct may produce that variable. For `append`, this is expressed by the constraints:

$$
\begin{array}{c}
(A_\epsilon \leftrightarrow A_{\mathsf{d}_1}) \wedge (B_\epsilon \leftrightarrow B_{\mathsf{d}_1}) \wedge (C_\epsilon \leftrightarrow C_{\mathsf{d}_1}) \wedge \\
(A_\epsilon \leftrightarrow A_{\mathsf{d}_2}) \wedge (B_\epsilon \leftrightarrow B_{\mathsf{d}_2}) \wedge (C_\epsilon \leftrightarrow C_{\mathsf{d}_2})
\end{array}
$$

We then process the disjuncts one after another. Both disjuncts are conjunctions. When processing a conjunction, our algorithm considers each variable occurring in the conjunction that has more than one potential producer. If a variable is nonlocal to the conjunction, then it may be produced either inside or outside the conjunction; if a variable is shared by two or more conjuncts, then it may be produced by any one of those conjuncts. The algorithm generates constraints that make sure that each variable has exactly one producer. If the variable is local, the constraints say that exactly one conjunct must produce the variable. If the variable is nonlocal, the constraints say that at most one conjunct may produce the variable.

In the first disjunct, there are no variables shared among the conjuncts, so the only constraints we get are those ones that say that each nonlocal is produced by the conjunction iff it is produced by the only conjunct in which it appears:

$$(A_{\mathsf{d}_1} \leftrightarrow A_{\mathsf{d}_1.\mathsf{c}_1}) \wedge (B_{\mathsf{d}_1} \leftrightarrow B_{\mathsf{d}_1.\mathsf{c}_2}) \wedge (C_{\mathsf{d}_1} \leftrightarrow C_{\mathsf{d}_1.\mathsf{c}_2})$$

The first conjunct in the first disjunct yields no nontrivial constraints. Intuitively, the lack of constraints from this goal reflects the fact that '`A = []`' can be used both to produce `A` and to test its value.

The second conjunct in the first disjunct yields one constraint, which says that the goal '`B = C`' can be used to produce at most one of `B` and `C`:

$$\neg(B_{\mathsf{d}_1.\mathsf{c}_2} \wedge C_{\mathsf{d}_1.\mathsf{c}_2})$$

For the second disjunct, we generate constraints analogous to those for the first conjunct for the nonlocal variables. But this disjunct, unlike the first, contains some shared local variables: `AH`, `CH`, `AT` and `CT`, each of which appears in two conjuncts. Constraints for these variables state that each of these variables must be produced by exactly one of the conjuncts in which it appears.

$$\begin{array}{l}(A_{\mathsf{d}_2} \leftrightarrow A_{\mathsf{d}_2.\mathsf{c}_1}) \wedge (B_{\mathsf{d}_2} \leftrightarrow B_{\mathsf{d}_2.\mathsf{c}_4}) \wedge (C_{\mathsf{d}_2} \leftrightarrow C_{\mathsf{d}_2.\mathsf{c}_2}) \wedge \\ \neg(AH_{\mathsf{d}_2.\mathsf{c}_1} \leftrightarrow AH_{\mathsf{d}_2.\mathsf{c}_3}) \wedge \neg(CH_{\mathsf{d}_2.\mathsf{c}_2} \leftrightarrow CH_{\mathsf{d}_2.\mathsf{c}_3}) \wedge \\ \neg(AT_{\mathsf{d}_2.\mathsf{c}_1} \leftrightarrow AT_{\mathsf{d}_2.\mathsf{c}_4}) \wedge \neg(CT_{\mathsf{d}_2.\mathsf{c}_2} \leftrightarrow CT_{\mathsf{d}_2.\mathsf{c}_4})\end{array}$$

The first conjunct in the second disjunct shows how we handle unifications of the form $\ulcorner x = f(y_1, ..., y_n)\urcorner$ where $n > 0$. The key to understanding the behaviour of our algorithm in this case is knowing that it is trying to decide between only two alternatives: either this unification takes all the y_is as input and produces x, or it takes x as input and produces all the y_is. This is contrary to most people's experience, because in real programs, unifications of this form can also be used in ways that make no bindings, or that produce only a subset of the y_i. However, using this unification in a way that requires both x and some or all of the y_i to be input is possible only if those y_i have producers outside this unification. When we transform the program into hyperhomogeneous form, we make sure that each unification of this form has fresh variables on the right hand side. So if a y_i could have such a producer, it would be replaced on the right hand side of the unification with a new variable y_i', with the addition of a new unification $y_i = y_i'$. That is, we convert unifications that take both x and some or all of the y_i to be input into unifications that take only x as input, and produce all the variables on the right hand side.

If some of the variables on the right hand side appear only once then those variables must be unbound, and using this unification to produce x would create a nonground term. Since the simplified approach does not consider nonground terms, in such cases it generates an extra constraint that requires x to be input to this goal.

In the case of '`A = [AH | AT]`', both `AH` and `AT` appear elsewhere so we get the constraints:

$$(AH_{\mathsf{d}_2.\mathsf{c}_1} \leftrightarrow AT_{\mathsf{d}_2.\mathsf{c}_1}) \wedge \neg(A_{\mathsf{d}_2.\mathsf{c}_1} \wedge AH_{\mathsf{d}_2.\mathsf{c}_1})$$

The first says that either this goal produces all the variables on the right hand side, or it produces none of them. In conjunction with the first, the second constraint says that the goal cannot produce both the variable on the left hand side, and all the variables on the right hand side.

The constraints we get for '`C = [CH | CT]`' are analogous:

$$(CH_{\mathsf{d}_2.\mathsf{c}_2} \leftrightarrow CT_{\mathsf{d}_2.\mathsf{c}_2}) \wedge \neg(C_{\mathsf{d}_2.\mathsf{c}_2} \wedge CH_{\mathsf{d}_2.\mathsf{c}_2})$$

The equation '`AH = CH`' acts just as the equation in the first disjunct, generating:

$$\neg(AH_{\mathsf{d}_2.\mathsf{c}_3} \wedge CH_{\mathsf{d}_2.\mathsf{c}_3})$$

The last conjunct is a call, in this case a recursive call. We assume that all calls to a predicate in the same strongly-connected component (SCC) of the static call graph as the caller have the same mode. We are currently inferring this mode so we do not know yet exactly what it is, but already have the constraint variables we need.[2] This means that the call produces its ith argument iff the predicate body produces its ith argument. This leads to one constraint for each argument position:

$$(AT_{\mathsf{d}_2.\mathsf{c}_4} \leftrightarrow A_\epsilon) \wedge (B_{\mathsf{d}_2.\mathsf{c}_4} \leftrightarrow B_\epsilon) \wedge (CT_{\mathsf{d}_2.\mathsf{c}_4} \leftrightarrow C_\epsilon)$$

This concludes the set of constraints generated by our algorithm for `append`.

6.2.2 Inference and Checking

The constraints we generate for a predicate can be used to infer its modes. Projecting onto the head variables, the constraint set we just built up has five different solutions, so `append` has five modes:

$\neg A_\epsilon \wedge \neg B_\epsilon \wedge \neg C_\epsilon$	`append(in, in, in)`
$\neg A_\epsilon \wedge B_\epsilon \wedge \neg C_\epsilon$	`append(in, out, in)`
$A_\epsilon \wedge \neg B_\epsilon \wedge \neg C_\epsilon$	`append(out, in, in)`
$A_\epsilon \wedge B_\epsilon \wedge \neg C_\epsilon$	`append(out, out, in)`
$\neg A_\epsilon \wedge \neg B_\epsilon \wedge C_\epsilon$	`append(in, in, out)`

Of these five modes, two (`append(in, in, out)` and `append(out, out, in)`) are what we call *principal* modes. The other three are *implied* modes, because their existence is implied by the existence of the principal modes; changing the mode of an argument from `out` to `in` makes the job of a predicate strictly easier. In the rest of the chapter, we assume that every predicate's set

[2]This assumption somewhat restricts the set of allowable well-moded programs. However, we have not found this to be an unreasonable restriction in practice. We have not come across any cases in typical Mercury programs where one would want to make a recursive call in a different mode. Also, the restriction can be overcome simply by giving a set of mode declarations for at least one predicate in the SCC, as we will see in Section 6.3.3.

of modes is *downward closed*, which means that if it contains a mode *pm*, it also contains all the modes implied by *pm*. In practice, the compiler generates code for a mode only if it is declared or it is a principal mode, and synthesises the other modes in the caller by renaming variables and inserting extra unifications. This synthesis does the superhomogeneous form equivalent of replacing '`append([1], [2], [3])`' with '`append([1], [2], X), X = [3]`'.

When we are inferring modes, we can filter out the generated implied modes, and associate only the principal modes with the predicate. Since we generate separate code for each mode of a predicate, this can reduce the sizes of executables. When we are checking declared modes, we check that all the declared modes are either a principal or implied mode. A declared mode that does not pass this test is incorrect, and we generate an error message for it.

Each solution also implicitly assigns modes to the primitive goals in the body, by specifying where each variable is produced. For example, the solution that assigns true to the constraint variables C_ϵ, $C_{\mathtt{d}_1}$, $C_{\mathtt{d}_1.\mathtt{c}_2}$, $C_{\mathtt{d}_2}$, $AH_{\mathtt{d}_2.\mathtt{c}_1}$, $AT_{\mathtt{d}_2.\mathtt{c}_1}$, $C_{\mathtt{d}_2.\mathtt{c}_2}$, $CH_{\mathtt{d}_2.\mathtt{c}_3}$, $CT_{\mathtt{d}_2.\mathtt{c}_4}$ and false to all others, which corresponds to the mode `append(in,in,out)`, also shows that '`A = [AH|AT]`' is a deconstruction (i.e. uses the fields of `A` to define `AH` and `AT`) while '`C = [CH|CT]`' is a construction (i.e. it binds `C` to a new heap cell).

In most cases, the values of the constraint variables of the head variables uniquely determine the values of all the other constraint variables. Sometimes, there is more than one set of value assignments to the constraint variables in the body that is consistent with a given value assignment for the constraint variables in the head. In such cases, the compiler can choose the assignment it prefers. For example, if two assignments differ only in that the first causes a variable to be produced by a unification while the second causes it to be produced by a predicate call, and the called predicate happens to be nondeterministic, i.e. able to succeed more than once, in the mode in which it is called, then the compiler should prefer the first assignment, since it is likely to lead to more efficient code.

6.3 Full Mode Inference

6.3.1 Expanded Grammars

We now consider the problem of handling programs in which different parts of a variable may be instantiated by different goals. We need to ensure that if two distinct positions in a variable may have different instantiation behaviour, then we have a way of referring to each separately. Hence we need to expand the type grammar associated with that variable.

We begin with an empty grammar and with the original code of the predicate expressed in superhomogeneous normal form. We modify both the grammar and the predicate body during the first stage of mode analysis.

If the unification $\ulcorner x = f(y_1, \ldots, y_n) \urcorner$ appears in the definition of the predicate, then

- if x has no grammar rule for functor f/n, add a rule $x \to f(y_1, \ldots, y_n)$, and for each y_i which already occurs on the left hand side of a grammar rule or in the head of the clause, replace each occurrence of y_i in the program by y_i' and add a unification $y_i = y_i'$.
- if x has a grammar rule $x \to f(z_1, \ldots, z_n)$ replace the unification by the conjunction $\ulcorner \bigwedge \langle x = f(z_1, \ldots, z_n), y_1 = z_1, \ldots, y_n = z_n \rangle \urcorner$.

This process may add unifications of the form $y = z$ where one of y and z occurs nowhere else. Such unifications can be safely removed.

After processing all such unifications, add a copy of the rules in $\mathsf{rules}(t)$ for each grammar variable v of type t which does not have them all.

Example 6.3.1. The superhomogeneous form of the usual source code for append has (some variant of)

```
A = [AH | AT], C = [AH | CT], append(AT,B,CT)
```

as its second clause, which our algorithm replaces with

```
A = [AH | AT], C = [CH | CT], AH = CH, append(AT,B,CT)
```

yielding the form we have shown in Section 6.2. The expanded grammar I computed for append is

$$\begin{array}{ll} A \rightarrow [] \ ; \ [AH|AT] & AT \rightarrow [] \ ; \ [AE|AT] \\ B \rightarrow [] \ ; \ [BE|B] & \\ C \rightarrow [] \ ; \ [CH|CT] & CT \rightarrow [] \ ; \ [CE|CT] \end{array}$$

Note that the grammar rules for `A` and `C` have been "unrolled" more than the rules for `B`. This allows us, e.g. to refer to the position AH (representing both the variable `AH` and the head element of `A`) separately to the elements in `AT`, which are represented by position AE. This is not necessary for `B`. ◁

The nonterminals of this grammar constitute the set of positions for which we will create constraint variables when we generate constraints from the predicate body, so from now we will use "nonterminal" and "position" (as well as "node") interchangeably. Note that there is a nonterminal denoting the top-level functor of every variable, and that some variables (e.g. `A`) have other nonterminals denoting some of their non-top-level functors as well. Note also that a nonterminal can fulfil both these functions, when one variable is *strictly* part of another. For example, the nonterminal AH stands both for the variable `AH` and the first element in any list bound to variable `A`, but the variables `B` and `C`, which are unified on some computation paths, each have their own nonterminal.

A predicate needs three Boolean variables for each position v:

1. v_{in} is true if the position is produced outside the predicate,

2. v_{ϵ} is true if the position is produced inside the predicate, and

3. v_{out} is true if the position is produced *somewhere* (either inside or outside the predicate).

Note that $v_{\mathsf{out}} \leftrightarrow v_{\mathsf{in}} \vee v_{\epsilon}$.

Definition 6.3.1 Let $\alpha(p/n)$ be the tuple $\langle x_1, \ldots, x_n \rangle$ of head variables (i.e. formal parameters) for predicate p/n. ◁

Definition 6.3.2 For an expanded grammar I and position x, we define the *immediate descendants* of x as

$$\delta_I(x) = \bigcup_{x \to f(y_1,\ldots,y_n) \ \in \ I} \{ \, y_1, \ldots, y_n \, \}$$

and the set of positions *reachable* from x as

$$\rho_I(x) = \{ \, x \, \} \cup \bigcup_{x \to f(y_1,\ldots,y_n) \ \in \ I} \{ \, \rho_I(y) \ \mid \ y \in \{ \, y_1, \ldots, y_n \, \} \, \} \qquad \triangleleft$$

When we are generating constraints between two variables (which will always be of the same type) we will need to be able to refer to positions within the two variables which *correspond* to each other, as e.g. AH and CH denote corresponding positions inside `A` and `C`. The notion of correspondence which we use allows for the two variables to have been expanded to different extents in the expanded grammar. For example, `A` has more descendant nonterminals in `append`'s grammar than `B`, even though they have the same type. In the unification '`A = B`', the nonterminal B would correspond to AT as well as A.

Definition 6.3.3 For expanded grammar I and positions x and y, we define the set of pairs of *corresponding nodes* in x and y as

$$\chi_I(x, y) = \{ \, \langle x, y \rangle \, \} \ \cup \bigcup_{\langle v,w \rangle \ \in \ \chi'_I(x,y)} \chi_I(v, w)$$

where

$$\mathsf{lhs}(I) = \{ \, x \ \mid \ \exists f/n \in \Sigma_{tree}. \ x \to f(y_1, \ldots, y_n) \ \in \ I \, \}$$

and

$$\chi'_I(x,y) = \begin{cases} \displaystyle\bigcup_{\substack{x \to f(v_1,\ldots,v_n) \ \in \ I \\ y \to f(w_1,\ldots,w_n) \ \in \ I}} \{ \, \langle v_1, w_1 \rangle, \ldots, \langle v_n, w_n \rangle \, \} & \text{if } x \in \mathsf{lhs}(I) \text{ and } y \in \mathsf{lhs}(I), \\ \displaystyle\bigcup_{x \to f(v_1,\ldots,v_n) \ \in \ I} \{ \, \langle v_1, y \rangle, \ldots, \langle v_n, y \rangle \, \} & \text{if } x \in \mathsf{lhs}(I) \text{ and } y \notin \mathsf{lhs}(I), \\ \displaystyle\bigcup_{y \to f(w_1,\ldots,w_n) \ \in \ I} \{ \, \langle x, w_1 \rangle, \ldots, \langle x, w_n \rangle \, \} & \text{if } x \notin \mathsf{lhs}(I) \text{ and } y \in \mathsf{lhs}(I), \\ \emptyset & \text{otherwise.} \end{cases}$$

For convenience, we also define χ for a pair of n-tuples:

$$\chi_I(\langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_n \rangle) = \bigcup_{i=1}^{n} \chi_I(x_i, y_i) \qquad \triangleleft$$

Definition 6.3.4 Given an expanded grammar I and a rule $x \ \to \ f(y_1, \ldots, y_n) \in I$ we say that x is the *parent node* of each of the nodes $y_1, \ldots, y_n$. $\triangleleft$

6.3.2 Mode Inference Constraints

We ensure that no variable occurs in more than one predicate, renaming them as necessary. We construct an expanded grammar I for P, the program module being compiled. We next group the predicates in the module into strongly-connected components (SCCs), and process these SCCs bottom up, creating a function $\mathcal{C}_{\mathsf{SCC}}$ for each SCC. This represents the Boolean constraints that we generate for the predicates in that SCC. The remainder of this section defines $\mathcal{C}_{\mathsf{SCC}}$.

