THESIS FOR THE DEGREE OF
MASTER OF COMPUTER SCIENCE BY RESEARCH

The Analysis Framework of HAL

Nicholas Nethercote

Department of Computer Science and Software Engineering
University of Melbourne, Melbourne, Australia.
September 2001 (revised April 2002).

Abstract

This thesis describes the implementation of an abstract interpretation-based generic analysis framework for the strongly typed, weakly moded, third generation constraint logic programming language HAL. The framework is domain independent, supports both top-down and bottom-up analyses, and uses a sophisticated algorithm and specialised data structures to compute analysis fixpoints efficiently.

We start by defining top-down and bottom-up semantics for the framework, and give a detailed description of its implementation, including discussion of non-obvious but important points for ensuring correctness and efficiency.

We then present top-down analyses for groundness, sharing and freeness that are implemented within the generic framework. The information they infer is used for optimising HAL programs that use Herbrand constraint solving by replacing the general unification operation with more specialised primitives. Importantly, we use mode information provided by the programmer to improve the efficiency and accuracy of such analyses. We then evaluate the cost of each analysis and the potential and actual benefits of these optimisations, concentrating particularly on the relative effect of each optimisation. The results show that the specialised primitives are up to twice as fast as general unification, and the effects of the optimisations for programs that use Herbrand constraint solving heavily are encouraging. Importantly, we also use the experimental results to identify another optimisation that will enable the use of even more highly specialised unifications — reduced to a handful of instructions — that run five to eight times faster than general unification.

We then describe HAL's determinism analysis, a bottom-up analysis implemented within the framework which aims to infer the number of solutions a procedure may have, and whether it may fail before producing its first solution. We evaluate the analysis cost, finding that it accounts for only 2–6% of compilation time, while the more specialised execution algorithms it allows speed up programs by an average of 70%. We also discuss the impact of determinism declarations on programming style, and how they promote program correctness.

Finally, we tackle the difficult problem of combining practical and accurate inter-module analysis with separate compilation. We describe in detail the compilation model used in the HAL compiler that supports context-sensitive inter-module analysis and multi-variant specialisation. The model allows an executable to be created after each module is compiled only once, but for complete analysis information each module may need to be compiled more than once. In the absence of procedure-level cyclic inter-module dependencies, this complete analysis information is as accurate as that found by a global program analysis.

Acknowledgments

I would like to thank my supervisors, Peter Stuckey and María García de la Banda, for all their help. They were always knowledgeable and friendly, helpful and interested, and generous with their time. In particular, I thank them for choosing me to work on HAL as a summer student when I didn't even realise that I wanted to study programming languages.

I wish to thank other members of the HAL team: Kim Marriott, for supervising me while Peter and María were away; and David Jeffery, for driving me out to Monash all those times, and answering all my questions about Mercury. Thanks also to Inga Sitzmann for proof-reading part of this thesis.

I also give heartfelt thanks and love to my family for the unquestioning support they have given me this year and every other.

Finally, I would like to acknowledge the Commonwealth of Australia for the financial support which made this work possible.

Contents

List of Figures

List of Tables

Preface

This thesis comprises eight chapters, including an introduction and a conclusion. Following the introduction, Chapter 2 provides sufficient background on constraint logic programming and HAL to understand the rest of the thesis. Chapters 3 and 4 describe the theory and implementation of the analysis framework of HAL. Chapters 5 and 6 describe and evaluate various analysis domains implemented within this framework. Chapter 7 describes a compilation model that supports accurate inter-module analysis and optimisation. Chapter 8 concludes.

This thesis is derived entirely from my own research. None of the material in this thesis has been previously published.

I certify that:

(i) this thesis comprises only my original work;

(ii) due acknowledgment has been made in the text to all other material used;

(iii) the thesis is approximately 30,000 words in length, exclusive of tables, maps, bibliographies, appendices and footnotes.

Chapter 1

Introduction

Compilers for modern declarative programming languages typically perform multiple phases of static analysis. These analyses can be broadly categorised as having one of two purposes.

- **Analyses for correctness:** these are used to check programmer declarations and assertions, and to infer unspecified program properties. Perhaps the best known example is type analysis.

  The primary purpose of such analyses is to improve program correctness. In some cases, the information about program properties may also be used to improve program efficiency, for example by allowing the use of a more efficient execution mechanism.

- **Analyses for optimisation:** these are used to predict safe and computable approximations of values and behaviours arising dynamically during program execution, in order to avoid redundant and superfluous computations.

  The primary purpose of such analyses is to improve program efficiency.

The benefits of improving program correctness and efficiency are clear. High-level declarative languages such as constraint programming languages are particularly amenable to analysis, due to their relatively clean semantics. Being high-level, they also have a lot to gain from performance optimisations.

One common approach for analysing constraint logic programs is to provide a generic analysis framework onto which multiple analyses can be "hooked". This makes it simple to add new analyses. It also means that any improvement to the framework will benefit multiple analyses.

The goal of this thesis is to comprehensively describe such an analysis framework for the constraint logic programming language HAL. In doing so, we aim to provide the following.

- A clear justification of the form of the framework.
- A sound theoretical basis for the framework.
- A comprehensive description of the implementation of the framework.

- Detailed descriptions of analyses implemented within the framework, and any enabled optimisations.
- Empirical evaluation of the costs and benefits of these analyses.

To satisfy this goal, we must consider potentially awkward details that are often overlooked in descriptions of program analyses, but are important in a real implementation.

- Analysis of full programs, without any restrictions on language features used.
- The use of higher-order programming.
- Practical and accurate inter-module analysis.
- Appropriate treatment of library modules.
- The use of programmer declarations to improve the accuracy and efficiency of analysis.
- Efficiency considerations, to minimise the cost of analysis.

Finally, this thesis is intended to focus on techniques that have been implemented and tested; anything not implemented should be clearly identified, and its potential for future inclusion within the compiler should be clearly stated.

With this goal clearly in mind, let us describe the organisation of this thesis. Chapter 2 provides a brief introduction to and history of constraint logic programming, and introduces HAL. Chapter 3 outlines the reasons for choosing HAL's particular analysis framework, and also provides its theoretical basis. Chapter 4 describes the actual implementation of the analysis framework, including detailed pseudocode of the main algorithm. Chapter 5 describes the implementation within the framework of groundness, sharing and freeness analyses, three traditional top-down (or goal-dependent) analyses for optimisation; Chapter 6 describes the implementation within the framework of a determinism analysis, a non-traditional bottom-up (or goal independent)[1] analysis for correctness. Both chapters contain evaluations of the cost of the analyses, and their benefits. Chapter 7 describes and evaluates the compilation model used in the HAL compiler that allows accurate, context-sensitive analysis information to be gathered across module boundaries. Chapter 8 concludes by considering how well our goals have been achieved, highlighting our contributions and identifying directions for future research.

[1] The terms *top-down* and *goal-dependent* will be used interchangeably in this thesis, as will the terms *bottom-up* and *goal-independent*.

Chapter 2

A New CLP Language

Constraint logic programming (CLP) languages are designed to solve a broad range of difficult problems that involve constraints, such as combinatorial optimisation and constraint satisfaction problems. Many different techniques can be used to solve these problems, but there is no single best technique — different techniques are suitable for different problems.

This chapter provides an introduction to constraint logic programming, briefly describes its history to show how the paradigm has evolved to provide programmers with greater control over the techniques used to solve their problems, and culminates with an introduction to HAL, a third generation CLP language designed specifically to facilitate easy experimentation with different constraint solving techniques. This description of HAL will serve as a sufficient background for the rest of this thesis.

2.1 Constraint Logic Programming

Constraint logic programming (CLP) is a relatively recent paradigm that combines the constraint solving facilities of constraint programming with the powerful and flexible search capabilities of logic programming. Here we introduce the basics of CLP.

Let us first consider the building blocks of constraint logic programs. A *variable* is a place-holder for a value, and is represented by a string of alphanumeric characters beginning with an upper case letter. A *constructor* is a string of characters beginning with a lowercase letter. A *constant* is a constructor or a value of some primitive type from a constraint domain, such as an integer. A *term* is either a variable, a constant, or a *compound term* — a *constructor* applied to an ordered list of one or more terms, written $f(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms. Constructors are often called *functors*. The *arity* of a term is its number of arguments. The functor f and arity n of a term is often written f/n.

All variables are initially unconstrained, but may become progressively more constrained. Unconstrained variables of term types are *free*, and may become *bound* to a term. A variable whose value is constrained to have a unique fixed value is *ground*.

Given a constraint domain $\mathcal{D}$, a *primitive constraint* is a constraint relation symbol from $\mathcal{D}$ used with the appropriate number of arguments. For example, $X \leq 0$ is a primitive constraint from the domain $\mathcal{R}$ (real numbers). A *user-defined constraint* has the form

$p(t_1, \ldots, t_n)$ where p is an n-ary predicate and $t_1, \ldots, t_n$ are expressions from the constraint domain. User-defined constraints are also known as *atoms*. A *literal* is either a primitive constraint or a user-defined constraint. A *goal* is a sequence of literals, $L_1, \ldots, L_n$. A *rule* has the form $H \leftarrow B$ where the head H is a user-defined constraint and B is a *goal*. A *predicate* consists of multiple rules each having a head with the same functor and arity. A program is a collection of predicates.

A *renaming* is a bijective mapping from variables to variables. The result of applying a renaming σ to a syntactic object s, written $\sigma(s)$, is that object s with each variable x replaced by the variable $\sigma(x)$. The inverse of a renaming σ is written σ^{-1}.

Programs are executed in response to a starting goal, called a *query*. Execution is based around the process of *derivation*. At each step in the derivation a literal is processed (usually the left-most one). If the literal is a user-defined constraint, it is replaced by the body of one of the predicate's rules (typically the first applicable one), with variables appropriately renamed. If a literal is a primitive constraint it is added to the *constraint store*. If we can process all the literals in a predicate, that predicate *succeeds*. If the constraint store becomes unsatisfiable at some point, the particular derivation has *failed*, but we may *backtrack* to previous rule choices and explore alternative execution paths. If no other paths succeed, the predicate *fails*. We may also backtrack after succeeding to search for further solutions. Thus the execution of a program is simply the exploration of a derivation tree, in which each node represents a failed or a successful derivation.

2.2 First Generation CLP Languages

First generation constraint languages were characterised by the *constraint logic programming scheme* introduced by Jaffar and Lassez in [30], described in [42]:

> "This *constraint logic programming scheme* defines a family of languages. Given a particular constraint domain, constraint solver and constraint simplifier, the scheme defines a language for writing programs and a mechanism for evaluating goals and programs written in this language."

For example, the language *CLP*(*Tree*) solves problems over the *Tree* (or Herbrand) domain.[1] Equality and disequality are the only primitive constraints provided for trees. Tree constraints are present in almost all CLP languages, as they allow complex data structures to be used.

More typical first generation CLP languages combined the *Tree* domain with a second constraint domain. The language CLP($\mathcal{R}$) [31, 32, 20] combines the *Tree* and $\mathcal{R}$ (real) domains; elements of the domain are trees that may have real numbers as leaves. Real (floating-point) numbers can be operated on using arithmetic and trigonometric functions, and can be constrained by equality, inequality, and (in some implementations) disequality relations. Similarly, the language *CLP*(*FD*) ([14] describes one implementation) combines tree and finite domain constraints. Finite domain variables may be constrained by equality,

[1] This is the pure component of the logic programming language Prolog.

inequality and disequality constraints; implementations typically also provide more complex built-in constraints useful for particular problems such as job-shop scheduling.

A classic example of constraint logic programming is provided by the CLP($\mathcal{R}$) program `mortgage` that models a bank loan (this version from [42]). It has five parameters: the principal, or amount still owed (`P`), the number of time periods in the mortgage (`T`), the interest rate (`I`), the repayment per time period (`R`), and the final balance owing (`B`).

```
mortgage(P, T, I, R, B) :-
    T = 0.0, B = P.
mortgage(P, T, I, R, B) :-
    T >= 1.0,
    mortgage(P + P*I - R, T - 1.0, I, R, B).
```

The first rule states that when the loan completes (`T = 0.0`), the balance remaining is equal to the amount still owed. The second rule uses recursion to define the mortgage in terms of itself, one time period hence; the amount owed will increase due to interest on the principal and decrease due to repayment.

Two advantages of CLP are demonstrated by `mortgage`: it is extremely concise, and it can be run in multiple *modes of usage* — many different questions can be answered by varying the instantiation of the five parameters. For example, this straightforward query of the CLP($\mathcal{R}$) system,

```
?- mortgage(10000.0, 10.0, 0.1, 1000.0, B).
```

asks "for a loan of $10,000, with an interest rate of 10% and repayments of $1,000 per time period, what will the balance be after ten periods?" The system's answer is:

```
B = 10000
```

Another query,

```
?- mortgage(P, 10.0, 0.1, 1000.0, 0.0).
```

asks "for a loan of ten periods, with an interest rate of 10% and repayments of $1,000 per period, how much can be borrowed so that the final balance will be $0?" The answer is:

```
P = 6144.57
```

A more complex query,

```
?- mortgage(P, 10.0, 0.1, R, B),
```

asks "for a loan of ten periods, with an interest rate of 10%, what is the relationship between the principle, the repayment, and the final balance?" The answer is:

```
P = 0.385543*B + 6.14457*R
```

By contrast, a typical imperative version of `mortgage` would be considerably longer, and would only be able to answer one type of query, such as the first one above which calculates `B`.

2.3 Second Generation CLP Languages

First generation CLP languages lacked flexibility — their solvers were non-extensible "black boxes" over which programmers had no control. Because different problems are suited to different constraint solving techniques, this was a significant drawback.

This limitation was gradually lessened as CLP languages evolved to provide support for problem-specific constraint solving. At first they provided "glass box" solvers which the programmer could extend to include problem-specific constraints (e.g. [14]). More recent versions of CLP languages such as ECLiPSe [64], SICStus Prolog [58] and clp(Q,R) [26] also support the addition of new constraint solvers by providing features such as dynamic scheduling [41], constraint handling rules [16] and attributed variables [25]. In a similar vein, toolkits such as ILOG SOLVER [53], designed for solving finite domain problems in C++, allow the programmer to define new primitive constraints by extending the solver and search mechanism.

However, few if any compilers for these languages perform analysis of programmer extensions, so they cannot be optimised, which may penalise their use.

2.4 HAL, a Third Generation CLP Language

HAL [12] is a strongly typed, weakly moded CLP language designed to allow easy experimentation with different constraint solvers, to support extension of solvers and construction of hybrid solvers, and to call solvers written in other languages such as C with little overhead.

The basic HAL syntax follows that of standard CLP languages, with variables, rules and predicates defined as usual. Beyond that, HAL also supports functions, and has a variety of declarations allowing the programmer to provide extra information about programs. For example, predicates and functions have semi-optional type, mode and determinism declarations.[2] Information from these declarations assists program correctness and allows the generation of fast target code and efficient integration with foreign languages. The declarations are very similar to those of the logic programming language Mercury [56], which HAL uses as a target language.

2.4.1 An Example HAL Program: `mortgage`

HAL is best introduced by an example. Consider the HAL version of `mortgage`, from [12].

```
:- module mortgage.                                                    (L1)
:- import simplex.                                                     (L2)
:- export pred mortgage(cfloat, cfloat, cfloat, cfloat, cfloat).       (L3)
:-        mode mortgage(in, in, in, in, out) is nondet.                (L4)
:-        mode mortgage(oo, oo, oo, oo, oo) is nondet.                 (L5)
mortgage(P, T, I, R, B) :-                                             (L6)
     T = 0.0, B = P.                                                   (L7)
```

[2] By "semi-optional" we mean they are compulsory for predicates exported from a module. Section 6.1 explains the reasoning behind this.

```
mortgage(P, T, I, R, B) :-                                    (L8)
    T >= 1.0,                                                 (L9)
    mortgage(P + P*I - R, T - 1.0, I, R, B).                  (L10)
```

Module name and imports: A module is defined within a file, exports items by annotating them with an `export` declaration (the `export_abstract` declaration exports items abstractly), and imports all the exported items from another module using the `import` declaration (selective import is also possible). HAL source file names must end in the suffix "`.hal`".

Line (L1) declares the module name. Line (L2) imports a library module `simplex`, which provides a simplex-based linear arithmetic constraint solver for constrained floating point numbers, called `cfloat`s.

Types: Every term in a program has a *type*, which determines its internal representation. The core language provides integer, float, character and string types, plus polymorphic constructor types (such as lists) based on these types. It only supports basic operations such as assignment, tests for equality, and construction and deconstruction of ground terms. If more sophisticated constraint solving is required for a type, a constraint solver for that type must be used.

Types are created using polymorphic regular tree type statements. Type renamings are also allowed. For example:

```
:- typedef list(T) -> ( [] ; [T | list(T)] ).
:- typedef tree(T) -> ( leaf ; branch(tree(T), T, tree(T)) ).
:- typedef int_list = list(int).
```

The first definition defines the polymorphic `list/1` constructor type, where the constant `[]` represents the empty list, `[Head | Tail]` is the usual syntactic sugar for the non-empty list `.(Head, Tail)`, and `T` is a *type variable*. The definition of the `tree/1` type shows the standard term syntax for user-defined types. The third definition is a monomorphic variant of `list/1` — a list of integers.

Overloading of predicates and functions is allowed, as long as identically named predicates and functions are defined in different modules. This allows the programmer to overload the standard arithmetic operations and relations (including equality), resulting in a natural syntax for constraints across multiple solvers. The type-checking algorithm that allows this overloading is described in [13].

Line (L3) of the example program exports the `mortgage` predicate, and also provides its *type declaration*, which prescribes the type of its arguments — all five are `cfloat`s, which are represented differently to built-in `float`s, and support full constraint solving.

Type classes: Type classes [63] provide *ad-hoc* or *constrained* polymorphism, which allow programmers to write code that can operate on any type that implements certain *methods*. Such a type is called an *instance* of the type class. Type classes provide a natural way of specifying a solver's capabilities, allowing a well-defined interface for solvers. HAL's type

class system is similar to Mercury's [34], the major difference being that HAL has a built-in class hierarchy characterising different solvers, ranging from the most basic class `eq` with a single equality method `=/2`, through to complex solver classes such as `float_solver`, which has methods for addition, subtraction, multiplication and comparison of constrained floats. Instances of some built-in classes can be automatically generated for types by using a "`deriving`" annotation. See [17] for details of the built-in hierarchy.

Type classes are not directly evident in `mortgage`, but the `cfloat` type imported from the `simplex` module would be an instance of the built-in `float_solver` class. The presence of this type class could be made explicit by generalising `mortgage`'s type declaration to this:

```
:- export pred mortgage(T, T, T, T, T) <= float_solver(T).
```

The "`<= float_solver(T)`" annotation is a *type class constraint* which states that `mortgage` may be passed arguments of any type that is an instance of the class `float_solver/1`. This demonstrates the ease of experimentation with different solvers in HAL — the solver used by a predicate can be changed very easily.

Instantiations: At each point in the execution of a program, every term in scope has an *instantiation*. The base instantiations are `new`, `old` and `ground`. A variable is `new` if it has not been seen by a constraint solver, `old` if it has, and `ground` if it has a fixed value. For types that have no solver, such as the built-in types, `old` is equivalent to `ground`.

More complex instantiations, lying between `old` and `ground`, can be defined. Polymorphic instantiation definitions are allowed, as are instantiation renamings. For example:

```
:- instdef list(I)    -> ( [] ; [I | list(I)] ).
:- instdef ground_list = list(ground).
```

These definitions define a fixed-length list whose elements all have the same instantiation, and a ground list.

Modes: Each argument of a predicate has a *mode*, i.e. a mapping $Inst_{in} \rightarrow Inst_{out}$ describing its input and (if the predicate succeeds) output instantiations.

The base modes are constructed from the base instantiations. We use two letter abbreviations (`nn`, `no`, `ng`, `oo`, `og`, `gg`) using the first letter of each instantiation (e.g. `ng` represents `new` $\rightarrow$ `ground`). The frequently used modes `in` and `out` are renamings of `gg` and `ng` respectively.

Mode definitions are created from existing instantiations. Polymorphic mode definitions and mode renamings are allowed. For example:

```
:- modedef ng      -> (new -> ground).
:- modedef same(I) -> (I -> I).
:- modedef in       = ng.
```

These definitions define the base mode `ng`; a mode that leaves an argument's instantiation unchanged; and the renaming `in`.

Lines (L4) and (L5) of `mortgage` are *mode declarations*, which define the modes of usage with which a predicate may be used. We call each mode of usage a *procedure*; for efficiency, separate code is generated for each procedure.

The first mode declaration uses the modes `in` and `out`; for this procedure to be used, when called the first four arguments must be `ground` and the last argument must be `new`. When the predicate returns all arguments must be `ground`. In the second mode all five arguments are `oo`, which means they must be "constrained" (`old`) upon predicate entry and exit. This more flexible procedure allows arbitrary uses of the predicate, like the CLP($\mathcal{R}$) version, but will execute less efficiently than the first procedure which is specialised for one particular kind of query.

It is worth noting that HAL's mode system is *prescriptive* rather than *descriptive* — the HAL compiler will reject a program if it cannot prove it satisfies its mode declarations.

Determinism declarations: Each procedure has a *determinism*, which describes how many solutions it may have. The six values used in HAL are the same as those used in Mercury: `det` means exactly one solution, `semidet` means zero or one solution, `multi` means one or more solutions, `nondet` means any number of solutions, `failure` means zero solutions, and `erroneous` means a run-time abort, exception, or infinite loop.

A *determinism declaration* can accompany each mode declaration. Both procedures of `mortgage` are `nondet`, meaning they may return any number of solutions.[3]

HAL's determinism system is described in detail in Chapter 6.

Predicate body: Lines (L6)–(L10) define the two rules for `mortgage`. They are identical to the rules in the CLP($\mathcal{R}$) version given in Section 2.2.

2.4.2 Further Language Features

HAL has several other features not used by `mortgage` to facilitate easy experimentation with different constraint solvers.

Higher-order programming: Procedures are first-class values, and higher-order terms can be created, passed around, and called using the `call/n` built-in.

In the following example, `map/3` is used to define a predicate that prepends a string to every element in a list of strings (the underscore '`_`' represents an *anonymous variable* which is not used). It uses the predicate `append_strings/3` (from the string library) to perform the prepending.

```
:- pred prepend_all(string, list(string), list(string)) is det.      (L1)
:- mode prepend_all(in, in, out) is det.                             (L2)
prepend_all(Str, List, NList) :-                                     (L3)
    P = append_strings(Str),                                         (L4)
    map(List, P, NList).                                             (L5)
```

[3] The first mode cannot actually succeed more than once, however the HAL compiler cannot determine this. See Section 6.7.3 for an explanation and work-around.

```
:- pred map(list(X), pred(X, Y), list(Y)).                    (L6)
:- mode map(in, pred(in, out) is det, out) is det.            (L7)
map([], _, []).                                               (L8)
map([X | Xs], Pred, [Y | Ys]) :-                              (L9)
    call(Pred, X, Y),                                         (L10)
    map(Xs, Pred, Ys).                                        (L11)
```

The higher-order unification in line (L4) creates a higher-order term, or *closure*, by a partial application of `append_strings/3`, which *captures* the variable `Str`. This closure is now treated like a procedure of arity two, and can be passed to `map/3`. When the closure is applied in line (L10) the arguments `X` and `Y` are added to the already captured arguments, and the closure executes equivalently to the first-order call `append_strings(Str, X, Y)`.

Herbrand constraint solving: As mentioned, the core language only supports basic operations such as assignment, tests for equality, and the construction and deconstruction of ground terms. However, HAL also provides a Herbrand constraint solver for each term type that is an instance of the built-in `herbrand` type class.[4] To use the Herbrand solver for a type within a module requires a `herbrand` declaration such as:

```
:- herbrand list/1.
```

The Herbrand solver supports full unification and, thus, the use of true logic variables and support for logic programming idioms such as difference lists. Herbrand constraint solving is discussed in more detail in Chapter 5.

Constraint handling rules: Constraint handling rules (CHRs) [16] are a flexible mechanism for writing incremental solvers. They consist of multi-headed guarded rules that repeatedly rewrite a constraint store until it is in a solved state. CHRs typically "sit on top" of the host language; HAL CHRs may contain HAL code, and are compiled into HAL code at an early stage of compilation. HAL CHRs are described in detail in [27].

Dynamic scheduling: HAL provides a *delay* construct for performing dynamic scheduling [41], allowing goals to be delayed until a condition is satisfied. Dynamic scheduling is useful for writing solvers, and for extending and combining existing solvers.

Global variables: Restricted "global variables", only visible within a module, allow efficient implementation of persistent constraint stores. They are not intended for general use, but only for use within solvers. They behave similarly to C's `static` variables by "remembering" their values between predicate calls. They are available in backtrackable and non-backtrackable flavours. HAL does not support the standard Prolog predicates `assert` and `retract`, but global variables can be used as a replacement for some of their typical uses.

[4] See [17] for details of the `herbrand` type class. Constructor types can be made instances of `herbrand` simply by annotating their type declaration with "`deriving herbrand`".

2.4.3 Current Implementation

The HAL compiler is written in the intersection of HAL and SICStus Prolog [58]; it has not bootstrapped yet, so it is currently executed using the SICStus Prolog compiler.

The HAL compiler consists of approximately 34,000 lines of code (excluding blanks and comments). Most of it is written in HAL, including the analysis framework, which comprises just over 3,000 lines, and the operations for the four analysis domains examined (groundness, sharing, freeness and determinism) which make up about 2,500 lines. The HAL standard library contains approximately 9,000 lines of code.

The generated Mercury files are compiled by the Melbourne Mercury Compiler. The only special requirement of the Mercury compiler is that the "reserved tag" compile grade [21] must be used in order to support Herbrand constraint solving. This is explained in Section 5.2.2.

Chapter 3

Program Analysis

In this chapter, we survey the analyses for correctness required for checking HAL programs, and some analyses which could be used to optimise generated code. From the characteristics of these analyses, we justify the chosen form of the analysis framework. We then describe the representation of programs used within the HAL compiler, and some preliminary transformations that convert a program into this representation before the analysis framework is used. Finally, we introduce the basics of abstract interpretation and formally define several semantics and uses of the framework.

3.1 Analysis in HAL: An Overview

When considering analyses in HAL, it is important to remember that HAL uses Mercury as a target language. Mercury was chosen as HAL's target language to take advantage of the many optimisations the Melbourne Mercury Compiler performs to generate efficient code. To avoid repeating work, the analyses performed by HAL should overlap with those performed by Mercury as little as possible. With this fact in mind, let us survey the relevant analyses for correctness and optimisation in HAL.

3.1.1 Analyses for Correctness

The HAL compiler must reject any program that does not safely satisfy its type, mode and determinism declarations.[1] These three kinds of declarations capture a large number of common program errors, reducing program development time, and improving correctness. They also allow more specialised execution algorithms to be used, improving performance.[2]

The HAL compiler must perform type, mode and determinism analyses — even though the Mercury compiler also performs these analyses — for two reasons. Firstly, for useability's sake the HAL compiler must perform these analyses to allow informative error messages; it is not reasonable to require a programmer to inspect a machine-generated Mercury file in order

[1] Some "safe" violations of these declarations are allowed, but cause warnings to be issued.

[2] See [11] for figures on the performance improvements attributable to type and mode declarations, and Section 6.7.2 for figures on the performance improvements due to determinism declarations.

to understand an error message, and then determine how this error in the machine-generated file corresponds to that of the original program![3]

Secondly, there are significant differences between the analyses required by the two languages. For example, HAL's type analysis contains a phase for inferring class constraints for the built-in type classes `eq/1` and `solver/1` [17]; this phase has no corresponding phase in Mercury. HAL's type analysis also uses a different algorithm for performing type inference, which translates a HAL program with type definitions into a logic program from which predicate and variable types can be inferred, using efficient constraint solving methods [13]. This method is more efficient than that currently used in the Mercury compiler; there are plans to change the Mercury compiler to use the same technique.