The constraint function $\mathcal{C}_{\mathsf{SCC}}(I, S)$ for an SCC S is the conjunction of the constraint functions $\mathcal{C}_{\mathsf{Pred}}(I, \pi/n)$ we generate for all predicates π/n in that SCC, i.e.

$$\mathcal{C}_{\mathsf{SCC}}(I, S) = \bigwedge_{\pi/n \in S,\ \ulcorner \pi(x_1,\ldots,x_n) \leftarrow G \urcorner \in P} \mathcal{C}_{\mathsf{Pred}}(I, \ulcorner \pi(x_1,\ldots,x_n) \leftarrow G \urcorner)$$

The constraint function we *infer* for a predicate p/n is the constraint function of its SCC, i.e. $\mathcal{C}_{\mathsf{Inf}}(I, \pi/n) = \mathcal{C}_{\mathsf{SCC}}(I, S)$ for each $\pi/n \in S$. $\mathcal{C}_{\mathsf{Inf}}(I, \pi/n)$ may be stricter than $\mathcal{C}_{\mathsf{Pred}}(I, \pi/n)$ if π/n is not alone in its SCC because it may be constrained by other members of the SCC. For predicates defined in other modules, we derive their $\mathcal{C}_{\mathsf{Inf}}$ from their mode declarations using the mechanism we will describe in Section 6.3.3.

$\mathcal{C}_{\mathsf{Pred}}$ itself is the conjunction of two functions: $\mathcal{C}_{\mathsf{Struct}}$, the structural constraints relating in and out variables, and $\mathcal{C}_{\mathsf{Goal}}$, the constraints for the predicate body goal:

$$\mathcal{C}_{\mathsf{Pred}}(I,\ \pi(x_1,\ldots,x_n) \leftarrow G) \ = \ \mathcal{C}_{\mathsf{Struct}}(I,\ \{\ x_1,\ldots,x_n\ \},\ \nu(G)) \ \wedge\ \mathcal{C}_{\mathsf{Goal}}(I, \epsilon, G)$$

We define $\mathcal{C}_{\mathsf{Struct}}$ and $\mathcal{C}_{\mathsf{Goal}}$ below.

Structural Constraints

The Boolean formula v_{in} is the proposition that program variable v is bound at call. The Boolean formula v_{out} is the proposition that v is bound at return. The Boolean formula v_ϵ is the proposition that v is bound by this predicate. These constraints relate the relationships between the above variables and relationships of boundness at different times.

If a node is not reachable from one of the predicate's argument variables, then it cannot be bound at call.

A node is bound at return if it is bound at call or it is produced within the predicate body. A node may not be both bound at call and produced in the predicate body.

In this version of the mode system, we require that if a node is bound at call then its parent node must also be bound at call. Similarly, if a node is bound at return then its parent node must also be bound at return. If we wanted to support co-routining and, in particular, the paradigm of communication via incomplete messages (see Section 5.6), we would need to relax this constraint.

$$\begin{aligned}\mathcal{C}_{\mathsf{Struct}}(I, H, N) = & \bigwedge_{v \in \rho_I(N) \setminus \rho_I(H)} \neg v_{\mathsf{in}} \ \wedge\ \bigwedge_{v \in N} \big((v_{\mathsf{out}} \leftrightarrow v_{\mathsf{in}} \vee v_\epsilon)\ \wedge\ \neg(v_{\mathsf{in}} \wedge v_\epsilon)\big) \\ \wedge & \bigwedge_{v \in \rho_I(N)} \bigwedge_{w \in \delta_I(v)} \{\ (w_{\mathsf{in}} \rightarrow v_{\mathsf{in}})\ \wedge\ (w_{\mathsf{out}} \rightarrow v_{\mathsf{out}})\ \}\end{aligned}$$

Example 6.3.2. For `append`, the structural constraints are:

$$\begin{array}{l}
A_{\mathsf{out}} \leftrightarrow A_{\mathsf{in}} \vee A_{\epsilon}, \neg(A_{\mathsf{in}} \wedge A_{\epsilon}), AH_{\mathsf{out}} \leftrightarrow AH_{\mathsf{in}} \vee AH_{\epsilon}, \neg(AH_{\mathsf{in}} \wedge AH_{\epsilon}),\\
AT_{\mathsf{out}} \leftrightarrow AT_{\mathsf{in}} \vee AT_{\epsilon}, \neg(AT_{\mathsf{in}} \wedge AT_{\epsilon}), AE_{\mathsf{out}} \leftrightarrow AE_{\mathsf{in}} \vee AE_{\epsilon}, \neg(AE_{\mathsf{in}} \wedge AE_{\epsilon}),\\
B_{\mathsf{out}} \leftrightarrow B_{\mathsf{in}} \vee B_{\epsilon}, \neg(B_{\mathsf{in}} \wedge B_{\epsilon}), BE_{\mathsf{out}} \leftrightarrow BE_{\mathsf{in}} \vee BE_{\epsilon}, \neg(BE_{\mathsf{in}} \wedge BE_{\epsilon}),\\
C_{\mathsf{out}} \leftrightarrow C_{\mathsf{in}} \vee C_{\epsilon}, \neg(C_{\mathsf{in}} \wedge C_{\epsilon}), CH_{\mathsf{out}} \leftrightarrow CH_{\mathsf{in}} \vee CH_{\epsilon}, \neg(CH_{\mathsf{in}} \wedge CH_{\epsilon}),\\
CT_{\mathsf{out}} \leftrightarrow CT_{\mathsf{in}} \vee CT_{\epsilon}, \neg(CT_{\mathsf{in}} \wedge CT_{\epsilon}), CE_{\mathsf{out}} \leftrightarrow CE_{\mathsf{in}} \vee CE_{\epsilon}, \neg(CE_{\mathsf{in}} \wedge CE_{\epsilon}),\\
AH_{\mathsf{in}} \rightarrow A_{\mathsf{in}}, AH_{\mathsf{out}} \rightarrow A_{\mathsf{out}}, AT_{\mathsf{in}} \rightarrow A_{\mathsf{in}}, AT_{\mathsf{out}} \rightarrow A_{\mathsf{out}},\\
AE_{\mathsf{in}} \rightarrow AT_{\mathsf{in}}, AE_{\mathsf{out}} \rightarrow AT_{\mathsf{out}}, BE_{\mathsf{in}} \rightarrow B_{\mathsf{in}}, BE_{\mathsf{out}} \rightarrow B_{\mathsf{out}},\\
CH_{\mathsf{in}} \rightarrow C_{\mathsf{in}}, CH_{\mathsf{out}} \rightarrow C_{\mathsf{out}}, CT_{\mathsf{in}} \rightarrow C_{\mathsf{in}}, CT_{\mathsf{out}} \rightarrow C_{\mathsf{out}},\\
CE_{\mathsf{in}} \rightarrow CT_{\mathsf{in}}, CE_{\mathsf{out}} \rightarrow CT_{\mathsf{out}}
\end{array} \quad \triangleleft$$

Goal Constraints

There is a Boolean variable v_p for each path p which contains a program variable x such that $v \in \rho_I(x)$. This variable represents the proposition that position v is produced in the goal referred to by the path p.

The constraints we generate for each goal fall into two categories: general constraints that apply to all goal types ($\mathcal{C}_{\mathsf{Gen}}$), and constraints specific to each goal type ($\mathcal{C}_{\mathsf{Goal}}$). The complete set of constraints for a goal ($\mathcal{C}_{\mathsf{Comp}}$) is the conjunction of these two sets.

The general constraints have two components. The first, $\mathcal{C}_{\mathsf{Local}}$, says that any node reachable from a variable that is local to a goal will be bound at return iff it is produced by that goal. The second, $\mathcal{C}_{\mathsf{Ext}}$, says that a node reachable from a variable v that is external to the goal G (i.e. does not occur in G) cannot be produced by G. (Note that a node cannot be reachable from both a variable that is local to the goal and one that is not.) The conjunction in the definition of $\mathcal{C}_{\mathsf{Ext}}$ could be over all the variables in the predicate that do not occur in G. However, if a variable v does not occur in G's parent goal, then the parent's $\mathcal{C}_{\mathsf{Goal}}$ constraints won't mention v, so there is no point in creating constraint variables for v for G.

$$\begin{aligned}
\mathcal{C}_{\mathsf{Comp}}(I,p,G) &= \mathcal{C}_{\mathsf{Gen}}(I,p,G) \ \wedge \ \mathcal{C}_{\mathsf{Goal}}(I,p,G)\\
\mathcal{C}_{\mathsf{Gen}}(I,p,G) &= \mathcal{C}_{\mathsf{Local}}(I,p,G) \ \wedge \ \mathcal{C}_{\mathsf{Ext}}(I,p,G)\\
\mathcal{C}_{\mathsf{Local}}(I,p,G) &= \bigwedge_{v \in \rho_I(\nu(G) \setminus \eta(G))} (v_p \leftrightarrow v_{\mathsf{out}})\\
\mathcal{C}_{\mathsf{Ext}}(I,p,G) &= \bigwedge_{v \in \rho_I(\eta(\mathsf{parent}(p)) \setminus \nu(G))} \neg v_p
\end{aligned}$$

Compound Goals

The constraints we generate for each kind of compound goal (conjunction, disjunction and if-then-else) are shown in Figure 6.1 on the next page. In each case, the goal-type-specific constraints are conjoined with the complete set of constraints from all the subgoals.

In each conjunctive goal a position can be produced by at most one conjunct.

In each disjunctive goal a node either must be produced in each disjunct or not produced in each disjunct.

For an if-then-else goal a node is produced by the if-then-else if and only if it is produced in either the condition, the then branch or the else branch. A node may not be produced by both the condition and the then branch because they are effectively conjoined. Nodes reachable from variables that are nonlocal to the if-then-else must not be produced by the condition because the negation of the condition is effectively conjoined to the else branch. If a node reachable from a nonlocal variable is produced by the then branch then it must also be produced by the else branch, and vice versa. This is because the then and else branches are effectively disjoined.

$$
\begin{aligned}
\mathcal{C}_{\mathsf{Goal}}(I,\ p,\ \ulcorner \bigwedge \langle G_1,\ldots,G_n\rangle \urcorner) &= \bigwedge_{i=1}^{n} \mathcal{C}_{\mathsf{Comp}}(I,\ p.\mathsf{c}_i,\ G_i)\ \wedge\ \mathcal{C}_{\mathsf{Conj}}(I,\ p,\ \langle G_1,\ldots,G_n\rangle)\\
\mathcal{C}_{\mathsf{Goal}}(I,\ p,\ \ulcorner \bigvee \langle G_1,\ldots,G_n\rangle \urcorner) &= \bigwedge_{i=1}^{n} \mathcal{C}_{\mathsf{Comp}}(I,\ p.\mathsf{d}_i,\ G_i)\ \wedge\ \mathcal{C}_{\mathsf{Disj}}(I,\ p,\ \langle G_1,\ldots,G_n\rangle)\\
\mathcal{C}_{\mathsf{Goal}}(I,\ p,\ \ulcorner \mathsf{if}\ G_c\ \mathsf{then}\ G_t\ \mathsf{else}\ G_e \urcorner) &= \mathcal{C}_{\mathsf{Comp}}(I,\ p.\mathsf{c},\ G_c)\ \wedge\ \mathcal{C}_{\mathsf{Comp}}(I,\ p.\mathsf{t},\ G_t)\\
&\quad \wedge\ \mathcal{C}_{\mathsf{Comp}}(I,\ p.\mathsf{e},\ G_e)\ \wedge\ \mathcal{C}_{\mathsf{Ite}}(I,\ p,\ G_c,\ G_t,\ G_e)\\
\mathcal{C}_{\mathsf{Conj}}(I,\ p,\ \langle G_1,\ldots,G_n\rangle) &= \bigwedge_{v\in\rho_I(\nu(\{\,G_1,\ldots G_n\,\}))} \begin{pmatrix} (v_p \leftrightarrow v_{p.\mathsf{c}_j} \vee \cdots \vee v_{p.\mathsf{c}_n}) \\ \wedge \quad \bigwedge_{i=1}^{n}\bigwedge_{j=1}^{i-1} \neg\left(v_{p.\mathsf{c}_i} \wedge v_{p.\mathsf{c}_j}\right)\end{pmatrix}\\
\mathcal{C}_{\mathsf{Disj}}(I,\ p,\ \langle G_1,\ldots,G_n\rangle) &= \bigwedge_{v\in\rho_I(\eta(\{\,G_1,\ldots G_n\,\}))} \bigwedge_{i=1}^{n} (v_p \leftrightarrow v_{p.\mathsf{d}_1})\\
\mathcal{C}_{\mathsf{Ite}}(I,\ p,\ G_c,\ G_t,\ G_e) &= \bigwedge_{v\in\rho_I(\nu(G))} \big((v_p \leftrightarrow v_{p.\mathsf{c}} \vee v_{p.\mathsf{t}} \vee v_{p.\mathsf{e}})\ \wedge\ \neg(v_{p.\mathsf{c}} \wedge v_{p.\mathsf{t}})\big)\\
&\quad \wedge \bigwedge_{v\in\rho_I(\eta(G))} \big(\neg v_{p.\mathsf{c}}\ \wedge\ (v_{p.\mathsf{t}} \leftrightarrow v_{p.\mathsf{e}})\big)
\end{aligned}
$$

Figure 6.1: Constraints for conjunctions, disjunctions and if-then-elses

Atomic Goals

We consider three kinds of atomic goals (we will consider higher-order calls and unifications with higher-order terms later in Section 6.3.4):

1. Unifications of the form $x = y$.
2. Unifications of the form $x = f(y_1,\ldots,y_n)$ where $x \rightarrow f(y_1,\ldots,y_n) \in I$.
3. Calls of the form $\pi(y_1,\ldots,y_n)$.

A unification of the form $x = y$ may produce at most one of each pair of corresponding nodes. Mercury does not currently allow aliases to exist between unbound nodes so each node reachable from a variable involved in a unification must be produced somewhere. (This is true even with the free(aliased) inst we discussed in Section 5.4, although HAL has relaxed this restriction.) For a unification $x = y$ at goal path p, the constraint is

$$\mathcal{C}_{\mathsf{Goal}}(I,\ p,\ \ulcorner x = y \urcorner) \ = \bigwedge_{(v,w)\in\chi_I(x,y)} \big(v_{\mathsf{out}} \ \wedge\ w_{\mathsf{out}} \ \wedge\ \neg(v_p \ \wedge\ w_p)\big)$$

During the goal scheduling phase, we further require that a node must be produced before it is aliased to another node. These two restrictions together disallow uses of partially instantiated data structures which would require some sort of reference chains (rather than just a single reference to the location where the value must be place) in the implementation. As we discussed in Chapter 5, it would be possible to use such a scheme. We could then remove this restriction from the mode system by simply removing the v_{out} and w_{out} constraints, above.

Example 6.3.3. For the unification '`B = C`' in `append` at goal path $\mathsf{d}_1.\mathsf{c}_2$ the constraints generated are:

$$\begin{array}{l} B_{\mathsf{out}} \wedge C_{\mathsf{out}} \wedge \neg(B_{\mathsf{d}_1.\mathsf{c}_2} \wedge C_{\mathsf{d}_1.\mathsf{c}_2}) \ \wedge \\ BE_{\mathsf{out}} \wedge CH_{\mathsf{out}} \wedge \neg(BE_{\mathsf{d}_1.\mathsf{c}_2} \wedge CH_{\mathsf{d}_1.\mathsf{c}_2}) \ \wedge \\ B_{\mathsf{out}} \wedge CT_{\mathsf{out}} \wedge \neg(B_{\mathsf{d}_1.\mathsf{c}_2} \wedge CT_{\mathsf{d}_1.\mathsf{c}_2}) \ \wedge \\ BE_{\mathsf{out}} \wedge CE_{\mathsf{out}} \wedge \neg(BE_{\mathsf{d}_1.\mathsf{c}_2} \wedge CE_{\mathsf{d}_1.\mathsf{c}_2}) \end{array}$$

◁

A unification of the form $x = f(y_1, \ldots, y_n)$ at path p does not produce any of the arguments $y_1, \ldots, y_n$ because the transformation into hyperhomogeneous normal form ensures that var-functor unifications are never responsible for producing the arguments of the functor. The node x must be produced somewhere (either at p or somewhere else). The constraint is

$$\mathcal{C}_{\mathsf{Goal}}(I,\ p,\ \ulcorner x = f(y_1, \ldots, y_n) \urcorner) \ = \ x_{\mathsf{out}} \ \wedge \bigwedge_{v\in\rho_I(\{\, y_1,\ldots,y_n \,\})} \neg v_p$$

Note that x will never be a member of $\rho_I(\{\, y_1, \ldots, y_n \,\})$ because of the transformation we have done at the start of Section 6.3.1.

Example 6.3.4. For the unification '`A = [AH|AT]`' in `append` at goal path $\mathsf{d}_2.\mathsf{c}_1$ the constraints generated are:

$$A_{\mathsf{out}} \wedge \neg AH_{\mathsf{d}_2.\mathsf{c}_1} \wedge \neg AT_{\mathsf{d}_2.\mathsf{c}_1} \wedge \neg AE_{\mathsf{d}_2.\mathsf{c}_1}$$

In the `(in, in, out)` mode of `append`, `A`, `AH`, `AT` and `AE` are all input to the procedure, so this unification doesn't bind any of these nodes. In the `(out, out, in)` mode, `AT` and `AE` are bound by the recursive call and `AH` is bound by the unification with `CH`, leaving only the `A` node to bound by this unification. ◁

A call $\pi(y_1, \ldots, y_n)$ will constrain the nodes reachable from the arguments of the call. For predicates in the current SCC, we only allow recursive calls that are in the same mode as the caller. The constraint is

$$\mathcal{C}_{\mathsf{Goal}}(I,\ p,\ \ulcorner \pi(y_1, \ldots, y_n) \urcorner) \ = \bigwedge_{\langle v,w\rangle\in\chi_I(\alpha(\pi/n),\langle y_1,\ldots,y_n\rangle)} \big((v_\epsilon \leftrightarrow w_p) \ \wedge\ (v_{\mathsf{in}} \rightarrow w_{\mathsf{out}})\big)$$

The first part ensures that the call produces the position if the position is produced by the predicate in the SCC. The second part ensures that call variable w is produced somewhere if it is required to be bound at call to the call (v_{in}). Since if v_{in} is true v_ϵ cannot be true, we can't mistakenly use this call site to produce w.

Example 6.3.5. For the recursive call `append(AT,B,CT)` at goal path $\mathsf{d}_2.\mathsf{c}_4$ in `append` the constraints generated on the first argument are:

$$\begin{array}{l} (A_\epsilon \leftrightarrow AT_{\mathsf{d}_2.\mathsf{c}_4}) \wedge (A_{\mathsf{in}} \rightarrow AT_{\mathsf{out}}) \wedge \\ (AH_\epsilon \leftrightarrow AE_{\mathsf{d}_2.\mathsf{c}_4}) \wedge (AH_{\mathsf{in}} \rightarrow AE_{\mathsf{out}}) \wedge \\ (AT_\epsilon \leftrightarrow AT_{\mathsf{d}_2.\mathsf{c}_4}) \wedge (AT_{\mathsf{in}} \rightarrow AT_{\mathsf{out}}) \wedge \\ (AE_\epsilon \leftrightarrow AE_{\mathsf{d}_2.\mathsf{c}_4}) \wedge (AE_{\mathsf{in}} \rightarrow AE_{\mathsf{out}}) \end{array} \qquad \triangleleft$$

For calls to predicates in lower SCCs the constraints are similar, but we must obtain the $\mathcal{C}_{\mathsf{Inf}}(I, \pi/n)$ constraint for the called predicate π/n and existentially quantify the head variables so that it is possible to call the predicate in different modes from different places within the current SCC:

$$\begin{array}{rcl} \mathcal{C}_{\mathsf{Goal}}(I,\ p,\ \pi(y_1,\ldots,y_n)) & = & \exists \rho_I(\alpha(\pi/n)).\ \mathcal{C}_{\mathsf{Inf}}(I,\ \pi/n) \\ & & \wedge \displaystyle\bigwedge_{\langle v,w\rangle \in \chi_I(\alpha(\pi/n),\langle y_1,\ldots,y_n\rangle)} \big((v_\epsilon \leftrightarrow w_p)\ \wedge\ (v_{\mathsf{in}} \rightarrow w_{\mathsf{out}})\big) \end{array}$$

6.3.3 Mode Declaration Constraints

For any predicate which has modes declared, the mode analysis system should check these declarations against the inferred mode information. This involves generating a set of constraints for the mode declarations and ensuring that they are consistent with the constraints generated from the predicate body.

The declaration constraint $\mathcal{C}_{\mathsf{Decls}}(I, \mathcal{D})$ for a predicate with a set of mode declarations $\mathcal{D}$ is the disjunction of the constraints $\mathcal{C}_{\mathsf{Decl}}(I, d)$ for each mode declaration d:

$$\mathcal{C}_{\mathsf{Decls}}(I, \mathcal{D}) = \bigvee_{d \in \mathcal{D}} \mathcal{C}_{\mathsf{Decl}}(I, d)$$

The constraint $\mathcal{C}_{\mathsf{Decl}}(I, d)$ for a mode declaration $d = \langle x_1 :: \mu_1, \ldots, x_n :: \mu_n\rangle$ is the conjunction of the constraints $\mathcal{C}_{\mathsf{Arg}}(I, \mu_i, x_i)$ for each argument mode μ_i with corresponding head variable x_i:

$$\mathcal{C}_{\mathsf{Decl}}(I, \langle x_1 :: \mu_1, \ldots, x_n :: \mu_n\rangle) \;=\; \bigwedge_{i=1}^{n} \mathcal{C}_{\mathsf{Arg}}(I, \mu_i, x_i) \;\wedge\; \mathcal{C}_{\mathsf{Struct}}(I, \{\, x_1, \ldots, x_n \,\}, \{\, x_1, \ldots, x_n \,\})$$

The structural constraints are used to determine the x_ϵ variables from x_{in} and x_{out}.

The constraint $\mathcal{C}_{\mathsf{Arg}}(I, \mu, x)$ for an argument mode $\mu = \iota \rightarrowtail \iota'$ of head variable x is the conjunction of the constraint $\mathcal{C}_{\mathsf{Init}}(I, \iota, x)$ for the initial instantiation state ι, and the constraint

$\mathcal{C}_{\mathsf{Fin}}(I, \iota', x)$ for the final instantiation state ι':

$$\mathcal{C}_{\mathsf{Arg}}(I, \iota \rightarrowtail \iota', x) = \mathcal{C}_{\mathsf{Init}}(I, \iota, x) \ \wedge \ \mathcal{C}_{\mathsf{Fin}}(I, \iota', x)$$

The constraint $\mathcal{C}_{\mathsf{Init}}(I, \iota, x)$ for an initial instantiation state ι of a head variable x is given below:

$$\begin{array}{rcl}
\mathcal{C}_{\mathsf{Init}}(I, \mathsf{free}, x) & = & \neg x_{\mathsf{in}} \\
\mathcal{C}_{\mathsf{Init}}(I, \mathsf{ground}, x) & = & \displaystyle\bigwedge_{w \in \rho_I(x)} w_{\mathsf{in}} \\
\mathcal{C}_{\mathsf{Init}}(I, \iota, x) & = & x_{\mathsf{in}} \quad \wedge \\
& & \displaystyle\bigwedge_{\substack{\iota \rightarrow f(\iota_1, \ldots, \iota_n) \in \mathsf{rules}(\iota) \\ x \rightarrow f(y_1, \ldots, y_n) \in I}} \big(\mathcal{C}_{\mathsf{Init}}(I, \iota_1, y_1) \wedge \cdots \wedge \mathcal{C}_{\mathsf{Init}}(I, \iota_n, y_n)\big)
\end{array}$$

The constraint $\mathcal{C}_{\mathsf{Fin}}(I, \iota, x)$ for a final instantiation state ι of a head variable x is isomorphic to $\mathcal{C}_{\mathsf{Init}}(I, \iota, x)$, differing only in that it constrains variables of the form v_{out} instead of v_{in}.

$$\begin{array}{rcl}
\mathcal{C}_{\mathsf{Fin}}(I, \mathsf{free}, x) & = & \neg x_{\mathsf{out}} \\
\mathcal{C}_{\mathsf{Fin}}(I, \mathsf{ground}, x) & = & \displaystyle\bigwedge_{w \in \rho_I(x)} w_{\mathsf{out}} \\
\mathcal{C}_{\mathsf{Fin}}(I, \iota, x) & = & x_{\mathsf{out}} \quad \wedge \\
& & \displaystyle\bigwedge_{\substack{\iota \rightarrow f(\iota_1, \ldots, \iota_n) \in \mathsf{rules}(\iota) \\ x \rightarrow f(y_1, \ldots, y_n) \in I}} \big(\mathcal{C}_{\mathsf{Fin}}(I, \iota_1, y_1) \wedge \cdots \wedge \mathcal{C}_{\mathsf{Fin}}(I, \iota_n, y_n)\big)
\end{array}$$

Mode checking is simply determining if the declared modes are at least as strong as the inferred modes. For each declared mode d of predicate π/n we check whether the implication

$$\mathcal{C}_{\mathsf{Decl}}(I, d) \rightarrow \mathcal{C}_{\mathsf{Inf}}(I, \pi/n)$$

holds or not. If it doesn't, the declared mode is incorrect.

If we are given declared modes $\mathcal{D}$ for a predicate π/n, they can be used to short-circuit the calculation of SCCs, since we can use $\mathcal{C}_{\mathsf{Decls}}(I, \mathcal{D})$ in mode inference for any predicate π'/m that calls π/n. This allows programmers to get around the restriction that all calls to a predicate in the same SCC as the caller must be in the same mode by simply giving a mode declaration for the called predicate.

Example 6.3.6. Given the mode definition:

```
:- mode lsg == (listskel(free) >> ground).
```

the mode declaration $d_1 = \langle \mathtt{A} :: \mathtt{lsg}, \mathtt{B} :: \mathtt{lsg}, \mathtt{C} :: \mathtt{in} \rangle$ for '`append/3`' gives $\mathcal{C}_{\mathsf{Decl}}(I, d_1)$ (ignoring v_{out} variables):

$$\begin{array}{l}
A_{\mathsf{in}} \wedge \neg AH_{\mathsf{in}} \wedge AT_{\mathsf{in}} \wedge \neg AE_{\mathsf{in}} \wedge B_{\mathsf{in}} \wedge \neg BE_{\mathsf{in}} \ \wedge \\
C_{\mathsf{in}} \wedge CH_{\mathsf{in}} \wedge CT_{\mathsf{in}} \wedge CE_{\mathsf{in}} \ \wedge \\
\neg A_{\epsilon} \wedge AH_{\epsilon} \wedge \neg AT_{\epsilon} \wedge AE_{\epsilon} \wedge \neg B_{\epsilon} \wedge BE_{\epsilon} \ \wedge \\
\neg C_{\epsilon} \wedge \neg CH_{\epsilon} \wedge \neg CT_{\epsilon} \wedge \neg CE_{\epsilon}
\end{array}$$

We can show that $\mathcal{C}_{\mathsf{Decl}}(I, d_1) \rightarrow \mathcal{C}_{\mathsf{Inf}}(I, \texttt{append/3})$. $\triangleleft$

6.3.4 Constraints for Higher-Order Code

Recall that construction of a higher-order term in our Mercury abstract syntax is through a unification of the form

$$x = \lambda(y_1 :: \mu_1, \ldots, y_n :: \mu_n) \leftarrow G$$

where $y_1, \ldots, y_n$ are the lambda-quantified arguments, $\mu_1, \ldots, \mu_n$ are the modes of the arguments, and G is the lambda body goal. Mercury requires the argument modes to be declared when constructing higher-order terms.

To analyse such a construction, we must do three things:

1. analyse the unification itself;

2. analyse the lambda body goal; and

3. record the higher-order argument modes so that we can mode-check any calls to the higher-order term.

For a unification at goal path p, the node x is produced by the unification. All nodes which are reachable from variables non-local to the lambda body goal must not be produced by the lambda body goal. Thus the $\mathcal{C}_{\mathsf{Goal}}$ constraint for the unification is

$$\mathcal{C}_{\mathsf{Goal}}(I,\ p,\ \ulcorner x = \lambda(y_1 :: \mu_1, \ldots, y_n :: \mu_n) \leftarrow G \urcorner) \quad = \quad x_p \ \wedge \bigwedge_{v \in \rho_I(\eta(p) \setminus \{\ x\ \})} \neg v_p$$

For analysing the lambda body goal, we can treat it like any other predicate and analyse it as described above. We make use of the given modes of the higher-order term as a mode declaration (see Section 6.3.3). We must also ensure that any non-local variables are not produced by the goal. The constraint for the body goal is thus

$$\begin{aligned}\mathcal{C}_{\lambda}(I,\ \langle y_1 :: \mu_1, \ldots, y_n :: \mu_n \rangle\ ,\ G) \quad = \quad & \mathcal{C}_{\mathsf{Decl}}(I,\ \langle y_1 :: \mu_1, \ldots, y_n :: \mu_n \rangle) \\ & \wedge\ \mathcal{C}_{\mathsf{Struct}}(I,\ \{\ y_1, \ldots, y_n\ \}\ ,\ \nu(G))\ \wedge\ \mathcal{C}_{\mathsf{Goal}}(I,\ \epsilon,\ G)\ \wedge \bigwedge_{v \in \rho_I(\eta(G))} \neg v_\epsilon\end{aligned}$$

For analysing higher-order calls we use the mode information associated with the higher-order term to obtain a "mode declaration" for the "predicate" we are calling. If the higher-order term was constructed by a higher-order unification in the current predicate then it will have exactly one mode declaration associated with it. If it was returned from another predicate then it may have multiple mode declarations—one for each mode of the predicate it was returned from (a predicate with a higher-order argument *must* have mode declarations).

Assume that we have collected information about higher-order terms such that for a variable x bound to a higher-order term $\lambda(y_1 :: \mu_1, \ldots, y_n :: \mu_n) \leftarrow G$ we have $\alpha(x) = \langle y_1, \ldots, y_n \rangle$ and $\mathcal{D}(x)$ is the set of mode declarations $\langle y_1 :: \mu_1, \ldots, y_n :: \mu_n \rangle$ associated with the higher-order term.

The goal constraint for the higher-order call $x(z_1, \ldots, z_n)$ is given below. We require x to be produced somewhere. The other constraints for the higher-order call resemble the constraints for a call to a predicate for which we have mode declarations.

$$\begin{aligned}\mathcal{C}_{\mathsf{Goal}}(I,\ p,\ x(z_1,\ldots,z_n)) \quad = \quad & x_{\mathsf{out}} \ \wedge\ \exists \rho_I(\alpha(x)).\ \mathcal{C}_{\mathsf{Decls}}(I,\ \mathcal{D}(x)) \\ & \wedge \bigwedge_{\langle v,w\rangle \in \chi_I(\alpha(x),\langle z_1,\ldots,z_n\rangle)} \big((v_\epsilon \leftrightarrow w_p)\ \wedge\ (v_{\mathsf{in}} \rightarrow w_{\mathsf{out}})\big)\end{aligned}$$

6.4 Selecting Procedures and Execution Order

Once we have generated the constraints for an SCC, we can solve those constraints. If the constraints have no solution, then some position has consumers but no producer, so we report a mode error. If the constraints have some solutions, then each solution gives a mode for each predicate in the SCC; the set of solutions thus defines the set of modes of each predicate. We then need to find a feasible execution order for each mode of each predicate in the SCC. The algorithm for finding feasible execution orders takes a solution as its input. If a given mode of a predicate corresponds to several solutions, it is sufficient for *one* of them to have a feasible ordering.

The main problem in finding a feasible schedule is that the mode analyser and the code generator [39] have distinct views of what it means to produce a (position in a) variable. In the grammar we generate for `append`, for example, the nonterminal AH represents both the value of the variable `AH` and the value of the first element of the variable `A`. In the forward mode of `append`, AH_{in} is true, so the mode analyser considers `AH` to have been produced by the caller even *before* execution enters `append`. However, as far as the code generator is concerned, `AH` doesn't really exist until the unification '`A = [AH|AT]`'. It is to cater for these divergent views that we separate the notion of a variable being *produced* from the notion of a variable being *visible*.

Definition 6.4.1 Given an expanded grammar I, an assignment M of boolean values to the constraint variables of a predicate π/n that makes the constraint $\mathcal{C}_{\mathsf{Inf}}(I, \pi/n)$ true is a *model* of $\mathcal{C}_{\mathsf{Inf}}(I, \pi/n)$. We write $M \models \mathcal{C}_{\mathsf{Inf}}(I, \pi/n)$. ◁

Definition 6.4.2 For a given model $M \models \mathcal{C}_{\mathsf{Inf}}(I, \pi/n)$ we define the set of nodes *produced* at a goal path p by

$$\mathsf{produced}_M(I, p) = \{\ v \ \mid\ M(v_p) = 1\ \}$$ ◁

Definition 6.4.3 For a given model $M \models \mathcal{C}_{\mathsf{Inf}}(I, \pi/n)$ we define the set of nodes *consumed* by the goal G at a goal path p by the formula shown in Figure 6.2 on the facing page. ◁

For a unification of the form $x = y$, we say that a node on one side of the equation is consumed iff a corresponding node on the other side is produced. (Due to the symmetric nature of this relationship, if v_1 and v_2 both correspond to w, then v_1 is consumed iff v_2 is consumed, and v_1 is produced iff v_2 is produced.) It is also possible for a pair of corresponding nodes to be neither produced nor consumed by the unification. This can mean one of two things. If the subterms of x and y at that node are already bound, then the unification will test the equality of those subterms; if they are still free, then it will create an alias between them. Note that if the unification produces either of the top level nodes x or y, then we call it an *assignment* unification.

$$
\begin{array}{rcl}
\mathsf{consumed}_M(I, p, \ulcorner x = y \urcorner) & = & \left\{ v \mid \langle v, w \rangle \in \chi_I(x, y) \wedge w \in \mathsf{produced}_M(I, p) \right\} \\
& \cup & \left\{ w \mid \langle v, w \rangle \in \chi_I(x, y) \wedge v \in \mathsf{produced}_M(I, p) \right\} \\
\\
\mathsf{consumed}_M(I, p, \ulcorner x = f(y_1, \ldots, y_n) \urcorner) & = & \{ x \} \setminus \mathsf{produced}_M(I, p) \\
\\
\mathsf{consumed}_M(I, p, \ulcorner x = \lambda(\overline{y :: \mu}) \leftarrow G \urcorner) & = & \mathsf{consumed}_M(I, \epsilon, G) \setminus \rho_I(\overline{y}) \\
\\
\mathsf{consumed}_M(I, p, \ulcorner \pi'(y_1, \ldots, y_n) \urcorner) & = & \left\{ v \mid \langle v, w \rangle \in \chi \ \wedge \ M'(w_{\mathsf{in}}) = 1 \right\} \\
& & \quad \text{where } \ \chi = \chi_I(\langle y_1, \ldots, y_n \rangle, \alpha(\pi'/n)) \\
& & \qquad \text{and } M' \models \mathcal{C}_{\mathsf{Inf}}(I, \pi'/n) \\
& & \qquad \text{such that } \forall \langle v, w \rangle \in \chi \ . \ M(v_p) \leftrightarrow M'(w_\epsilon) \\
\\
\mathsf{consumed}_M(I, p, \ulcorner x(y_1, \ldots, y_n) \urcorner) & = & \{ x \} \cup \left\{ v \mid \langle v, w \rangle \in \chi \ \wedge \ M'(w_{\mathsf{in}}) = 1 \right\} \\
& & \quad \text{where } \ \chi = \chi_I(\langle y_1, \ldots, y_n \rangle, \alpha(x)) \\
& & \qquad \text{and } M' \models \mathcal{C}_{\mathsf{Decls}}(I, \mathcal{D}(x)) \\
& & \qquad \text{such that } \forall \langle v, w \rangle \in \chi \ . \ M(v_p) \leftrightarrow M'(w_\epsilon) \\
\\
\mathsf{consumed}_M(I, p, \ulcorner \bigwedge \langle G_1, \ldots, G_n \rangle \urcorner) & = & \displaystyle\bigcup_{i=1}^{n} \mathsf{consumed}_M(I, p.\mathsf{c}_i, G_i) \setminus \mathsf{produced}_M(I, p) \\
\\
\mathsf{consumed}_M(I, p, \ulcorner \bigvee \langle G_1, \ldots, G_n \rangle \urcorner) & = & \displaystyle\bigcup_{i=1}^{n} \mathsf{consumed}_M(I, p.\mathsf{d}_i, G_i) \setminus \mathsf{produced}_M(I, p) \\
\\
\mathsf{consumed}_M(I, p, \ulcorner \mathsf{if}\ G_c\ \mathsf{then}\ G_t\ \mathsf{else}\ G_e \urcorner) & = & \big(\mathsf{consumed}_M(I, p.\mathsf{c}, G_c) \cup \mathsf{consumed}_M(I, p.\mathsf{t}, G_t) \\
& & \quad \cup\ \mathsf{consumed}_M(I, p.\mathsf{e}, G_e)\big) \setminus \mathsf{produced}_M(I, p)
\end{array}
$$

Figure 6.2: Calculating which nodes are "consumed" at which positions

For a unification of the form $x = f(y_1, \ldots, y_n)$ we say that the node x is consumed iff it is not produced, and that no other nodes are ever consumed. The reason for the latter half of that rule is that our grammar will use the same nonterminal for e.g. y_1 as for the first subterm of x. Since this unification merely creates aliases between the y_i and the corresponding subterms of x, the nonterminals of the y_i cannot be produced by this unification; if they are produced at all, they have to be produced elsewhere. Note that if the unification produces x, then we call it a *construction* unification; if it consumes x, then we call it a *deconstruction* unification.