Mode analysis is also quite different in HAL and Mercury; HAL allows `old` variables, whereas Mercury does not; constraint solvers introduce further significant differences. Mode analysis is a quite complex operation in which the compiler performs multi-variant specialisation, generating separate code for each procedure of a predicate. See [19] for a description of the algorithm used.

Determinism analysis is the simplest of the three analyses for correctness, being a bottom-up analysis. It is the analysis for correctness that is most similar to Mercury's [22].

3.1.2 Analyses for Optimisation

The Mercury compiler performs a large number of optimisations, both high-level (e.g. inlining, higher-order and type specialisation, and deforestation [61]) and low-level (e.g. early discarding of `nondet` frames, jump-to-jump short-circuiting, tail call optimisations, etc. [56, 54]). Because the Mercury compiler does a very good job with these, the optimisations we concentrate on at the HAL level are those involving language features not supported by Mercury.

One prime candidate is Herbrand constraint solving — in particular the unification of terms involving `old` instantiations.[4] If certain properties are known about the terms being unified, unification can be simplified. For this purpose, several well-known analyses used in Prolog compilers are applicable, such as those inferring groundness [10, 40], sharing [28, 57] and freeness [47, 8] information. A typical approach to such analyses is to provide a general top-down abstract interpretation-based analysis framework into which different analysis domains can be easily "hooked" by providing certain basic domain operations. These analyses all fit naturally within such a framework.

Some solver-specific optimisations can also significantly improve the speed of programs that use certain constraints heavily. For example, information from the domain *LSign* [43] can be used to optimise programs using linear arithmetic constraints. The analyses required for these optimisations typically also fit within generic top-down analysis frameworks. Also,

[3] A better alternative is to avoid having the Mercury compiler perform these analyses. Since a compiled HAL program is known to have correct types, modes and determinisms, they need not be checked by the Mercury compiler. A long-term aim for HAL is to bypass these analyses by passing the appropriate data structures directly to the "middle" of the Mercury compiler. This would speed up compilation times considerably.

[4] Mercury does not support Herbrand constraint solving, as it only provides a limited form of unification.

because HAL is designed for writing new solvers, it is quite feasible that new solvers will benefit from new analyses over different abstract domains. The ability to "plug in" a new analysis domain easily is therefore very useful.

3.1.3 Analyses Chosen for HAL

Having examined the required and desired analysis phases for the HAL compiler, we can draw the following conclusions.

- Type analysis must be done first, as it selects the predicate used by every call in the program, resolving any overloading. It uses a highly specific algorithm to do this.
- Mode analysis must be done second, because it specialises each predicate as one or more procedures. It too uses a highly specific algorithm.
- Determinism analysis can be performed any time after mode analysis. It is less specialised, being a fairly straightforward bottom-up analysis.
- Analyses for optimisation can also be performed any time after mode analysis. Analyses for optimising built-in language operations such as unification can be performed within a generic top-down framework. Solver-specific analyses can also be easily added to such a framework.

The decision to implement a generic top-down analysis framework for the HAL compiler was decided by the final point. A framework supporting goal-dependent analysis was chosen; broadly speaking, goal-dependent analysis is generally more accurate than goal-independent analysis.[5] One recognised problem with goal-dependent analysis is the difficulty of handling context-sensitive calls to external modules and libraries. For example, Bagnara *et al.* state in [1] that this "is one of the reasons why focusing only on goal-dependent analyses is, in our opinion, a mistake." To counter this problem, the HAL compiler implements a new approach to inter-module analysis, first described in [6]. The implementation is described in Chapter 7.

Since type and mode analysis use highly specific algorithms, they are not suitable for fitting within an analysis framework. More generally, mode analysis cannot be performed within the chosen generic analysis framework since it reorders literals, and the framework relies on a fixed literal order. Type analysis also cannot be performed within the framework for the same reason since it must be performed before mode analysis to determine which procedures are available. This leaves only determinism analysis. One useful feature of the chosen framework is that it can accommodate bottom-up analysis if certain parametric operations are defined appropriately. To avoid the effort of implementing a dedicated

[5] Although this is not always true; [18] shows that in general the two approaches are incomparable, and states that "the goal-dependent approach can be more accurate. However, the goal-independent based approach may also be more accurate, and, in a certain sense, is inherently more accurate" (page 15). Despite this, the empirical results given indicate that in practice goal-dependent approaches are generally more accurate than the goal-independent approach.

determinism analysis phase or a separate bottom-up analysis framework, the top-down analysis framework is also used to perform determinism analysis. Chapter 6 shows how this is achieved.

3.2 Program Representation

This section compares possible structures of constraint logic programs, contrasting the restricted canonical form typically used when describing analyses with the more expressive *super-homogeneous* form used for HAL programs. It also describes the internal representation of programs in super-homogeneous form used within the HAL compiler.

3.2.1 Canonical Form of Constraint Logic Programs

Algorithms and analyses that operate on constraint logic programs are often defined only on programs containing rules that are in a restricted *canonical form*, to simplify the analysis.

Such programs are expressed as multiple predicates, each containing one or more rules. Each rule is a single conjunction of literals; multiple rules form an implicit disjunction. If-then-elses typically aren't covered.[6] Unifications are typically broken up into multiple simpler *unification constraints*, and procedure heads and user-defined constraints are *normalised* so that their parameters are distinct. Higher-order programming is usually not considered.

3.2.2 Super-homogeneous Form of HAL Programs

The HAL compiler performs a source-to-source transformation that converts programs into *super-homogeneous* form [56] before type analysis. This form is similar to canonical form, but is more expressive and closer to the original code written by the programmer. The steps of the conversion are as follows.

- Functions are converted to predicates, and nested function applications are flattened into multiple predicate calls.[7]
- Rule heads and procedure calls are normalised so all arguments are distinct variables (this means no implicit unifications take place in a rule head).
- Complex unifications are flattened by introducing intermediate variables and breaking them into several simpler unifications having one of two forms.
  1. $X = Y$, where both X and Y are variables.
  2. $X = f(Y_1, \ldots, Y_n)$, where $n \geq 0$ and $X, Y_1, \ldots, Y_n$ are distinct variables; f may be a predicate symbol, in which case $f(Y_1, \ldots, Y_n)$ is a higher-order term.
- Head variables of rules are renamed to be the same across all rules of a predicate.

[6] Most analysers treat an if-then-else `( I -> T ; E )` as a disjunction `( I, T ; E )`. This might lead to a loss of accuracy.

[7] Because of this, we will only consider predicates from this point onwards.

- Multiple rules are combined into an explicit disjunction.

For example, the predicate `append/3` is typically written like this:

```
append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :-
    append(Xs, Ys, Zs).
```

After conversion to super-homogeneous form, it looks like this:

```
append(Xs, Ys, Zs) :-
    ( Xs = [],
      Ys = Zs
    ;
      Xs = [X | Xs1],
      Zs = [X | Zs1],
      append(Xs1, Ys, Zs1)
    ).
```

3.2.3 Internal Representation of HAL Programs

Once mode analysis is complete, the program is in a form suitable for the analysis framework. Each procedure contains a head and a single body, and all the complex language features such as type classes, delay constructs, and CHRs have been converted to calls to compiler-generated procedures. These are represented internally the same as user-defined procedures, so the analysis framework does not need to treat them specially.

The four kinds of body remaining are as follows.

- Conjunctions: *and*(list(*body*)).
- Disjunctions: *or*(list(*body*)).
- If-then-elses: *if_then_else*(*body*, *body*, *body*).
- Literals: *literal*(*literal_kind*).

A conjunction contains at least two conjuncts. A disjunction contains at least two disjuncts. An if-then-else contains three branches. Different kinds of bodies can be arbitrarily nested, just as they were in the original program. Conjunctions, disjunctions and if-then-elses will be referred to as *compound bodies*, or just compounds. Other information is also stored in each body by this stage, such as its local variables, and the instantiation of each variable at the program point immediately after the body. For simplicity this is omitted from this presentation.

The three kinds of literal are as follows.

- Unifications: *unify*(*var*, *term*), where *term* is either a variable or non-variable term of the form $f(Y_1, \ldots, Y_n)$ and $Y_1, \ldots, Y_n$ are distinct variables.

- Higher order unifications: *unifyHO*(*var*, *ho_term*), where *ho_term* is a closure of the form $h(Y_1, \ldots, Y_n)$ and $Y_1, \ldots, Y_n$ are distinct variables.

- Procedure calls: *call*(*atom*), where *atom* has the form $p(Y_1, \ldots, Y_n)$ and $Y_1, \ldots, Y_n$ are distinct variables.

Like bodies, literals also have other information associated with them, but again for simplicity this information is omitted.

For example, the internal representation of the body of the super-homogeneous form of `append/3` given above is:

or([*and*([*literal*(*unify*(`Xs`, `[]`)),
literal(*unify*(`Ys`, `Zs`))]),
and([*literal*(*unify*(`Xs`, `[X | Xs1]`)),
literal(*unify*(`Zs`, `[X | Zs1]`)),
literal(*call*(`append(Xs1, Ys, Zs1)`))])]).

After mode analysis, the recursive call to the predicate `append` will be replaced by a call to a specific procedure, and some of the literals may be reordered.

3.3 Theoretical Foundations

This section provides the theoretical basis for the analysis framework. It introduces abstract interpretation — the formalism used to prove correctness of the analysis framework — and describes the semantics used by top-down and bottom-up analyses.

3.3.1 Abstract Interpretation

Dataflow analysis is the process of statically inferring properties of variables, data structures, and program fragments. A standard approach is to approximate the operations and data used in the *concrete* execution of a program with abstract operations and *descriptions* of data used in an *abstract* execution.

Since control may potentially pass through certain program points any number of times during program execution, we cannot hope to achieve any kind of "complete" approximation, as the analysis would not be computable in general. However, if we settle for cruder descriptions, we can perform analyses that are practical, yet accurate enough to be useful. The general goal of the abstract execution is to annotate every program point with a description approximating the values that data items may take at that point during the concrete execution. In some cases, not every program point needs to be annotated.

For example, let us consider a simple groundness domain in which a description is a set of all the variables in scope that are known to be ground,[8] and the following HAL program (adapted from [18]), where each program point is annotated with a number ⓧ.

[8] A more sophisticated groundness domain is described in Section 5.3.

```
:- typedef nat -> ( zero ; s(nat) ) deriving herbrand.
:- herbrand nat.
:- pred le(nat, nat).
:- mode le(oo, oo) is nondet.
le(X, Y) :-
    ( ① X = Y ②
    ;
      ③ Y = s(Z), ④ le(X, Z) ⑤
    ). ⑥
```

The use of the `deriving herbrand` and `herbrand` declarations and the `oo` modes means the arguments of `le/2` can have arbitrary instantiations. Let us assume that this predicate is called from a context such that the second argument `Y` is known to be ground. Our analyser should give the following descriptions of the ground variables at each program point:

① = {Y},
② = {X, Y},
③ = {Y},
④ = {Y, Z},
⑤ = {X, Y, Z},
⑥ = {X, Y}.

Note that the descriptions 1, 2 and 6 are in terms of the variables {X, Y}, whereas descriptions 3–5 are in terms of the variables {X, Y, Z}, since Z is local to the second disjunct.[9]

This program point information tells us that if the second argument of a call to `le/2` is known to be ground then the second argument for all subsequent calls will also be ground, and that all answers for such calls to `le/2` will have both arguments ground.

The correctness of this kind of dataflow analysis is normally formalised in terms of *abstract interpretation* [9]. The concrete domain ($\mathcal{C}$) is *approximated* by descriptions in the description domain ($\mathcal{D}$), both of which are usually complete lattices related by a pair of functions:

$\alpha : \mathcal{C} \to \mathcal{D}$, the abstraction function;

$\gamma : \mathcal{D} \to \mathcal{C}$, the concretisation, or "semantic function" for approximations.

The functions α and γ should be monotonic and form a *Galois connection*, i.e. for all $D \in \mathcal{D}$ and for all $C \in \mathcal{C}$:

$$\alpha(C) \leq_{\mathcal{D}} D \iff C \leq_{\mathcal{C}} \gamma(D),$$

where $\leq_{\mathcal{C}}$ and $\leq_{\mathcal{D}}$ are partial orderings on $\mathcal{C}$ and $\mathcal{D}$ respectively.

The idea of approximation is made precise as follows:

$$D \in \mathcal{D} \text{ approximates } C \in \mathcal{C} \text{ iff } C \leq_{\mathcal{C}} \gamma(D).$$

[9] The treatment of local variables during analysis is made precise in the following section.

This is abbreviated as $D \propto C$.

The top (least precise) and bottom (most precise) elements of a lattice are denoted $\top$ and $\bot$ respectively.

Finally, let us call an element of $\mathcal{P}(Con)$ a *constraint description*, where $\mathcal{P}$ is the powerset function and Con is the set of all constraints. A domain $\mathcal{D}$ of constraint descriptions is *downwards closed* if for all $D \in \mathcal{D}$ and $e \in Con$, $D \propto \{e\}$ whenever $D \propto \{e' \mid e' \vDash e\}$, where $\vDash$ represents implication. Similarly, a domain $\mathcal{D}$ of constraint descriptions is *upwards closed* if for all $D \in \mathcal{D}$ and $e \in Con$, $D \propto \{e\}$ whenever $D \propto \{e' \mid e \vDash e'\}$. Intuitively, conjoining two descriptions from a downwards closed domain will result in an equal or more precise description; conjoining two descriptions from an upwards closed domain will result in an equal or less precise description.

3.3.2 Top-down Analysis

The top-down goal-dependent approach [3] is commonly used for analysing constraint logic programs. Its aim is to, for a particular domain, analyse a program for a set of initial *calling patterns* that describe the contexts from which the program's predicates (or procedures, for HAL) are called. For each calling pattern a corresponding *answer description* is inferred.

The equations defining the *general goal-dependent semantics* for a description domain $\mathcal{D}$, $gen_{\mathcal{D}}$, for canonical constraint logic programs were given in [18]. The semantics defined below, $top^{HAL}_{\mathcal{D}}$, is based on this definition of $gen_{\mathcal{D}}$, with two major differences. Firstly, it is extended to deal with full HAL programs, including explicit disjunctions, if-then-elses, and the use of higher-order procedures. Secondly, since HAL is not an interactive language (unlike many CLP implementations, it does not have an interactive environment), we do not use an arbitrary goal G as the starting point for analysis as is done for $gen_{\mathcal{D}}$. Instead, we begin from a set of calling patterns, S, that describes the contexts from which the procedures exported from a module may be called. Since HAL has a module system that supports separate compilation, when compiling a single module we may not know all the calling patterns with which the exported predicates might be called. Chapter 7 describes the compilation approach used in the HAL compiler that allows an accurate initial set of calling patterns S to be determined. For the moment, let us assume that we are given the set of calling patterns to analyse.

The definition of the $top^{HAL}_{\mathcal{D}}$ semantics is parametric. The domain $\mathcal{D}$ can be varied together with the following (monotonic) auxiliary functions:

$$
\begin{array}{ll}
comb_{\mathcal{D}}: & \mathcal{P}(Vars) \times \mathcal{D} \times \mathcal{P}(Vars) \times \mathcal{D} \to \mathcal{D}, \\
add_{\mathcal{D}}: & Con \times \mathcal{D} \to \mathcal{D}, \\
disj_{\mathcal{D}}: & \mathcal{P}(\mathcal{D}) \to \mathcal{D}, \\
if\_to\_then_{\mathcal{D}}: & \mathcal{D} \to \mathcal{D}, \\
ite_{\mathcal{D}}: & \mathcal{D} \times \mathcal{D} \times \mathcal{D} \to \mathcal{D}, \\
inrestrict_{\mathcal{D}}: & \mathcal{P}(Vars) \times \mathcal{D} \to \mathcal{D}, \\
outrestrict_{\mathcal{D}}: & \mathcal{P}(Vars) \times \mathcal{D} \to \mathcal{D}, \\
extend_{\mathcal{D}}: & \mathcal{P}(Vars) \times \mathcal{D} \to \mathcal{D}.
\end{array}
$$

Vars is the domain of all variables. The function $comb_{\mathcal{D}}$ is used to combine the answer description for a call with the description inferred for the procedure so far. This is similar to abstract conjunction, but more general; in Section 3.3.3 we will see why we use it instead of abstract conjunction. The function $add_{\mathcal{D}}$ is similar, but "adds" a primitive constraint to a description. The function $disj_{\mathcal{D}}$ combines descriptions from the disjuncts of a disjunction. The function $if\_to\_then_{\mathcal{D}}$ determines the context in which the *then* branch of an if-then-else will be analysed, and $ite_{\mathcal{D}}$ combines the three descriptions of an if-then-else. The function $inrestrict_{\mathcal{D}}$ is used to restrict (or project) a description onto the variables of a procedure call, $outrestrict_{\mathcal{D}}$ is used to project a description onto a smaller set of variables, and $extend_{\mathcal{D}}$ is used to extend a description onto a larger set of variables.

Definition 3.1: The *top-down semantics*, $top^{HAL}_{\mathcal{D}}$, for a domain $\mathcal{D}$, HAL program P and set of initial calling patterns S is the least fixpoint of the following equations:

$$
\begin{aligned}
&top^{HAL}_{\mathcal{D}}(P,S) = \{\text{let } B = \text{body of } H \text{ in}\\
&\qquad\qquad H : D_H \mapsto \mathbf{Proc}(H \leftarrow B, D_H) \mid H : D_H \in S\},\\
&\mathbf{Proc}(H \leftarrow B, D_H) = \text{let } V = vars(H) \cup local\_vars(B) \text{ in}\\
&\qquad\qquad outrestrict_{\mathcal{D}}(vars(H), \mathbf{Body}(B, V, extend_{\mathcal{D}}(V, D_H))),\\
&\mathbf{Body}(and(Bs), V, D_{pre}) = \mathbf{AndBodies}(Bs, V, D_{pre}),\\
&\mathbf{Body}(or(Bs), V, D_{pre}) = \mathbf{OrBodies}(Bs, V, D_{pre}),\\
&\mathbf{Body}(if\_then\_else(B_{if}, B_{th}, B_{el}), V, D_{pre}) = \mathbf{ITEBodies}(B_{if}, B_{th}, B_{el}, V, D_{pre}),\\
&\mathbf{Body}(literal(L), V, D_{pre}) = \mathbf{Literal}(L, V, D_{pre}),\\
&\mathbf{AndBodies}(nil, V, D_{pre}) = D_{pre},\\
&\mathbf{AndBodies}(B : Bs, V, D_{pre}) =\\
&\quad \text{let } V' = V \cup local\_vars(B) \text{ in}\\
&\quad \mathbf{AndBodies}(Bs, V, outrestrict_{\mathcal{D}}(V, \mathbf{Body}(B, V', extend_{\mathcal{D}}(V', D_{pre})))),\\
&\mathbf{OrBodies}(Bs, V, D_{pre}) =\\
&\quad disj_{\mathcal{D}}(\{\text{let } V' = V \cup local\_vars(B) \text{ in}\\
&\qquad outrestrict_{\mathcal{D}}(V, \mathbf{Body}(B, V', extend_{\mathcal{D}}(V', D_{pre}))) \mid B \in Bs\}),\\
&\mathbf{ITEBodies}(B_{if}, B_{th}, B_{el}, V, D_{pre}) =\\
&\quad \text{let } V_{ifth} = V \cup if\_then\_local\_vars(B_{if}, B_{th}) \text{ in}\\
&\quad \text{let } V_{if} = V_{ifth} \cup local\_vars(B_{if}) \text{ in}\\
&\quad \text{let } V_{th} = V_{ifth} \cup local\_vars(B_{th}) \text{ in}\\
&\quad \text{let } V_{el} = V \cup local\_vars(B_{el}) \text{ in}\\
&\quad \text{let } D'_{if} = outrestrict_{\mathcal{D}}(V_{ifth}, \mathbf{Body}(B_{if}, V_{if}, extend_{\mathcal{D}}(V_{if}, D_{pre}))) \text{ in}\\
&\quad \text{let } D_{th} = outrestrict_{\mathcal{D}}(V, \mathbf{Body}(B_{th}, V_{th}, extend_{\mathcal{D}}(V_{th}, if\_to\_then_{\mathcal{D}}(D'_{if})))) \text{ in}\\
&\quad \text{let } D_{el} = outrestrict_{\mathcal{D}}(V, \mathbf{Body}(B_{el}, V_{el}, extend_{\mathcal{D}}(V_{el}, D_{pre}))) \text{ in}\\
&\quad ite_{\mathcal{D}}(outrestrict_{\mathcal{D}}(V, D'_{if}), D_{th}, D_{el}),\\
&\mathbf{Literal}(L, V, D_{pre}) = add_{\mathcal{D}}(L, D_{pre}) \text{ when } L \in Con,\\
&\mathbf{Literal}(L, V, D_{pre}) = comb_{\mathcal{D}}(V, D_{pre}, vars(A), \mathbf{Atom}(A, D_{pre})) \text{ when } L = call(A),\\
&\mathbf{Atom}(A, D_{pre}) = \text{let } R = defn_P(A) \text{ in } \mathbf{Proc}(R, inrestrict_{\mathcal{D}}(vars(A), D_{pre})),
\end{aligned}
$$

where the auxiliary functions $add_{\mathcal{D}}$, $disj_{\mathcal{D}}$, $if\_to\_then_{\mathcal{D}}$, $ite_{\mathcal{D}}$, $inrestrict_{\mathcal{D}}$, $outrestrict_{\mathcal{D}}$ and $extend_{\mathcal{D}}$ must approximate the equivalent functions over $\mathcal{P}(Con)$: *add*, *disj*, *if_to_then*, *ite*,

restrict, *restrict* and *extend* respectively.

Note that the properties of the $comb_{\mathcal{D}}$ function are discussed below in Section 3.3.3.

Most of the equations have one or more body, literal, or atom parameters, plus two extra parameters V and D_H or V and D_{pre}. D_H is the *calling description* for the procedure H, i.e. the context from which H is called. A calling pattern consists of a procedure head and a calling description, e.g. $H : D_H$. D_{pre} is the *pre-description*, i.e. the description that applies to the program point immediately before a compound body, literal or atom; similarly a *post-description* is the description that applies to the program point immediately after a body;[10] V is the *variables of interest*, i.e. the set of variables that D_H or D_{pre} is in terms of.

The function *vars* returns the variables of an atom. The function *local_vars* returns the local variables of a body; as is usual, variables are given the smallest possible enclosing scope. The function *if_then_local_vars* returns the shared local variables of the *if* and *then* branches of an if-then-else. The function $defn_P$ returns the procedure in P that has a variant of the function's argument as its head.

The result of the analysis is a set of calling pattern and answer description pairs of the form $H : D_H \mapsto D_{Ans}$.

To find the answer for a predicate $H \leftarrow B$ in the context of a calling description D, the description is first extended onto B's variables of interest. The answer for B is then restricted onto the head variables to give the predicate's answer description.

Conjuncts are considered one at a time. Each conjunct's pre-description is extended onto the conjunct to account for any local variables, then an answer is obtained and restricted back onto the variables of the parent conjunction. The post-description of each conjunct becomes the pre-description of the next, except for the final conjunct, whose post-description is returned as the conjunction's overall answer via the equation for the empty conjunction *nil*.

A disjunction's pre-description is extended onto the variables of interest of each of its disjuncts. The answers from the disjuncts are restricted back onto the disjunction's variables of interest, and then combined using $disj_{\mathcal{D}}$ to give the disjunction's overall answer.[11] For most domains, $disj_{\mathcal{D}}$ finds the least upper bound.[12]

The equations for an if-then-else treat an $if\_then\_else(B_{if}, B_{th}, B_{el})$ similarly to a disjunction of the form $or(and(B_{if}, B_{th}), B_{el})$, but with two differences.

1. The pre-description for the *then* branch is obtained by applying $if\_to\_then_{\mathcal{D}}$ to the post-description of the *if* branch.

2. The overall answer for the if-then-else is obtained by applying $ite_{\mathcal{D}}$ to the answers for the three branches.

The first difference is required for bottom-up analysis as shown in Section 3.3.4. The second difference is required for determinism analysis; see Section 6.4.5 for more detail. Typically

[10] Note that in a conjunction one body's post-description is the following body's pre-description.
[11] Alternatively, we could first disjoin the descriptions and then restrict the result.
[12] But determinism is an exception; see Sections 6.4.3 and 6.4.4 for more detail.

for top-down analyses, an $\mathit{if\_then\_else}(B_{if}, B_{th}, B_{el})$ can be analysed exactly as if it has the form $\mathit{or}(\mathit{and}(B_{if}, B_{th}), B_{el})$. This is achieved by defining $\mathit{if\_to\_then}_{\mathcal{D}}$ and $\mathit{ite}_{\mathcal{D}}$ as follows:

$$\mathit{if\_to\_then}_{\mathcal{D}}(D) = D,$$
$$\mathit{ite}_{\mathcal{D}}(D_{if}, D_{th}, D_{el}) = \mathit{disj}_{\mathcal{D}}(\{D_{th}, D_{el}\}).$$

Primitive constraints (which are assumed to include (higher-order) unifications) are "added" to the pre-description using $\mathit{add}_{\mathcal{D}}$. All remaining literals are atoms "calling" a procedure. The pre-description is restricted onto the atom's variables, giving the calling description; the answer for that calling pattern is then combined with the pre-description to obtain the post-description.

For clarity, the top-down semantics does not mention program annotation. The annotation can be made explicit by threading an annotation function and the parent procedure calling description D_H through every equation, and modifying the annotation function when literals are encountered with a call $\mathit{annotate}(\mathit{pp}(L), D_H, D)$. This call associates the pair (D_H, D) with the program point immediately before L.[13]

One complication not addressed by the above semantics is the construction and application of higher-order terms. If a predicate is only called via a higher-order application, its body will not be annotated if we exactly follow the above semantics. This can be fixed by calling **Proc** for the procedure referred to by the higher-order term; to be safe the calling description used should be $\top$, except that any downwards closed information available for the variables captured by the closure can be included (upwards closed information must be discarded since the captured variables might be used between closure construction and application). This approach can be seen in Section 4.4.

The top-down semantics and parametric functions together specify a dataflow analysis. The implementation technique used to compute the least fixpoint is described in Chapter 4.

3.3.3 Two Variations of $\mathit{top}_{\mathcal{D}}^{\mathit{HAL}}$

The definition of the $\mathit{comb}_{\mathcal{D}}$ function in the $\mathit{top}_{\mathcal{D}}^{\mathit{HAL}}$ semantics in the previous section is deliberately vague. This is because there are two basic variations of the semantics.

Variation one: The first variation of the top-down semantics captures the usual theoretical view of goal-dependent analysis. It is obtained by requiring that:

$$\mathit{comb}_{\mathcal{D}}(V, D, V', D') = \mathit{conj}_{\mathcal{D}}(D, \mathit{extend}_{\mathcal{D}}(V, D')),$$

and that the function $\mathit{conj}_{\mathcal{D}}$ approximates the equivalent function conj over $\mathcal{P}(\mathit{Con})$.

This corresponds to the semantics called $\mathit{std}_{\mathcal{D}}$ in [18]. This variation is adequate for many abstract domains. However, the accuracy of the analysis for some domains can be improved by introducing a more precise $\mathit{comb}_{\mathcal{D}}$ operation.

[13] It would be simple to annotate every program point, including those before and after every conjunction, disjunction, and if-then-else; but for the optimisations used in the HAL compiler, it is only useful to record literal pre-descriptions.