For a higher-order unification, any node consumed by the body goal, except nodes reachable from the lambda-quantified variables, is consumed by the unification goal.

For a call to a predicate π', we know which nodes of the actual parameters of the call the model M of the predicate π we are analysing says should be produced by the call. We need to find a model M' of the constraints of π' that causes the corresponding nodes in the actual parameters of π' to be output. Since the first stage of the analysis succeeded we know such a model M' exists. The consumed nodes of the call are then the nodes of the actual parameters that correspond to the nodes of the formal parameters of π' that M' requires to be input.

For a higher-order call $x(y_1, \ldots, y_n)$, the set of consumed nodes is similar to the set of consumed nodes for a predicate call, except that we use the mode information from the higher-order term instead of from the predicate. The node x is also consumed because we require x to be bound to a higher-order term (i.e. produced) before the higher-order call.

For compound goals, the consumed nodes are the union of the consumed nodes of the subgoals, minus the nodes that are produced within the compound goal.

Example 6.4.1. In the `(in, in, out)` mode of `append`, the produced and consumed sets of the conjuncts are:

Path	Goal	produced	consumed
$\mathsf{d}_1.\mathsf{c}_1$	`A = []`	$\emptyset$	$\{A\}$
$\mathsf{d}_1.\mathsf{c}_2$	`B = C`	$\{C, CH, CT, CE\}$	$\{B, BE\}$
$\mathsf{d}_2.\mathsf{c}_1$	`A = [AH | AT]`	$\emptyset$	$\{A\}$
$\mathsf{d}_2.\mathsf{c}_2$	`C = [CH | CT]`	$\{C\}$	$\emptyset$
$\mathsf{d}_2.\mathsf{c}_3$	`AH = CH`	$\{CH\}$	$\{AH\}$
$\mathsf{d}_2.\mathsf{c}_4$	`append(AT, B, CT)`	$\{CT, CE\}$	$\{AT, AE, B, BE\}$

◁

Neither disjunct produces any position that it also consumes. Therefore, if our ordering algorithm only required a node to be produced before it is consumed, it would find any order acceptable. On the other hand, the code generator is more fussy; for example, before it can emit code for the recursive call, it needs to know where variables `AH` and `AT` are stored, even if they have not been bound yet. This is why we need the concept of *visibility*.

Definition 6.4.4 A variable is *visible* at a goal path p if the variable is a head variable or has appeared in the predicate body somewhere to the left of p. The functions make_visible and need_visible, defined in Figure 6.3 on the next page, respectively determine whether a goal makes a variable visible or requires it to be visible. Note that visibility is applied to variables rather than to nodes. ◁

Example 6.4.2. Given the `(in, in, out)` mode of `append` the make_visible and need_visible sets of the conjuncts are:

Path	Goal	make_visible	need_visible
$\mathsf{d}_1.\mathsf{c}_1$	`A = []`	$\emptyset$	$\{A\}$
$\mathsf{d}_1.\mathsf{c}_2$	`B = C`	$\{C\}$	$\{B\}$
$\mathsf{d}_2.\mathsf{c}_1$	`A = [AH | AT]`	$\{AH, AT\}$	$\{A\}$
$\mathsf{d}_2.\mathsf{c}_2$	`C = [CH | CT]`	$\{C, CH, CT\}$	$\emptyset$
$\mathsf{d}_2.\mathsf{c}_3$	`AH = CH`	$\{CH\}$	$\{AH\}$
$\mathsf{d}_2.\mathsf{c}_4$	`append(AT, B, CT)`	$\{CT\}$	$\{AT, B\}$

◁

Our algorithm needs to find, in each conjunction in the body, an ordering of the conjuncts such that the producer of each node comes before any of its consumers, and each variable is made visible before any point where it needs to be visible. We do this by traversing the predicate body top down. At each conjunction, we construct a directed graph whose nodes are the conjuncts. The initial graph has an edge from c_i to c_j iff c_i produces a node that c_j consumes. If this graph is cyclic, then mode ordering fails. If it isn't, we try to add more edges while keeping the graph acyclic.

We sort the variables that need to be visible anywhere in the conjunction that are also made visible in the conjunction into two classes: those where it is clear which conjunct should make them

$$\mathsf{make\_visible}_M(I,p,G) = \begin{cases} \nu(G) \setminus \mathsf{consumed}_M(I,p,G) & \text{if } G \text{ is atomic;} \\ \bigcup_{i=1}^{n} \mathsf{make\_visible}_M(I,p.\mathsf{c}_i,G_i) & \text{if } G = \ulcorner \bigwedge \langle G_1,\ldots,G_n\rangle \urcorner; \\ \bigcap_{i=1}^{n} \mathsf{make\_visible}_M(I,p.\mathsf{d}_i,G_i) & \text{if } G = \ulcorner \bigvee \langle G_1,\ldots,G_n\rangle \urcorner; \\ \begin{array}{l} (\mathsf{make\_visible}_M(I,p.\mathsf{c},G_c) \\ \cup\ \mathsf{make\_visible}_M(I,p.\mathsf{t},G_t)) \\ \cap\ \mathsf{make\_visible}_M(I,p.\mathsf{e},G_e) \end{array} & \text{if } G = \ulcorner \mathsf{if}\ G_c\ \mathsf{then}\ G_t\ \mathsf{else}\ G_e \urcorner. \end{cases}$$

$$\mathsf{need\_visible}_M(I,p,G) = \begin{cases} \nu(G) \cap \mathsf{consumed}_M(I,p,G) & \text{if } G \text{ is atomic;} \\ \bigcup_{i=1}^{n} \begin{array}{l} \mathsf{need\_visible}_M(I,p.\mathsf{c}_i,G_i) \\ \setminus \mathsf{make\_visible}_M(I,p,G) \end{array} & \text{if } G = \ulcorner \bigwedge \langle G_1,\ldots,G_n\rangle \urcorner; \\ \bigcup_{i=1}^{n} \mathsf{need\_visible}_M(I,p.\mathsf{d}_i,G_i) & \text{if } G = \ulcorner \bigvee \langle G_1,\ldots,G_n\rangle \urcorner; \\ \bigcup_{pc\in\{\,\mathsf{c,t,e}\,\}} \begin{array}{l} \mathsf{need\_visible}_M(I,p.pc,G_{pc}) \\ \setminus \mathsf{make\_visible}_M(I,p,G) \end{array} & \text{if } G = \ulcorner \mathsf{if}\ G_c\ \mathsf{then}\ G_t\ \mathsf{else}\ G_e \urcorner. \end{cases}$$

Figure 6.3: Calculating make_visible and need_visible

visible and those where it isn't. A variable falls into the first class iff it is made visible in only one conjunct, or if a conjunct that makes it visible is also the producer of its top level node. (In the forward mode of `append`, all variables fall into the first class; the only variable that is made visible in more than one conjunct, `CH`, does not need to be visible in any conjunct in that conjunction.) For each of these variables, we add an edge from the conjunct that makes the variable visible to all the conjuncts c_j that need it to be visible. If the graph is still acyclic, we then start searching the space of mappings that map each variable in the second class to a conjunct that makes that variable visible, looking for a map that results in an acyclic graph when we add links from the selected make_visible conjunct of each variable to all the corresponding need_visible conjuncts.

It can also happen that some of the conjuncts need a variable visible that none of the goals in the conjunction make visible. If this variable is made visible by a goal to the left of the whole conjunction, by another conjunction that encloses this one, then everything is fine. If it isn't, then the ordering of the enclosing conjunction would have already failed, because if no conjunct makes the variable visible, then the conjunction as a whole needs it visible.

If no mapping yields an acyclic graph, the procedure has a mode error. If some mappings do, then the algorithm in general has two choices to make: it can pick any acyclic graph, and it can pick any order for the conjuncts that is consistent with that graph.

All the nodes the forward mode of `append` consumes are input to the predicate, so there are no ordering constraints between producers and consumers. The first disjunct has no visibility constraints either, so it can be given any order. In the second disjunct, visibility requirements dictate that '`A = [AH|AT]`' must occur before both '`AH = CH`' and $\texttt{append}(AT, B, CT)$, to make AH and AT visible where required. This leaves the compiler with this graph:

$$\begin{tikzcd}
\texttt{A=[AH|AT]} \arrow[rr, bend left] \arrow[dr] & \texttt{C=[CH|CT]} & \texttt{AH=CH} \\
 & \texttt{append}(AT, B, CT) &
\end{tikzcd}$$

This graph does not completely fix the order of the conjuncts. A parallel implementation may choose to execute several conjuncts in parallel, and a coroutining implementation may choose to interleave their execution, although in this case that would not be worthwhile. More likely, an implementation may choose to schedule the recursive call last to ensure tail recursion. (In Chapter 5 we needed a program transformation separate from mode analysis to introduce tail recursion in predicates like this.)

Example 6.4.3. Consider the predicate `p/3` defined by

```
:- pred p(int,int,int).
:- mode p(out,out,out) is det.
p(X,Y,Z) :- X = Y, Y = Z, Z = X.
```

The mode constraints are satisfiable, and one model is $X_\epsilon \wedge X_{\mathtt{c}_1} \wedge Y_\epsilon \wedge Y_{\mathtt{c}_2} \wedge Z_\epsilon \wedge Z_{\mathtt{c}_3}$, where each variable is produced by the equation on which it occurs on the left. The initial graph of the conjunction is

$$\begin{tikzcd}
X = Y \arrow[rr, bend left] & Y = Z \arrow[l] & Z = X \arrow[l]
\end{tikzcd}$$

Since this graph and all other possible graphs are cyclic, the procedure has a mode error. ◁

6.5 Implementation Issues

Our analysis is implemented within the Melbourne Mercury compiler. We represent the Boolean constraints as *reduced ordered binary decision diagrams* (ROBDDs) [19, 20] using a highly-optimised implementation by Schachte [118, 119] who has shown that ROBDDs provide a efficient representation for other logic program analyses based on Boolean domains.

ROBDDs are directed acyclic graphs with common-subexpressions merged. They provide an efficient, canonical representation for Boolean functions.

In the worst case, the size of an ROBDD can be exponential in the number of variables. In practice, however, with care this worst-case behaviour can usually be avoided. We use a number of techniques to keep the ROBDDs as small and efficient as possible.

6.5.1 Reducing the Number of Variables

One of the most effective ways of reducing the size of an ROBDD is to avoid putting unrelated constraints into the same ROBDD. Rather than analysing the whole program using a single ROBDD, we use a separate ROBDD for each SCC of predicates. We analyse the SCCs bottom-up so that mode information from lower SCCs is available when analysing higher SCCs. Predicates that have mode declarations can be analysed separately from their SCCs. Such predicates are analysed after all the SCCs have been analysed. When inferring modes for the SCC predicates we can make use of given mode declarations. By the time we get to analysing predicates which already have mode declarations, we have all the mode information we need from other predicates which they might call.

Another way that we can reduce the number of variables in an ROBDD, and thus its size, is to note that in our analysis, certain pairs of variables are always constrained to be equivalent. For a disjunction with path p containing a disjunct with path $p.\mathsf{d}_n$

$$v_p \leftrightarrow v_{p.\mathsf{d}_n}$$

always applies. We can also assume

$$v_p \leftrightarrow v_{p.\mathsf{e}}$$

for an if-then-else. This constraint obviously applies for non-local variables. For local variables, it is slightly stronger than the constraints previously given, since a position in a local variable may be bound by the condition or then branch rather than the else branch. To allow us to add this constraint we note that, due to Mercury's quantification rules, if a variable is local to the if-then-else, it must either be local to the else branch or not occur in the else branch. If a variable is local to the else branch, $v_p \leftrightarrow v_{p.\mathsf{e}}$ obviously holds. If it does not occur in the else branch then we have an extra constraint added which is never used since $v_{p.\mathsf{e}}$ will not occur in any other constraints.

Since the above equivalence constraints occur for all positions in all disjunctions and if-then-elses, we can reduce the number of constraint variables we use by using the same variable for each side of the equivalence. That is, whenever we create a constraint with $v_{p.\mathsf{d}_n}$ or $v_{p.\mathsf{e}}$, we can use v_p instead.

6.5.2 Restriction and Variable Ordering Trade-Offs

The size of an ROBDD is very sensitive to the ordering of variables chosen [19, 107]. For example the Boolean function $F = (x_1 \wedge y_1) \vee \ldots \vee (x_n \wedge y_n)$ with variable ordering $x_1 < y_1 < x_2 < y_2 < \cdots < x_n < y_n$ is represented by an ROBDD with $2n + 2$ nodes, whereas if we change the ordering to $x_1 < x_2 < \cdots < x_n < y_1 < y_2 < \cdots < y_n$ then the ROBDD requires 2^{n+1} nodes. That is, we go from an ROBDD whose size is linear in the number of variables to one whose size is exponential in the number of variables just by changing the ordering. While it is not practical to try to compute an optimal ordering of variables (finding an optimal ordering is an NP-complete problem [13, 107]), with a bit of care and some intuition into why certain orderings are bad, we can come up with a good heuristic for ordering. The intuition that we use is that variables that are "closely related" should be close to each other in the variable ordering. For example, in F, variables x_i and y_i are closely related so placing them close together in the variable ordering produces a smaller ROBDD.

An important operation for reducing the size of the ROBDD is to remove (through existential quantification of the Boolean function) intermediate variables once they are no longer needed.

This operation is known as *restriction* of the ROBDD. Schachte [118] describes two algorithms for doing this: a general purpose algorithm restrict for removing a single variable, and restrict_threshold for removing all variables greater than or equal to some variable in the variable ordering. The function restrict_threshold is much more efficient than restrict so we want to choose a variable order that allow us to use it as much as possible.

Choosing a good variable ordering involves balancing these two considerations. We use an ordering that allows us to use restrict_threshold whenever we need to do restriction, and that, secondarily, attempts to keep related variables close to each other.

Variables that are needed at the end of the analysis are given the lowest numbers so that extra variables can easily be restricted away at the end of the analysis. The variables that we need at the end are, for all nodes v reachable from the arguments of the predicate,v_{in}, v_{out} and v_ϵ (we refer to these as the *interface* variables); and, for each path p to a unification or call goal and nodes v reachable from variables in the goal, v_p (we refer to these as *leaf* variables).

Starting from the argument nodes, we visit each reachable node in the instantiation grammar, numbering the interface variables for each node in order. These three variables for each node are always constrained together so we number them consecutively. We then traverse the body of the predicate and assign numbers to all the leaf variables.

During the analysis, before analysing a goal at path p we work out what constraint variables we need to keep once we have finished with this goal. This is all constraint variables v_p for nodes reachable from program variables occurring in this goal. We assign numbers to these variables, then analyse the goal and (recursively) its sub-goals, then use restrict_threshold to remove any intermediates that are no longer needed.

6.5.3 Order of Adding Constraints

The order that constraints are added can have a significant impact on the size of intermediate ROBDDs. In general, we want to generate stronger constraints before weaker ones. If a predicate has mode declarations, we generate the constraints corresponding to these first. This generates the final solution set for all the interface variables and the rest of the analysis is effectively reduced to just checking that these constraints are satisfied and working out exactly where each node is produced. Adding mode declaration constraints first is particularly effective if there is only one mode declaration for the predicate (a very common occurrence in typical Mercury programs) since this completely constrains the interface variables.

We also note that for any node v where v_{in} is known to be true from the mode declaration(s), we immediately know that for any goal path p, v_p must be false since if a node is input to the predicate, it cannot be produced within the predicate. Therefore, these variables can be constrained to false immediately after adding the mode declaration constraints.

For the predicate body, we generate constraints bottom-up, starting at the call and unify goals. This also helps to keep the size of the intermediate ROBDDs smaller because these goals usually involve fairly strong constraints on a small number of variables, whereas disjunctions and conjunctions require generating relatively weak constraints on a large number of variables.

An exception to this is that for an if-then-else goal we have the constraints that the condition of the if-then-else may not produce any nodes of non-local variables. These constraints are strong so it is beneficial to generate them before generating constraints for the enclosed goal.

6.5.4 Removing Information from ROBDDs

While ROBDDs are efficient compared to other methods for storing general Boolean constraints, certain kinds of simple constraints can be stored more efficiently using other representations. If we know that these sorts of constraints will appear often, and the overhead of keeping them separate does not outweigh the benefits, we can use these alternative representations to improve the efficiency of the analysis.

Bagnara [7] describes an implementation of groundness analysis in Pos [88], the domain of positive Boolean functions, using ROBDDs. He uses an optimisation where all variables that are definitely true are removed from the ROBDD and kept as a separate set (implemented as a bit vector). This improves the efficiency of his groundness analysis in two ways: it makes it much easier to determine which variables are definitely ground at any point during the analysis, and it reduces the size of the ROBDDs, thus making operations on them more efficient.

Bagnara and Schachte [8] improve on this by also removing equivalent variable pairs from the ROBDD and representing them with an array. They find that this yields significant efficiency gains for groundness analysis using Pos since the resulting ROBDD is nearly always the single node **1**. A more detailed description of the algorithms they use can be found in Schachte [119].

Our analysis uses a larger subset of Boolean functions than Pos. In particular, it is possible (and very likely) that some variables may be definitely false. An obvious augmentation to Bagnara and Schachte's approach which proves useful for us is to include a separate set for definitely false variables as well as the set for definitely true variables.

After removing definite variables and equivalence constraints from the ROBDD, we found it is also useful to remove other 2-SAT constraints. In particular, when analysing programs involving large data structures, a lot of implication constraints are created in the early stages of the analysis, between parent and child nodes in the instantiation grammar. The constraint $\neg(x \wedge y)$, which occurs often in the analysis, is also a 2-SAT constraint. We find that removing these and storing them in a more efficient data structure is very beneficial to performance.

Detecting and removing 2-SAT constraint in an ROBDD is very easy so we remove all of them. We keep these constraints in clausal form and use resolution to obtain a maximal set of clauses. This makes operations such as restriction and disjunction easier. It also has the potential to cause the set of clauses to grow very large. However, we find that in practice this is not a problem. Very few of the constraints added cause a large number of 2-SAT clauses to be generated. Also, we only represent 2-SAT constraints in clausal form if they cannot be represented more compactly using the definite variable and equivalent variables sets.

Below we describe the algorithms required for keeping 2-SAT constraints separate from the ROBDD. We use the notation of Schachte [119] as closely as possible. An ROBDD node $R \in \mathsf{Robdd}$ is either one of the terminal nodes True or False, or a nonterminal node which has three components: $R_{\mathsf{val}} \in \mathsf{BVar}$, $R_{\mathsf{true}} \in \mathsf{Robdd}$ and $R_{\mathsf{false}} \in \mathsf{Robdd}$. We assume a total order $<$ on BVar and require $R_{\mathsf{val}} < (R_{\mathsf{true}})_{\mathsf{val}}$ and $R_{\mathsf{val}} < (R_{\mathsf{false}})_{\mathsf{val}}$ for all $R \in \mathsf{Robdd}$.

An ROBDD node can be created only through the function $\mathsf{make\_node} : \mathsf{BVar} \times \mathsf{Robdd} \times \mathsf{Robdd} \rightarrow \mathsf{Robdd}$. A call $\mathsf{make\_node}(v, R_1, R_2)$ returns an ROBDD node R such that $R_{\mathsf{val}} = v$,

$R_{\mathsf{true}} = R_1$, and $R_{\mathsf{false}} = R_2$. If a matching node R already exists then $\mathsf{make\_node}$ returns it, otherwise it creates a new node. The re-use of existing nodes ensures that each ROBDD has no duplicate sub-graphs.

We refer to an ROBDD by its top level node R.

The semantics of an ROBDD is given by the function $[\![\cdot]\!]_{\mathsf{R}} : \mathsf{Robdd} \to \mathsf{BConstr}$ defined as

$$[\![R]\!]_{\mathsf{R}} = \begin{cases} 0 & \text{if } R = \mathsf{False} \\ 1 & \text{if } R = \mathsf{True} \\ (R_{\mathsf{val}} \wedge [\![R_{\mathsf{true}}]\!]_{\mathsf{R}}) \vee (\neg R_{\mathsf{val}} \wedge [\![R_{\mathsf{false}}]\!]) & \text{otherwise} \end{cases}$$

The operators $\wedge_{\mathsf{R}} : \mathsf{Robdd} \times \mathsf{Robdd} \to \mathsf{Robdd}$ and $\vee_{\mathsf{R}} : \mathsf{Robdd} \times \mathsf{Robdd} \to \mathsf{Robdd}$ provide conjunction and disjunction, respectively, for ROBDDs. That is

$$\begin{aligned} [\![R \wedge_{\mathsf{R}} R']\!]_{\mathsf{R}} &= [\![R]\!]_{\mathsf{R}} \wedge [\![R']\!]_{\mathsf{R}} \\ [\![R \vee_{\mathsf{R}} R']\!]_{\mathsf{R}} &= [\![R]\!]_{\mathsf{R}} \vee [\![R']\!]_{\mathsf{R}} \end{aligned}$$

We make use of the functions $\mathsf{vars\_entailed} : \mathsf{Robdd} \to \mathcal{P}\,\mathsf{BVar}$ and $\mathsf{vars\_disentailed} : \mathsf{Robdd} \to \mathcal{P}\,\mathsf{BVar}$ which are defined by Schachte [119].

$$\begin{aligned} \mathsf{vars\_entailed}(R) &= \{\, v \in \mathsf{BVar} \,|\, \forall \mathcal{B} \in \mathsf{BVal}.\ \mathcal{B} \models [\![R]\!]_{\mathsf{R}} \Rightarrow \mathcal{B}(v) = 1 \,\} \\ \mathsf{vars\_disentailed}(R) &= \{\, v \in \mathsf{BVar} \,|\, \forall \mathcal{B} \in \mathsf{BVal}.\ \mathcal{B} \models [\![R]\!]_{\mathsf{R}} \Rightarrow \mathcal{B}(v) = 0 \,\} \end{aligned}$$

The application $\mathsf{vars\_entailed}(R)$ returns the set of Boolean variables entailed by $[\![R]\!]_{\mathsf{R}}$. The application $\mathsf{vars\_disentailed}(R)$ returns the set of Boolean variables disentailed by $[\![R]\!]_{\mathsf{R}}$.

We also define $R[1/V]$, where $R \in \mathsf{Robdd}$ and $V \subseteq \mathsf{BVar}$, to be the ROBDD obtained by replacing each node R' in R such that $R'_{\mathsf{val}} \in V$ with R'_{true}. Similarly, $R[0/V]$ is the ROBDD obtained by replacing each node R' in R such that $R'_{\mathsf{val}} \in V$ with R'_{false}. The semantics of these functions are given below.

$$\begin{aligned} \big[\!\big[R[1/V]\big]\!\big]_{\mathsf{R}} &= \exists V.\ \left([\![R]\!]_{\mathsf{R}} \ \wedge \ \bigwedge_{v \in V} v \right) \\ \big[\!\big[R[0/V]\big]\!\big]_{\mathsf{R}} &= \exists V.\ \left([\![R]\!]_{\mathsf{R}} \ \wedge \ \bigwedge_{v \in V} \neg v \right) \end{aligned}$$

$R[1/V]$ is the ROBDD which results when we constrain R to require every variable $v \in V$ to have the value 1, and then restrict away these variables from the ROBDD. Likewise, $R[0/V]$ is the result of constraining R such that all $v \in V$ have the value 0, and then restricting away these variables.

Schachte [119] defines a representation for sets of pairs of equivalent variables:

$$\mathsf{Equiv} = \big\{\, E \subseteq \mathsf{BVar} \times \mathsf{BVar} \,\big|\, \forall \langle v, v' \rangle \in E.\ \exists v''.\ \big(v < v' \ \wedge \ (\langle v', v'' \rangle \in E \Rightarrow \langle v, v'' \rangle \in E)\big) \,\big\}$$

and the *leader function* $\lambda_E : \mathsf{BVar} \to \mathsf{BVar}$ for $E \in \mathsf{Equiv}$:

$$\lambda_E(v) = \min\bigl(\{\, v \,\} \cup \{\, v' \in \mathsf{BVar} \,|\, \langle v', v\rangle \in E \,\}\bigr)$$

The semantics of Equiv are given by the function $[\![\cdot]\!]_{\mathsf{E}} : \mathsf{Equiv} \to \mathsf{BConstr}$:

$$[\![E]\!]_{\mathsf{E}} = \bigwedge_{\langle v,v'\rangle \in E} (v \leftrightarrow v')$$

Schachte also defines $\wedge_{\mathsf{E}} : \mathsf{Equiv} \times \mathsf{Equiv} \to \mathsf{Equiv}$ and $\vee_{\mathsf{E}} : \mathsf{Equiv} \times \mathsf{Equiv} \to \mathsf{Equiv}$ which give the operations of conjunction and disjunction, respectively. That is

$$E \wedge_{\mathsf{E}} E' = \mathsf{trans}^*(E \cup E') \qquad [\![E \wedge_{\mathsf{E}} E']\!]_{\mathsf{E}} = [\![E]\!]_{\mathsf{E}} \wedge [\![E']\!]_{\mathsf{E}}$$
$$E \vee_{\mathsf{E}} E' = E \cap E' \qquad [\![E \vee_{\mathsf{E}} E']\!]_{\mathsf{E}} = [\![E]\!]_{\mathsf{E}} \vee [\![E']\!]_{\mathsf{E}}$$

Requiring $E \in \mathsf{Equiv}$ to be a transitive relation on BVar ensures that it explicitly represents all the equivalence relations implied by its semantics. This allows the disjunction to be implemented as a simple set intersection.

The function $\mathsf{equiv\_vars} : \mathsf{Robdd} \to \mathsf{Equiv}$ obtains all pairs of equivalent variables in an ROBDD, and the function $\mathsf{squeeze\_equiv} : \mathsf{Robdd} \times (\mathsf{BVar} \to \mathsf{BVar}) \to \mathsf{Robdd}$ removes all equivalence constraints from an ROBDD given a leader function representing those equivalences. These two functions can be used together to identify and remove all equivalence constraints from an ROBDD.

We now define our extension to add a separate representation for 2-SAT constraints, which we use mostly to represent implications. A literal $\ell \in \mathsf{Literal}$ is either a variable $v \in \mathsf{BVar}$ or its negation $\neg v$. If $\ell = v$ or $\ell = \neg v$ then we define $\ell_{\mathsf{var}} = v$. We let the pair $\langle \ell, \ell' \rangle \in \mathsf{Literal} \times \mathsf{Literal}$ represent the clause $\ell \vee \ell'$. We define $\mathsf{res}^* : \mathsf{Literal} \times \mathsf{Literal} \to \mathsf{Literal} \times \mathsf{Literal}$ such that $\mathsf{res}^*(I)$ is the closure of I under resolution, that is, the least set containing all clauses in I plus all clauses derivable from I through resolution.

Example 6.5.1.

$$\mathsf{res}^*(\{\, \langle \neg w, x\rangle \,, \langle \neg x, y\rangle \,, \langle \neg y, z\rangle \,\}) = \{\, \langle \neg w, x\rangle \,, \langle \neg x, y\rangle \,, \langle \neg y, z\rangle \, \langle \neg w, y\rangle \,, \langle \neg w, z\rangle \,, \langle \neg x, z\rangle \,\} \qquad \triangleleft$$

We represent the 2-SAT constraints as a set of pairs where each pair represents a clause containing two literals. We require each pair to be ordered using the total order $<$ on BVar, and we require that the set of clauses be closed under resolution:

$$\mathsf{2Sat} = \{\, I \subseteq \mathsf{Literal} \times \mathsf{Literal} \,|\, I = \mathsf{res}^*(I) \;\wedge\; \forall \langle \ell, \ell' \rangle \in I.\; \ell_{\mathsf{var}} < \ell'_{\mathsf{var}} \,\}$$

The semantics of $\mathsf{2Sat}$ are given by the function $[\![\cdot]\!]_{\mathsf{2Sat}} : \mathsf{2Sat} \to \mathsf{BConstr}$:

$$[\![I]\!]_{\mathsf{2Sat}} = \bigwedge_{\langle \ell, \ell' \rangle \in I} (\ell \vee \ell')$$

We define operations $\wedge_2 : \mathsf{2Sat} \times \mathsf{2Sat} \to \mathsf{2Sat}$ and $\vee_2 : \mathsf{2Sat} \times \mathsf{2Sat} \to \mathsf{2Sat}$ which provide conjunction and disjunction, respectively, for the 2Sat domain:

$$I \wedge_2 I' = \mathsf{res}^*(I \cup I') \qquad\qquad [\![I \wedge_2 I']\!]_{\mathsf{2Sat}} = [\![I]\!]_{\mathsf{2Sat}} \wedge [\![I']\!]_{\mathsf{2Sat}}$$
$$I \vee_2 I' = I \cap I' \qquad\qquad [\![I \vee_2 I']\!]_{\mathsf{2Sat}} = [\![I]\!]_{\mathsf{2Sat}} \vee [\![I']\!]_{\mathsf{2Sat}}$$

The requirement that all $I \in \mathsf{2Sat}$ be closed under resolution allows the disjunction operation to be implemented as a simple set intersection. It also makes other operations, such as restriction, much easier.

To find all the 2-SAT constraints in an ROBDD we use the function $\mathsf{find2sat} : \mathsf{Robdd} \rightarrow \mathsf{2Sat}$ defined in Figure 6.4. For the ROBDD True, find2sat returns the empty set. For the ROBDD False, find2sat returns the set of all possible 2-SAT constraints. For any other ROBDD R, find2sat computes the 2-SAT constraints involving R_{val} by obtaining the sets of variables entailed and disentailed by R_{true} and R_{false}. It then calls find2sat recursively on R_{true} and R_{false} to compute the 2-SAT constraints for each branch, and then takes the intersection of these sets. A potentially more efficient method for computing the 2-SAT constraints might be to use a branch and bound algorithm which would allow information to be shared between the computations for each branch rather than computing each branch independently and then taking the intersection. The algorithm we have given is complete (i.e. finds all 2-SAT constraints in the ROBDD) only if the ROBDD does not constrain any variables to be either definitely 0 or definitely 1. However, we only require it to operate on ROBDDs where such constraints have already been removed.

$$\begin{array}{l}
\mathsf{find2sat}(R) = \\
\quad \text{if } R = \mathsf{True} \text{ then} \\
\quad\quad \emptyset \\
\quad \text{else if } R = \mathsf{False} \text{ then} \\
\quad\quad \bigcup \mathsf{2Sat} \\
\quad \text{else} \\
\quad\quad \{\, \langle \neg R_{\mathsf{val}}, v \rangle \mid v \in \mathsf{vars_entailed}(R_{\mathsf{true}}) \,\} \;\cup \\
\quad\quad \{\, \langle \neg R_{\mathsf{val}}, \neg v \rangle \mid v \in \mathsf{vars_disentailed}(R_{\mathsf{true}}) \,\} \;\cup \\
\quad\quad \{\, \langle R_{\mathsf{val}}, v \rangle \mid v \in \mathsf{vars_entailed}(R_{\mathsf{false}}) \,\} \;\cup \\
\quad\quad \{\, \langle R_{\mathsf{val}}, \neg v \rangle \mid v \in \mathsf{vars_disentailed}(R_{\mathsf{false}}) \,\} \;\cup \\
\quad\quad \big(\mathsf{find2sat}(R_{\mathsf{false}}) \;\cap\; \mathsf{find2sat}(R_{\mathsf{true}})\big)
\end{array}$$

Figure 6.4: The function find2sat

After we have found all the 2-SAT constraints in an ROBDD, we can remove them using the function $\mathsf{remove2sat} : \mathsf{2Sat} \times \mathsf{Robdd} \rightarrow \mathsf{Robdd}$ defined in Figure 6.5 on the next page. In our implementation, both find2sat and remove2sat are memoed in order to avoid repeated traversals of the same sub-graph.

We represent our Boolean constraints by a tuple $\langle T, F, E, I, R \rangle \in \mathsf{TFEIR}$. We give the definition for TFEIR and its semantic function $[\![\cdot]\!]_{\mathsf{X}} : \mathsf{TFEIR} \rightarrow \mathsf{BConstr}$ in Figure 6.6 on the facing page. Intuitively, $T \subseteq \mathsf{BVar}$ is the set of variables that are definitely true; $F \subseteq \mathsf{BVar}$ is the set of variables that are definitely false; $E \in \mathsf{Equiv}$ is the set of pairs of equivalent variables that do not occur in T or F; $I \in \mathsf{2Sat}$ is the set of constraints in 2-SAT, except those represented by T, F and E; and $R \in \mathsf{Robdd}$ contains any remaining constraints that cannot be represented by other means. The function vars returns the set of Boolean variables contained in a member of Equiv, 2Sat or Robdd.