Variation two: The operation $comb_{\mathcal{D}}(V_{pre}, D_{pre}, V_{call}, D_{Ans})$ is used to combine the answer description D_{Ans} for a call, which is in terms of the variables V_{call}, with the call's pre-description D_{pre} which is in terms of the variable set V_{pre}, where $V_{call} \subseteq V_{pre}$. Consider this predicate:

```
p(X,Y) :- q(X), r(Y).
```

The pre-description D_{pre} for `q(X)` will be in terms of the variables $\{X, Y\}$. D_{pre} is restricted onto $\{X\}$ to give the calling description for `q(X)`. Once an answer description D_{Ans} is obtained for the call, it is combined with D_{pre} to give the pre-description for `r(Y)`.

The call to $comb_{\mathcal{D}}$ is required to combine the answer description D_{Ans} with the variables not involved in the call. But in doing so, the information about the variables that are involved in the call is added twice. In the example above, the pre-description for `r(Y)` has had information about X added twice: once directly from D_{pre}, and a second time from D_{Ans} which was derived from D_{pre}. This can be a problem for domains that are not downwards closed — adding information twice can result in a much weaker and thus less useful description.

Generally, the abstract combination operation $comb_{\mathcal{D}}$ does not need to be defined in terms of abstract conjunction. By using a specialised version of $comb_{\mathcal{D}}$, called $comb^{spec}_{\mathcal{D}}$, we can obtain more accurate results, and sometimes improve efficiency. The only restriction is that $comb^{spec}_{\mathcal{D}}$ must be *safe* according to the following definition from [18].

Definition 3.2: Let $V_{call} \subseteq V_{pre}$, and $C, C'' \in \mathcal{P}(Con)$, and $D_{pre}, D_{Ans} \in \mathcal{D}$, and let $C' = conj(restrict(V_{call}, C), restrict(V_{call}, C''))$. The *specialised abstract combination* operation $comb^{spec}_{\mathcal{D}}$ is *safe* if, whenever $D_{pre} \propto C$ and $D_{Ans} \propto C''$, we also have:

$$comb^{spec}_{\mathcal{D}}(V_{pre}, D_{pre}, V_{call}, D_{Ans}) \propto conj(C, C').$$

The second variation of the top-down semantics is obtained by replacing $comb_{\mathcal{D}}$ by $comb^{spec}_{\mathcal{D}}$ which must be safe.

This corresponds to the semantics called $spec_{\mathcal{D}}$ in [18]. This variation will be at least as accurate as the first; for downwards closed domains it will be equally accurate. On a practical note, figures from [18] show that this variation is both more accurate and more efficient than the first variation for the abstract domains *SS* and ASub, which are described in Chapter 5.

3.3.4 Bottom-up Analysis

For bottom-up analyses each program fragment does not need to be considered in any kind of context, but instead has a single "inherent" answer. Examples of bottom-up analyses are type analysis (each term has only one type), and determinism analysis (each body in a procedure has only one determinism).

The key idea for adapting the top-down framework to perform bottom-up analysis is that every calling pattern should have the form $P : true_{\mathcal{D}}$. This means that each body can be analysed in a context-free manner.

Definition 3.3: The *bottom-up semantics*, $bot^{HAL}_{\mathcal{D}}$, for a domain $\mathcal{D}$, HAL program P and set of initial calling patterns S is obtained from the $top^{HAL}_{\mathcal{D}}$ semantics by making two changes. Firstly, we must impose restrictions to ensure $true_{\mathcal{D}}$ is the only calling description used.

1. The set S should only contain calling patterns in which the calling description is $true_{\mathcal{D}}$.
2. The auxiliary functions $inrestrict_{\mathcal{D}}$ and $if\_to\_then_{\mathcal{D}}$ should be defined as follows:

$$inrestrict_{\mathcal{D}}(V, D) = true_{\mathcal{D}},$$
$$if\_to\_then_{\mathcal{D}}(D) = true_{\mathcal{D}}.$$

Secondly, we must replace the four **Body** equations from the $top^{HAL}_{\mathcal{D}}$ semantics with the following equations:

$$\mathbf{Body}(and(Bs), V, D_{pre}) = comb_{\mathcal{D}}(V, D_{pre}, V, \mathbf{AndBodies}(Bs, V, inrestrict_{\mathcal{D}}(V, D_{pre}))),$$
$$\mathbf{Body}(or(Bs), V, D_{pre}) = comb_{\mathcal{D}}(V, D_{pre}, V, \mathbf{OrBodies}(Bs, V, inrestrict_{\mathcal{D}}(V, D_{pre}))),$$
$$\mathbf{Body}(if\_then\_else(B_{if}, B_{th}, B_{el}), V, D_{pre}) =$$
$$comb_{\mathcal{D}}(V, D_{pre}, V, \mathbf{ITEBodies}(B_{if}, B_{th}, B_{el}, V, inrestrict_{\mathcal{D}}(V, D_{pre}))),$$
$$\mathbf{Body}(literal(L), V, D_{pre}) = \mathbf{Literal}(L, V, D_{pre}),$$

where $comb_{\mathcal{D}}(V, D_{pre}, V, D_{Ans})$ is defined in the manner described for the first variation of $top^{HAL}_{\mathcal{D}}$ in Section 3.3.3.

By using $inrestrict_{\mathcal{D}}$, we ensure the compounds have the calling description $true_{\mathcal{D}}$, and are thus analysed in a context-free manner. Once an answer is obtained, it is conjoined with the pre-description using $comb_{\mathcal{D}}$ — if defined in the manner described for the first variation of $top^{HAL}_{\mathcal{D}}$ in Section 3.3.3, $comb_{\mathcal{D}}$ finds the conjunction of its second and fourth arguments when its first and third arguments are the same, as they are here.

Also, when performing a bottom-up analysis the calls to *annotate* are not necessary, since we are only interested in the answer descriptions for procedures.

See Section 4.9.7 for a discussion of the practical efficiency consequences of "under-utilising" the top-down framework in this way. See Chapter 6 for details on how determinism analysis is performed in this bottom-up manner.

Chapter 4

Intra-module Analysis

The previous chapter provided the theoretical basis for HAL's analysis framework. However, an implementation is much more complex. This is because it is not a simple task to efficiently find the least fixpoint of a set of semantic equations. In this chapter we describe the implementation of HAL's analysis framework in detail.

Several generic abstract interpretation engines have been built for constraint logic programming languages (e.g. [48, 7, 33, 36]). They all have sophisticated algorithms and data structures for fixpoint computation, with the aim of producing the *least program analysis graph*, i.e. the most precise finite representation of the (possibly infinite) set of (possibly infinite) AND-OR trees explored by the concrete execution of a program [3].

HAL's analysis framework is similar, being based on the algorithm from [23]. As presented, that algorithm is suitable only for canonical constraint logic programs, computing the least program analysis graph corresponding to the least fixed point of the equations for the standard semantics $std_{\mathcal{D}}$ given in [18]. The algorithm given in Section 4.4 extends the original algorithm to handle HAL programs and allow bottom-up analysis; it computes the least program analysis graph corresponding to the least fixed point of the equations for either the top-down semantics $top_{\mathcal{D}}^{HAL}$ defined in Section 3.3.2 or the bottom-up semantics $bot_{\mathcal{D}}^{HAL}$ defined in Section 3.3.4.

Before presenting the algorithm itself, we outline some types and notation, and describe the data structures and important operations. Afterwards, some important implementation details are discussed.

4.1 Preliminaries

Let us now introduce the types and notation used in the algorithm.

Descriptions: D, possibly subscripted, represents a description in an abstract domain. We assume that all description comparisons in the algorithm are purely syntactic, so descriptions must be in some kind of normal form to ensure all equivalences are found.

Parents and children: A procedure head is considered a parent of the procedure body. A compound body is a parent of the bodies it contains. Each parent has one or more corresponding children, which may be compound bodies or literals. For example, consider this procedure:

```
p(X, Y) :- q(X), r(Y).
```

The procedure head `p(X, Y)` is the parent of the conjunction `q(X), r(Y)`. The conjunction is the parent of the two literals `q(X)` and `r(Y)`.

Calling patterns: $H : D_H$, $P : D_P$, $B : D_B$, $A : D_A$ and $C : D_C$ are calling patterns for procedure heads, parents, bodies, atoms (procedure calls) and closures respectively. Each consists of the program part and its calling description, and each may be augmented with the variables of interest that the calling description is in terms of, e.g. $H(V_H) : D_H$. These variables are only shown where needed, although they are attached to all calling patterns in the actual implementation.

Answers: *Ans*, possibly subscripted, represents an *answer*. The two possible answers are *unreached* and *answer*(D), where D is an abstract description. An *unreached* answer indicates that analysis has encountered a body or procedure, but has not yet determined its answer description. An *answer* wraps the answer description of a body or procedure.

Once analysis is complete, a procedure's answer may remain *unreached*. Because the abstract execution approximates the concrete execution, this means the end of the procedure is never reached, and the procedure must never succeed. This is possible if the procedure always fails, infinitely loops, always aborts or always throws an exception.[1]

Arcs: The fixpoint computation is divided into units called *arcs*. Each arc represents a "snap-shot" of the analysis just before a body. The variable *Arc* represents an arc, and has this form:

$$P(V_P) : D_P \Rightarrow [D_{pre}]\ B(V_B).$$

$P(V_P) : D_P$ is the calling pattern of the parent procedure head or body. D_{pre} is the pre-description of the next body B. V_B is B's variables of interest.

4.2 Data Structures

There are three primary data structures used by the generic algorithm: the priority queue, the answer table, and the arc table.

[1] Often abstract domains have an artificial bottom element that means "unreached". But often they don't; by explicitly including *unreached* in the framework, we don't have to add such an element to domains that don't naturally have them.

4.2.1 The Priority Queue

The priority queue stores arcs. The analysis of one arc is the minimal unit of work performed by the algorithm, and is called an *event*. The priority queue is initialised from the initial set of calling patterns S. Arcs are dequeued and analysed one at a time; each event may cause more arcs to be added to the priority queue. Analysis is complete when the priority queue is empty.

The priority queue allows the analysis to calculate the least fixed point without specifying a fixed order of the traversal of bodies. The order used depends on the priority strategy of the queue, which is a parameter of the algorithm. Any strategy will work correctly, but the efficiencies of different strategies will vary. The strategy used in the implementation is described in Section 4.9.1.

One requirement for this generality is that when adding a new arc, any "similar" arcs already in the queue must be replaced. This ensures old events that are now out-of-date cannot be processed after more recent events. This requirement is explained in detail in Section 4.6.

4.2.2 The Answer Table

The answer table records answers for procedures and compound bodies. It has four parts, one for each kind of answer that needs to be recorded.

- Procedure answers have the form $H : D_H \mapsto Ans$.
- Compound body answers have the form $B : D_B \mapsto Ans$.
- Disjunct answers are recorded in the form $B : D_B \mapsto \text{list}(D_{Ans})$.
- If-then-else branch answers are recorded in the form $B : D_B \mapsto (Ans_{if}, Ans_{th}, Ans_{el})$.

The last two parts require some explanation. Whenever an answer changes for a single disjunct, the answer for the parent disjunction must be computed from all the disjunct answers, each of which must be restricted onto the parent disjunction's variables of interest. To avoid repeatedly restricting disjunct answer descriptions, we record them separately in the answer table.

Most other generic analysis frameworks do not work this way. Instead they join each new disjunct answer description with the current parent disjunction answer description. We cannot do this because we are interested in a domain, determinism, where $disj_{\mathcal{D}}$ is not a least upper bound. In this case, simply joining the new disjunct answer description with the current disjunction answer description could give the wrong answer for the disjunction. For example, consider this procedure:

```
p :- ( a ; b ).
```

Assume the first answer descriptions obtained for `a/0` and `b/0` are `det` and `failure` respectively. The disjunction of these is `det`. Now assume the answer description for `a/0` is subsequently changed to `semidet`. If we disjoin the new disjunct answer description `semidet` with the old disjunction answer description `det`, we get `nondet`, which is inaccurate. If we

instead compute the new description from all the disjunct answer descriptions, we get the more accurate answer `semidet`.

If-then-else branch answers are treated similarly, except that three answers must be recorded, rather than a list. We use answers instead of descriptions to allow for the possibility of a branch answer being *unreached*; this is not necessary for disjunctions because we only record answer descriptions for those disjuncts that have them (an empty answers list thus implies the answer *unreached* for a disjunction).

4.2.3 The Arc Table

The arc table is used to ensure that the effect of each new answer is fully propagated. It also allows a minimal amount of reanalysis to be performed whenever a procedure or body's answer is updated. Imagine a conjunction of the form `(..., B, ...)`. Once this conjunction has been analysed, the answer for B (for the appropriate calling pattern) may be subsequently improved. If this occurs, part of the conjunction must be reanalysed. The simplest approach is to reanalyse the whole procedure from scratch, but this unnecessarily repeats prior work. It is more efficient to reanalyse from B onwards. This requires a "snapshot" of the analysis state just prior to B. By recording B's arc in the arc table when we encounter it, we can easily restart the analysis from exactly that point.

We need to record arcs for compound bodies and calls to local procedures, as their answer descriptions can be improved during the course of the analysis. We don't need to record arcs for "constraints", since their answers cannot change.

The arc table is also used once analysis is complete, for computing the module's call graph, and for determining which variants must be generated if multi-variant specialisation is being performed. Sections 7.4.4 and 7.4.5 describe how this information is used. To compute the call graph, arcs must also be recorded for calls to external procedures, even though their answers can never change. Similarly, we must record arcs for procedures that are only called via higher-order calls so they are included in this post-processing.

The arc table has three parts.

- Arcs for compound bodies (the *compound arc table*).
- Arcs for procedure calls (the *call arc table*).
- Arcs for procedures called only through higher-order calls (the *higher-order call arc table*).

The first two parts are used in two different ways during analysis. Firstly, whenever a new answer description is obtained for the child of a compound, we look in the compound arc table to find the arc of the child's parent. We can then add the parent's arc to the priority queue, ensuring it is processed again now that its answer may be different (note that the parent's answer can immediately be looked up — the compound does not need to be reanalysed). This is how answer descriptions are propagated upwards within procedures.

Secondly, whenever an answer description for a local procedure is improved, we look through the call arc table to find all the locations where that procedure is called with the

relevant calling description, allowing for variable renamings. Arcs are added to the priority queue for all matching calls, which ensures that all procedures calling the updated calling pattern are reanalysed from the point of the call onwards.

Because call arcs have the form $P(V_P) : D_P \Rightarrow [D_{pre}]\ B(V_B)$, they do not contain the calling description D_B for the body B. This is for space efficiency, as D_B does not need to be computed for (higher-order) unifications and constraints. However, since the arc table lookups are for a full calling pattern $B : D_B$, the calling description D_B must be stored with the *Arc* itself in the arc table.

When adding arcs to the arc table, as when adding arcs to the priority queue, we must replace any "similar" arcs to avoid processing out-of-date arcs after more recent ones. This is explained in Section 4.6.

4.3 Operations

The following operations are used in the algorithm. We do not provide a definition for them because they are quite straightforward. Note that the priority queue, answer table and arc table are all assumed to be implicit global variables.

- add_event(Arc) adds an arc to the priority queue. It must replace any "similar" arcs in the queue, as explained in Section 4.6.
- next_event() dequeues the next event from the priority queue.
- add_compound_arc(Arc, D_B) adds an arc and its body's calling description to the compound arc table. It must replace "similar" arcs in the compound arc table, as explained in Section 4.6.
- add_call_arc(Arc, D_A) adds an arc and its body's calling description to the call arc table. It too must replace "similar" arcs in the call arc table.
- add_HO_call_arc(Arc, D_C) adds an arc and its captured procedure's calling description to the higher-order call arc table. Again, it must replace "similar" arcs.
- add_proc_answer($H : D_H \mapsto Ans$) records a new local procedure answer in the answer table, overwriting any old answer.
- add_compound_answer($B : D_B \mapsto Ans$) records a new compound answer in the answer table, overwriting any old answer.
- add_disjunct_answer_desc($P : D_P, B, D$) records the restricted answer description D for disjunct B in disjunction P's list of disjunct answer descriptions, overwriting any old answer description for B. If this is P's first disjunct answer description, it creates the list before adding D.
- add_ite_branch_answer_desc($P : D_P, B, D$) records the restricted answer description for branch B[2] of if-then-else P as $answer(D)$, overwriting any old answer for B. If this is

[2] In the implementation it is possible to tell which if-then-else branch a body B is from its parent P.

P's first branch answer description, it creates a triple (*unreached*, *unreached*, *unreached*) before adding D.

- get_disjunct_answer_descs($P : D_P$) gets the list of restricted disjunct answer descriptions for the disjunction with the calling pattern $P : D_P$.
- get_ite_branch_answers($P : D_P$) gets the triple of restricted branch answers for the if-then-else with the calling pattern $P : D_P$.
- annotate(L, D_P, D) annotates a literal with the description pair (D_P, D).[3]
- vars(A) returns the set of variables in an atom. Due to normalisation, every variable in an atom is distinct, so the set's cardinality will be equal to the atom's arity.
- local_vars(B) finds the set of local variables of a body.
- if_then_local_vars(B) finds the set of local variables shared by the *if* and *then* branches of an if-then-else.
- arity(A) returns the arity of an atom.
- compute_call_graph() uses the arc table to compute the call graph and determine which variants need to be generated.

In the framework implementation, the parametric operations of the algorithm become the methods of a type class named `abstract_domain`. An abstract description type must be an instance of this class to be used within the framework. The methods are as follows.

- Acomb, Aadd, Adisj, Aif_to_then, Aif_then_else, Aoutrestrict, Ainrestrict and Aextend should implement the functions from Section 3.3.2: $comb_{\mathcal{D}}$, $add_{\mathcal{D}}$, $disj_{\mathcal{D}}$, $if\_to\_then_{\mathcal{D}}$, $ite_{\mathcal{D}}$, $outrestrict_{\mathcal{D}}$, $inrestrict_{\mathcal{D}}$ and $extend_{\mathcal{D}}$ respectively; the only variation is that Aadd and Acomb should return an answer rather than just an answer description (to allow *unreached*). Each function should be monotonic and approximate the corresponding concrete functions.

  Note that the variation of the $top_{\mathcal{D}}^{HAL}$ semantics used depends entirely on the definition of Acomb, as described in Section 3.3.3.
- Abottom(V) returns $\bot$ over the variables in V for the abstract domain.
- Ainitial_guess($H : D_H$) returns an initial answer for an encountered local procedure. The simplest option is to return *unreached*, although if an answer for a more specific calling pattern has been found, it can be used as a first approximation. If an answer description is returned, it must be below the least fixed point for the procedure.

[3] Attentive readers will note that Section 3.3.2 discussed annotation pairs of the form (D_H, D), i.e. pairing the literal's description D with the procedure head description D_H rather than the description D_P of the parent, which may be a compound. In the implementation, we record D_P rather than D_H. Since the (D_H, D) form is required for multi-variant specialisation (see Section 7.4.5), compute_call_graph converts from the (D_P, D) form to the (D_H, D) form when post-processing the arc table.

- Ais_constraint(A) decides if an atom is considered a "constraint" for the domain. If it is, its post-description will be obtained by Aadd rather than treating it like a normal procedure call. This is useful if it is possible to return answers more precise than what would be inferred by applying the algorithm to the definition of the constraint, e.g. if specific domain knowledge can be used. Also, it allows answers to be provided for constraints for which the code cannot be analysed (e.g. if it is written in C).

- Acalling_descHO(V_H : D_H) is used when computing the calling description for procedures called via higher-order calls. V_H is the set of head variables of the procedure captured by the closure. The calling description D_H will only contain information about the variables that were captured by the closure.

  For top-down analyses, it must return $\top$ over V_H, except it can retain downwards closed information from D_H. For bottom-up analyses, it must return $true_{\mathcal{D}}$, in the same way that Ainrestrict does.

- Aexternal_proc(P : D_P) obtains an answer description for a call to a procedure not defined in the analysed module. One approach is to return $\top$ as the answer description — this is simple, safe, but inaccurate. Chapter 7 describes a compilation approach that leads to much more accurate answers for calls to external procedures. This operation must also handle higher-order applications made using `call/n`.[4]

4.4 The Generic Algorithm

The algorithm is shown in Figures 4.1, 4.2 and 4.3. It is defined in terms of the following main operations.

- `analyse` adds an arc to the priority queue and an initial answer to the answer table for every calling pattern in S. It also contains the main loop which processes arcs one at a time until completion. Finally, it calls compute_call_graph to finish processing.

- `new_proc_CP` is called whenever a local procedure is encountered as part of a previously unseen calling description. The calling description is extended onto the variables of interest of the procedure body to obtain its pre-description. The arc for the body is then added to the priority queue, and the initial answer obtained using Ainitial_guess is added to the answer table and returned.

- `process_arc` forms the heart of the analysis. It performs a single step of the left-to-right traversal of a body.

  If the body is a (higher-order) unification or "constraint", we annotate the literal with the pre-description, and then use Aadd to determine the post-description. If the literal is a higher-order unification, we also call `handle_unifyHO` to ensure the procedure from which the closure is built is analysed.

[4] Being a built-in, `call/n` is defined in HAL's `system` module and thus is always external.

```
analyse(S)
    foreach H : D_H ∈ S
        new_proc_CP(H(vars(H)) : D_H)
    while Arc := next_event()
        process_arc(Arc)
    compute_call_graph()

new_proc_CP(H(V_H) : D_H)
    let B := body of procedure H
    V_B := V_H ∪ local_vars(B)
    D_pre := Aextend(V_B, D_H)
    add_event(H(V_H) : D_H ⇒ [D_pre] B(V_B))
    Ans_init := Ainitial_guess(H : D_H)
    add_proc_answer(H : D_H ↦ Ans_init)
    return Ans_init

process_arc(Arc)
    let P(V_P) : D_P ⇒ [D_pre] B(V_B) := Arc
    if (B = literal(L))
        if (L = unify(_,_)) or (L = unifyHO(_,_)) or (L = call(A) and Ais_constraint(A))
            annotate(L, D_P, D_pre)
            Ans := Aadd(L, D_pre)
            if (L = unifyHO(_,C))
                handle_unifyHO(Arc, D_pre, C)
        elseif (L = call(A))
            D_A := Ainrestrict(V_B, D_pre)
            annotate(L, D_P, D_A)
            Ans := get_proc_answer(Arc, V_P, D_pre, A(V_B) : D_A)
    else
        D_B := Ainrestrict(V_B, D_pre)
        Ans := get_compound_answer(Arc, V_P, D_pre, B(V_B) : D_B)
    if (Ans = answer(D_post))
        if (P is a procedure head)
            update_proc_answer(P : D_P, Aoutrestrict(V_P, D_post))
        elseif (P = and(Bs)) and (there is a conjunct B_next following B in Bs)
            V_next := local_vars(B_next)
            add_event(P(V_P) ⇒ [Aextend(V_next, Aoutrestrict(V_P, D_post))] B_next(V_next))
        elseif (P = if_then_else(B_if, B_th, _)) and (B = B_if)
            V_ifth := if_then_local_vars(B_if, B_th)
            D'_post := Aoutrestrict(V_ifth, D_post)
            add_ite_branch_answer_desc(P : D_P, B_if, Aoutrestrict(V_P, D'_post))
            V_th := V_ifth ∪ local_vars(B_th)
            add_event(P(V_P) : D_P ⇒ [Aextend(V_th, Aif_to_then(D'_post))] B_th(V_th))
        else
            update_compound_answer(P(V_P) : D_P, B, Aoutrestrict(V_P, D_post))
```

Figure 4.1: Generic analysis algorithm (I)

If the body is a procedure call, we restrict the pre-description onto the call variables, annotate it, and then obtain an answer using `get_proc_answer`.

Otherwise, the body is a compound. First we Ainrestrict the pre-description onto

```
handle_unifyHO(Arc, D_pre, C)
    let H := head of procedure used in closure C
    let V_C^H := first arity(C) vars in H
    D_C := Ainrestrict(vars(C), D_pre)
    D_C^H := rename D_C replacing vars(C) with V_C^H
    D_H := Acalling_descHO(vars(H) : D_C^H)
    if not(there exists a renaming σ s.t. σ(H : D_H) ↦ _ in answer table)
        add_HO_call_arc(Arc, D_C)
        new_proc_CP(H(vars(H)) : D_H)

get_proc_answer(Arc, V_P, D_pre, A(V_A) : D_A)
    add_call_arc(Arc, D_A)
    if (A is a call to an external procedure)
        Ans := answer(Aexternal_proc(A : D_A))
    elseif (there exists a renaming σ s.t. σ(A : D_A) ↦ Ans in answer table)
        Ans := σ^{-1}(Ans)
    else
        Ans := new_proc_CP(A(V_A) : D_A)
    return do_comb(V_P, D_pre, V_A, Ans)

get_compound_answer(Arc, V_P, D_pre, B(V_B) : D_B)
    add_compound_arc(Arc, D_B)
    if not(B : D_B ↦ Ans is in the answer table)
        Ans := new_compound_CP(B(V_B) : D_B)
    if (analysis is bottom-up)
        return do_comb(V_P, D_pre, V_B, Ans)
    else
        return Ans

new_compound_CP(B(V_B) : D_B)
    if (B = and(B_1 : _))
        V_1 := V_B ∪ local_vars(B_i)
        add_event(B(V_B) : D_B ⇒ [Aextend(V_1, D_B)] B_1(V_1))
    elseif (B = or(Bs))
        foreach disjunct B_i in Bs
            V_i := V_B ∪ local_vars(B_i)
            add_event(B(V_B) : D_B ⇒ [Aextend(V_i, D_B)] B_i(V_i))
    elseif (B = if_then_else(B_if, B_th, B_el))
        V_if := V_B ∪ if_then_local_vars(B_if, B_th) ∪ local_vars(B_if)
        V_el := V_B ∪ local_vars(B_el)
        add_event(B(V_B) : D_B ⇒ [Aextend(V_if, D_B)] B_if(V_if))
        add_event(B(V_B) : D_B ⇒ [Aextend(V_el, D_B)] B_el(V_el))
    add_compound_answer(B : D_B ↦ unreached)
    return unreached
```

Figure 4.2: Generic analysis algorithm (II)

the body. This does nothing for top-down analyses, but is required for bottom-up analyses so that the compound is analysed with the calling pattern $true_{\mathcal{D}}$. We then use `get_compound_answer` to get the answer for the compound.

Now the answer obtained for the body is dealt with. If it is *unreached*, we can do

```
update_proc_answer(H : D_H, D_new)
    Ans_new := answer(D_new)
    find the renaming σ s.t. σ(H : D_H) ↦ Ans_old in answer table
    if (Ans_new ≠ σ^-1(Ans_old))
        add_proc_answer(H : D_H ↦ Ans_new)
        foreach entry (Arc, D_B) in the call arc table
            let _ : _ ⇒ [_] B := Arc
            if (there exists a renaming σ s.t. σ(H : D_H) = B : D_B)
                add_event(Arc)

update_compound_answer(P(V_P) : D_P, B, D_new)
    if (P = and(_))
        Ans_new := answer(D_new)
    elseif (P = or(_))
        add_disjunct_answer_desc(P : D_P, B, D_new)
        Ds := get_disjunct_answer_descs(P : D_P)
        Ans_new := answer(Adisj(Ds))
    elseif (P = if_then_else(_, _, _))
        add_ite_branch_answer_desc(P : D_P, B, D_new)
        (Ans_if, Ans_th, Ans_el) := get_ite_branch_answers(P : D_P)
        D_if := ite_answer_to_desc(Ans_if, V_P)
        D_th := ite_answer_to_desc(Ans_th, V_P)
        D_el := ite_answer_to_desc(Ans_el, V_P)
        Ans_new = answer(Aif_then_else(D_if, D_th, D_el))
    find entry P : D_P ↦ Ans_old in answer table
    if (Ans_new ≠ Ans_old)
        add_compound_answer(P : D_P ↦ Ans_new)
        foreach entry (Arc, D'_B) in the compound arc table
            let _ : _ ⇒ [_] B' := Arc
            if (P : D_P = B' : D'_B)
                add_event(Arc)

ite_answer_to_desc(Ans, V_P)
    if (Ans = answer(D_Ans))
        return D_Ans
    else
        return Abottom(V_P)

do_comb(V_P, D_pre, V_B, Ans)
    if (Ans = answer(D_Ans))
        return answer (Acomb(V_P, D_pre, V_B, D_Ans))
    else
        return unreached
```

Figure 4.3: Generic analysis algorithm (III)

nothing more for this body; if the program point immediately after the body is reachable a non-*unreached* answer will eventually be obtained for it, and an event will be scheduled so the analysis will re-commence from this point.