$\mathsf{remove2sat}(I, R) = \mathsf{remove2sat}'(I, \emptyset, \emptyset, R)$

where

$$
\begin{array}{l}
\mathsf{remove2sat}'(I, T, F, R) = \\
\quad \text{if } R = \mathsf{True} \ \vee \ R = \mathsf{False} \text{ then} \\
\quad\quad R \\
\quad \text{else if } R_{\mathsf{val}} \in T \text{ then} \\
\quad\quad \mathsf{remove2sat}'(I, T, F, R_{\mathsf{true}}) \\
\quad \text{else if } R_{\mathsf{val}} \in F \text{ then} \\
\quad\quad \mathsf{remove2sat}'(I, T, F, R_{\mathsf{false}}) \\
\quad \text{else} \\
\quad\quad \mathsf{make\_node}\big(R_{\mathsf{val}}, \\
\quad\quad\quad \mathsf{remove2sat}'(I,\ T \cup \{\, v \mid \langle \neg R_{\mathsf{val}}, v\rangle \in I \,\},\ F \cup \{\, v \mid \langle \neg R_{\mathsf{val}}, \neg v\rangle \in I \,\},\ R_{\mathsf{true}}), \\
\quad\quad\quad \mathsf{remove2sat}'(I,\ T \cup \{\, v \mid \langle R_{\mathsf{val}}, v\rangle \in I \,\},\ F \cup \{\, v \mid \langle R_{\mathsf{val}}, \neg v\rangle \in I \,\},\ R_{\mathsf{false}})\big)
\end{array}
$$

Figure 6.5: The function remove2sat

$$
\mathsf{TFEIR} = \left\{ \langle T, F, E, I, R\rangle \;\middle|\; \begin{array}{l}
T \subseteq \mathsf{BVar} \ \wedge \ F \subseteq \mathsf{BVar} \ \wedge \ E \in \mathsf{Equiv} \ \wedge \\
I \in \mathsf{2Sat} \ \wedge \ R \in \mathsf{Robdd} \ \wedge \\
T \cap \mathsf{vars}(E) = T \cap \mathsf{vars}(I) = T \cap \mathsf{vars}(R) = \emptyset \ \wedge \\
F \cap \mathsf{vars}(E) = F \cap \mathsf{vars}(I) = F \cap \mathsf{vars}(R) = \emptyset \ \wedge \\
\forall \langle v, v'\rangle \in E.\ v' \notin \mathsf{vars}(I) \ \wedge v' \notin \mathsf{vars}(R) \ \wedge \\
\mathsf{vars\_entailed}(R) = \mathsf{vars\_disentailed}(R) = \emptyset \ \wedge \\
\mathsf{equiv\_vars}(R) = \mathsf{find2sat}(R) = \emptyset
\end{array} \right\}
$$

$$
[\![\langle T, F, E, I, R\rangle]\!]_{\mathsf{X}} = \bigwedge_{v \in T} v \ \wedge \ \bigwedge_{v \in F} \neg v \ \wedge \ [\![E]\!]_{\mathsf{E}} \ \wedge \ [\![I]\!]_{\mathsf{2Sat}} \ \wedge \ [\![R]\!]_{\mathsf{R}}
$$

Figure 6.6: Definition and semantics for TFEIR

This is an extension of the representation used by Schachte [119] which consisted of only the T, E and R components.

Given an arbitrary tuple $X \in \mathcal{P}\,\mathsf{BVar} \times \mathcal{P}\,\mathsf{BVar} \times \mathsf{Equiv} \times \mathsf{2Sat} \times \mathsf{Robdd}$ we want to *normalise* it into an equivalent tuple $X' \in \mathsf{TFEIR}$. This will ensure that each constraint is represented in the most efficient manner. To do this we define the function $\mathsf{normalise} : \mathcal{P}\,\mathsf{BVar} \times \mathcal{P}\,\mathsf{BVar} \times \mathsf{Equiv} \times \mathcal{P}\,\mathsf{2Sat} \times \mathsf{Robdd} \to \mathsf{TFEIR}$. The algorithm we use is shown in Figure 6.7 on the next page.

We first check whether the intersection of T_0 and F_0 is empty. If it is not, then the constraints are inconsistent so we return a TFEIR representing 0. To understand the rest of the algorithm, note that it involves propagating constraints between different components of the TFEIR in order to move constraints as far to the "left" as possible and reduce the size of components towards the "right" as far as possible. The equations defining T_1, F_1 and E_1 propagate constraints from the E component to the T and F components. The equations defining T_2, F_2 and I_1 propagate constraints from the I component to the T and F components. The equations defining T_3, F_3 and R_1 propagate constraints from the R component to the T and F components. The equations defining E_2 and I_2 propagate constraints from the I component to the E component. The equations defining E_3 and R_2 propagate constraints from the R component to the E component.

$$
\begin{array}{l}
\mathsf{normalise}(T_0, F_0, E_0, I_0, R_0) = X \iff \\
\quad \text{if } T_0 \cap F_0 \neq \emptyset \text{ then} \\
\qquad X = \langle \emptyset, \emptyset, \emptyset, \emptyset, \mathsf{False} \rangle \\
\quad \text{else} \\
\qquad T_1 = T_0 \cup \{ x, y \mid \langle x, y \rangle \in E_0 \ \wedge \ \{ x, y \} \cap T_0 \neq \emptyset \} \ \wedge \\
\qquad F_1 = F_0 \cup \{ x, y \mid \langle x, y \rangle \in E_0 \ \wedge \ \{ x, y \} \cap F_0 \neq \emptyset \} \ \wedge \\
\qquad E_1 = E_0 \setminus T_1 \times T_1 \setminus F_1 \times F_1 \ \wedge \\
\qquad T_2 = T_1 \cup \left\{ y \,\middle|\, x \in T_1 \ \wedge \ \big(\langle \neg x, y \rangle \in I_0 \ \vee \ \langle y, \neg x \rangle \in I_0 \big) \right\} \\
\qquad\qquad \cup \left\{ y \,\middle|\, x \in F_1 \ \wedge \ \big(\langle x, y \rangle \in I_0 \ \vee \ \langle y, x \rangle \in I_0 \big) \right\} \ \wedge \\
\qquad F_2 = F_1 \cup \left\{ y \,\middle|\, x \in T_2 \ \wedge \ \big(\langle \neg x, \neg y \rangle \in I_0 \ \vee \ \langle \neg y, \neg x \rangle \in I_0 \big) \right\} \\
\qquad\qquad \cup \left\{ y \,\middle|\, x \in F_1 \ \wedge \ \big(\langle x, \neg y \rangle \in I_0 \ \vee \ \langle \neg y, x \rangle \in I_0 \big) \right\} \ \wedge \\
\qquad I_1 = I_0 \setminus \{ \langle \ell, \ell' \rangle \mid \{ \ell_{\mathsf{var}}, \ell'_{\mathsf{var}} \} \cap (T_2 \cup F_2) \neq \emptyset \} \ \wedge \\
\qquad T_3 = T_2 \cup \mathsf{vars\_entailed}(R_0) \ \wedge \\
\qquad F_3 = F_2 \cup \mathsf{vars\_disentailed}(R_0) \ \wedge \\
\qquad R_1 = R_0[1/T_3][0/F_3] \ \wedge \\
\qquad E_2 = E_1 \wedge_{\mathsf{E}} \left\{ \langle x, y \rangle \,\middle|\, \{ \langle \neg x, y \rangle, \langle x, \neg y \rangle \} \subseteq I_1 \right\} \ \wedge \\
\qquad I_2 = \left\{ \mathcal{L} \in I_1 \,\middle|\, (\mathcal{L} = \langle \neg x, y \rangle \ \vee \ \mathcal{L} = \langle x, \neg y \rangle) \Rightarrow \langle x, y \rangle \notin E_2 \right\} \\
\qquad E_3 = E_2 \wedge_{\mathsf{E}} \mathsf{equiv\_vars}(R_1) \ \wedge \\
\qquad R_2 = \mathsf{squeeze\_equiv}(R_1, \lambda_{E_3}) \ \wedge \\
\qquad I_3 = I_2 \wedge_2 \mathsf{find2sat}(R_2) \ \wedge \\
\qquad R_3 = \mathsf{remove2sat}(I_3, R_2) \ \wedge \\
\qquad \text{if } T_3 = T_0 \ \wedge \ F_3 = F_0 \ \wedge \ E_3 = E_0 \ \wedge \ I_3 = I_0 \ \wedge \ R_3 = R_0 \text{ then} \\
\qquad\quad X = \langle T_3, F_3, E_3, I_3, R_3 \rangle \\
\qquad \text{else} \\
\qquad\quad X = \mathsf{normalise}(T_3, F_3, E_3, I_3, R_3)
\end{array}
$$

Figure 6.7: Normalisation function for TFEIR

Finally, the equations defining I_3 and R_3 propagate constraints from the R component to the I component. We iterate over these steps until we reach a fixpoint. Termination is guaranteed because the sets E_0, I_0 and R_0 are finite and at least one of these three components is reduced in size in each iteration except the last.

The operators $\wedge_{\mathsf{X}} : \mathsf{TFEIR} \times \mathsf{TFEIR} \to \mathsf{TFEIR}$ and $\vee_{\mathsf{X}} : \mathsf{TFEIR} \times \mathsf{TFEIR} \to \mathsf{TFEIR}$ provide conjunction and disjunction, respectively, for TFEIRs. Their definitions are shown in Figure 6.8 on the facing page. These definitions are extended from the definitions given in Schachte [119].

For conjunction, we could just return $\mathsf{normalise}(T_0 \cup T_1,\ F_0 \cup F_1,\ E_0 \wedge_{\mathsf{E}} E_1,\ I_0 \wedge_{\mathsf{I}} I_1,\ R_0 \wedge_{\mathsf{R}} R_1)$. However, the definition given in Figure 6.8 is more efficient because it reduces the size of the two ROBDDs as much as possible before taking their conjunction.

For disjunction, we take the disjunction of each component of the TFEIRs from left to right. For example, we first take the disjunction of the components T_0 and T_1 which gives us $T_0 \cap T_1$. Any constraints that occur in only one of T_0 and T_1 must be added back into the E, I and R components of their respective TFEIRs before those components are processed.

In practice, we rarely use these generalised algorithms. Instead we use algorithms specialised for each type of constraint we need to add. We also try to avoid calling normalise where possible. For example, if we wish to conjoin the constraint represented by $\langle T, F, E, I, R \rangle$ with the constraint v for some $v \in \mathsf{BVar}$ then we simply return $\langle T \cup \{ v \}, F, E, I, R \rangle$. Normalising the TFEIR each time we add such a simple constraint would add a large overhead so we delay the normalisation

$$\langle T_0, F_0, E_0, I_0, R_0\rangle \wedge_{\mathsf{X}} \langle T_1, F_1, E_1, I_1, R_1\rangle = \mathsf{normalise}(T_0' \cup T_1',\ F_0' \cup F_1',\ E_0' \wedge_{\mathsf{E}} E_1',\ I_0' \wedge_2 I_1',\ R_0' \wedge_{\mathsf{R}} R_1')$$

where

$$\begin{aligned} T &= T_0 \cup T_1 & E &= E_0 \wedge_{\mathsf{E}} E_1 \\ F &= F_0 \cup F_1 & I &= I_0 \wedge_2 I_1 \end{aligned}$$
$$\langle T_0', F_0', E_0', I_0', R_0'\rangle = \mathsf{normalise}(T, F, E, I, R_0)$$
$$\langle T_1', F_1', E_1', I_1', R_1'\rangle = \mathsf{normalise}(T, F, E, I, R_1)$$

$$\langle T_0, F_0, E_0, I_0, R_0\rangle \ \vee_{\mathsf{X}}\ \langle T_1, F_1, E_1, I_1, R_1\rangle \quad = \quad \langle T_0 \cap T_1,\ F_0 \cap F_1,\ E,\ I_0' \vee_2 I_1',\ R_0' \vee_{\mathsf{R}} R_1'\rangle$$

where

$$\begin{aligned}
E_0' &= E_0 \wedge_{\mathsf{E}} \left\{ \langle x, y\rangle \,\middle|\, x < y \ \wedge\ \big(\{\, x, y \,\} \subseteq T_0 \setminus T_1 \ \vee\ \{\, x, y \,\} \subseteq F_0 \setminus F_1\big) \right\} \\
E_1' &= E_1 \wedge_{\mathsf{E}} \left\{ \langle x, y\rangle \,\middle|\, x < y \ \wedge\ \big(\{\, x, y \,\} \subseteq T_1 \setminus T_0 \ \vee\ \{\, x, y \,\} \subseteq F_1 \setminus F_0\big) \right\} \\
E &= E_0' \vee_{\mathsf{E}} E_1' \\
I_0' &= I_0 \wedge_2 \left\{ \langle \neg x, y\rangle, \langle x, \neg y\rangle \,\middle|\, x < y \ \wedge\ \langle x, y\rangle \in E_0' \setminus E_1' \right\} \\
I_1' &= I_1 \wedge_2 \left\{ \langle \neg x, y\rangle, \langle x, \neg y\rangle \,\middle|\, x < y \ \wedge\ \langle x, y\rangle \in E_1' \setminus E_0' \right\} \\
R_0' &= R_0 \wedge_{\mathsf{R}} \bigwedge\nolimits_{\mathsf{R}}{}_{v \in T_0 \setminus T_1} \lambda_E(v) \ \wedge_{\mathsf{R}} \bigwedge\nolimits_{\mathsf{R}}{}_{v \in F_0 \setminus F_1} \neg\lambda_E(v) \ \wedge_{\mathsf{R}} \bigwedge\nolimits_{\mathsf{R}}{}_{\langle x, y\rangle \in (E_0' \setminus E_1') \vee_{\mathsf{E}} E_0} (x \leftrightarrow y) \wedge_{\mathsf{R}} \\
&\quad \bigwedge\nolimits_{\mathsf{R}}{}_{\langle \ell, \ell'\rangle \in (I_0' \setminus I_1') \vee_2 I_0} (\ell \vee_{\mathsf{R}} \ell') \\
R_1' &= R_1 \wedge_{\mathsf{R}} \bigwedge\nolimits_{\mathsf{R}}{}_{v \in T_1 \setminus T_0} \lambda_E(v) \ \wedge_{\mathsf{R}} \bigwedge\nolimits_{\mathsf{R}}{}_{v \in F_1 \setminus F_0} \neg\lambda_E(v) \ \wedge_{\mathsf{R}} \bigwedge\nolimits_{\mathsf{R}}{}_{\langle x, y\rangle \in (E_1' \setminus E_0') \vee_{\mathsf{E}} E_1} (x \leftrightarrow y) \wedge_{\mathsf{R}} \\
&\quad \bigwedge\nolimits_{\mathsf{R}}{}_{\langle \ell, \ell'\rangle \in (I_1' \setminus I_0') \vee_2 I_1} (\ell \vee_{\mathsf{R}} \ell')
\end{aligned}$$

Figure 6.8: Conjunction and disjunction for TFEIR

until it is actually needed. Another option would be to write specialised normalisation algorithms for each type of constraint we want to add. This is the approach taken by Schachte [119].

6.6 Experimental Results

We now present some experimental results to show the feasibility of our analysis. The timings are all taken from tests run on a Gateway Select 950 PC with a 950MHz AMD Athlon CPU, 256KB of L2 cache and 256MB of memory, running Linux kernel 2.4.16.

Table 6.1 compares times for mode checking some standard logic programming benchmarks. The column labelled "simple" is the time for the simple constraint-based system for ground variables presented in Section 6.2. The column labelled "full" is for the full constraint-based system presented in Section 6.3. The column labelled "old" is the time for mode checking in the "current" Mercury mode checker (as described in Chapter 3). The next two columns show the ratios between the new and old systems. The final column is taken from Table 5.2 on page 115 and shows, for reference, the ratio between the alias tracking system of Chapter 5 and the old system. All times are in milliseconds. We measured the time for 10 repeats of the mode analysis for each benchmark and divided the result by 10.

	simple	full	old	simple/old	full/old	alias/old
cqueens	407	405	17	23	23	1.40
crypt	1032	1335	38	27	35	1.44
deriv	13166	32541	59	223	551	1.73
nrev	520	569	42	12	13	0.88
poly	1348	5245	63	21	83	1.76
primes	356	358	12	29	29	1.70
qsort	847	1084	112	7	9	0.74
queens	386	381	9	42	42	1.37
query	270	282	11	24	25	1.65
tak	204	201	2	102	100	1.56

Table 6.1: Times for mode checking logic programming benchmarks

The constraint-based analyses are significantly slower than the current system and the alias tracking system. This is partly because they are obtaining much more information about the program and thus doing a lot more work. For example, the current system selects a fixed sequential order for conjunctions during the mode analysis — an order that disallows partially instantiated data structures — whereas the constraint-based approaches allow all possible orderings to be considered while building up the constraints. The most appropriate scheduling can then be selected based on the execution model considering, for example, argument passing conventions (e.g. for the possibility of introducing tail calls) and whether the execution is sequential or parallel.

Profiling shows that much of the execution time is spent in building and manipulating the ROBDDs. It may be worth investigating different constraint solvers, such as propagation-based solvers [4, 35, 93, 94, 123]. Further optimisation of the BDD-based approach [97, 107], and an investigation of more efficient 2-SAT solvers [48, 158] may also yield improved performance. Another possible method for improving overall analysis time would be to run the old mode analysis first and only use the new analysis for predicates for which the old analysis fails.

It is interesting to observe the differences between the simple constraint system and the full system. None of these benchmarks require partially instantiated data structures so they are all able to be analysed by the simple system. In some cases, the simple system is not very different to the full system, but in others—particularly in the bigger benchmarks—it is significantly faster. We speculate that this is because the bigger benchmarks benefit more from the reduced number of constraint variables in the simple analysis.

Table 6.2 shows some timings for programs that make use of partially instantiated data structures, which the current Mercury system (and the simple constraint-based system) is unable to analyse. Again the times are in milliseconds, measured by repeating each test 10 times and dividing the result by 10.

	check	infer	infer/check
iota	384	472	1.22
append	327	488	1.49
copytree	150	6174	41.16

Table 6.2: Times for checking and inferring modes with partially instantiated data structures

The “iota” benchmark is the program from Example 5.1.4 on page 91. The “append” benchmark is the classic `append/3`. The “copytree” benchmark is a small program that makes a structural copy of a binary tree skeleton, with all elements in the copy being new free variables, as shown below:

```
:- type tree(T) ---> empty
                ;    tree(tree(T), T, tree(T)).
:- inst tree(I) == bound(empty
                    ;     tree(tree(I), I, tree(I))).
:- pred copytree(tree(T), tree(U)).
:- mode copytree(in(tree(free)), out(tree(free))).

copytree(empty, empty).
copytree(tree(L1, _, R1), tree(L2, _, R2)) :-
        copytree(L1, L2), copytree(R1, R2).
```

The times in the “check” columns are for checking programs with mode declarations whereas the “infer” column shows times for doing mode inference with all mode declarations removed. It is interesting to note the saving in analysis time achieved by adding mode declarations. This is particularly notable for the “copytree” benchmark where mode inference is able to infer many more modes than the one we have declared. (We can similarly declare only the `(in, in, out)` mode of `append` and reduce the analysis time for that to 210ms.)

6.7 Related Work

Moded Flat GHC [23, 140–143] is the only other system we are aware of that uses constraints for mode analysis. However, as we discussed earlier, Moded Flat GHC relies on position in the clause (in the head or guard versus in the body) to determine if a unification is allowed to bind any variables, which significantly simplifies the problem. The constraints generated are equational, and rely on delaying the complex cases where there are three or more occurrences of a variable in a goal.

Boolean constraints have become a proven technique for use in groundness and sharing analyses of logic programs [e.g. 31–33, 81, 84, 88, 119]. In such analyses, each program variable is typically represented, in the abstract domain, by a single Boolean variable. The abstraction of the program is a set of constraints on the Boolean variables. To discover the state of a variable at a point during program execution, one must examine the constraint store at the corresponding point during the abstract interpretation. In contrast, in our approach, we have a separate Boolean variable for each (position in a) variable at each program point. This is necessary to enable us to express constraints between variables at different program points. However, the large number of Boolean variables which our approach necessitates makes it more challenging to represent the constraints in an efficient manner.

Recent work on analysis of functional programs has also used Boolean constraints. For example, Glynn et al. have had some success with using Boolean constraints for binding-time analysis [59] and strictness analysis [58].

There has been other work on constraint-based analysis of Mercury. In particular, the work of Vanhoof [148, 149] on a constraint-based binding-time analysis is notable. It has a similar basic approach to ours of using constraints between the "positions" in the type trees of variables to express data flow dependencies. However, binding-time analysis requires the results of mode analysis to be available. Given that mode analysis has determined which argument are input and which other variables are computed from those arguments, it divides the input arguments into two classes which are bound at different times, and seeks to find out when each of the other variables are bound so that it can schedule each body goal to be evaluated as early as possible. The main constraint here is that a variable cannot be bound until the variables it is computed from are bound. These constraints are significantly simpler than the ones we deal with here.

6.8 Limitations and Possible Future Work

We have defined a constraint based approach to mode analysis of Mercury. While it is not as efficient as the current system for mode checking, it is able to check and infer more complex modes than the current system, and decouples reordering of conjuncts from determining producers. The implementation handles all Mercury constructs, including the higher-order constructs.

The constraint-based mode analysis does not yet handle subtyping or unique modes. We plan to extend it to handle these features as well as explore more advanced mode systems: complicated uniqueness modes, where unique objects are stored in and recovered from data structures; polymorphic modes, where Boolean variables represent a pattern of mode usage; and the circular modes needed by client-server programs, where client and server processes (modelled as recursive loops) cooperate to instantiate different parts of a data structure in a coroutining manner.

For handling unique modes we plan to look at using a modified version of the sharing analysis of Lagoon and Stuckey [81], as we discussed in Section 5.5. Another approach may be something similar to that of the linearity analysis of Ueda [142] for Moded Flat GHC, which we discussed in Section 3.5. However, we would need to extend that work considerably to handle the kinds of data flow we would like to allow in Mercury programs.

Chapter 7

Conclusion

In this chapter we provide a review of contents of this thesis, followed by a discussion of the benefits it provides to various groups. The discussion includes possible future research and implementation directions.

7.1 Review

This thesis has been based on the premise that a strong prescriptive mode system can allow for an efficient implementation of a logic programming language without the need for non-logical features that would destroy the declarative semantics of the language. We have shown that providing a precise mode analysis domain and algorithm, based on a strong static type system, can allow such a system to retain much of the expressiveness of an unmoded logic programming language while also contributing significantly to the robustness required for a real world industrial strength language.

In Chapter 3 we presented a detailed description of the mode system of Mercury 0.10.1, including a formalisation of the mode analysis rules and an overview of the algorithms used for mode checking and inference. Previous descriptions of Mercury's mode system, e.g. [66, 131], have been intended only as a high-level overview of the system and do not contain the detail necessary for the in-depth understanding required to reason about and extend the system, instead referring the reader to the papers on which the mode system is based [65, 128, 129]. These papers, and papers on related systems such as HAL [55], describe similar mode systems to that of Mercury. However, there has been no description of the design choices that led to the Mercury mode system being what it is, and the only detailed description of how it worked was in the source code for the Melbourne Mercury compiler.

We have clarified the relationship between the mode system and the formalism of abstract interpretation. By providing the concretisation and abstract functions we have formalised the semantics of the abstract domain of instantiation states. The "matches" partial order $\sqsubseteq$ formalises the comparison of the information content of different instantiation states. We have not provided a formal proof that the mode system is sound. However, we are confident that the framework we have given could be used to provide the basis for such a proof.

The $\sqsubseteq$ partial order also allows a comparison to similar mode systems, such as that of HAL [55], which differs from Mercury's mode system mainly in that it defines $\sqsubseteq$ slightly differently. This small difference, however, has significant consequences for how strong the mode system is and how suitable it is for a particular purpose — Mercury's mode system, designed with software engineering goals in mind, aims to be as strong as possible, whereas the mode system of HAL has made compromises which make the mode system weaker, in order to make constraint logic programming more convenient while still gaining most of the benefits of a prescriptive mode system.

Formalising and writing down the mode analysis rules has been very beneficial in enabling us to reason about the mode system. For example, formalising the notions of abstract unification, merging and matching through the definition of the two partial orders $\preceq$ and $\sqsubseteq$ greatly facilitated the design and development of the constrained polymorphic modes extension described in Chapter 4.

The extension in Chapter 4 allows the relationship between the modes of the input and output arguments of a predicate to be expressed more precisely. The benefit is analogous to that provided by parametric polymorphic type systems. The caller of a predicate will often have more precise information about the instantiation state of the predicate's input arguments than is required by the predicate's mode declaration. This extension allows the caller to deduce more precise information about the instantiation states of the output arguments after that particular call succeeds, based on the information it has about the input arguments. This increases the expressiveness of the language in a way that has extremely useful practical consequences. One major benefit is that it allows higher-order values to be stored in and retrieved from polymorphic data structures, such as lists and trees, without losing their higher-order mode information. This makes it much easier to use certain styles of higher-order programming which are common in functional programming languages. It allows higher-order values to be treated much more like the "first class" values that they are intended to be.

In Chapter 5 we presented an extension to the mode system to keep track of definite aliases between variables within a predicate body. The alias tracking system uses inst keys to indicate which parts of each inst in an instmap are aliased. Inst keys are in some ways a compile time analog of logic variables. However, they can represent sharing between bound terms as well as between unbound variables.

The increase in the precision of the mode system resulting from alias tracking has two major benefits. The first is that it allows some uses of partially instantiated data structures. One use we saw for this was to enable the introduction of tail recursion into many predicates which could not otherwise be made tail recursive. We have restricted the use of partially instantiated data structures to cases that require only a single reference to the part of the data structure which needs to be filled in. This avoids the need for pointer chains and trailing in the abstract machine and can therefore be implemented without any efficiency costs. If it were desired, the analysis could be extended to allow cases where multiple references are required. The machinery to support this has been implemented for the HAL project and can be compiled into the Mercury run time system through a compiler flag. However, such features incur efficiency costs at run time, even in parts of the code where they are not being used, so we try to avoid them if possible. In any case, most uses of partially instantiated data structure which require multiple references are likely to be un-modeable in the Mercury mode system without the use of dynamic modes (i.e. the `any` inst) and are therefore more amenable to the Herbrand constraint solving approach of HAL [49].

The second major benefit of the alias tracking analysis is that it allows more complex uses of unique modes. Unique modes are important for modelling input/output and destructive update (e.g. for arrays) in a logically sound way. They can also provide the compiler with the information it needs to implement automatic compile time garbage collection and structure re-use. The alias tracking extension makes the mode system precise enough to handle cases where unique objects are stored inside other unique objects. This was not previously possible, making the use of unique modes in large programs inconvenient. A large program may have several unique objects which must be threaded through parts of the program. It is convenient to be able to bundle them into one data structure rather than requiring a separate pair of `(di, uo)` arguments for each predicate for each unique object. This is now possible.

An unfortunate side effect of the alias tracking system is that it slows down mode analysis significantly, even in cases where the extra information it provides is not required. We would like to investigate ways of making the system more efficient.

In Chapter 6 we presented an alternative approach to mode analysis using Boolean constraints to express the instantiation relationships between variables. The Boolean constraints determine, for each node in the expanded type graph of each variable, which subgoal is responsible for producing that node. This information can then be used to determine an execution order for conjuncts in each conjunction and can be used to check the correctness of mode declarations. The system we have presented is as precise and expressive as the alias tracking system of Chapter 5. One advantage it provides, however, is that it allows re-ordering of conjunctions when doing mode inference rather than just mode checking. It also separates out the conceptual stages of mode analysis: first determine producers and consumers, then schedule goals. We plan to add further stages to handle sub-typing constraints and unique modes.

One significant benefit of the work of Chapter 6 is that it provides an explicit representation of the instantiation graphs and a means of referring to individual nodes in the graph, rather than just to the insts of program variables. This gives us much more precision than previously possible in the mode system. We believe this will make it easier to add further extensions required to describe complex patterns of data flow, such those required for allowing co-operative co-routining and the paradigms of incomplete messages and synchronous streams.

The first phase of the analysis provides the information needed to determine which goals in a conjunction may be executed in parallel. This could be used for automatic parallelisation of programs in a framework such as that described by Conway [38].

We have implemented a prototype of the constraint-based mode analysis system. While it is currently significantly slower than the existing mode system, it does show the feasibility of the approach. More work is required to improve its efficiency.

7.2 Contributions of this Thesis

In this section we discuss the contributions of this thesis to the three main groups of beneficiaries: Mercury application programmers, Mercury implementors, and language designers/theoreticians.

7.2.1 Benefits to Programmers

The main benefit this thesis provides for Mercury application programmers is that it increases the expressiveness of the language, allowing programming constructs which were not previously possible.

The limitations of the pre-existing mode system, as described in Chapter 3, restrict programmers to using mostly just the simple modes `in` and `out` as well as the unique modes `di` and `uo` for some special cases, such as I/O and destructively updatable arrays. This mode system gives Mercury a level of expressiveness which is roughly comparable to a strict functional programming language. Actually, in some ways it provides considerably more expressiveness than functional languages, such as providing nondeterminism and backtracking. However, in other ways it provides slightly less expressiveness, for example by making it harder to use higher-order objects which have been stored inside polymorphic data structures.

The polymorphic mode system described in Chapter 4 alleviates the above problem by allowing the programmer to declare modes for polymorphic predicates which can propagate the higher-order mode information from input arguments to output arguments.

The mode system of Mercury is very complex and there may be some concern about how usable it is for the average application programmer. Fortunately, we do not believe that this is a huge problem. Most of the time, programmers will continue to need only the simple modes `in`, `out`, `di` and `uo`. Our experience has been that most programmers, even those with no previous experience of logic programming, find these modes intuitive and easy to use because they represent the familiar concepts of input and output arguments and destructive update. In addition, we believe that the mode declarations act a valuable documentation which aids the programmer in determining the intended meaning and use of a predicate. Despite the positive anecdotal evidence, it would be interesting to conduct formal usability testing of the language. Such testing, however, is beyond the scope of this thesis.

The polymorphic mode system extensions of Chapter 4 will mainly be used by programmers writing libraries of polymorphic code. Such programmers are likely to be more experienced with the language and to be familiar the extra concepts required. Application programmers will still need to understand polymorphic mode declarations to some degree in order to make use of such libraries. However, so long as they can recognise that, say, `in(list_skel(I))` is an input argument and `out(list_skel(I))` is an output argument, they should not have too much trouble. Furthermore, we expect that most polymorphically-moded predicates will also have corresponding polymorphic types. Using the "theorems for free" principle described in Section 4.4 would allow much polymorphically-typed code to take advantage of polymorphic modes without any extra effort required on the part of either the library writer or application programmer.

The extensions described in Chapters 5 and 6 increase the precision of mode analysis and thus allow more programs to be classed as mode correct. However, they do not change the syntax of the language in any way and thus should not complicate the language for application programmers. In fact, they should make the application programmer's task simpler because programs which might otherwise have resulted in the generation of mode errors, which the novice programmer may find hard to understand, will now be allowed by the compiler.

At present, these two extensions only allow for alias tracking within the body of a predicate.

In future, we may like to allow modes to be declared or inferred which contain aliases between predicate arguments. This *would* require changes to the syntax for mode declarations and it is not yet clear how such changes could be made in a way that causes the least confusion to Mercury programmers.

7.2.2 Benefits to the Mercury Implementors

Probably the major benefit for the implementors of the Mercury language is the formalisation of the mode system presented in Chapter 3. This provides a description of Mercury's mode system that is more detailed than that provided in the language reference manual [66], but at a higher level of abstraction than the actual implementation. We believe that it provides the right level of detail and abstraction to allow new implementors to get up to speed with the workings of the mode analyser in the existing compiler or even to develop a new implementation of Mercury from scratch.

Chapters 4–6 provide documentation for the implementation of the extensions to the mode system which they respectively describe. Chapters 5 and 6, in particular, describe experimental prototypes which still require significant implementation work before becoming part of the Mercury distribution. They also provide much potential for extension and future experimentation (see below). These parts of the thesis should be invaluable to whoever eventually takes on these tasks.

7.2.3 Benefits to Language Designers/Theoreticians

The work in this thesis provides several benefits for language designers and logic programming theoreticians. First, we have shown that a strongly typed logic programming language with a strong prescriptive mode system is capable of being expressive enough to allow a rich variety of logic programming constructs and idioms. We would like to encourage future language designers to continue to experiment with extending the boundaries of what can be achieved in such a system. The constraint-based mode system of Chapter 6 provides a good framework for experimenting with the different properties of mode systems by modifying the constraints. For example, by relaxing some of the constraints which enforce sequentiality we can obtain a system which allows co-operative co-routining and the paradigms of incomplete messages and synchronous streams.

In Chapter 3 we have provided an important link between theory and practice in the form of a Galois connection and a set of inference rules. This gives a formal connection between the pre-existing implementation of the Mercury mode system and the theory of abstract interpretation which is an essential precursor to eventually proving the soundness of the system.

Bibliography

[1] A. Aiken and B. Murphy. Implementing regular tree expressions. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pages 427–447. ACM Press, 1991.

[2] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[3] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):147–147, March 1976.

[4] Krzysztof R. Apt. Some remarks on Boolean constraint propagation. In *New Trends in Constraints, Joint ERCIM/Compulog Net Workshop*, volume 1865 of *Lecture Notes in Artificial Intelligence*. Springer, 2000.

[5] Krzysztof R. Apt and Roland N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[6] Michael A. Arbib, A. J. Kfoury, and Robert N. Moll. *A Basis for Theoretical Computer Science*. Springer, 1981.

[7] Roberto Bagnara. A reactive implementation of *Pos* using ROBDDs. In H. Kuchen and S. D. Swierstra, editors, *Programming Languages: Implementations, Logics and Programs, Proceedings of the Eighth International Symposium*, volume 1140 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 1996.

[8] Roberto Bagnara and Peter Schachte. Factorizing equivalent variable pairs in ROBDD-based implementations of *Pos*. In A. M. Haeberer, editor, *Proceedings of the Seventh International Conference on Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 1999.

[9] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.

[10] Ralph Becket. Thinking about mode analysis. Email to the mercury-developers mailing list, January 2003. URL `http://www.mercury.cs.mu.oz.au/mailing-lists/mercury-developers/mercury%-developers.0301/0000.html`.

[11] Peter A. Bigot and Saumya Debray. Return value placement and tail call optimization in high level languages. *Journal of Logic Programming*, 38(1):1–29, January 1999.

[12] Hans-Juergen Boehm. Dynamic memory allocation and garbage collection. *Computers in Physics*, 9(3):297–303, May 1995.

[13] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, September 1996.

[14] George Boole. *The Mathematical Analysis of Logic: Being an Essay towards a Calculus of Deductive Reasoning*. Macmillan, Barclay, and Macmillan, Cambridge, 1847.

[15] George Boole. *Investigation of the Laws of Thought*. Dover, New York, 1854.

[16] J. Boye and J. Małuszyński. Directional types and the annotation method. *Journal of Logic Programming*, 33(3):179–220, 1997.

[17] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, third edition, 2000.

[18] Maurice Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, 1991.

[19] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[20] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[21] W. Chen, M. Kifer, and D. S. Warren. HiLog: A first-order semantics for higher-order logic programming. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 1090–1114. MIT Press, 1989.