If an answer description was obtained, the action taken depends on what kind of parent the body has. If it's a procedure head, we restrict the description onto the head

variables and update the procedure answer. If the body is part of a conjunction and there is at least one more conjunct to analyse, we restrict the answer onto the parent conjunction's variables, extend it onto the next conjunct's variables, and then add an arc. If the body is the *if* branch of an if-then-else, we treat the *if* and *then* branches like a two-body conjunction, except that we use Aif_to_then to determine the *then* branch's pre-description. We must also record the *if* branch answer, which is restricted onto the variables of the parent if-then-else. If none of the above is true, we restrict the answer onto the parent compound's variables and `update_compound_answer` deals with it.

- `handle_unifyHO` is called whenever we encounter a higher-order unification. It ensures that any procedure only called via a higher-order call is analysed.

  It restricts the description onto the closure's captured variables, and then renames this description and the closure to use the variables from the head of the captured procedure. It then calls Acalling_descHO which computes the best safe calling description possible: $\top$ plus any downwards closed information known about the captured variables. Finally, if this calling pattern has not been seen before, we add the arc to the higher-order call arc table (to be used by compute_call_graph) and call `new_proc_CP` which schedules it for analysis (but we ignore `new_proc_CP`'s return value).

  If the constructed closure is never actually used, this will analyse the captured procedure unnecessarily. However, since reachability is undecidable in general, we err on the side of safety. In practice, it is extremely rare for a closure to be built but never called.

- `get_proc_answer` is called whenever we encounter a procedure call. First we add an arc for the call to the arc table (recall that arcs for external procedures are needed to compute the call graph once analysis is finished).

  If the call is to an external procedure, we use Aexternal_proc to obtain an answer. Otherwise the call is local. If the calling pattern has been seen before and we have an answer for it (modulo variable renaming), we rename the answer appropriately (although the answer may be *unreached*). If it hasn't been seen before we use `new_proc_CP`, which will return an initial answer and schedule an event so the procedure will be analysed.

  If the answer obtained for the call is not *unreached*, we Acomb it with the pre-description to obtain the post-description.

- `get_compound_answer` is called whenever we encounter a compound. First we add an arc for the compound to the arc table. If we have an answer for the compound — *unreached* or otherwise — we use that (no renamings are necessary). Otherwise we call `new_compound_CP`, which will return an initial answer and also schedule events so that the children bodies will be analysed.

  If we are performing a top-down analysis, this answer can be returned immediately. However, if we are performing a bottom-up analysis, we must combine the answer with the compound's pre-description first, because the compound was analysed with the context-free calling description $true_{\mathcal{D}}$.

- `new_compound_CP` is called when we encounter a compound as part of a new calling description; the behaviour depends on the compound type.

  If it's a conjunction, we extend the pre-description D_B onto the variables of the first conjunct and schedule an event for it to be analysed. If it's a disjunction, we schedule one event per disjunct, first extending the pre-description onto each disjunct's variables of interest. If it's an $\mathit{if\_then\_else}(B_{if}, B_{th}, B_{el})$, we treat it similarly to how we would treat an $\mathit{or}(\mathit{and}(B_{if}, B_{th}), B_{el})$.

  Regardless of the compound type, we install and return *unreached* as the compound's initial answer.

- `update_proc_answer` is called whenever we infer a new answer description for a local procedure. If there was a previous answer and this answer is the same (modulo variable renaming), we do nothing. Otherwise, we add the answer to the answer table, look through the arc table to find all the other procedures in which this calling pattern has been encountered (again modulo variable renaming), and schedule events so that all those procedures are reanalysed from that point onwards.

- `update_compound_answer` is called whenever we infer a new answer description for a child of a compound; the behaviour depends on the compound type.

  If it's a conjunction, we use the answer as given. If it's a disjunction, we record the restricted disjunct answer description, and then combine all the restricted disjunct answer descriptions obtained so far, giving the new disjunction answer. If it's an if-then-else, we record the restricted branch description, obtain answer descriptions for each of the three branches — using $\bot$ for any branch with the answer *unreached* — and then combine them with Aif_then_else. Note that this function is only called if an answer has been obtained for one or both of the *then* and *else* branches, so Aif_then_else will never be called with all three descriptions as $\bot$.

  If there was a previous answer and this answer is the same, we do nothing. Otherwise, we add the answer to the answer table, find any matching arcs for the compound in the compound arc table, and schedule events so they are reanalysed. Note that for top-down analyses at most one matching arc will still be up-to-date but the answers obtained for any out-of-date arcs will not be propagated upwards because their parent arcs will have been removed from the arc table (see Section 4.6). For bottom-up analyses, there will always be exactly one match (that of the arc with the compound calling description $\mathit{true}_{\mathcal{D}}$).

- `ite_answer_to_desc` turns a branch answer into a description; if the answer is *unreached*, it returns $\bot$.

- `do_comb` is a simple wrapper for Acomb that lifts it to operate on answers.

4.5 An Example

We are now ready for an example. Consider the predicate `le/2` from Section 3.3.1, reproduced here:

```
:- pred le(nat, nat).
:- mode le(oo, oo) is nondet.
le(X, Y) :-
    ( X = Y
    ;
      Y = s(Z),
      le(X, Z)
    ).
```

First, let us consider the information flow that takes place when analysing this procedure, independent of any particular analysis domain, as represented in Figure 4.4. The term structure of the procedure is clear. Dotted arrows represent information flow; each arrow is marked with a number that indicates the accompanying operation. Program point annotations indicate the variables that an abstract description would be in terms of, for example XYZ indicates that any description would be in terms of X, Y and Z. The head, compound bodies and `le(X,Z)` call are annotated with their calling and answer descriptions.

The diagram does not show the addition and use of arcs in the arc table, nor does it show the order in which events are processed. Only the first argument of each label operation is shown.

We start from the calling pattern `le(X,Y)` : XY. All the downward pointing arrows (1,3,6,8) extend parent descriptions onto the local variables of the child; the conjunction is the only child body with local variables, so extension 6 is the only one that does anything.

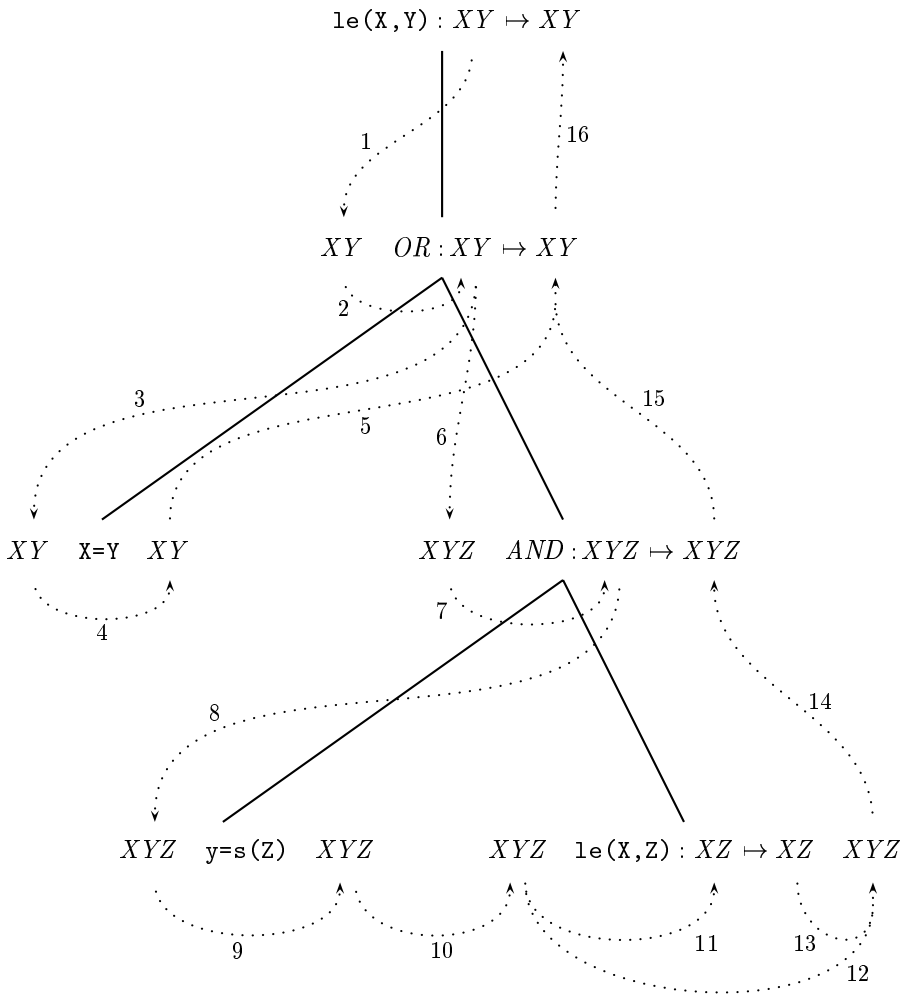
The constraints `X=Y` and `Y=s(Z)` are handled by adding the literal to the pre-description (4,9). The compound bodies and the call are handled differently. First the pre-description is inrestricted onto the variables of the body (2,7,11); for top-down analyses this only changes the description for the call `le(X,Z)`, restricting it from XYZ to XZ.

The disjunction's answer is obtained by disjoining the restricted answer descriptions of its children (5,15). The pre-description for the call `le(X,Z)` is obtained from the post-description of `X=Y` by restricting it back onto the variables of the parent conjunction, and then extending it to include any singleton variables (10); in this case the restriction and extension makes no difference. The call's post-description is obtained by combining its pre-description and answer description (12,13). The answer for the conjunction is the restricted post-description of its last conjunct (14).

Finally, the procedure answer is found by restricting the answer of its child body (16).

This example shows that many of the calls to Aextend and Aoutrestrict are unnecessary; five calls were made to each operation, and only one of each changed the description. This observation inspired the optimisation described in Section 4.9.5.

Figure 4.5 shows how this form is "instantiated" for a simple groundness analysis of the calling pattern `le(X,Y)` : $\{Y\}$, which corresponds to that described in Section 3.3.1. Let us



1. Aextend($\{X, Y\}$)
2. Ainrestrict($\{X, Y\}$)
3. Aextend($\{X, Y\}$)
4. Aadd(X=Y)
5,15. Aoutrestrict($\{X, Y\}$), Adisj
6. Aextend($\{X, Y, Z\}$)
7. Ainrestrict($\{X, Y, Z\}$)
8. Aextend($\{X, Y, Z\}$)
9. Aadd(le(X,Z))
10. Aoutrestrict($\{X, Y, Z\}$), Aextend($\{X, Y, Z\}$)
11. Ainrestrict($\{X, Z\}$)
12,13. Acomb
14. Aoutrestrict($\{X, Y, Z\}$)
16. Aoutrestrict($\{X, Y\}$)

Figure 4.4: Information flow during analysis of `le(X,Y)`

assume that the priority queue is a stack, and that disjunctions are placed on the stack in reverse order so that the first disjunct is processed first.

We start with a single arc on the queue for `le(X,Y)` : $\{Y\}$. Processing that leads to an event for $OR : \{Y\}$. When it is processed, arcs are added to the queue for the two disjuncts. We process the base case `X=Y` first. Its answer is disjoined with the other disjunction answers obtained so far (none), and the disjunction's answer is updated to $\{X, Y\}$. At this point we search the compound arc table for an arc matching $OR : \{Y\}$, and find one; this arc

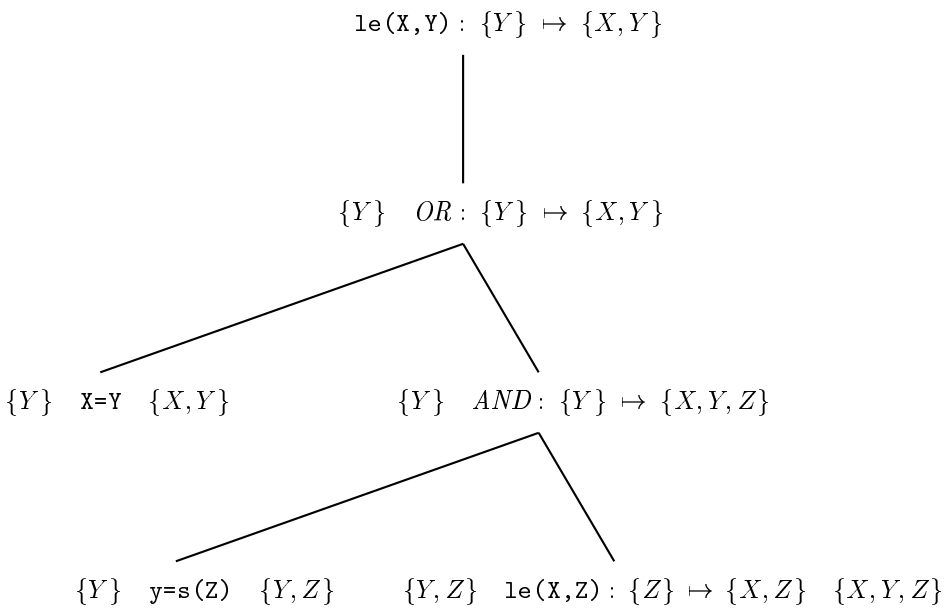


Figure 4.5: Simple groundness analysis of `le(X,Y)` : $\{Y\}$

is added to the front of the queue. Processing it just involves looking up the disjunction's current answer; the first answer for the calling pattern `le(X,Y)` : $\{Y\}$ is thus $\{X,Y\}$.

The remaining arc on the queue is for the second disjunct's calling pattern $AND : \{Y\}$. We process its conjuncts one at time to obtain its answer {X,Y}, looking up the current answer for `le(X,Y)` : $\{Y\}$ in the process for the call `le(X,Z)` : $\{Z\}$. The conjunction's answer is updated and its arc from the arc table re-scheduled and processed, propagating its answer up to its parent disjunction. The answer $\{X,Y\}$ for the second disjunct is disjoined with the previous answer $\{X,Y\}$ for the first disjunct, resulting in $\{X,Y\}$. Since this is the same as the disjunction's old answer, we don't schedule any arcs from the arc table, and as there are no remaining events in the queue, analysis ends.

This example showed how compound arcs are rescheduled when compound answers change, but it didn't show how call arcs are used when a procedure's answer changes. They would be used during a simple groundness analysis of `le/2` if the priority queue strategy is different to that above: if the recursive disjunct is processed first, the answer for the `le(X,Z)` : $\{Z\}$ call when first encountered is *unreached* and so analysis of that disjunct cannot continue. The `X=Y` disjunct is then analysed and the answer $\{X,Y\}$ obtained for the calling pattern `le(X,Y)` : $\{Y\}$. This is a renaming of the `le(X,Z)` : $\{Z\}$ arc in the call arc table, so it will be added to the queue and analysis of the recursive disjunct can continue using the new answer.

4.6 Replacing Similar Arcs

When adding arcs to the priority queue and arc table during analysis, we must replace any "similar" arcs to avoid processing out-of-date arcs after more recent ones, which could lead

to the wrong answer. A "similar" arc is one with the same parent calling pattern and body; the pre-description (and calling description, for arc table entries) may be different. When adding an arc $P : D_P \Rightarrow [D_{pre}]\ B$ to the priority queue, we must replace any arc of the form $P : D_P \Rightarrow [\_]\ B$, and when adding an entry $(P : D_P \Rightarrow [D_{pre}]\ B, D_B)$ to the arc table, we must replace any entry of the form $(P : D_P \Rightarrow [\_]\ B, \_)$. Let us consider why.

Replacing similar priority queue arcs: Consider this predicate:

```
p(X,Y) :- q(X), r(Y).
```

Assume `p/2` is called with the calling description $D_{\mathtt{p}}$. After processing the call `q(X)`, the following arc is added to the priority queue:

$$Arc = AND : D_{AND} \Rightarrow [D_{pre}]\ \mathtt{r(Y)}$$

(where $AND : D_{AND}$ is the conjunction calling pattern). Assume that the answer for `q/1` is now updated before that arc is processed. The updated answer means another arc is added to the priority queue:

$$Arc' = AND : D_{AND} \Rightarrow [D'_{pre}]\ \mathtt{r(Y)}.$$

Arc is now out-of-date, but because any priority queue strategy is allowed, it is possible that Arc' could be processed before *Arc*. The incorrect answer resulting from *Arc* would replace the correct answer from Arc'. Hence "similar" arcs must be replaced in the priority queue.

Replacing similar call arcs: Let us reuse the same example predicate. Assume we process $\mathtt{p(X,Y)} : D_{\mathtt{p}}$ and obtain an answer, adding the following entries to the call arc table:

$$Entry_{\mathtt{q}} = (AND : D_{AND} \Rightarrow [D_{\mathtt{q}}]\ \mathtt{q(X)}, D_{\mathtt{q}c}),$$
$$Entry_{\mathtt{r}} = (AND : D_{AND} \Rightarrow [D_{\mathtt{r}}]\ \mathtt{r(Y)}, D_{\mathtt{r}c}).$$

Assume the answer for $\mathtt{q(X)} : D_{\mathtt{q}c}$ is then changed. Analysis restarts at `q(X)` and proceeds to `r(Y)` with a different pre-description $D'_{\mathtt{r}}$; the following arc is now added to the call arc table:

$$Entry'_{\mathtt{r}} = (AND : D_{AND} \Rightarrow [D'_{\mathtt{r}}]\ \mathtt{r(Y)}, D'_{\mathtt{r}c}).$$

If $D_{\mathtt{r}c} = D'_{\mathtt{r}c}$ and a new answer is subsequently obtained for $\mathtt{r(Y)} : D_{\mathtt{r}c}$, both $Entry_{\mathtt{r}}$ and $Entry'_{\mathtt{r}}$ will match the calling pattern and have their arcs added to the priority queue by `update_proc_answer`. Because similar priority queue arcs are replaced, if the out-of-date arc from $Entry_{\mathtt{r}}$ is added second it could replace the arc from $Entry'_{\mathtt{r}}$, which would result in a wrong answer. An alternative would be to store the two arcs in such a way that the more recent $Entry'_{\mathtt{r}}$ will be chosen in preference to $Entry_{\mathtt{r}}$, but since $Entry_{\mathtt{r}}$ is not of any use once $Entry'_{\mathtt{r}}$ arises, it is simpler to replace it.

The problem arises because different call pre-descriptions can lead to the same calling description, (i.e. $\mathsf{Ainrestrict}(V, D)$ can equal $\mathsf{Ainrestrict}(V, D')$ when $D \neq D'$). Thus when a procedure answer is updated, more than one arc for a particular call could match under the same parent calling pattern, one of which will be out-of-date. Hence "similar" arcs must be replaced in the call arc table.

Replacing similar compound arcs: Consider the following predicate:

```
p(X,Y) :- a(X,Y), ( b(X) ; b(Y) ).
```

Assume that the initial answer for `a/2` leads to the following arc being added to the compound arc table:

$$Entry = (AND : D_{AND} \Rightarrow [D_{pre}]\ OR, D_{OR})$$

(where $OR : D_{OR}$ is the disjunction calling pattern). Assume `a/2`'s answer is then updated before the two disjuncts under *Entry* are analysed, which leads to the following arc being added to the compound arc table:

$$Entry' = (AND : D_{AND} \Rightarrow [D'_{pre}]\ OR, D'_{OR}).$$

Assume that the two disjuncts `b(X)` and `b(Y)` under $Entry'$ are analysed and an answer is obtained for $OR : D'_{OR}$. Since $Entry'$ matches the updated calling pattern, it is added to the priority queue and analysed; the up-to-date answer is propagated upwards, and the answer for `p(X,Y)` : $D_{\mathtt{p}}$ is updated.

Assume the two disjuncts under *Entry* are now analysed, and a new (but out-of-date) answer is obtained for $OR : D_{OR}$. Since *Entry* matches this updated calling pattern, it is added to the priority queue, leading to an out-of-date answer for `p(X,Y)`.

If *Entry* is replaced by $Entry'$ when $Entry'$ is added, this problem is avoided — when the disjuncts under *Entry* are analysed and the answer for $OR : D_{OR}$ is updated, there is no out-of-date matching arc in the compound arc table. Hence "similar" arcs must be replaced in the compound arc table.

4.7 Correctness

The algorithm computes the least fixpoint of the equations for either of the $top^{HAL}_{\mathcal{D}}$ or $bot^{HAL}_{\mathcal{D}}$ semantics defined in Section 3.3 using chaotic iteration [9]. Although the order in which events are processed is not fixed, the events themselves encode a left-to-right traversal of the procedures, ensuring a unique result. For the least fixed point to be well defined the abstract operations must be monotonic, Ainitial_guess must return a value below the least fixed point, and Acomb must be safe. Under these assumptions we have the same correctness result as the original algorithm from [23].

Theorem 4.1: For a program P and initial calling patterns S, the generic analysis algorithm returns an answer table which represents the least program analysis graph of P and S.

The corollary of this theorem is that the strategy used by the priority queue does not affect the correctness of the algorithm.

4.8 Differences Between Algorithms

Although the presented algorithm was based on the algorithm from [23], there are several significant differences. This is because the original algorithm is not intended to closely represent an implementation, while the presented algorithm faithfully represents the framework implemented in the HAL compiler. The main differences follow.

Compound bodies: The biggest difference is the move from canonical constraint logic programs to HAL programs including explicit disjunctions and if-then-elses; the presence of arbitrarily nested compound bodies necessitated the introduction of arcs and answers for compounds. The different handling of calls and compounds throughout completely changed the structure of the algorithm and the data structures. The scoping of variables within nested bodies also complicated the extension and restriction of descriptions significantly.

Higher-order terms: The original algorithm did not mention higher-order terms at all. The presented algorithm does, by ensuring that procedures only called via a closure are analysed, and by keeping downwards closed information in calling descriptions from higher-order unifications and applications.

Non-least upper bound $disj_{\mathcal{D}}$: The original algorithm always joined new disjunct answer descriptions with the current disjunction answer. The presented algorithm instead records the disjuncts individually, and performs Adisj on all the disjunct answer descriptions. This is necessary to perform accurate determinism analysis, because it has a $disj_{\mathcal{D}}$ operation that is not a least upper bound.

Bottom-up analysis: Unlike the original algorithm, the presented algorithm can be used for performing bottom-up analysis, increasing its generality and usefulness.

Optimisation: The original algorithm deliberately ignored several obvious optimisations. For example, it used three types of events: *newcall*, *updated* and *arc* events. The *newcall* and *updated* events indirectly caused more *arc* events to be scheduled; the presented algorithm instead adds the appropriate *arc* events directly.

Also, the original algorithm restricted the pre-description of all literals to find the calling description and stored it in all arcs, even though it is not necessary for (higher-order) unifications and "constraints".

Calls to external procedures: Unlike the original algorithm, the presented algorithm distinguishes between calls to local and external procedures.

A more general semantics: The original algorithm used abstract conjunction, $conj_{\mathcal{D}}$, and thus found the least program analysis graph corresponding to the least fixed point of the equations for the standard semantics $std_{\mathcal{D}}$ from [18]. The presented algorithm uses abstract combination, $comb_{\mathcal{D}}$, resulting in a more flexible semantics that accommodates two

variations, as seen in Section 3.3.3, the second of which can be more accurate and efficient for some domains.

Incrementality: The original algorithm was designed for incremental analysis. Its main part computed the least program analysis graph from scratch, and ancillary parts were used to incrementally analyse a program that had had predicates added, removed or arbitrarily changed.

The presented algorithm does not have these incremental parts, and the current implementation does not perform incremental analysis.[5] Part of the reason for this is that the HAL compiler currently does not perform incremental compilation, and has no mechanism for determining which parts of a module have changed since the last compilation. However, the presented algorithm could be extended quite straightforwardly to be incremental — none of the changes made affect its potential for incremental analysis. Doing so would require adding code for determining which procedures have changed, adjusting the initial set of calling patterns S, and deleting affected entries in the answer table. Once this is done, S and the answer table can be passed to the framework as is.

4.9 Efficiency Considerations

The presented algorithm is a fairly faithful representation of the implementation in the HAL compiler. However, for the sake of clarity and generality, some details of the implementation were omitted. If the presented algorithm is implemented naïvely, performance will be poor. This section discusses several important efficiency considerations.

4.9.1 Priority Queue Strategy

We have seen that the priority strategy used does not affect correctness, but it does affect efficiency. Different priority strategy approaches are discussed in [50]. There is a trade-off between the complexity of the strategy and the number of arcs processed — using a more sophisticated strategy can reduce the number of arcs processed, but adding and removing events may take longer.

The strategy used in the implementation is extremely simple: the priority queue is a stack, which treats the most recent events as having the highest priority. The only complication is that disjunct events are reversed before adding, and *else* branch events are added before *if* branch events; this is so the base case of recursive predicates is (usually) processed before the recursive case. This was found to reduce execution time by about 10–15% for determinism analysis (see Chapter 6) of reasonably sized modules (e.g. more than 100 lines of code).

[5] By *incremental* we mean that it supports incremental compilation in which only the parts of a program that have changed (and related parts) must be recompiled and reanalysed. The presented algorithm *is* incremental in a different sense — procedures are not reanalysed from scratch when an answer they depend on changes, but are only reanalysed from the affected point onwards.

Several more complicated strategies were experimented with, such as ensuring all children of a compound are analysed before the compound's answer is propagated upwards. Surprisingly, these strategies were no better than the simple stack-based strategy.

One drawback of the stack-based priority queue is that the cost of replacing "similar" arcs in the queue is proportional to the length of the queue. When analysing larger modules, the queue can become quite large (e.g. more than 300 events), and if the abstract domain operations are cheap, the replacement check can take up a significant proportion of analysis time (e.g. close to 40% for determinism analysis of large files, e.g. 2,000 lines of code). This is annoying, since the check succeeds extremely rarely.

However, attempts to reduce the overhead of the check — such as breaking the queue into multiple buckets, and using a hash function that allocates arcs with the same parent calling patterns to the same bucket — did not improve execution time, since the other overheads introduced negated any reduction in the time taken for the check.

Despite this, the analysis times shown in Section 6.7.1 for determinism analysis are still quite fast. Also, determinism is an extreme example, because its abstract operations are very simple. The relative cost of the replacement check is much smaller for other domains which have more expensive abstract operations.

4.9.2 Arc Table Structure

The arc table is accessed in four different ways.

1. In `update_proc_answer`, we must find all call arc table entries that match a renaming of a given calling pattern (there may be zero, one or many).

2. "Similar" arcs must be replaced in the call arc table.

3. In `update_compound_answer`, we must find all compound arcs that match a given calling pattern (there may be zero, one or many).

4. "Similar" arcs must be replaced in the compound arc table.

If we do not choose our data structure carefully, one or more of these accesses will be linear in the size of the table. Since all four accesses are frequent, this will adversely affect execution time. The structure of the arc table used in the implementation is as follows:

```
:- typedef arc_tbl(D)      -> arc_tbl(call_arc_tbl(D), comp_arc_tbl(D)).
:- typedef call_arc_tbl(D) = tree(proc_key, call_arcs(D)).
:- typedef call_arcs(D)    = list(bdkey, list(pair(arc(D), D))).
:- typedef comp_arc_tbl(D) = tree(bdkey, list(pair(arc(D), D))).
```

Each type is parameterised by the abstract description `D`. The type `tree/2` is a balanced tree, and `pair/2` is a simple pair type; both are from HAL's standard library.

The call arc table's primary index is a `proc_key` uniquely identifying each procedure; the secondary index is a `bdkey` which uniquely identifies each body calling that procedure. For each call to the particular procedure, we record the arc itself and its calling description.

Access 1 finds the entry for a particular `proc_key`, and then traverses the list of all calls to that procedure to find those which match the calling description (modulo renaming). Access 2 finds a specific call by `proc_key` and `bdkey`, and then traverses the inner list to find and replace the arc with matching parent description, if there is one.

If the number of procedures in a module is X, the number of calls to a given procedure P in that module is Y, and the maximum number of different calling patterns any particular call to P is involved in is Z, access 1 for P will have a complexity of $O(\log(X) + YZ)$, and access 2 will be $O(\log(X) + Y + Z)$. This is good since X can be reasonably large (e.g. more than 100 in larger modules), whereas Y and Z are usually very small (Z is often equal to one; for bottom-up analyses it is always one).

The compound arc table's primary index is the `bdkey` uniquely identifying the compound body (whereas there can be multiple calls to a procedure in a module, there is only one occurrence of each compound body, so the compound arc table is simpler). For each body there is a list of associated arcs.

Access 3 finds the entry for a particular compound by `bdkey`, and then traverses the list to find all the arcs with a matching calling description. Access 4 also looks up a body by `bdkey`, and again traverses the list, this time to find and replace the arc with matching parent description, if there is one.

If the number of compounds in a module is X, and the number of different calling patterns a given compound is involved in is Z, accesses 3 and 4 for that compound will both have a complexity of $O(\log(X) + Z)$. Again this is good since X can be quite large (e.g. more than 1000 in larger modules), but Z is usually very small (often equal to one; for bottom-up analyses it is always one).

A simple experiment was performed in which the trees in the above types were replaced with lists, various modules from the compiler itself were compiled, and determinism analysis was timed using the SICStus Prolog timing predicates.[6] For very small modules (e.g. 87 lines of code) the list representation was just as fast, or possibly faster (difficult to tell, since the times were very small). But not surprisingly, the tree version was up to 1.7 times faster for larger modules (e.g. 1381 lines of code).

4.9.3 Dead Variable Removal

Normalisation of HAL programs introduces many intermediate variables, which can lead to prohibitively large abstract descriptions. Many variables local to conjunctions will have their last-use before the end of the conjunction, and thus they can be removed from descriptions before then. For example:

```
p(X, Y) :- q(X, A), r(A, B), s(B, Y).
```

In this predicate the variables local to the conjunction are `A` and `B`. `A` is last used in the call `r(A, B)` and so can be removed from its post-description before the conjunction's end. `B` cannot be removed before the conjunction's end since it is used in the last conjunct.

[6] Recall that HAL has not yet bootstrapped, and is currently executed through the SICStus Prolog.

The framework already restricts a conjunct's post-description onto the variables of its parent conjunction immediately; the dead variable removal can be neatly incorporated into this operation. In the example above, we would restrict the post-description of `r(A, B)` onto the variables of the parent conjunction, {`A`, `B`, `X`, `Y`}; but because `A` dies at this point, we can instead restrict onto {`B`, `X`, `Y`}. We must also adjust the parent conjunction's variables of interest so that `A` is not considered for any subsequent conjuncts.

Prior to the analysis framework's invocation, the compiler performs a textual liveness pass that determines the last-use of every non-head variable in each procedure, specifically to support this dead variable removal. Each abstract domain is required to provide a parameter that indicates if dead variable removal should be performed. This is because for some domains, such as *LSign* [43], restriction is very expensive. However, for domains where restriction is not so expensive it is definitely worthwhile; much of the time it makes only a slight improvement to efficiency, but in certain cases the difference is drastic. For example, the benchmark `hamil` contains one predicate which defines a 50-node Hamiltonian path graph. Normalisation breaks up the graph's creation into more than 400 unifications, all in a single conjunction. Without dead variable removal, sharing analysis (see Section 5.4) does not complete within 10 minutes. With dead variable removal, sharing analysis takes only 25 seconds.

4.9.4 Avoiding Unnecessary Adisj and Aif_then_else Calls

For generality, the algorithm presented in Section 4.4 performs Adisj on all disjuncts every time a new disjunct answer is obtained, to allow for domains in which Adisj is not a least upper bound. Because this generality is only required for unusual domains such as determinism (see Sections 6.4.3 and 6.4.4), abstract domains are required to provide a parameter that indicates whether Adisj is a least upper bound, and thus whether each new disjunct answer can be disjoined with the previous disjunction answer. This saves time by avoiding unnecessary calls to Adisj, and saves space because disjunct answers do not need to be stored in the answer table.

Similarly, for many domains $\mathsf{Aif\_then\_else}(D_{if}, D_{th}, D_{el}) = \mathsf{Adisj}([D_{th}, D_{el}])$. Unnecessary calls to Aif_then_else can be avoided by combining new *then* and *else* branch answers with the previous if-then-else answer using Adisj. If so, the abstract operation Abottom is not required, since it is only used when performing a non-least upper bound Aif_then_else.

A related optimisation is that the implementation only performs Adisj and Aif_then_else operations if the answer for the disjunct or branch has changed.

Some experiments showed that the effect of these optimisations is quite small. Nonetheless, it is worthwhile because they are very simple, and result in strictly less time and space being used. Also, if an abstract domain was implemented in which these operations were expensive, the gains would be larger.

4.9.5 Avoiding Unnecessary Aextend and Aoutrestrict Calls

As we saw in Section 4.5, the framework calls Aextend and Aoutrestrict very frequently. A simple analysis of several modules of the HAL compiler showed that approximately 85% of

bodies do not contain any local variables. Because of this, the framework implementation only calls Aextend and Aoutrestrict when the extension/restriction is onto a different set of variables. For analyses in which these operations are not cheap, this could make a significant difference. For analyses in which these operations are cheap, the difference is negligible.

It should be noted that this optimisation is carefully intertwined with the dead variable removal described in Section 4.9.3. Sometimes a body has no local variables, but is the last body in which one or more of a conjunction's local variables are used. In such cases, we are careful not to skip the call to Aoutrestrict which is needed to remove the dead variable(s).

4.9.6 Selective Annotations

As the algorithm is presented, all literals are annotated. In the implementation, abstract domains are required to perform more selective annotation by providing a predicate that succeeds if a unification or "constraint" should be annotated (this predicate is part of the `abstract_domain` type class). Also, it allows abstract domains to annotate a literal with something other than a description. This reduces space usage by avoiding unnecessary annotations, and by reducing the size of annotations. Note that procedure calls and higher-order unifications are always annotated with full descriptions to support multi-variant specialisation (see Section 7.4.5).

4.9.7 Efficiency of Bottom-up Analysis

It is worth considering the cost of performing a bottom-up analysis within the top-down framework. We cannot perform an empirical comparison, since there is no dedicated bottom-up analysis in the HAL compiler, and it would not be reasonable to compare the cost with that of a bottom-up analysis from another compiler for a similar language. However we can make some observations.

Unnecessary calling descriptions: Every calling pattern will have the form $B : true_{\mathcal{D}}$, so the calling description $true_{\mathcal{D}}$ is not necessary. This adds a small time overhead to the framework, for superfluous construction and deconstruction of calling patterns, and unnecessary calls to Ainrestrict, Aif_to_then and Acalling_descHO (although these are extremely cheap, as they return $true_{\mathcal{D}}$). Compared to the more complex operations performed by the framework, the overhead should be negligible.

It also adds a space overhead to the priority queue, arc table and answer table, because $true_{\mathcal{D}}$ must be stored with every calling pattern. For many domains $true_{\mathcal{D}}$ is a "small" value (e.g. an empty set), in which case the overhead should again be negligible. Some domains have a "large" $true_{\mathcal{D}}$, in which case the overhead may be more substantial. We have not implemented any domains with a "large" $true_{\mathcal{D}}$, so it is hard to say how large this overhead would be.

Unnecessary reanalysis: The analysis framework traverses procedure bodies left-to-right. Whenever the answer for a procedure or compound is updated, the bodies to the right of the updated body will be reanalysed. For example, consider this procedure:

```
p(X, Y) :- q(X, A), r(A, B), s(B, Y).
```

Imagine the entire procedure has been analysed, and an answer obtained. The answer description for the calling pattern `q(X, A)` : $true_{\mathcal{D}}$ is then updated. The entire procedure will be reanalysed from `q(X, A)` onwards. During a top-down analysis this will be necessary, since the post-description for one conjunct becomes the pre-description for the next conjunct. During a bottom-up analysis, this may not be necessary, because Ainrestrict always returns $true_{\mathcal{D}}$, and individual conjuncts are independent. But when the subsequent conjuncts are reanalysed, their answers can be found quickly by looking up the answer table; the expense of the re-analysis will largely depend on the complexity of Acomb.

Conclusion: Judging from these observations, using the top-down framework for performing bottom-up analysis is not as efficient as a dedicated bottom-up framework would be. However, the figures in Section 6.7.1 on the cost of determinism analysis show the overhead is quite acceptable for a simple domain. Further experimentation with more complex domains would be required to confirm that the overhead is reasonable in general.

4.10 Performance Evaluation

The performance of the analysis framework cannot be measured independently of an abstract domain. For this reason, we present no performance figures for the framework in this chapter. Instead, we refer the reader to Section 5.6.1, which evaluates the cost of groundness, sharing and freeness analyses, and Section 6.7.1, which evaluates the cost of determinism analysis.

Chapter 5

Herbrand Analysis

Three of the best known and most widely studied analyses for CLP languages infer information about the *groundness*, *sharing* and *freeness* of program variables.

- Groundness analysis is used to determine whether a variable is ground at a certain point in a program's execution.
- Sharing analysis is used to determine when variables may affect each other. During a program's execution, two variables *share* if at some point, they are bound to terms that share a common variable. Similarly, a *non-linear* variable is one that shares with itself, i.e. is bound to a term which contains multiple occurrences of one or more variables.
- Freeness analysis is used to determine whether a variable is *free*, i.e. has not been bound to any non-variable term.

These three analyses are closely linked: groundness and freeness information can improve sharing analysis, while sharing information can improve both groundness and freeness analyses. Related to freeness analysis is *reference chain length* analysis, which determines the length of free variables' reference chains. For the purpose of optimisation in HAL, we are only interested in knowing whether free variables have a reference chain length of one. Such variables have not been aliased with any other, and will be described as *lonely*. We will refer to these analyses collectively as *Herbrand analyses*.

In traditional CLP languages, the programmer provides no information that encompasses groundness, sharing and freeness information, so any inferred information can be highly useful. In the strongly-moded language Mercury the programmer provides almost all of such information; Mercury only supports the use of ground terms. HAL's weak mode system supports programs written in a style anywhere between these two extremes. This means that Herbrand analyses must allow for information provided by the programmer via mode declarations, which influences them significantly.

This chapter starts by describing common uses of and approaches to inferring groundness, sharing and freeness information. It then describes how these three analyses are performed within HAL's analysis framework, the optimisations enabled by the inferred information, and concludes with an evaluation of their costs and benefits.

5.1 Common Uses and Approaches

This section briefly outlines some of the more common uses of groundness, sharing and freeness information, and some typical approaches to performing the respective analyses.

5.1.1 Groundness

Groundness analysis is arguably the most fundamental and important dataflow analysis for CLP programs. It can be used for optimising unification and constraint solving. Good groundness information is also vital for the accuracy of many other analyses, in particular most kinds of sharing analysis.

Groundness analysis is most commonly performed using one of the two domains Def and Pos. In these domains, the groundness of variables is abstractly described by *definite* and *positive* boolean functions, respectively, which allow accurate capturing of both definite groundness information, and groundness dependencies.

5.1.2 Sharing

The term "sharing" encompasses a range of different kinds of information. The different kinds of sharing information have multiple uses, such as occur-check reduction [57], improving exploitation of independent AND-parallelism [29], compile-time garbage collection [35, 44], and for improving the accuracy of freeness analysis [38, 4].

Many domains have been proposed for sharing analysis. They typically include groundness, sharing, linearity and even freeness information. The two most influential domains have been ASub, which contains groundness, *pair-sharing*, and linearity elements [57]; and Sharing, which uses *set-sharing* to encode information about groundness and groundness dependencies, and sharing and sharing dependencies [28].

5.1.3 Freeness

Freeness information too has several uses, such as improving utilisation of AND-parallelism [24, 65], improving the accuracy of sharing analysis [47], reducing the overhead of dynamic scheduling [49], and optimising unification. This last use is how we use freeness analysis information in HAL. Reference chain length information has also been used for optimising the execution of Prolog programs [59].

5.2 Term Representation and Herbrand Unification

The most obvious use of Herbrand analysis information for HAL, and the one we will concentrate on, is the optimisation of Herbrand constraint solving, i.e. general term unification. Recall that a variable whose type is an instance of the `herbrand` type class and has a "`:- herbrand`" declaration within its module can be involved in arbitrary unifications. A general unification operation is provided for these types, but groundness, sharing and freeness information can be used to replace this general operation with more specialised and efficient versions. Before we can understand these optimisations, we must first understand

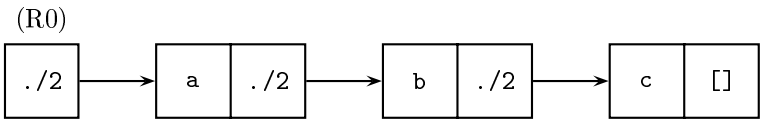


Figure 5.1: Mercury representation of `[a, b, c]`

HAL's term representation, how it compares with Mercury's term representation, and how it supports Herbrand constraint solving.

5.2.1 Mercury

Mercury's strong static typing means that the type of every term is known at compile-time. This allows the use of a very compact term representation. On 32-bit machines with aligned addressing, the two low bits of a pointer are zero. In Mercury these two bits are used to store "tag" values. For types with with up to four functors, the tag bits are sufficient to distinguish them. If the term has any arguments, the remainder of the word is a pointer to a sequence of contiguous words on the heap that store the term's arguments; otherwise the remaining bits are zero. For types with more than four non-constant functors, an extra word is used to represent the functor. This case occurs rarely in practice and in what follows we will ignore it for simplicity. True logic variables do not need to be represented, since Mercury does not allow them. Native types that fit within a word, such as 32-bit integers, are not tagged.

For example, Figure 5.1 shows the representation of a list `[a, b, c]`, where `a`, `b` and `c` are functors of a single type. The cells labelled with "`./2`" contain pointers with a tag that identifies them as pointing to a cons cell; the cells labelled with `a`, `b`, `c` and `[]` contain tags representing those functors. The head of the list is pointed to by a tagged pointer in the register R0; this is how terms are typically passed as arguments to procedures. Note that the representation is almost identical to that of a linked-list written in C.

Like HAL, Mercury performs normalisation, so only unifications of the form $X = Y$ and $X = f(Y_1, \ldots, Y_n)$ must be considered. The two cases for unifications of the form $X = Y$ are as follows.

- If X is `new`[1] and Y is `ground`, or vice versa (an *assignment*) the value of the `ground` variable is copied into the `new` variable.
- If both X and Y are `ground` a specialised procedure that is generated for each type is used to check if the two terms are equal; we will call this generated procedure `unify_gg`.

The three cases for unifications of the form $X = f(Y_1, \ldots, Y_n)$ are as follows.

- If X is `new` and $Y_1, \ldots, Y_n$ are `ground` (a *construct*), a block of n contiguous cells is allocated and X is set as a pointer with an appropriate tag to this block.

[1] In Mercury "`new`" is called "`free`".

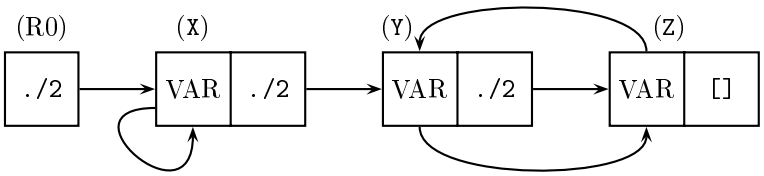


Figure 5.2: HAL representation of `[X, Y, Z]`

- If X is `ground` and $Y_1, \ldots, Y_n$ are all `new` (a *deconstruct*), after checking X is of the correct functor, the values in the contiguous block of n cells it points to are copied into $Y_1, \ldots, Y_n$.
- If all variables are `ground`, `unify_gg` is used as for the $X = Y$ case.

See [56] for full details on Mercury's term representation.

5.2.2 HAL

HAL supports not only basic term manipulation equivalent to Mercury's, but also more general term manipulation via the Herbrand solver. Since HAL compiles to Mercury, it makes considerable sense for its basic term representation to be the same as Mercury's. However, we also need a way to distinguish and represent free variables.

This is achieved by reserving one of the four tags used to distinguish the functors of a type to represent unbound variables (this is why the Mercury programs generated by the HAL compiler must be compiled in the "reserved tag" grade of the Melbourne Mercury Compiler, as mentioned in Section 2.4.3). We call this reserved tag the VAR tag. One consequence of this is that the remaining tag bits can only distinguish types with three or fewer functors without requiring extra space. Variables are represented using the same technique as the PARMA system [60]: lonely variables point to themselves, and aliased free variables point to each other, forming a cycle. When a Herbrand variable is created, it is initialised by the predicate `herbrand_init` (calls to which are inserted where necessary during mode analysis) which allocates a new cell on the heap for the variable, and sets it to point to itself, tagged with the VAR tag. For example, Figure 5.2 shows HAL's representation of the list `[X, Y, Z]` after `Y` and `Z` have been unified. `X` is lonely and points to itself; `Y` and `Z` are aliased and form a cycle.

The advantage of this variable representation is that when a variable becomes bound to a term, all the variables in its cycle are updated to point to that term. Thus when a term becomes ground it is a legitimate Mercury term. Furthermore, even when a term is only partially bound, the bound part may be able to be manipulated by the efficient Mercury operations. For example, HAL's representation of the ground list `[a, b, c]` is the same as Mercury's (Figure 5.1).

The various unification cases that can occur in HAL are implemented by a mix of Mercury and C code. When applicable, the Mercury unifications are used, e.g. when one argument is `new` and the other `ground` or bound, or when both arguments are `ground`. Otherwise the unifications must be handled specially.

First let us consider unifications of the form $X = Y$. If X is `new` and Y is not, or vice versa, Mercury's assignment works as is. If X and Y are both `new`, one variable is initialised by `herbrand_init` making it `old`, and the previous case applies. If X and Y are both `old`, it is a "true" unification, and is replaced by a call to the Herbrand unification predicate `unify_oo`, which is automatically generated for each type `t`. It is defined as follows:

```
:- pred unify_oo(t, t).
:- mode unify_oo(oo, oo) is semidet.
unify_oo(X, Y) :-
    ( nonvar(X) -> ( nonvar(Y) -> unify_val_val(X, Y)
                   ;              unify_var_val(Y, X)
                   )
    ;              ( nonvar(Y) -> unify_var_val(X, Y)
                   ;              unify_var_var(X, Y)
                   )
    ).
```

The predicate `nonvar/1` succeeds if the argument is bound, i.e. if it has a non-VAR tag. If it succeeds, it also dereferences the bound term if necessary (dereferencing will be explained shortly). The predicate `unify_var_var` unifies two variables. It first checks that the two variables are not the same, and then joins the cycles together, trailing the changes. The predicate `unify_var_val` unifies a variable and a bound term by modifying all variables in the cycle to point to the bound term and trailing the changes. Both `unify_var_var` and `unify_var_val` are implemented in C as follows:

```
unify_var_var(X, Y) {
    QX = *X; QY = *Y;
    while (QX != Y && QY != X) {
        if (QX != X && QY != Y) {
            QX = *QX; QY = *QY;
        } else {
            trail(X); trail(Y);
            Tmp = *Y; *Y = *X; *X = Tmp;
            break;
        }
    }
}
```

```
unify_var_val(X, Y) {
    QX = X;
    do {
        trail(QX);
        Next = *QX;
        *QX = Y;
        QX = Next;
    } while (QX != X)
}
```

The predicate `unify_val_val` unifies two bound terms; it is defined similarly to `unify_gg` except it calls `unify_oo` on any arguments of the bound terms.

Similarly, there are various cases for unifications of the form $X = f(Y_1, \ldots, Y_n)$. Some are handled correctly by Mercury, and some are not. The optimisation of these unifications is not considered in this thesis, so we omit the details of how they work; the interested reader can consult [11] for more information.

There is one complication that arises from HAL's unbound variable representation. Only heap variables can be placed in an alias cycle — a free variable in a register or on the stack

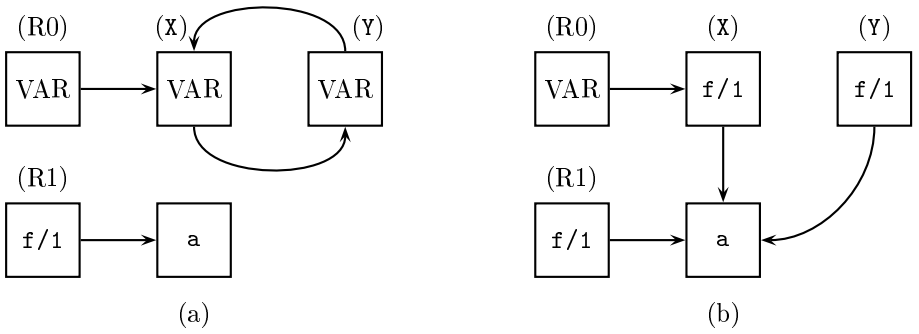


Figure 5.3: Bound variable dereferencing

points into a cycle on the heap. When the cycle is bound, an extra pointer dereference is required to get from the register or stack variable to the term. For example, consider the pointer in register R0 in Figure 5.3(a) pointing into the variable cycle. This situation will occur if `X` and `Y` form a cycle on the heap, and then a `new` variable in the register R0 is unified with `Y`. If the variables are unified with the term `f(a)` pointed to by register R1 we obtain the heap shown in Figure 5.3(b). Accessing the term from R0 requires an extra pointer dereference.

Note that the implementation of dynamic scheduling in HAL is intertwined with Herbrand representation; delayed goals are "attached" to variable chains, and can be triggered by various events, such as a variable becoming bound. For efficiency reasons, all the unifications used have two versions — one for types that support delay, and one for types that do not. For simplicity, all the following material is targeted towards the non-delay unifications, although much of it is also applicable to the unifications that support delay.

5.2.3 Optimisation of Herbrand Unifications in HAL

Calls to the general unification procedure `unify_oo` can be replaced with calls to more specialised and efficient procedures if we know that one or both arguments are lonely, free, bound or ground, or that the arguments do not share with each other.

Groundness, boundedness and freeness optimisations: If we know both arguments of the unification are free, we can avoid the two `nonvar/1` tests in `unify_oo` by replacing it with `unify_var_var`. If we know one argument is free and the other bound or ground, we can replace it with `unify_var_val`. If we know both arguments are bound, we can replace it with `unify_val_val`. If we know one argument is `ground` and the other is bound, we can replace `unify_oo` with `unify_val_gnd`, which is defined similarly to `unify_val_val` except that `unify_gnd_old` (defined below) is called on any arguments rather than `unify_oo`, possibly avoiding further `nonvar/1` tests. If we know that both arguments are `ground`, we can replace `unify_oo` with Mercury's `unify_gg`.

If we know the freeness/boundedness of only one argument, we can avoid one `nonvar/1` test by replacing `unify_oo` with `unify_var_old` or `unify_val_old`, defined as follows:

```
unify_var_old(X, Y) :-
```

```
    ( nonvar(Y) -> unify_var_val(X, Y)
    ;              unify_var_var(X, Y)
    ).

unify_val_old(X, Y) :-
    ( nonvar(Y) -> unify_val_val(X, Y)
    ;              unify_var_val(Y, X)
    ).
```

If we know one argument is ground and nothing about the other, we can replace `unify_oo` with `unify_gnd_old`, defined as follows:

```
unify_gnd_old(X, Y) :-
    ( nonvar(Y) -> unify_val_gnd(Y, X)
    ;              unify_var_val(Y, X)
    ).
```

Sharing optimisations: If both arguments are free and do not share, their cycles cannot intersect, so we can replace `unify_var_var` with `unify_var_var_noshare`, which replaces the cycle traversal with a single pointer swap, as follows:

```
unify_var_var_noshare(X, Y) {
    trail(X); trail(Y);
    Tmp = *Y; *Y = *X; *X = Tmp;
}
```

If one argument is free, the other has an unknown instantiation, and they do not share, calls to `unify_var_old` can be replaced by `unify_var_old_noshare`, defined identically to `unify_var_old` but with the call to `unify_var_var` replaced by `unify_var_var_noshare`.

If the freeness/boundedness of the two arguments is not known, but it is known that they do not share, `unify_oo` can be replaced with `unify_oo_noshare`, which is derived from `unify_oo` in the same fashion that `unify_var_old_noshare` is derived from `unify_var_old`.

Loneliness optimisations: If one argument is lonely, and the other argument is lonely or free, we can replace `unify_oo` with `unify_var1_var`:

```
unify_var1_var(X, Y) {
    trail(X); trail(Y);
    Tmp = *Y; *Y = X; *X = Tmp;
}
```

This is very similar to `unify_var_var_noshare` above. The difference is in the second assignment — because `X` is lonely, we know it points to itself, so we can avoid the dereference because `X == *X`.

If one argument is lonely and the other argument is bound or ground, we can replace `unify_oo` with `unify_var1_val` — because the first argument is lonely, we can do a simple pointer re-assignment, avoiding the loop used in `unify_var_val`.

$X \setminus Y$	lonely	free	bound	ground	unknown
lonely	`var1_var`	`var1_var`	`var1_val`	`var1_val`	`var1_old`
free		`var_var`*	`var_val`	`var_val`	`var_old`*
bound			`val_val`	`val_gnd`	`val_old`
ground				`gg`	`gnd_old`
unknown					`oo`*

Table 5.1: Specialisations of `unify_oo`

```
unify_var1_val(X, Y) {
    trail(X);
    X = Y;
}
```

If one argument is lonely and nothing is known about the other argument, we can replace `unify_oo` with `unify_var1_old`, which is defined similarly to `unify_var_old`.

Summary: The possible specialisations of `unify_oo` are summarised in Table 5.1. Their names have been shortened by removing the "`unify_`" prefix. Empty entries are equivalent to the symmetrical case (although the arguments must be swapped in some cases). Those marked with a '*' have a sharing and non-sharing version. Note that `unify_var1_var` and `unify_var1_old` are always non-sharing, because a lonely variable cannot share with anything else. Note also that there is no `unify_var1_var1`, `unify_var1_gnd` or `unify_var_gnd`, as they would be identical to `unify_var1_var`, `unify_var1_val` and `unify_var_val` respectively.

To gain full optimisation benefits from the freeness analysis information, the HAL compiler performs multi-variant specialisation. A variant is generated for each calling description with which a procedure is called. See Section 7.4.5 for more details.

5.3 Groundness Analysis in HAL

The domain Pos can be used for highly accurate groundness analysis. It can also be implemented very efficiently using *reduced order binary decision diagrams* (ROBDDs) [5], directed acyclic graph structures that compactly represent boolean functions.