[22] M. Cheng, M. van Emden, and B. Richards. On Warren's method for functional programming in logic. In *Proceedings of the Seventh International Conference on Logic Programming*, 1990.

[23] Kenta Cho and Kazunori Ueda. Diagnosing non-well-moded concurrent logic programs. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 215–229, 1996.

[24] Alonzo Church. A set of postulates for the foundation of logic: Part I. *Annals of Mathematics*, 33:346–366, 1932.

[25] Alonzo Church. A set of postulates for the foundation of logic: Part II. *Annals of Mathematics*, 34:839–864, 1933.

[26] Keith Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[27] Keith Clark and Steve Gregory. Notes on systems programming in Parlog. In *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, pages 299–306, 1984.

[28] Keith Clark and Steve Gregory. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[29] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer, fourth edition, 1994.

[30] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Theoretical Computer Science*, 124(1):93–125, 1994.

[31] Michael Codish and Bart Demoen. Analyzing logic programs using "PROP"-ositional logic programs and a magic wand. *Journal of Logic Programming*, 25(3):249–274, 1995.

[32] Michael Codish and Harald Søndergaard. The Boolean logic of set sharing analysis. In *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 89–100. Springer, 1998.

[33] Michael Codish, Harald Søndergaard, and Peter J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21 (5):948–976, 1999.

[34] C. Codognet, P. Codognet, and M. Corsini. Abstract interpretation of concurrent logic languages. In *Proceedings of the North American Conference on Logic Programming*, pages 215–232. MIT Press, 1990.

[35] P. Codognet and D. Diaz. A simple and efficient Boolean constraint solver for constraint logic programming. *Journal of Automated Reasoning*, 17(1):97–128, 1996.

[36] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, October 2002.

[37] Tom Conlon. *Programming in PARLOG*. Addison-Wesley, 1989.

[38] Thomas Conway. *Towards Parallel Mercury*. PhD thesis, The University of Melbourne, August 2002.

[39] Thomas Conway, Fergus Henderson, and Zoltan Somogyi. Code generation for Mercury. In *Proceedings of the International Symposium on Logic Programming*, pages 242–256, 1995.

[40] Thomas C. Conway and Zoltan Somogyi. Deep profiling: Engineering a profiler for a declarative programming language. Technical Report 2001/24, Department of Computer Science and Software Engineering, The University of Melbourne, July 2001. URL `http://www.mercury.cs.mu.oz.au/information/papers.html`.

[41] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[42] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[43] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[44] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer, 1992.

[45] Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190. ACM Press, 2002.

[46] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[47] Saumya K. Debray and David S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229, September 1988.

[48] Alvaro del Val. On 2-SAT and renamable horn. In *Proceedings of the Seventeenth (U.S.) National Conference on Artificial Intelligence*, pages 343–348. AAAI Press/The MIT Press, 2000.

[49] Bart Demoen, María García de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. Herbrand constraint solving in HAL. In *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 260–274, 1999.

[50] Bart Demoen, María García de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. An overview of HAL. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 174–188, 1999.

[51] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer, 1996.

[52] Tyson Dowd, Zoltan Somogyi, Fergus Henderson, Thomas Conway, and David Jeffery. Run time type information in Mercury. In *Proceedings of the First International Conference on the Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 224–243. Springer, 1999.

[53] S. Etalle and M. Gabbrielli. Layered modes. *Journal of Logic Programming*, 39:225–244, 1999.

[54] John P. Gallagher and D. André de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[55] María García de la Banda, Peter J. Stuckey, Warwick Harvey, and Kim Marriott. Mode checking in HAL. In *Proceedings of the First International Conference on Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 1270–1284. Springer, 2000.

[56] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[57] Ferenc Gécseg and Magnus Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 1–68. Springer, 1996.

[58] Kevin Glynn, Martin Sulzmann, and Peter J. Stuckey. Efficient strictness analysis with HORN constraints. In P. Cousot, editor, *Proceedings of the Eighth International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 73–92. Springer, 2001.

[59] Kevin Glynn, Martin Sulzmann, Peter J. Stuckey, and Harald Søndergaard. Boolean constraints for binding-time analysis. In O. Danvy and A. Filinksi, editors, *Proceedings of the Second Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 36–92. Springer, 2001.

[60] Thomas Hallgren and Magnus Carlsson. Programming with Fudgets. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 137–182. Springer, May 1995.

[61] Paul Halmos. *Naïve Set Theory*. Springer, 1974.

[62] Dana G. Harrington. A type system for destructive updates in declarative programming languages. Master's thesis, Department of Computer Science, University of Calgary, 2001.

[63] R. Haygood. *The Aquarius Prolog Users' Manual*. Department of Computer Science, University of Southern California, April 1993.

[64] Andy Heaton, Muhamed Abo-Zaed, Michael Codish, and Andy King. A simple polynomial groundness analysis for logic programs. *Journal of Logic Programming*, 45(1–3):143–156, 2000.

[65] Fergus Henderson. Strong modes can change the world! Honour's report, Department of Computer Science, The University of Melbourne, November 1992. URL `http://www.mercury.cs.mu.oz.au/information/papers.html`.

[66] Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, Chris Speirs, Tyson Dowd, and Ralph Becket. *The Mercury Language Reference Manual*. Department of Computer Science and Software Engineering, The University of Melbourne, April 2001. URL `http://www.mercury.cs.mu.oz.au/information/documentation.html`. Version 0.10.1.

[67] Fergus Henderson, Zoltan Somogyi, and Thomas Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the Australian Computer Science Conference*, pages 337–346, January 1996.

[68] Pat M. Hill and John W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[69] J. R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[70] Alfred Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, 16(1):14–21, 1951.

[71] ISO. Standard for the programming language Prolog. Standard Number ISO/IEC 13211-1:1995, International Organisation for Standardisation (ISO), 1995.

[72] D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 154–165. MIT Press, 1989.

[73] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM Press, 1987.

[74] Gerda Janssens and Maurice Bruynooghe. Deriving descriptions of possible value of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2 & 3): 205–258, 1992.

[75] David Jeffery. *Expressive Type Systems for Logic Programming Languages*. PhD thesis, The University of Melbourne, February 2002.

[76] David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in Mercury. In J. Edwards, editor, *Proceedings of the Twenty-Third Australasian Computer Science Conference*, volume 22 of *Australian Computer Science Communications*, pages 128–135. IEEE Press, 2000.

[77] Kenneth W. Kennedy. A survey of data flow analysis techniques. In Steven S. Muchnik and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.

[78] Feliks Kluźniak. Type synthesis for Ground Prolog. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 788–816. MIT Press, 1987.

[79] Robert A. Kowalski. Predicate logic as a programming language. In *Proceedings of the Congress of the International Federation for Information Processing*, pages 569–574. Elsevier/North Holland, 1974.

[80] Robert A. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7): 424–436, July 1979.

[81] Vitaly Lagoon and Peter J. Stuckey. Precise pair-sharing analysis of logic programs. In Claude Kirchner, editor, *Proceedings of the Fourth International Conference on Principles and Practice of Declarative Programming*, pages 99–108. ACM Press, 2002.

[82] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, February 1965.

[83] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part II. *Communications of the ACM*, 8(3):158–165, March 1965.

[84] Baudouin Le Charlier and Pascal Van Hentenryck. Groundness analysis for Prolog: Implementation and evaluation of the domain *Prop*. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation*, pages 99–110. ACM Press, 1993.

[85] Baudouin Le Charlier and Pascal Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

[86] Thomas Lindgren, Per Mildner, and Johan Bevemyr. On Taylor's scheme for unbound variables. Technical Report 116, Computing Science Department, Uppsala University, Sweden, October 1995.

[87] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.

[88] Kim Marriott and Harald Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.

[89] Kim Marriott, Harald Søndergaard, and Neil D. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[90] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[91] Nancy Mazur, Gerda Janssens, and Maurice Bruynooghe. A module based analysis for memory reuse in Mercury. In *Proceedings of the First International Conference on Computational Logic*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1255–1269. Springer, 2000.

[92] Nancy Mazur, Peter Ross, Gerda Janssens, and Maurice Bruynooghe. Practical aspects for a working compile time garbage collection system for Mercury. In *Proceedings of the Seventeenth International Conference on Logic Programming*, volume 2237 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 2001.

[93] D. McAllester. An outlook on truth maintenance. AI Memo 551, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1980.

[94] D. McAllester. Truth maintenance. In *Proceedings of the Eighth (U.S.) National Conference on Artificial Intelligence*, pages 1109–1116, 1990.

[95] Christopher Mellish. The automatic derivation of mode declarations for Prolog programs. Research Paper 163, Department of Artificial Intelligence, University of Edinburgh, 1981.

[96] Christopher Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2(1):43–66, April 1985.

[97] D. M. Miller and R. Drechsler. Dual edge operations in reduced ordered binary decision diagrams. In *Proceedings of the 1998 IEEE International Symposium on Circuits and Systems*, volume 6, pages 159–162. IEEE Press, 1998.

[98] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[99] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[100] A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the practicality of abstract equation systems. In *International Conference on Logic Programming*. MIT Press, June 1995.

[101] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[102] Lee Naish. Automatic control for logic programs. *Journal of Logic Programming*, 2(3): 167–183, November 1985.

[103] Lee Naish. *Negation and Control in Prolog*, volume 238 of *Lecture Notes in Computer Science*. Springer, 1986.

[104] Lee Naish. A declarative view of modes. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 185–199. MIT Press, 1996.

[105] Lee Naish. Higher-order logic programming. Technical Report 96/2, Department of Computer Science, The University of Melbourne, February 1996. URL `http://www.cs.mu.oz.au/publications/tr_db/TR.html`.

[106] Lee Naish. Mode checking using constrained regular trees. Technical Report 98/3, Department of Computer Science, The University of Melbourne, March 1998. URL `http://www.cs.mu.oz.au/publications/tr_db/TR.html`.

[107] Amit Narayan. Recent advances in BDD based representations for Boolean functions: A survey. In *Twelfth International Conference on VLSI Design*, pages 408–413. IEEE Press, 1999.

[108] Nicholas Nethercote. The analysis framework of HAL. Master's thesis, Department of Computer Science and Software Engineering, The University of Melbourne, April 2002.

[109] Nicholas Nethercote and Alan Mycroft. The cache behaviour of large lazy functional programs on stock hardware. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance*. ACM Press, 2002.

[110] Richard A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[111] David Overton, Zoltan Somogyi, and Peter J. Stuckey. Constraint-based mode analysis of Mercury. In Claude Kirchner, editor, *Proceedings of the Fourth International Conference on Principles and Practice of Declarative Programming*, pages 109–120. ACM Press, 2002.

[112] Rinus Plasmeijer and Marko van Eekelen. *Concurrent Clean Language Report, Version 1.3.1.* Department of Software Technology, University of Nijmegen, September 2001.

[113] A. Porto. A language for extended programming in logic. In M. van Carneghem, editor, *Proceedings of the First International Logic Programming Conference*, pages 31–37, 1982.

[114] Steve Reeves and Michael Clarke. *Logic for Computer Science.* Addison Wesley, 1990.

[115] Olivier Ridoux, Patrice Boizumault, and Frédéric Malésieux. Typed static analysis: Application to groundness analysis of Prolog and Lambda-Prolog. In *Proceedings of the International Symposium on Functional and Logic Programming*, volume 1722 of *Lecture Notes in Computer Science*, pages 267–283. Springer, 1999.

[116] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[117] Peter Ross, David Overton, and Zoltan Somogyi. Making Mercury programs tail recursive. In *Proceedings of the Ninth International Workshop on Logic-Based Program Synthesis and Transformation*, volume 1817 of *Lecture Notes in Computer Science*, pages 196–215. Springer, 2000.

[118] Peter Schachte. Efficient ROBDD operations for program analysis. In Kotagiri Ramamohanarao, editor, *Proceedings of the Ninteenth Australasian Computer Science Conference*, pages 347–356. Australian Computer Science Communications, 1996.

[119] Peter Schachte. *Precise and Efficient Static Analysis of Logic Programs.* PhD thesis, The University of Melbourne, July 1999.

[120] Tom Schrijvers and Bart Demoen. Combining an improvement to PARMA trailing with trailing analysis. In Claude Kirchner, editor, *Proceedings of the Fourth International Conference on Principles and Practice of Declarative Programming*, pages 88–98. ACM Press, 2002.

[121] Ehud Y. Shapiro. Systems programming in Concurrent Prolog. In *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages*, pages 93–105. ACM Press, 1984.

[122] Ehud Y. Shapiro. A subset of Concurrent Prolog and its interpreter. In *Concurrent Prolog: Collected Papers*, volume 1, pages 27–83. MIT Press, 1987.

[123] H. Simonis. Test generation using the constraint logic programming language CHIP. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 101–112. MIT Press, 1989.

[124] Jan-Georg Smaus. *Modes and Types in Logic Programming.* PhD thesis, University of Kent at Canterbury, December 1999.

[125] Jan-Georg Smaus, Pat M. Hill, and Andy M. King. Termination of logic programs with `block` declarations running in several modes. In C. Palamidessi, editor, *Proceedings of*

the Tenth Symposium on Programming Languages, Implementations, Logics, and Programs, volume 1490 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1998.

[126] Jan-Georg Smaus, Pat M. Hill, and Andy M. King. Preventing instantiation errors and loops for logic programs with multiple modes using `block` declarations. In P. Flener, editor, *Proceedings of the Eight International Workshop on Logic-Based Program Synthesis and Transformation*, volume 1559 of *Lecture Notes in Computer Science*, pages 289–307. Springer, 1999.

[127] Jan-Georg Smaus, Pat M. Hill, and Andy M. King. Mode analysis domains for typed logic programs. In *Proceedings of the Ninth International Workshop on Logic-Based Program Synthesis and Transformation*, volume 1817 of *Lecture Notes in Computer Science*, pages 82–101. Springer, 2000.

[128] Zoltan Somogyi. A system of precise modes for logic programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 769–787, 1987.

[129] Zoltan Somogyi. *A Parallel Logic Programming System Based on Strong and Precise Modes*. PhD thesis, The University of Melbourne, January 1989.

[130] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: An efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, 1995.

[131] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[132] Zoltan Somogyi, Fergus Henderson, Thomas Conway, and Richard O'Keefe. Logic programming for the real world. In Donald A. Smith, editor, *Proceedings of the ILPS'95 Postconference Workshop on Visions for the Future of Logic Programming*, pages 83–94, 1995.

[133] Harald Søndergaard. An application of abstract interpretation to logic programming: Occur check reduction. In B. Robinet and R. Wilhelm, editors, *Proceedings of the European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 1986.

[134] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, second edition, 1994.

[135] Jichang Tan and I-Peng Lin. Recursive modes for precise analysis of logic programs. In J. Małuszyński, editor, *Proceedings of the Fourteenth International Logic Programming Symposium*, pages 277–290. MIT Press, 1997.

[136] Andrew Taylor. *High Performance Prolog Implementation*. PhD thesis, University of Sydney, 1991.

[137] Andrew Taylor. Parma—bridging the performance GAP between imperative and logic programming. *Journal of Logic Programming*, 29(1–3):5–16, 1996.

[138] Simon Taylor. Optimization of Mercury programs. Honour's report, Department of Computer Science and Software Engineering, The University of Melbourne, 1998. URL `http://www.mercury.cs.mu.oz.au/information/papers.html`.

[139] James A. Thom and Justin Zobel. NU-Prolog reference manual. Technical Report 86/10, Department of Computer Science, The University of Melbourne, 1986.

[140] Kazunori Ueda. I/O mode analysis in concurrent logic programming. In *Theory and Practice of Parallel Programming*, volume 907 of *Lecture Notes in Computer Science*, pages 356–368. Springer, 1995.

[141] Kazunori Ueda. Experiences with strong moding in concurrent logic/constraint programming. In *Proceedings of the International Workshop on Parallel Symbolic Languages and Systems*, volume 1068 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 1996.

[142] Kazunori Ueda. Linearity analysis of concurrent logic programs. In Michael Leuschel, editor, *Proceedings of the International Workshop on Optimization and Implementation of Declarative Programs*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.

[143] Kazunori Ueda and Masao Morita. Moded flat GHC and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43, 1994.

[144] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.

[145] Pascal Van Hentenryck, Agostino Cortesi, and Baudouin Le Charlier. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, 1995.

[146] Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.

[147] Peter Van Roy and Alvin M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 25(1):54–68, January 1992.

[148] Wim Vanhoof. Binding-time analysis by constraint solving: A modular and higher-order approach for Mercury. In M. Parigot and A. Voronkov, editors, *Proceedings of the Seventh International Conference on Logic for Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 399–416. Springer, 2000.

[149] Wim Vanhoof. *Techniques for On- and Off-Line Specialisation of Logic Programs.* PhD thesis, Katholieke Universiteit Leuven, Belgium, June 2001.

[150] Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, pages 45–52, 1984.

[151] Philip Wadler. Theorems for free! In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.

[152] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[153] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, 1990.

[154] David H. D. Warren. Implementing Prolog — compiling predicate logic programs. Research Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[155] David H. D. Warren. Higher-order extensions to Prolog: Are they needed? In J. E. Hayes, Donald Michie, and Y.-H. Pao, editors, *Machine Intelligence 10*, volume 125 of *Lecture Notes in Mathematics*, pages 441–454. Ellis Horwood, 1982.

[156] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, California, USA, October 1983.

[157] Joseph L. Zachary and Katherine A. Yelick. Moded type systems to support abstraction. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 8, pages 229–243. MIT Press, 1992.

[158] Lei Zheng and Peter J. Stuckey. Improving SAT using 2SAT. In *Proceedings of the 25th Australasian Computer Science Conference*, pages 331–340, 2002.

Index