However, ROBDDs are best implemented in languages that, unlike HAL, allow explicit destructive update of data structures. Once HAL bootstraps and compiles through Mercury, it will be relatively straightforward to use an existing ROBDD implementation written in C, such as the one described by Schachte in [55], to implement Pos in HAL. In the meantime, we needed a groundness analyser for the short-term. We had two basic choices: port a C-based ROBDD implementation to SICStus Prolog, or choose a different representation. The former would have been quite difficult,[2] so we instead chose a groundness domain and representation that was straightforward to implement in HAL and still quite accurate: Def,

[2] From personal communication with Peter Schachte.

implemented using a *dual Blake canonical form* ($\text{DBCF}_{\mathsf{Def}}$) representation specialised for Def, which has been implemented previously and found to be at least as fast, if not faster than ROBDDs [55].

Figure 5.4 shows the lattices for Def and Pos. The difference is that Def has no way of expressing boolean disjunction. However, in practice the two domains tend to give similar results.

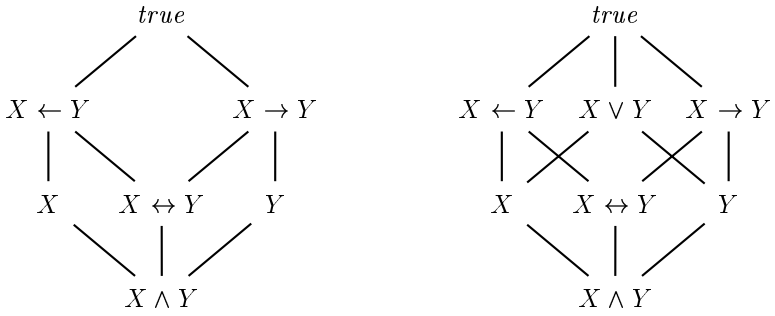


Figure 5.4: Two-variable Def and Pos lattices

5.3.1 $\text{DBCF}_{\mathsf{Def}}$ Representation and Operations

$\text{DBCF}_{\mathsf{Def}}$ is a simple representation for definite boolean functions. Since the implementation of Def is only intended to be for the short term, the following material is intended only as a brief overview of $\text{DBCF}_{\mathsf{Def}}$. Please see [55] for a more formal coverage.

Descriptions in $\text{DBCF}_{\mathsf{Def}}$ have the following form:

$$\bigwedge_{x \in Var} (x \leftarrow M_x)$$

where $(x \leftarrow M_x)$ is a *clause*, M_x is a conjunction of variables that does not contain x, and each x can appear on the left-hand side of more than one clause. For example, the formula:

$$((a \wedge b) \leftrightarrow c) \wedge d \wedge (e \leftarrow (f \vee g))$$

has this $\text{DBCF}_{\mathsf{Def}}$ representation:

$$(a \leftarrow c) \wedge (b \leftarrow c) \wedge (c \leftarrow (a \wedge b)) \wedge (d \leftarrow \mathit{true}) \wedge (e \leftarrow f) \wedge (e \leftarrow g).$$

$\text{DBCF}_{\mathsf{Def}}$ representations are minimal, e.g. $(x \leftarrow y) \wedge (x \leftarrow (y \wedge z))$ is normalised to $(x \leftarrow y)$. We will use a function *minimise* to normalise $\text{DBCF}_{\mathsf{Def}}$ descriptions. Also, all dependencies are made explicit, e.g. $(x \leftarrow y) \wedge (y \leftarrow z)$ is written $(x \leftarrow y) \wedge (x \leftarrow z) \wedge (y \leftarrow z)$.

The description derived from a unification is found using the following function:

$$\begin{aligned} \mathit{unify}_{\mathsf{Def}}(X = Y) &= (X \leftarrow Y) \wedge (Y \leftarrow X), \\ \mathit{unify}_{\mathsf{Def}}(X = f(Y_1, \ldots, Y_n)) &= (X \leftarrow (Y_1 \wedge \cdots \wedge Y_n)) \wedge (Y_1 \leftarrow X) \wedge \cdots \wedge (Y_n \leftarrow X). \end{aligned}$$

Important $\mathrm{DBCF}_{\mathsf{Def}}$ operations include the least upper bound (or *join*, written $\sqcup$), greatest lower bound (or *meet*, written $\sqcap$), and restriction. Assume that F and F' are two formulae in $\mathrm{DBCF}_{\mathsf{Def}}$. The join of F and F' is defined as:

$$F \sqcup F' = \mathit{minimise}(\bigwedge_{x \in Var} \{(x \leftarrow M_x \wedge M'_x) \mid x \leftarrow M_x \text{ is in } F \text{ and } x \leftarrow M'_x \text{ is in } F'\}).$$

The meet of F and F' is defined as follows, using the function *close* which performs the transitive closure of implication:

$$\begin{aligned}
&F \sqcap F' = \mathit{minimise}(\mathit{close}(F \wedge F')),\\
&\mathit{close}(F) = \text{if there exists clauses } (x \leftarrow (y \wedge M_x)) \text{ and } (y \leftarrow M'_x) \text{ in } F \text{ and}\\
&\qquad\qquad (x \leftarrow M'_x \wedge M_x)) \text{ is not in } F \text{ then}\\
&\qquad\qquad\quad \mathit{close}(F \wedge (x \leftarrow M'_x \wedge M_x))\\
&\qquad\qquad \text{else}\\
&\qquad\qquad\quad F.
\end{aligned}$$

Because all dependencies are explicit, a variable y can be restricted from F simply by removing any clauses that contain it:

$$\exists y.F = \bigwedge_{x \in Var} \{x \leftarrow M_x \mid y \text{ is not in } (x \leftarrow M_x)\}.$$

Note that the implementation uses a modification of $\mathrm{DBCF}_{\mathsf{Def}}$ for efficiency — each description consists of a term $\mathit{def}(G, \mathit{Deps})$ in which the definitely ground variables are stored in the set G, and the groundness dependencies are stored in $\mathrm{DBCF}_{\mathsf{Def}}$ in Deps. This is because it is frequently necessary to extract all the ground variables from a description (particularly for sharing and freeness analyses; see Sections 5.4.4 and 5.5.4). This changes the above operations slightly, but in a straightforward manner; for simplicity we will ignore this difference in what follows.

5.3.2 Integrating Mode Information

HAL's weak mode system greatly affects groundness analysis. A great deal of information is known about groundness from the mode declaration of each procedure. For example, the groundness answer for the procedure `append(in, in, out)`, is known without requiring any analysis — all three arguments are ground after the call. This is the only possible (and thus the best) answer for this procedure. We will describe answers like this that can be determined without analysis as *innate* answers. We will also use the symbol $\top^*$ to indicate the top-most description allowed with respect to other sources of information, such as mode declarations or previous analyses. For example, the calling description $\top^*$ for `append(X, Y, Z)` with the mode `(in, in, out)` is $\mathit{def}(\{X, Y\}, \emptyset)$; $\top^*$ for the mode `(oo, oo, oo)` is $\mathit{def}(\emptyset, \emptyset)$. Any calling pattern of the form $P : \top^*$ for which the answer is innate we will describe as *perfect*, or a *perfect call*. For example, the calling pattern `p(X)` : $\mathit{def}(\emptyset, \emptyset)$ for the procedure `p(og)` is perfect, whereas the calling pattern `p(X)` : $\mathit{def}(\{\mathtt{X}\}, \emptyset)$ is not, since $\mathit{def}(\{\mathtt{X}\}, \emptyset) \neq \top^*$.

When a perfect external call is encountered during analysis, an answer can be returned by Aexternal_proc immediately. When a non-perfect calling pattern for an external procedure with an innate answer is encountered, although an answer could be obtained immediately, a call must be made to `get_ext_answer` so that the calling pattern can be registered and an appropriate variant can be generated during multi-variant specialisation the next time that module is compiled. This is explained in detail in Section 7.4.

In addition to groundness information from declarations, mode analysis also records which variables are known to be ground at each program point. This information can also be used to augment groundness analysis.

To incorporate the extra mode information from procedure mode declarations and program point annotations into a groundness analysis we use the following operations.

- add_mode_ground(L, D) looks up the groundness information from mode analysis for the program point immediately after the literal L, and adds it to D.
- has_innate_answer(H) determines if a predicate has an innate answer, i.e. whether the output instantiation of every argument of H is known regardless of the calling context. This is true if every argument has a mode of the form `(new -> new)`, `(_ -> ground)`, or any other form in which the output instantiation forces the argument to become ground.[3]
- is_perfect_call$(P : D_P)$ determines if a calling pattern is perfect, i.e. if its calling pattern is $\top^*$ with respect to the groundness information known from the procedure's mode, and its answer is innate.
- output_mode_ground(H) returns an answer description based on the mode of a procedure, i.e. one in which every argument with a mode of the form `(_ -> ground)` (or another form with an output instantiation which forces the argument to become ground) is ground.

We will call Def augmented with groundness information from the program's modes by the name Def^{HAL}.

5.3.3 Definition of Def^{HAL}

The methods of the `abstract_domain` instance (the type class was defined in Section 4.3) for Def^{HAL} are given in Figure 5.5.

- Acomb returns the meet of the two descriptions (the variable sets are ignored). Since groundness information is downwards closed, the two variations of the $top_{\mathcal{D}}^{HAL}$ semantics discussed in Section 3.3.3 are equivalent.
- Aadd first obtains the description of the literal using $unify_{\mathsf{Def}}$, and then adds any extra groundness information known from mode analysis, and returns the meet of the two descriptions.

[3] For example `(new -> ground_list)`, where `ground_list` was defined in Section 2.4.1.

```
Acomb(_, D1, _, D2)
    return answer(D1 ⊓ D2)

Aadd(L, D)
    D_lit := unify_Def(L)
    D' := add_mode_ground(L, D)
    return answer(D' ⊓ D_lit)

Adisj(D1, D2)
    return D1 ⊔ D2

Aif_to_then(D)
    return D

Aif_then_else(_, D_th, D_el)
    return Adisj(D_th, D_el)

Aoutrestrict(V, D)
    foreach variable X in D not in V
        D := ∃X.D
    return D

Ainrestrict(V, D)
    return Aoutrestrict(V, D)

Aextend(_, D)
    return D

Ainitial_guess(H : _)
    if (has_innate_answer(H))
        D_Ans := output_mode_ground(H)
        return answer(D_Ans)
    else
        return unreached

Ais_constraint(_)
    return false

Acalling_descHO(_ : D_H)
    return D_H

Aexternal_proc(P : D_P)
    if (is_perfect_call(P : D_P))
        return output_mode_ground(H)
    else
        return get_ext_answer(P : D_P)
```

Figure 5.5: Def^{HAL} abstract operations

- Adisj returns the join of two descriptions. Groundness is an abstract domain in which Adisj is a least upper bound, and is thus a candidate for the optimisation described in Section 4.9.4. Hence it is shown as having two description arguments, rather than a single description set argument.
- Aif_to_then returns D untouched, since we are doing a top-down analysis.
- Aif_then_else is found by joining D_{th} and D_{el}. Its operation can be optimised in the same manner as Adisj. As mentioned in Section 4.9.4, this means the Abottom operation is not required for Def^{HAL}, and is thus omitted.
- Aoutrestrict restricts away the unwanted variables one at a time.
- Ainrestrict is the same as Aoutrestrict.
- Aextend does not change the description.
- Ainitial_guess returns an innate answer immediately, if a procedure has one. If not, it returns *unreached*.
- Ais_constraint returns *false*. No procedure calls need to be considered specially.
- Acalling_descHO returns the description D_H unchanged — because groundness is downwards closed, the information from D_H can be safely used.
- Aexternal_proc returns an innate answer immediately, if the calling pattern is perfect. If not, it looks up an external answer using `get_ext_answer`, which is defined in

Section 7.4.2. Calls to `call/n` are handled by appropriate entries in the `system.reg` file (explained in Chapter 7) that retain all information from the calling pattern; this is possible because groundness information is downwards closed.

5.4 Sharing Analysis in HAL

The sharing analysis used in the HAL compiler is based on the domain ASub. Descriptions in the ASub domain contain two components: a groundness description, and a structure sharing description. This domain was chosen because it is relatively simple, can be implemented quite efficiently, and fits within the framework more easily than a set-sharing based domain such as Sharing. Also, since ASub includes linearity information, its accuracy cannot be improved by freeness information (as opposed to Sharing), which means the sharing and freeness analyses are decoupled. This made their implementation and comparison easier. Finally, the structure sharing component of the ASub implementation could be reused in the future for compile-time garbage collection.

5.4.1 Groundness

Any groundness domain can be used with ASub, and the more accurate the groundness domain, the more accurate the sharing analysis will be. The obvious candidate is Pos, due to its accuracy and efficiency, and also for a third reason: it is *condensing* [38], which means that a goal-independent analysis will give exactly the same information as a goal-dependent analysis. For the purposes of ASub, a bottom-up Pos analysis could be performed before a top-down ASub analysis, and the groundness answers computed for each calling pattern could be used for the groundness component.

However, since the groundness analyser implemented uses Def^{HAL} which is not condensing, the groundness analysis must be combined with the sharing analysis to obtain the maximum accuracy possible.

5.4.2 Structure Sharing

The structure sharing domain (SS) is one domain for sharing analysis (see e.g [18]). A description is a set of variable pairs. The pair XY indicates that X and Y may share. The pair XX indicates that X may be non-linear.

The elements of the domain SS form a complete lattice, ordered by the subset relation $\subseteq$. The lattice for SS for descriptions with two variables is shown in Figure 5.6.
The interesting abstract operations for the domain SS (those required for ASub) are defined as follows:

$$
\begin{aligned}
&conj_{ss}(D, D') = \{XY \mid \text{there is a path from } X \text{ to } Y \text{ using arcs} \\
&\qquad\qquad\qquad \text{alternately from } D \text{ and } D'\}, \\
&add_{ss}(L, D) = conj_{ss}(\{XY_i \mid 1 \leq i \leq n\}, D), \\
&\qquad\qquad \text{where } L \text{ is } X = f(Y_1, \ldots, Y_n) \text{ or } X = Y_1, \\
&disj_{ss}(D_1, D_2) = D_1 \cup D_2,
\end{aligned}
$$

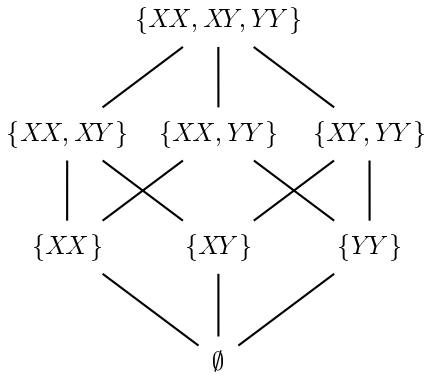


Figure 5.6: Two-variable SS lattice

$$restrict_{ss}(V, D) = \{XY \mid X, Y \in V \wedge XY \in D\},$$
$$comb_{ss}^{spec}(V, D, V', D') = \text{let } D'' = D \setminus \{XY \mid X, Y \in V'\} \text{ in } D'' \cup new_{ss}(D'', D'),$$
$$new_{ss}(D, D') = \bigcup_{XY \in D'} \{UV \mid (U = X \vee UX \in D) \wedge (Y = V \vee YV \in D)\},$$
$$top_{ss}(V, D) = \{XY \mid X, Y \in V\}.$$

The use of new_{ss} in the definition of $comb_{ss}^{spec}$ follows [18]. One problem with the definition of add_{ss} is that it uses $conj_{ss}$, which has worst-case behaviour that is exponential in the size of its arguments, and is difficult to program efficiently in practice. To obtain acceptable performance we can take advantage of the fact that the second argument of a call to $conj_{ss}$ from add_{ss} always has the form $\{XY_1\}$ or $\{XY_1, \ldots, XY_n\}$, depending on whether the unification is of the form $X = Y_1$ or $X = f(Y_1, \ldots, Y_n)$.

The specialised version of add_{ss} uses the operation new_{ss}. The new_{ss} operation provides an approximation to $conj_{ss}$ for the cases needed by add_{ss} but it misses some new sharing pairs, because the abstractions of $X = Y_1$ ($\{XY_1\}$) and $X = f(Y_1, \ldots, Y_n)$ ($\{XY_1, \ldots, XY_n\}$) do not take into account the information in D. We can avoid missing such non-linear pairs by adding some extra pairs to these abstractions to take into account the information in D, leading to this definition of add_{ss}:

$$add_{ss}(X = Y_1, D) =$$
$$\text{let } D' = \begin{cases} \{Y_1Y_1\} & \text{if } XX \in D \\ \emptyset & \text{otherwise} \end{cases} \text{ in}$$
$$\text{let } D'' = \begin{cases} \{XX\} & \text{if } Y_1Y_1 \in D \\ \emptyset & \text{otherwise} \end{cases} \text{ in}$$
$$D \cup new_{ss}(D, \{XY_1\} \cup D' \cup D''),$$

$$add_{ss}(X = f(Y_1, \ldots, Y_n), D) =$$
$$\text{let } D' = \begin{cases} \{AB \mid A, B \in \{Y_1, \ldots, Y_n\}\} & \text{if } XX \in D \\ \emptyset & \text{otherwise} \end{cases} \text{ in}$$
$$\text{let } D'' = \begin{cases} \{XX\} & \text{if } \exists i, j : Y_iY_j \in D \\ \emptyset & \text{otherwise} \end{cases} \text{ in}$$

$$D \cup new_{ss}(D, \{XY_1, \ldots, XY_n\} \cup D' \cup D'').$$

Note that the definition of $add_{ss}(X = Y_1, D)$ is equivalent to that for $add_{ss}(X = f(Y_1), D)$.

5.4.3 Groundness + Structure Sharing = ASub

A description in the ASub domain is a term, $asub(G, SS)$, containing a groundness description and a structure sharing description. The combination of the groundness and SS operations is quite straightforward. If a variable is ground, it cannot share with any other variable; whenever a variable becomes ground, we simply remove any sharing pairs that involve it. For example, if the SS component of the description at the program point prior to the unification `X = a` is $\{XY\}$, afterwards it will be $\emptyset$. To do this we will use a function unshare_ground_vars(SS, G) which removes any ground variables in G from SS.

The only interesting aspect of the combination of the two domains involves the exact order of the operations; roughly speaking, we want to remove any sharing pairs due to newly ground variables as soon as possible in order to make the sharing operations work on the smallest sets possible.

We will call ASub augmented with groundness information from the program's modes by the name ASub^{HAL}.

5.4.4 Definition of ASub^{HAL}

The methods of the `abstract_domain` instance for ASub^{HAL} are given in Figure 5.7. For generality, they are defined in terms of the groundness operations $comb_G$, add_G, $disj_G$, $restrict_G$ and $bottom_G$ — the implementation uses Def^{HAL}, although any groundness domain could be used.

- Acomb combines the groundness descriptions, and uses the result to remove any sharing pairs containing a variable that is now ground from SS_1 and SS_2, which are then combined. Since it uses a specialised $comb_{ss}^{spec}$ operation, ASub^{HAL} is an example of a domain that uses the second variation of the $top_{\mathcal{D}}^{HAL}$ semantics discussed in Section 3.3.3.
- Aadd follows the same pattern as Acomb, except that add_G and add_{ss} are used instead of $comb_G$ and $comb_{ss}$.
- Adisj simply disjoins the two components piecewise. As in Section 5.3.3, Adisj is a least upper bound and is shown having two arguments rather than a single set argument.
- Aif_to_then returns D untouched, since we are doing a top-down analysis.
- Aif_then_else is found by joining D_{th} and D_{el}; its operation can be optimised in the same manner as Adisj. As in Section 5.3.3, this means Abottom is not required and is thus omitted.
- Aoutrestrict restricts the two components piecewise.
- Ainrestrict is the same as Aoutrestrict.

Acomb(V_1, $asub(G_1, SS_1)$, V_2, $asub(G_2, SS_2)$)
  $G_3 := comb_G(V_1, G_1, V_2, G_2)$
  $SS_3 :=$ unshare_ground_vars(SS_1, G_3)
  $SS_4 :=$ unshare_ground_vars(SS_2, G_3)
  $SS_5 := comb_{ss}^{spec}(V_1, SS_3, V_2, SS_4)$
  return $answer(asub(G_3, SS_5))$

Aadd(L, $asub(G, SS)$)
  $G' := add_G(L, G)$
  $SS' :=$ unshare_ground_vars(SS, G')
  $SS'' := add_{ss}(L, SS')$
  return $answer(asub(G', SS''))$

Adisj($asub(G_1, SS_1)$, $asub(G_2, SS_2)$)
  $G := disj_G(G_1, G_2)$
  $SS := disj_{ss}(SS_1, SS_2)$
  return $asub(G, SS)$

Aif_to_then(D)
  return D

Aif_then_else(_, D_{th}, D_{el})
  return Adisj(D_{th}, D_{el})

Aoutrestrict(V, $asub(G, SS)$)
  $G' := restrict_G(V, G)$
  $SS' := restrict_{ss}(V, SS)$
  return $asub(G', SS')$

Ainrestrict(V, D)
  return Aoutrestrict(V, D)

Aextend(_, D)
  return D

Ainitial_guess(H : _)
  if (has_innate_answer(H))
    $G_{Ans} :=$ output_mode_ground(H)
    return $answer(asub(G_{Ans}, \emptyset))$
  else
    return *unreached*

Ais_constraint(_)
  return *false*

Acalling_descHO(V_H : $asub(G_H, \_)$)
  $SS_\top := top_{ss}(V_H)$
  $SS' :=$ unshare_ground_vars($SS_\top$, G_H)
  return $asub(G_H, SS')$

Aexternal_proc(P : D_P)
  if (is_perfect_call(P : D_P))
    $G_{Ans} :=$ output_mode_ground(H)
    return $asub(G_{Ans}, \emptyset)$
  else
    return `get_ext_answer`(P : D_P)

Figure 5.7: ASub^{HAL} abstract operations

- Aextend does not change the description.

- Ainitial_guess returns an innate answer immediately, if a procedure has one. If not, it returns *unreached*.

- Ais_constraint returns *false*. No procedure calls need to be considered specially.

- Acalling_descHO starts by finding $\top$ for SS over the head variables of the procedure captured by the closure — any sharing information about the captured variables from the calling description must be ignored, since it may not still be valid once the closure is called because SS is upwards closed. Then we remove any variables known to be ground, which we can do since the groundness component is downwards closed, so any information we have about groundness will still be valid when the closure is called.

- Aexternal_proc returns an innate answer immediately, if the calling pattern is perfect. If not, it looks up an external answer using `get_ext_answer`, which is defined in Section 7.4.2. Calls to `call/n` are handled by appropriate entries in the `system.reg` file (explained in Chapter 7) that return $\top$ over its variables except that groundness information can be kept (because groundness information is downwards closed).

5.5 Freeness Analysis in HAL

The domain used for freeness analysis in the HAL compiler is $\mathsf{Freeness}^{HAL}$. Descriptions in $\mathsf{Freeness}^{HAL}$ have the form $\mathit{free}(S, F, L)$, where S is a description from a sharing domain, F is a set of definitely free variables, and L is a set of definitely lonely variables.

5.5.1 Sharing

Accurate freeness analysis requires some form of sharing analysis. Any sharing domain could be used, but the more accurate the sharing analysis, the more accurate the freeness analysis will be. Our implementation uses ASub^{HAL}. Sharing information is necessary for accurate freeness analysis because when a free variable becomes bound, any variables that share with it are no longer definitely free.

5.5.2 Freeness and Loneliness

Figure 5.8 shows the freeness and loneliness lattice for two variables that do not share. Each element in the lattice is a (free set, lonely set) pair. Since any lonely variable is also free, the set of lonely variables is always a subset of the free variables; the lattice is ordered by the pairwise superset $(\supseteq, \supseteq)$ relation.

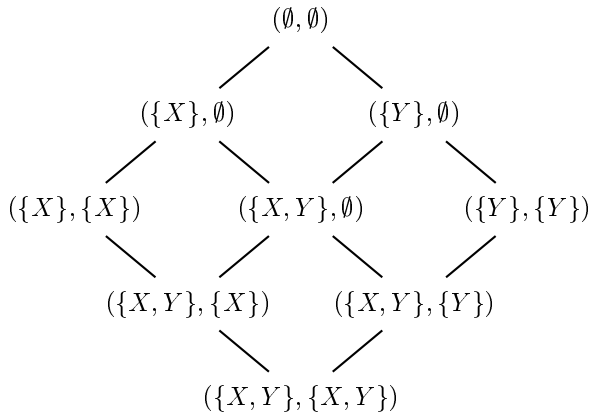


Figure 5.8: Two-variable (free set, lonely set) lattice

Note that the properties of freeness and loneliness are only applicable to `old` variables of types declared as `herbrand` (i.e. possibly non-ground variables that have been seen by the Herbrand constraint solver). They should not be confused with `herbrand` variables that are `new`, which have not been seen by the solver; such variables do not need to be considered in freeness descriptions, since they are never involved in `unify_oo` unifications, which is what we are trying to optimise. For example, consider the following procedure, as seen after the call to `herbrand_init` is inserted by the compiler.

```
:- herbrand list/1.
:- pred p(list(T), list(T)).
:- mode p(no, oo) is semidet.
```

```
p(X, Y) :- ①
    herbrand_init(X), ②
    X = Y. ③
```

Assume that the calling description is $(\{\texttt{Y}\}, \emptyset)$ upon entry to the procedure (`X` is not mentioned in the description as it is `new`). The (free set, lonely set) descriptions for each program point are as follows:

① = $(\{\texttt{Y}\}, \emptyset)$,
② = $(\{\texttt{X}, \texttt{Y}\}, \{\texttt{X}\})$,
③ = $(\{\texttt{X}, \texttt{Y}\}, \emptyset)$.

`X` becomes lonely after the call to `herbrand_init`, and then loses its loneliness when unified with `Y`.

5.5.3 Handling Unifications

The most interesting operation for Freeness^{HAL} is Aadd. For unifications of the form $X = Y$ the action taken depends on the state of the involved variables. Table 5.2 shows the (free set, lonely set) pair for X and Y after a $X = Y$ unification for all the possible combinations. The '*' indicates that any free variables that share with X must also be removed from the free set. The '**' indicates that any free variables that share with either X or Y must be removed from the free set. Empty entries are equivalent to the symmetric case.

HAL's mode system does not allow `new`–`new` unifications. If one variable is `new` and the other is lonely or free, the `new` variable also becomes lonely or free. It may sound strange for a variable to be lonely after it has been involved in a unification; however, recall that a unification involving a `new` variable is an assignment, and the content of the non-`new` variable is copied into the `new` variable. This results in the variable which was `new` pointing to the lonely variable, which has not been touched and is thus still lonely.

If both variables are lonely or free, they both end up free. If one variable is lonely or free and the other is not, the lonely/free variable no longer is, nor are any variables with which it shares (although recall that a lonely variable cannot share with anything else). If neither variable is free, we must unfree any variables that share with either of them.

For unifications of the form $X = f(Y)$ the possible results are shown in Table 5.3. This table can be extended to unifications of the form $X = f(Y_1, \ldots, Y_n)$ in the obvious manner. The '*' and '**' annotations have the same meaning as for Table 5.2.

X is bound by an $X = f(Y)$ unification, and thus is never free or lonely afterwards. If X shares with other free variables (possible if it is not `new` or lonely), they are also removed from the free set. Y is never lonely afterwards[4] and is only free afterwards if it was free or lonely beforehand, and X was `new`, free or lonely.

Note that neither of these tables mention the sharing component of the Freeness^{HAL} description, which must be computed separately.

[4] If X is `new`, lonely or free the code for an $X = f(Y)$ unification could be generated in a way that preserves Y's loneliness, but currently it is not.

$X \setminus Y$	new	lonely	free	other
new	—	$(\emptyset, \{X, Y\})$	$(\{X, Y\}, \emptyset)$	$(\emptyset, \emptyset)$
lonely		$(\{X, Y\}, \emptyset)$	$(\{X, Y\}, \emptyset)$	$(\emptyset, \emptyset)$
free			$(\{X, Y\}, \emptyset)$	$(\emptyset, \emptyset)$*
other				$(\emptyset, \emptyset)$**

Table 5.2: $X = Y$ for Freeness^{HAL}

$X \setminus Y$	new	lonely	free	other
new	$(\emptyset, \emptyset)$	$(\{Y\}, \emptyset)$	$(\{Y\}, \emptyset)$	$(\emptyset, \emptyset)$
lonely	$(\emptyset, \emptyset)$	$(\{Y\}, \emptyset)$	$(\{Y\}, \emptyset)$	$(\emptyset, \emptyset)$
free	$(\emptyset, \emptyset)$*	$(\{Y\}, \emptyset)$*	$(\{Y\}, \emptyset)$*	$(\emptyset, \emptyset)$*
other	$(\emptyset, \emptyset)$*	$(\emptyset, \emptyset)$*	$(\emptyset, \emptyset)$**	$(\emptyset, \emptyset)$**

Table 5.3: $X = f(Y)$ for Freeness^{HAL}

5.5.4 Definition of Freeness^{HAL}

The methods of the `abstract_domain` instance for Freeness^{HAL} are given in Figure 5.9. For generality, they are defined in terms of the sharing operations $comb_S$, add_S, $disj_S$, $restrict_S$, top_S, $bottom_S$ and $calling\_descHO_S$ — the implementation uses ASub^{HAL}, although any sharing domain could be used. The function unfree_sharers(F, V, S) removes from the free set F any variables that share (according to S) with any variables in V. The function output_mode_ground is redefined from earlier to return a sharing description containing the variables that are definitely ground upon procedure exit.

- Acomb first combines the sharing descriptions. To find the free and lonely variables, it removes all variables of the call from the pre-descriptions, and then adds back those variables known to still be lonely or ground after the call. If any of the call variables are not free after the call, any variables they shared with before the call must be removed from the free set, as they may now be bound.

  Freeness^{HAL} does not directly use a specialised Acomb operation for the freeness and loneliness components, although the sharing Acomb operation may be specialised.

- Aadd treats one "constraint" specially — the argument of a call to `herbrand_init` is added to the lonely set. Otherwise, unifications are handled as described in Table 5.2 and Table 5.3.

- Adisj first disjoins the sharing components. It then disjoins the free and lonely sets using intersection. As in Sections 5.3.3 and 5.4.4, Adisj is a least upper bound, and is shown having two arguments rather than a single set argument.

- Aif_to_then returns D untouched, since we are doing a top-down analysis.

- Aif_then_else is found by joining D_{th} and D_{el}; its operation can be optimised in the same manner as Adisj. As in Sections 5.3.3 and 5.4.4, this means Abottom is not required and is thus omitted.

```
Acomb(V1, free(S1, F1, L1), V2, free(S2, F2, L2))
    S3 := comb_S(V1, S1, V2, S2)
    F3 := (F1 \ V2) ∪ F2
    L3 := (L1 \ V2) ∪ L2
    F4 := unfree_sharers(F3, V2 \ F2, S1)
    return answer(free(S3, F4, L3))

Aadd(Lit, free(S, F, L))
    if (Lit = literal(herbrand_init(X))
        return free(S, F, L ∪ {X})
    else
        % See Tables 5.2 and 5.3

Adisj(free(S1, F1, L1), free(S2, F2, L2))
    S3 := disj_S(S1, S2)
    F3 := F1 ∩ F2
    L3 := L1 ∩ L2
    return free(S3, F3, L3)

Aif_to_then(D)
    return D

Aif_then_else(_, Dth, Del)
    return Adisj(Dth, Del)

Aoutrestrict(V, free(S, F, L))
    S' := restrict_S(V, S)
    F' := V ∩ F
    L' := V ∩ L
    return free(S', F', L')

Ainrestrict(V, D)
    return Aoutrestrict(V, D)

Aextend(V, D)
    return D

Ainitial_guess(H : _)
    if (has_innate_answer(H))
        S_Ans := output_mode_ground(H)
        return answer(free(S_Ans, ∅, ∅))
    else
        return unreached

Ais_constraint(A)
    if (A = herbrand_init(_))
        return true
    else
        return false

Acalling_descHO(V_H : free(S_H, _, _))
    S' := calling_descHO_S(V_H, S_H)
    return free(S', ∅, ∅)

Aexternal_proc(P : D_P)
    if (is_perfect_call(P : D_P))
        S_Ans := output_mode_ground(H)
        return free(S_Ans, ∅, ∅)
    else
        return get_ext_answer(P : D_P)
```

Figure 5.9: Freeness^{HAL} abstract operations

- Aoutrestrict restricts the three components piecewise.
- Ainrestrict is the same as Aoutrestrict.
- Aextend does not change the description.
- Ainitial_guess returns an innate answer immediately, if a procedure has one. If not, it returns *unreached*.
- Ais_constraint only returns *true* if the call is to `herbrand_init` — the only "constraint" considered specially.
- Acalling_descHO starts by finding the appropriate description for the sharing component. The freeness and loneliness components must be returned as the empty set, because Freeness^{HAL} is an upwards closed domain, so no information can be safely retained.
- Aexternal_proc returns an innate answer immediately, if the calling pattern is perfect. If not, it looks up an external answer using `get_ext_answer`, which is defined in Section 7.4.2. Calls to `call/n` are handled by appropriate entries in the `system.reg`

Module	Preds	Lits	G	S	F	All(F)	Prop(F)
`fast_mu`	8	41	920	1930	2170	7030	30.9%
`hanoidiff`	6	8	30	60	90	1150	8%
`qsortdiff`	11	22	90	110	140	11140	1.3%
`zebra`	7	25	130	240	270	3590	7.5%
`icomp`	30	116	1160	4220	5910	12940	45.6%
`term`	121	502	1620	1830	2080	32830	6.3%
`chrcmp`	86	870	2300	2580	2760	60250	4.6%
`term2body`	119	994	2720	3850	6100	53520	11.4%
`mode_analysis_framework`	122	1490	3710	4500	4920	69910	7.0%

Table 5.4: Cost of Freeness^{HAL} analysis (ms)

file (explained in Chapter 7) that return $\top$ over its variables except that groundness information can be kept (because groundness information is downwards closed).

5.6 Experimental Analysis Evaluation

This section evaluates HAL's groundness, sharing and freeness analyses by considering the time taken during compilation to perform each analysis, and the performance improvements for programs optimised using the obtained information.

5.6.1 Cost of Analysis

To determine the compile-time cost of groundness, sharing and freeness analysis, a number of modules of varying sizes were tested. These were: four small Prolog benchmarks `fast_mu`, `hanoidiff`, `qsortdiff` and `zebra` (also used to determine the effect of the Herbrand optimisations; see Section 5.6.2); one slightly larger program `icomp`; and four large modules from the HAL compiler that use Herbrand constraint solving, `term`, `term2body`, `chrcmp` and `mode_analysis_framework`. The tests were performed on an Intel Pentium II-400MHz with 384MB of RAM, running Red Hat Linux (version 6.0, kernel version 2.2).

As mentioned in Section 2.4.3, the HAL compiler is currently executed through SICStus Prolog, as it has not yet bootstrapped. These experiments used SICStus version 3.8.6 (compact code) with garbage collection on. Because the analysis framework uses the `abstract_domain` type class, and SICStus does not support type classes, it is currently only possible to use one abstract domain within the framework at a time. This means the analysis times given do not include the determinism analysis phase, since it is also implemented within the analysis framework (see Chapter 6).

Each module was compiled five times in a row, and the fastest time recorded. All times are SICStus run-times, measured in milliseconds. In all cases, the variation over the five compilations was less than 1%.

Table 5.4 shows the cost of analysing these modules. Columns two and three give the number of predicates and literals (before normalisation) in each module. Columns four, five and six give the analysis times for Def^{HAL}, ASub^{HAL} and Freeness^{HAL} respectively. Column

seven gives the overall compilation time, including $\mathsf{Freeness}^{HAL}$ analysis, and column eight gives the proportion of compilation time taken up by $\mathsf{Freeness}^{HAL}$ analysis (i.e. column six divided by column seven). Note that the overall compilation time is only the time taken by the HAL compiler; to run the programs, the generated Mercury code must then be compiled by the Mercury compiler.

Since Def^{HAL} is incorporated into ASub^{HAL}, which is incorporated into $\mathsf{Freeness}^{HAL}$, it is not surprising to see that the analysis times increase without exception when moving from Def^{HAL} to ASub^{HAL} to $\mathsf{Freeness}^{HAL}$. Generally, the ASub^{HAL} times are roughly 1.5–2 times longer than the Def^{HAL} times, and the $\mathsf{Freeness}^{HAL}$ times are slightly longer than the ASub^{HAL} times.

Of the four benchmarks, `fast_mu` uses `old` terms the most, and pays the price in analysis time. A disproportionately long mode analysis phase[5] skews the result downwards for `qsortdiff`. Of all the tested modules, `icomp` uses `old` terms the most — almost all its predicate arguments are `oo` — and it easily has the highest analysis time. Among the four compiler modules, the analysis times reflect quite accurately the proportion of old terms used — only one predicate in `chrcmp` has `oo` and `no` arguments, `term2body` uses them in a number of places, and `term` and `mode_analysis_framework` are in between.

The obvious conclusion from these figures is that freeness analysis is not too expensive when Herbrand constraint solving is used sparingly, but that the cost jumps quite quickly as `old` terms are used more frequently, providing an argument in favour of using ground terms wherever possible.

It should be noted that the compilation times for the HAL compiler are currently quite high. Most of this time is taken up by reading and pre-processing the source file and interface files, and performing type and mode analysis. However, Mercury programs are much faster than SICStus programs (see [11, 56] for figures); once the compiler bootstraps and runs as a HAL/Mercury executable rather than a SICStus program, we conservatively expect it will run 2–3 times faster, giving compilation times similar to those of the Mercury compiler when compiling similarly sized Mercury programs. The relative cost of the Herbrand analyses may change when this happens, although the difference is likely to be small.

5.6.2 Effect of Optimisations

This section evaluates the potential benefits of the Herbrand optimisations described in Section 5.2.3 by measuring the difference in cost between `unify_oo` and the specialised unifications. It then measures the actual benefits on a small number of benchmarks.

All the experiments in this section were performed on the same machine as those in Section 5.6.1. The tested programs were compiled by the Melbourne Mercury Compiler (July 2001 development version), and run with garbage collection turned off.[6] Each program was executed ten times, and the fastest elapsed time (user + system, reported by GNU `time` version 1.7) was recorded. In all cases, the maximum variation over each program's execution

[5] This is due to the presence of a large constant list, which is normalised into many simple unifications, which the current mode analysis algorithm does not handle very well.

[6] With garbage collection on, the times are much less consistent.

times was 10 milliseconds.

Relative unification speeds: The first experiments performed were intended to determine how much faster than `unify_oo` the specialised unifications are, and thus identify the maximum benefits the Herbrand optimisations from Section 5.2.3 would allow. To do this, a program was written that performed a large number of identical unifications many times. Using this program, four tests were performed.

1. 1,000,000 unifications of two lonely variables.
2. 1,000,000 unifications of a lonely variable with a ground term.
3. 250,000 unifications of two free variables, where each variable was part of a ten variable cycle chain, and the two cycles did not share.
4. 250,000 unifications of two ground terms of the form `[yes, no, yes, no, yes, no, yes, no, yes, no]`, where `yes` and `no` are the functors of the built-in `yesno` type.

Every unification from Table 5.1 was tested by one or more of these tests. For example, the unifications suitable for the lonely–ground unification were tested with the following predicate:

```
:- pred loop_LG(int::in, list(T)::oo) is semidet <= herbrand(T).
loop_LG(N, Y) :-
    ( N = 0 ->
        true
    ;
        <unify>(X,Y),
        N1 = N - 1,
        loop_LG(N1, Y)
    ).
```

The argument `Y` used was a ground list. The generated Mercury code was hand-edited for each run so that `<unify>` was set to, in turn: `unify_oo`, `unify_var_old`, `unify_var_val`, `unify_var1_old`, `unify_var1_val`, `unify_val_old`, and `unify_gnd_old` (the arguments were swapped for the last two). Predicates similar to `loop_LG` were used for the other three tests. An extra run was performed for each test in which `<unify>` was set to `null`, an empty C function taking two arguments; this was used to determine the cost of the loop operations. By subtracting this figure from each time result, we obtained a measure of the time taken by the unifications alone.

The results are summarised in Tables 5.5–5.8. In each table, column one gives the unification used (or `null`). Column two gives each run's execution time. Column three gives the execution time minus the time of the loop operations (i.e. minus the T value for `null`). Column four gives the improvement factor over the unspecialised unification `unify_oo`, where applicable. All times are in milliseconds.

Table 5.5 shows the times for 1,000,000 lonely–lonely unifications. The cost of the loop itself (120 ms) included allocating and initialising two lonely variables for each unification. The interesting results were as follows.

Unification	T	T'	Fctr
`null`	120	0	–
`oo`	740	620	1
`var_old`	580	460	1.3
`var_var`	500	380	1.6
`oo_noshare`	730	610	1.0
`var_old_noshare`	570	450	1.4
`var_var_noshare`	490	370	1.7
`var1_old`	560	440	1.4
`var1_var`	480	360	1.7

Table 5.5: 1,000,000 lonely–lonely unifications (ms)

Unification	T	T'	Fctr
`null`	160	0	–
`oo`	650	490	1
`var_old`	430	270	1.8
`var_val`	390	230	2.1
`var1_old`	420	260	1.9
`var1_val`	380	220	2.2
`val_old`	500	340	1.4
`gnd_old`	540	380	1.3

Table 5.6: 1,000,000 lonely–ground unifications (ms)

Unification	T	T'	Fctr
`null`	1870	0	–
`oo`	2090	220	1
`var_old`	2050	180	1.2
`var_var`	2030	160	1.4
`oo_noshare`	2020	150	1.5
`var_old_noshare`	1980	110	2.0
`var_var_noshare`	1960	90	2.4

Table 5.7: 250,000 free–free unifications (ms)

Unification	T	T'	Fctr
`null`	20	0	–
`oo`	1650	1630	1
`val_old`	1630	1610	1.01
`val_val`	1600	1580	1.03
`gnd_old`	1260	1240	1.31
`val_gnd`	1150	1130	1.44
`gg`	340	320	5.09

Table 5.8: 250,000 ground–ground unifications (ms)

- Removing one or both `nonvar/1` tests made the biggest difference, as shown by the difference between `unify_oo` (two tests), `unify_var_old` (one test) and `unify_var_var` (zero tests). Avoiding the first test saved 160 ms, avoiding the second saved only 80 ms. The corresponding differences between `unify_oo_noshare`, `unify_var_old_noshare` and `unify_var_var_noshare` were also 160 ms and 80 ms. The difference between `unify_var1_old` and `unify_var1_var` (one test avoided) was 80 ms. In all cases, the avoided `nonvar/1` tests in the condition failed, so the else case was executed.

- The `noshare` version of each unification is marginally (10 ms) faster than the normal version, which demonstrates that the single cycle traversal step performed by the normal versions for lonely variables is not expensive. For the same reason, both `var1` unifications are only slightly (20 ms) faster than their `var` counterparts.

- The unification `unify_var1_var` is just (10 ms) faster than `unify_var_var_noshare`, which shows the effect of the single pointer dereference avoided by `unify_var1_var`, which was explained in Section 5.2.3.

Table 5.6 shows the times for 1,000,000 lonely–ground unifications. The arguments of `unify_val_old` and `unify_gnd_old` were swapped so they were a ground–lonely unification. The pertinent results were as follows.

- The difference between the entries for `unify_oo`, `unify_var_old` and `unify_var_val` show that avoiding the first `nonvar/1` test (which failed) saved 220 ms, whereas avoiding the second (which succeeded) saved only 40 ms. For `unify_oo`, `unify_val_old`

and `unify_var_val`, avoiding the first test (which succeeded) saved 150 ms, avoiding the second (which failed) saved 110 ms. The difference between `unify_var1_old` and `unify_var1_val` (one successful test avoided) was only 40 ms.

- The unifications `unify_var1_old` and `unify_var1_val` were each only 10 ms faster than `unify_var_old` and `unify_var_val` respectively. This confirms that the single cycle traversal step performed by `unify_var_old` and `unify_var_val` for lonely variables is not expensive.

- The unifications `unify_val_old` and `unify_gnd_old` took quite different times (40 ms difference), which is surprising since they do exactly the same thing when the `old` argument is a variable (the `nonvar/1` test fails, then `unify_var_val` is called). There is no obvious reason for this difference.

Table 5.7 shows the times for 250,000 free–free unifications, where each free variable is in a cycle of length ten, and the two cycles do not share. The cost of the loop itself was quite high for this test (1870 ms) because twenty variables were allocated and initialised by `herbrand_init` for each unification. The notable results were as follows.

- Looking at `unify_oo`, `unify_var_old` and `unify_var_var`, again we see that avoiding the `nonvar/1` tests makes quite a difference — avoiding the first test saved 40 ms, avoiding the second test saved 20 ms (the times are smaller because only 250,000 unifications were performed for this test). The corresponding savings for the `noshare` versions were also 40 and 20 ms. All the avoided `nonvar/1` tests were ones that failed.

- The `noshare` unifications which avoid the unnecessary traversal of variable cycles are much faster than their normal counterparts (all by 70 ms). Admittedly, the `noshare` benefit is likely to be much less in practice; cycle lengths greater than one were rare in the programs tested in [11], and both cycles must be large for the saving to be significant.

Table 5.8 shows the times for unifying 250,000 ground–ground lists of length ten. The cost of the loop itself was very small compared to the other tests because no variables needed to be initialised. The salient results were as follows.

- The difference between `unify_oo`, `unify_val_old` and `unify_val_val` show that avoiding the first `nonvar/1` test saved 20 ms, whereas avoiding the second saved 30 ms (again the times are smaller because only 250,000 unifications were performed). Both of the avoided `nonvar/1` tests were ones that succeeded.

- The unifications in which one argument was known to be ground are clearly faster. This is because multiple `nonvar/1` tests can be avoided.

- Reverting to Mercury's `unify_gg` when both arguments were ground made a huge difference. This is primarily because when comparing two lists of ten elements that are known to be ground, 42 `nonvar/1` tests can be avoided (for each argument, 11 for the list backbone and 10 for the elements). If the terms were larger, the difference would be even greater.

Unification	T	T'	Fctr
`null`	120	0	–
`oo`	740	620	1
`var_var`	500	380	1.6
`var_var_notrail`	230	110	5.6
`var1_var`	480	360	1.7
`var1_var_notrail`	200	80	7.8

Table 5.9: 1,000,000 lonely–lonely `notrail` unifications (ms)

Unification	T	T'	Fctr
`null`	160	0	–
`oo`	650	490	1
`var_val`	390	230	2.1
`var_val_notrail`	240	80	6.1
`var1_val`	380	220	2.2
`var1_val_notrail`	230	70	7.0

Table 5.10: 1,000,000 lonely–ground `notrail` unifications (ms)

- The improvement from `unify_gnd_old` to `unify_gg` is larger than from `unify_oo` to `unify_gnd_old`. Yet each specialised unification avoids the same number of `nonvar/1` tests. The likely reason for the disparity is that several coercions (which were not shown in Section 5.2.3) are required for Mercury's mode analysis to believe that the sub-lists are bound at each unification step; they are not required for `unify_gg`.

It is also worthwhile to compare the relative speeds of unifications between tables. We see from Tables 5.5 and 5.6 that `unify_var_var` is 150 ms slower than `unify_var_val` when the first argument is lonely, and that `unify_var1_var` is 140 ms slower than `unify_var1_val`. Given that variable cycle traversal is cheap, the likely cause of the differences is that `unify_var_var` and `unify_var1_var` both trail two variables, whereas `unify_var_val` and `unify_var1_val` only trail one. To confirm this, we repeated the lonely–lonely test for `unify_var_var` and `unify_var1_var`, and the lonely–ground test for `unify_var_val` and `unify_var1_val`, but this time without trailing the variables.

Table 5.9 shows the times for 1,000,000 lonely–lonely unifications without trailing. Only the `notrail` unification results are new — the others are reproduced from Table 5.5. Removing both calls to `trail/1` made a large improvement of 270 and 280 ms to `unify_var_var` and `unify_var1_var`. This makes `unify_var_var_notrail` an impressive 3.5 times faster than `unify_var_var`, and `unify_var1_var_notrail` 4.5 times faster than `unify_var1_var`.

Table 5.10 shows the times for 1,000,000 lonely–ground unifications without trailing. Again, only the `notrail` unification results are new, the others being reproduced from Table 5.6. Removing the one `trail/1` call from `unify_var_val` and `unify_var1_val` saved 150 ms in both cases. This was about half the saving made by removing two variable trailings from `unify_var_var` and `unify_var1_var`, showing the improvement to be consistent. We see that `unify_var_val_notrail` is 2.9 times faster than `unify_var_val`, and `unify_var1_val_notrail` is 3.1 times faster than `unify_var1_val`.

In practice, when garbage collection is on, the `notrail` versions would be even faster relative to the normal versions because they require no memory for trailing the variables.

Tables 5.9 and 5.10 show that all four `notrail` unifications take quite similar times (70–110 ms), which makes sense since they have all been reduced to a handful of assignments and tests. There is very little scope for further optimisation of these unifications; indeed, `unify_var1_val_notrail` performs only a single C assignment.

Variable trailing is only necessary if a unification can be backtracked over. An analysis phase could be implemented in order to identify which unifications do not need their variables

trailed. This could be done straightforwardly using sharing and determinism (see Chapter 6) information.

We are now in a position to draw our conclusions. The results in Tables 5.5–5.10 show there are five main areas of benefit from these specialisations, of increasing worth.

1. The `var1` unifications provide only the slightest of improvements over the `var` unifications, especially `unify_var_var_noshare`. The difference is 10 ms per 1,000,000 unifications.

2. If both arguments are variables in long chains, `unify_var_var_noshare` is much faster than `unify_var_var` — almost twice as fast for chains of length ten — but that's rare and otherwise the difference is very small (e.g. 10 ms per 1,000,000 unifications if either argument is lonely).

3. Avoiding `nonvar/1` tests makes quite a difference, although the pay-off varies wildly. The cost of `nonvar/1` is 40–220 ms for 1,000,000 tests, with the average being around 110 ms. Avoiding tests that fail seems to save more time (80–220 ms, average about 125 ms for 1,000,000) than avoiding tests that succeed (40–150 ms, average about 85 ms for 1,000,000).

4. Avoiding unnecessary variable trailing makes a consistent difference of 135–150 ms per 1,000,000 variables trailed; the tested `notrail` unifications were 2.9–4.5 times faster than `unify_oo`. This indicates that an analysis that identifies when variable trailing can be avoided could provide substantial improvements.

5. Using Mercury's `unify_gg` where possible can make a big difference; particularly for large terms. This shows that when programming with performance in mind, ground terms should be used where possible.

Note that these are not the only optimisations enabled by Herbrand information; other potential optimisations are covered briefly in Section 5.7.

Benchmarks: Having carefully analysed the relative costs of all the different unifications, we are ready to examine the effect of these Herbrand optimisations on non-manufactured benchmark programs. Only a small number of modest benchmarks were analysed to evaluate the effect of the Herbrand optimisations on HAL programs. There are two main reasons for this.

1. At the time of writing, the HAL compiler's code generator was being extensively rewritten. The old code generator did not produce satisfactory code with respect to Herbrand optimisations. To work around this, the generated Mercury files were hand-edited somewhat to be as optimised as an improved code generator would make them. This meant that benchmarks tested had to be quite small.

2. As we have seen, Mercury's (and therefore HAL's) basic term manipulations are extremely fast. Therefore programs written with performance in mind should only use Herbrand constraint solving when the basic term manipulations cannot be used. Only

Module	Preds	Lits	oo calls	NoOpt	Opt	Factor
fast_mu	5	30	2,980,000	1930	1780	1.08
hanoidiff	2	6	2,620,000	4840	4240	1.14
qsortdiff	3	10	25,000	1190	1170	1.02
zebra	5	20	118,000	140	140	1.00

Table 5.11: Effect of Freeness^{HAL} analysis (ms)

programs that use true Herbrand constraint solving can benefit from Herbrand optimisations, and surprisingly few Prolog programs fall into this category.

Finding programs that used Herbrand constraint solving and were small enough that the generated code could be hand-edited was quite difficult. However, despite the modest number and size of the benchmarks, we believe the results still provide an insight into what kind of performance benefits Herbrand optimisation will provide for real programs.

The benchmarks chosen were four small Prolog benchmarks that used Herbrand constraint solving (those used in Section 5.6.1), translated into HAL by adding the necessary types and declarations. The experiments were performed under the same conditions as the unification experiments. The time reported was the best of five consecutive runs, and the maximum variation in times was 10 ms.

Table 5.11 shows the results of the Herbrand optimisations. Columns two and three give the number of predicates and literals (before normalisation) in each benchmark. Column three gives the number of run-time calls made to `unify_oo` in each benchmark. Columns four and five give the execution times for the un-optimised and optimised benchmarks respectively. Column five gives the improvement factor.

We saw in the first part of this section that specialised unifications ranged from 1.0–2.4 times faster than `unify_oo`, except for `unify_gg` which was substantially faster than that. For programs that use logic variables (and thus cannot use `unify_gg`), the improvement will depend on which unifications can be substituted for `unify_oo`, and what proportion of the un-optimised program's execution time is taken up by calls to `unify_oo`.

Of the benchmarks, `hanoidiff` showed the greatest improvement, of 14%. It contained two calls to `unify_oo` which were replaced by `unify_var_old` and `unify_var_val`. If boundedness information from mode analysis was exploited (it currently is not, but could be quite easily), the `unify_var_old` could be replaced by another `unify_var_val`, in which case the improvement rises to 18%. The improvement for `fast_mu` arises from replacing `unify_oo` calls with `unify_gg`. The improvement for `qsortdiff` arises from replacing two `unify_oo` calls with `unify_var_val`; however the number of calls is too small to make much difference. In `zebra`, some `unify_oo` calls are replaced with `unify_oo_noshare`, and others are replaced with `unify_var_val`; again the number of calls is too small to make a difference.

5.7 Conclusion

We have seen that the cost of Herbrand analysis is fairly modest for modules that use Herbrand constraint solving sparingly, with freeness analysis accounting for 1.3–11.4% of

compilation time of such modules, with around 5–8% being common. However, analysis times jump if `old` terms are used frequently, approaching 50% of compilation time.

We have also seen that `unify_oo` can be replaced with more specialised unifications using information from Herbrand analysis, and we carefully compared their relative speeds to determine which parts of unification are expensive and which are not. The experimental results showed that avoiding `nonvar/1` tests was a significant source of improvement. They also demonstrated that an analysis that determines when variable trailing can be avoided (using sharing and determinism analysis information) could further improve performance, and is worth implementing. Perhaps most importantly, they showed that true Herbrand constraint solving is much more expensive than HAL's basic manipulation of `new` and `ground` terms.

The benefit of the studied optimisations depends on how heavily a program uses Herbrand constraint solving; some small benchmarks that used Herbrand constraint solving intensively run up to 14% faster when optimised.

The experiments performed do not provide a convincing argument of the value of Herbrand analysis. However, one important point to consider is that it can be used for more sophisticated optimisations other than simply replacing calls to `unify_oo`. Work has begun on integrating the analysis information into HAL's code generator to further optimise Herbrand constraint solving. Also, the analysis information could be used to improve the modes of procedures — for example by replacing ground `old` arguments with `ground`, and lonely `old` arguments with `new`, and then re-running mode analysis on the new variants to obtain optimal code.

Having examined Herbrand constraint solving in such detail, our primary conclusion is somewhat contrarious — if performance is an important consideration, Herbrand constraint solving should be avoided where possible, since it is considerably slower than HAL's basic term manipulation.

Chapter 6

Determinism Analysis

A defining feature of CLP programs is that they can exhibit *non-determinism*: a predicate may return one or more solutions, or none at all. This feature allows quite powerful programming techniques to be used. However, there are two main problems with the unrestricted use of non-determinism.

1. **Correctness:** In practice, the majority of predicates in typical CLP programs are deterministic, or at least are intended to be deterministic. Unfortunately, it is very easy to unintentionally write predicates that can fail. If a predicate that should always succeed actually fails, it can be very difficult to determine where the failure occurred. Furthermore, if a query to a program that can legitimately fail does so, it can be even more difficult to know if the program failed "for the right reason". It is also possible to unintentionally write a program that returns multiple solutions, or the same answer multiple times.

2. **Efficiency:** Non-determinism has significant performance costs associated with it. For that reason, CLP programmers devote much effort to ensure that they do not accidentally leave behind choice-points. In particular, the infamous *cut* operator (!) is often used to prune potential backtracking. Unfortunately, cut is notoriously difficult to use correctly, and is the source of many subtle and not-so-subtle bugs in CLP programs. Furthermore, most CLP languages only provide one execution mechanism, so deterministic predicates are executed in the same manner as non-deterministic predicates, which adds unnecessary overhead.

This chapter describes how determinism declarations work within the language, and how bottom-up determinism inference is performed within HAL's analysis framework. Finally, it evaluates the cost and benefits of determinism analysis, and discusses how HAL's determinism system solves the above two problems.

6.1 HAL's Determinism System

Each procedure in a HAL program may be annotated with a determinism declaration. These declarations indicate whether the procedure can fail before producing its first solution, and

what is the maximum number of solutions it might return (zero, one, or more than one). Determinism declarations are semi-optional in the sense that procedures exported from a module are required to have a determinism declaration.[1] This is primarily for software engineering purposes: type, mode and determinism declarations together provide an excellent form of documentation for exported procedures. It also simplifies inter-module analysis; see Section 6.4.1 for an explanation.

Determinism declarations are checked by the compiler. If the compiler cannot prove a determinism declaration provided by the programmer is correct, it issues an error or warning. The problem of optimal determinism inference is undecidable in general, as solving it would require solving the halting problem [62]. As a result, any determinism analysis will sometimes obtain an inaccurate (but not incorrect) answer for a procedure. However, the algorithm used works extremely well in practice. The algorithm is conservative: it never underestimates a procedure's maximum number of solutions, and it never incorrectly reports that a procedure cannot fail.

HAL's determinism system is almost identical to that of Mercury. Thus, the following discussion regarding determinism declarations, switch detection, and common subexpression elimination has much in common with the description of the Mercury determinism system in [22].

6.2 The Determinism Domain

There are two characteristics that decide the determinism of a HAL procedure.

1. Whether the procedure can fail before producing a solution.
2. The maximum number of solutions produced by the procedure: zero, one, or many.

Combining these two characteristics gives us the six determinism categories, as follows.

- If a procedure has exactly one solution, it is *deterministic* (`det`).
- If a procedure has at most one solution, but can fail, it is *semi-deterministic* (`semidet`).
- If a procedure has at least one solution, but possibly more, it is *multi-deterministic* (`multi`).
- If a procedure might have more than one solution, and can fail, it is *nondeterministic* (`nondet`).
- If a procedure always fails, it has the determinism `failure`.
- If a procedure never fails or succeeds (i.e. it infinitely loops, always aborts, or always throws an exception) it has the determinism `erroneous`.

[1] The same is true for type and mode declarations.

The last two are rarely used, although the system predicate `fail/0` has a determinism of `failure`, and the system predicate `error/1` (used to abort) and the library predicate `throw/1` (used to throw an exception) are both `erroneous`. The categories are summarised in Table 6.1.

Max. solns.	0	1	> 1
Cannot fail	`erroneous`	`det`	`multi`
Can fail	`failure`	`semidet`	`nondet`

Table 6.1: The determinism categories

Note that the statements above about the number of solutions for the non-`erroneous` values assume that control returns to the caller. The determinism values do allow for control not to return to the caller of a non-`erroneous` procedure; for example, a `det` procedure may return exactly one solution, *or* abort, throw an exception or infinitely loop.

The determinism categories form a complete lattice, shown in Figure 6.1. Broadly speaking, the determinisms become less restrictive as we move up the lattice.

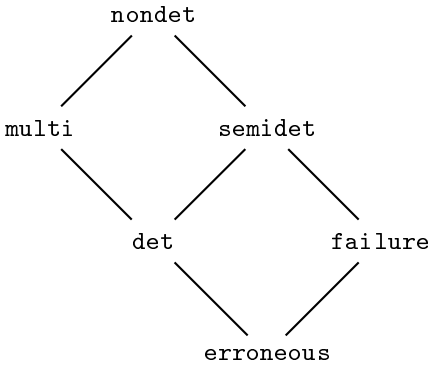


Figure 6.1: Determinism lattice

Determinism declarations are attached to mode declarations; different procedures of a predicate can have different determinisms. For example, consider the following four procedures of `append/3`:

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in,  in,  out) is det.
:- mode append(in,  in,  in)  is semidet.
:- mode append(out, out, in)  is multi.
:- mode append(oo,  oo,  oo)  is nondet.
```

The first three determinism declarations follow straightforwardly from the modes: concatenating two lists provides exactly one answer; checking if two lists concatenated equal a third will either succeed or fail; and a list can be broken into two parts in one or more ways. The fourth mode subsumes the other three cases, and thus can fail or succeed one or more times.

6.3 Preprocessing

Two source-to-source transformations are performed before determinism analysis takes place. The aim of these transformations is to find certain code features which, when identified, can improve the accuracy of determinism analysis.

The first of these is switch detection, which determines whether branches of a disjunction are mutually exclusive. The second is common subexpression elimination, which exposes some switches that aren't immediately obvious.

6.3.1 Switch Detection

Consider the following definition of one procedure of `append/3` as seen after conversion to super-homogeneous form and mode analysis has taken place:

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
append(Xs, Ys, Zs) :-
    ( Xs = [],
      Ys = Zs
    ;
      Xs = [X | Xs1],
      append(Xs1, Ys, Zs1),
      Zs = [X | Zs1]
    ).
```

This procedure contains a disjunction, but it is deterministic. This is because the two disjuncts are mutually exclusive when `Xs` is known to be bound: it must be either bound to the empty list `[]`, or the non-empty list `[X | Xs1]`. A disjunction that unifies a bound variable with a different functor in each disjunct is known as a *switch*, due to the superficial resemblance to switches in C programs. The associated unifications are called *switching unifications*. A disjunct containing a switching unification is called a *switching disjunct*.

There are two ways to categorise a switch. Firstly, a *full switch* is a disjunction containing a switching unification in every disjunct. At most one of the disjuncts can succeed. By contrast, a *partial switch* is a disjunction in which not every disjunct contains a switching unification. At most one of the disjuncts containing a switching unification can succeed.

Secondly, a *cannot_fail switch* is a disjunction in which every functor allowed by a variable's instantiation is involved in a switching unification, with at most one per disjunct. Exactly one of these unifications will succeed. A *can_fail switch* is similar, except that only a strict subset of the constructors allowed by a variable's instantiation are involved in switching unifications. Zero or one of these unifications will succeed.

Note that these categorisations assume that each functor is not involved in more than one switching unification per switch. We can assume this because common subexpression elimination (which will be described in Section 6.3.2) ensures it is true.

The following procedures demonstrate the four possible switch combinations, and their effect on the determinism of the switch. Note that if a predicate has only one procedure, the

type, mode and determinism declarations can be combined by pairing the type and mode of each argument with the `::/2` functor and appending the determinism declaration.

```
:- typedef abc -> ( a ; b ; c ).

:- pred full_cannot_fail(abc::in) is det.
full_cannot_fail(X) :-
    ( X = a ; X = b ; X = c ).

:- pred full_can_fail(abc::in) is semidet.
full_can_fail(X) :-
    ( X = a ; X = b ).

:- pred partial_cannot_fail(abc::in) is multi.
partial_cannot_fail(X) :-
    ( X = a ; X = b ; X = c ; true ).

:- pred partial_can_fail(abc::in) is multi.
partial_can_fail(X) :-
    ( X = a ; X = b ; true ).
```

Note that although the switching disjuncts of `partial_can_fail` can fail, in this case the non-switching disjunct will always succeed, so the disjunction will never fail.

In practice, the vast majority of switches are full and `cannot_fail`, since they are most commonly used to deal with constructor types where each arm of the switch handles one functor.

Switches are detected in a disjunction by considering all non-local variables that are bound before entry to the disjunction. If at least two of any variable's possible functors are tested in distinct disjuncts, a switch is known to be present. Note that unlike the first-argument indexing present in most CLP systems, switches will be detected on any clause variable.

As soon as a full switch is found, we can stop looking since we know the disjunction has at most one solution. If we find a partial switch we remember it but continue looking for more switches. If only partial switches are found, we choose the one that covers the most disjuncts. In the case of a tie, the first one found is chosen arbitrarily.[2]

If the best switch found is partial, the disjuncts containing the switching unifications are gathered into a new sub-disjunction. For example, `partial_cannot_fail` defined above is transformed into this form:

```
partial_cannot_fail
    ( ( X = a ; X = b ; X = c )
    ; true
    ).
```

[2] Henderson *et al.* state on page 8 of [22] that when choosing between partial switches, "the nature of the heuristic used does not seem to matter much in practice."

This simplifies determinism analysis, since it removes all partial switches.

If a disjunction is found to be a switch, it is marked as either `can_fail` or `cannot_fail`. The switching unifications are also marked. Both these markings are used when determining the disjunction's determinism, as will be explained in Section 6.4.

6.3.2 Common Subexpression Elimination

The described approach to switch detection finds the majority of switches in HAL programs. However there are some important cases that misses. For example, consider the following predicate:

```
:- pred count_elements(list(T)::in, int::out) is det.
count_elements([], 0).
count_elements([_], 1).
count_elements([_,_|_], 2).
```

When converted to super-homogeneous form, it becomes as follows:

```
count_elements(A, B) :-
    ( A = [], B = 0
    ; A = [C | D], D = [], B = 1
    ; A = [E | F], F = [G | H], B = 2
    ).
```

The problem is that these three disjuncts are mutually exclusive, but there are actually two switches present; one switches between `A` being empty or non-empty, and the other switches between the tail of `A` being empty or non-empty. To find both switches the common subexpression shared by the second and third disjuncts must be eliminated by "hoisting" it as follows:

```
count_elements(A, B) :-
    ( A = [], B = 0
    ; A = [C | D],
      ( D = [], B = 1
      ; D = [G | H], B = 2
      )
    ).
```

The two switches present are now clearly separated and can be found by the technique described in the previous section. Common subexpression elimination therefore must be performed before switch detection. Note that the first-argument indexing performed by most CLP systems would not be enough to avoid trying the second clause for lists of length two or more in a predicate such as `count_elements`.

The algorithm for performing common subexpression elimination is quite similar to that for switch detection, and the two phases share code in the compiler. It too searches for switching unifications in disjuncts; once it finds them, it combines any unifications that match against the same functor by hoisting out the first of the common unifications. Note

that some variables in the second and subsequent arms must be renamed or eliminated (in the example given, `E` is removed, and `F` is renamed as `D`).

Three phases of the compiler must then be reinvoked on all changed procedures.

1. Variable scopes are recomputed to take into account the new disjunction. For example, after `count_elements` is transformed, the variables `G` and `H` change from being local variables of the third disjunct to being local variables of the second disjunct of the inner disjunction. The local variable sets of affected bodies must also be updated to account for renamed and eliminated variables.

2. Mode analysis is rerun to compute the instantiations of the relevant variables at the new program points created by the introduction of the new disjunction. This is necessary because determinism analysis and other subsequent phases require these instantiations at every program point.

3. Common subexpression elimination is again performed to eliminate any nested common subexpressions.

These steps are repeated as many times as necessary. Once they are complete, switch detection can be performed. Note that common subexpression elimination cannot cause the compiler to enter an infinite loop, as the number of unifications that match the same functor is reduced by each iteration.

6.4 Determinism Analysis of Bodies

The determinism of each body in the program is inferred as follows.

6.4.1 Literals

Procedure calls: The determinism of a call to a local procedure is the determinism inferred for that procedure. The fixpoint that needs to be reached for the determinism inference of procedures that call each other is computed appropriately by the analysis framework.

The determinism of an external procedure call can be found in the imported module's interface file. Thus, before a module can be compiled, interface files must be generated for all the modules it uses. These interface files contain information about all exported items, including the determinism of all exported procedures. The determinism is guaranteed to be there, because all exported procedures must have a determinism declaration; this makes determinism analysis simpler. Of course, the determinism for the external procedures may be incorrect — interface file generation does not perform determinism checking, but trusts the programmer's declarations — but this is unavoidable when using separate compilation. This is not a problem in practice, since all declarations must be checked before the program can be run.

The determinism of a higher-order call using `call/n` is equal to the determinism of the higher-order argument's procedure. This determinism is always known because HAL only allows higher-order terms to be created from procedures whose determinism is declared.

	e	f	d	s	m	n
e	e	e	e	e	e	e
f	f	f	f	f	f	f
d	e	f	d	s	m	n
s	f	f	s	s	n	n
m	e	f	m	n	m	n
n	f	f	n	n	n	n

(a)

	e	f	d	s	m	n
e	e	e	d	s	m	n
f	e	f	d	s	m	n
d	d	d	m	m	m	m
s	s	s	m	n	m	n
m	m	m	m	m	m	m
n	n	n	m	n	m	n

(b)

	e	f	d	s	m	n
e	e	f	d	s	m	n
f	f	f	s	s	n	n
d	d	s	d	s	m	n
s	s	s	s	s	n	n
m	m	n	m	n	m	n
n	n	n	n	n	n	n

(c)

Table 6.2: Determinism of conjunctions, disjunctions, full `cannot_fail` switches

Unifications: All unifications are either `det` or `semidet`. The exact determinism depends on the instantiation of the arguments. There are only four forms of unifications that can appear after mode analysis; any unification not fitting one of these forms will be converted into two or more unifications that do.

- Unifications of the form $X = Y$ where one variable is `new` are *assignments*. They are always `det`.
- Unifications of the form $X = Y$ where neither X nor Y are `new` are usually `semidet`, but may be `det` if X and Y are known to be ground and equal.
- Unifications of the form $X = f(Y_1, \ldots, Y_n)$ where X is `new` are *constructs*. They are `det`.
- Unifications of the form $X = f(Y_1, \ldots, Y_n)$ where X is not `new` and $Y_1, \ldots, Y_n$ are distinct `new` variables are *deconstructs*. They are usually `semidet`, but can be `det` when X is of a type whose only functor is f/n or X is known to be bound to f/n.

There is one exception to these rules: unifications marked as switching unifications are always treated as `det`. This is explained in Section 6.4.4.

Higher-order unifications: The determinism of a higher-order unification is always `det`, since the variable on the left-hand side is guaranteed to always be `new`.

6.4.2 Conjunctions

The determinism of a conjunction of two bodies is found by the predicate `conj_two_dets`, the operation of which is summarised by Table 6.2(a). Each determinism value is represented by its first letter; the row entries represent the first conjunct's determinism, and the column entries represent the second conjunct's determinism. Note that the combination is not symmetric; if the first conjunct is `erroneous` or `failure`, the conjunction's overall determinism is `erroneous` or `failure`.

Most of the combinations are straightforward. One potentially confusing combination is that of `(semidet, erroneous)` conjunctions, which have the determinism `failure`. The reason is that such conjunctions can fail (if the first conjunct fails), and they will never produce a solution (either the first conjunct will fail or control will not return from the second).

Hence, their overall determinism is `failure`. The `(nondet, erroneous)` combination is similar.

The determinism of a conjunction with more than two bodies is obtained by folding `conj_two_dets` left-to-right through the list of individual determinisms. Essentially, a conjunction can fail if any of its conjuncts can fail; it will have no solutions if any of its conjuncts have no solutions; it will have many solutions if all of it conjuncts can succeed and at least one has many solutions; and it is deterministic only if all its goals are deterministic.

6.4.3 Disjunctions

The determinism of a non-switch disjunction is found similarly to that of a conjunction, but using the combinations from Table 6.2(b). Unlike conjunctions, the combinations are symmetric.

Essentially, a disjunction can fail only if all its disjuncts can fail; it will have many solutions if any of its disjuncts have many solutions or at least two of them have one solution; and it will be deterministic only if one disjunct is deterministic and all the others have no solutions. In practice, non-switch disjunctions are very rarely deterministic.

Note that the combination of two disjuncts is *not* a least upper bound operation. In particular, the least upper bound of any two elements in a lattice must be idempotent, i.e. $e \sqcup e = e$. This is not the case with $disj_{\mathcal{D}}$ for determinism, for example a disjunction of two `det` disjuncts has the determinism `multi`.

6.4.4 Switches

The determinism of a full switch is found similarly to that of a non-switch disjunction, but using the combinations from Table 6.2(c). Note that these are the combinations for a `cannot_fail` switch; a `can_fail` switch can always fail, changing the entries that cannot fail (`erroneous`, `det`, `multi`) to the corresponding values that can (`failure`, `semidet`, `nondet`). Also recall that all partial switches are removed during switch detection, so we only have to consider full switches.

A switch differs from a non-switch disjunction in that at most one of its disjuncts will produce a solution. Switching unifications would normally be `semidet`, being either deconstructs or equality tests, but we ignore them (by treating them as `det`, the identity element for conjunction) so their `semidet`-ness does not "taint" the determinism of the rest of each switching disjunct. The `semidet`-ness of the switching unifications is instead taken into account in the combination in Table 6.2(c).

Note that the combination of two switch arms in a `cannot_fail` switch *is* the least upper bound.

6.4.5 If-then-elses

The behaviour of if-then-elses is different in HAL to most other CLP languages — if the condition of an if-then-else can succeed multiple times, the second and subsequent solutions

will *not* be pruned. The *else* branch will only be taken if the condition fails before producing any solutions. HAL inherits this behaviour from Mercury.

To compute the determinism of an if-then-else, we must break up each branch's determinism into its components. This is done using the following types and predicate:

```
:- typedef determ   -> ( det    ; semidet ; multi
                       ; nondet ; failure ; erroneous ).
:- typedef can_fail -> ( can_fail ; cannot_fail ).
:- typedef max_soln -> ( zero ; one ; many ).

:- pred determinism_components(determ, can_fail, max_soln).
:- mode determinism_components(in,  out, out) is det.
:- mode determinism_components(out, in,  in)  is det.
determinism_components(erroneous, cannot_fail, zero).
determinism_components(failure,   can_fail,    zero).
determinism_components(det,       cannot_fail, one).
determinism_components(semidet,   can_fail,    one).
determinism_components(multi,     cannot_fail, many).
determinism_components(nondet,    can_fail,    many).
```

If the determinism of the three branches of an if-then-else are `IfDeterm`, `ThenDeterm`, and `ElseDeterm`, and they have been broken into their `can_fail` and `max_soln` parts by `determinism_components`, the overall determinism `Determ` of the if-then-else is found as follows:

```
( IfCanFail = cannot_fail ->
    conj_two_dets(IfDeterm, ThenDeterm, Determ)
;
  IfDeterm = failure ->
    Determ = ElseDeterm
;
  IfDeterm = erroneous ->
    Determ = erroneous
;
    conjunction_maxsoln(IfMaxSoln, ThenMaxSoln, IfThenMaxSoln),
    full_switch_maxsoln(IfThenMaxSoln, ElseMaxSoln, MaxSoln),
    full_switch_canfail(ThenCanFail, ElseCanFail, CanFail),
    determinism_components(Determ, CanFail, MaxSoln)
).
```

The actions of `conjunction_maxsoln`, `full_switch_maxsoln` and `full_switch_canfail` are summarised in Tables 6.3, 6.4 and 6.5.

If the condition cannot fail, the *else* branch can be ignored. If the condition always fails, the overall determinism is that of the *else* branch. If the condition is `erroneous`, the whole if-then-else is `erroneous`. Otherwise, the condition can both fail and succeed; in this case,

	zero	one	many
zero	zero	zero	zero
one	zero	one	many
many	zero	many	many

Table 6.3: Conjunction `max_soln` combinations

	zero	one	many
zero	zero	one	many
one	one	one	many
many	many	many	many

Table 6.4: Full switch `max_soln` combinations

	can_fail	cannot_fail
can_fail	can_fail	can_fail
cannot_fail	can_fail	cannot_fail

Table 6.5: Full switch `can_fail` combinations

the if-then-else is basically a `cannot_fail` switch between the *then* and *else* branches. But if the condition succeeds multiple times, the *then* branch can succeed multiple times even if it is deterministic.

6.5 Determinism Analysis of Modules

Determinism analysis is performed after type and mode analysis, common subexpression elimination and switch detection. It infers the determinism of all procedures, checks the declared determinism of procedures with declarations, and issues warnings and errors about determinism violations.

The fixpoint required for determinism inference is provided by the analysis framework. All that is required is a suitable instance of the type class `abstract_domain` (defined in Section 4.3), the methods of which are shown in Figure 6.2.

- Acomb conjoins two determinisms using `conj_two_dets` (the variable sets are ignored).
- Aadd obtains the determinism of the literal using `get_literal_determ`; its definition is not shown, but it behaves according to the rules for literals given in Section 6.4.1. The determinism is then combined with D using `conj_two_dets`.
- Adisj disjoins multiple determinisms according to the rules in Sections 6.4.3 and 6.4.4.
- Aif_to_then always returns `det`, the $true_{\mathcal{D}}$ element for determinism.
- Aif_then_else combines the determinism of the three branches according to the rules in Section 6.4.5.
- Aoutrestrict is trivial, as determinism doesn't involve variables.
- Ainrestrict always returns `det`, the $true_{\mathcal{D}}$ element for determinism.
- Aextend is also trivial, again because determinism doesn't involve variables.
- Abottom returns `erroneous`, the determinism lattice's bottom element.

```
Acomb(_, D1, _, D2)
    conj_two_dets(D1, D2, D)
    return answer(D)

Aadd(L, D)
    Dlit := get_literal_determ(L)
    conj_two_dets(D, Dlit, D')
    return answer(D')

Adisj(Ds)
    % See Sections 6.4.3, 6.4.4

Aif_to_then(_)
    return det

Aif_then_else(Dif, Dth, Del)
    % See Section 6.4.5

Aoutrestrict(_, D)
    return D

Ainrestrict(_, _)
    return det

Aextend(_, D)
    return D

Abottom(_)
    return erroneous

Ainitial_guess(_)
    return unreached

Ais_constraint(_)
    return false

Acalling_descHO(_ : _)
    return det

Aexternal_proc(P : _)
    if (P = call/n)
        D := determinism of higher-order term
    else
        D := lookup P in external interface file
    return D
```

Figure 6.2: Determinism analysis abstract operations

- Ainitial_guess doesn't attempt to obtain an initial answer; it returns *unreached*.

- Ais_constraint returns *false*. No procedure calls need to be considered specially for determinism analysis.

- Acalling_descHO returns `det`, the $true_{\mathcal{D}}$ element for determinism.

- Aexternal_proc finds the determinism of external procedures as explained in Section 6.4.1 — it looks up the external module's interface file unless it is a higher-order application using `call/n`, in which case the determinism is that of the procedure captured by the higher-order term.

These class methods provide an example of how to perform bottom-up analysis within the framework, as was described in Section 3.3.4 — Aif_to_then, Ainrestrict, and Acalling_descHO all return $true_{\mathcal{D}}$, which for determinism is `det`. The only other requirement for bottom-up analysis is that the initial set of calling patterns S only contain calling patterns of the form $P : true_{\mathcal{D}}$. For determinism analysis we initialise S by adding P : `det` for every procedure exported from the module.

If the final answer for a procedure is *unreached*, e.g. due to degenerate recursion, this is interpreted as `erroneous`.

6.6 Errors and Warnings

Warning and error messages are issued when a determinism declaration doesn't match that inferred.[3] If the declared determinism is strictly tighter than the inferred determinism according to the determinism lattice (e.g. declared `det`, inferred `semidet`), or if the two determinisms are incomparable (e.g. declared `semidet`, inferred `multi`), an error is issued.

If the declared determinism is strictly laxer than the inferred determinism according to the determinism lattice (e.g. declared `semidet`, inferred `det`), a warning is issued. This is not a serious problem — the determinism is not incorrect, just inaccurate, which may degrade performance slightly (see Section 6.7.2 for an explanation).

6.7 Experimental Analysis Evaluation

This section evaluates HAL's determinism analysis by considering the time taken during compilation to perform the analysis, and the performance improvements it provides; it also provides a discussion of the accuracy of the analysis and its effect on programming style.

6.7.1 Cost of Analysis

To determine the cost of determinism analysis at compile-time, a number of modules of varying sizes were tested: five traditional standalone Prolog benchmarks (`aiakl`, `boyer`, `fib`, `qsortdiff` and `warplan`), rewritten as HAL programs; five medium-sized solver modules (`bounds`, `dfd`, `domain`, `mset` and `simplex`); and five larger modules from the HAL compiler itself (`freeness_ops`, `swdet_cse`, `fixpoint_framework`, `global_declarations` and `mode_analysis_framework`). The experiments were run under the same conditions as those in Section 5.6.1, except that the determinism analysis phase replaced the Herbrand analysis phase.

Table 6.6 shows the results of analysing these modules. Columns two and three give the number of predicates and literals (before normalisation) in each module. Column four gives the time taken for switch detection and common subexpression elimination,[4] column five gives the time taken for determinism analysis, column six is the sum of the previous two columns, and column seven gives the overall compilation time. Column eight gives the proportion of compilation time taken up by switch detection, common subexpression elimination and determinism analysis (i.e. column six divided by column seven). All times are in milliseconds, rounded to the nearest ten.

Switch detection and common subexpression elimination were significantly cheaper than determinism analysis for all modules except `mode_analysis_framework`; switch detection was more expensive for that module because it contains a number of very long predicates (some with more than 100 literals), so a high proportion of unifications had to be tested to see if they could be switched on.

[3] Note that determinism declarations are not used during analysis, but only during the checking stage.
[4] The two phases are intertwined in the compiler which makes it difficult to measure them separately.

Module	Preds	Lits	SwCSE	Det	Both	All	Prop
`aiakl`	13	47	10	40	50	9810	0.5%
`boyer`	17	136	60	160	220	5730	3.8%
`fib`	5	14	0	10	10	750	1.3%
`qsortdiff`	11	22	10	30	40	10830	0.3%
`warplan`	34	166	60	320	380	14920	2.5%
`bounds`	34	199	10	120	130	4110	3.2%
`dfd`	22	219	20	240	260	6220	4.2%
`domain`	47	335	40	270	310	5940	5.2%
`mset`	36	242	50	200	250	4290	5.8%
`simplex`	47	223	50	220	270	4760	5.6%
`freeness_ops`	53	241	70	280	350	13760	2.5%
`swdet_cse`	49	444	340	430	770	21650	3.6%
`fixpoint_framework`	83	702	180	700	880	30650	2.9%
`global_declarations`	226	991	570	1640	2210	63090	3.5%
`mode_analysis_framework`	122	1490	2300	1630	3930	65700	6.0%

Table 6.6: Cost of determinism analysis (ms)

The three phases together account for 0.3–6.0% of compilation time, although if we ignore the figures of 0.3% and 0.5% for `qsortdiff` and `aiakl` (which were skewed by extremely long mode analysis phases) and 1.3% for `fib` (which is too small to be reliable) the range is 2.5%–6.0%.

This shows that determinism analysis is clearly quite cheap, and supports the claims of Section 4.9.7 that bottom-up analysis is quite efficient within the top-down analysis framework. It also shows that determinism analysis scales very well — the two largest modules `global_declarations` and `mode_analysis_framework` consist of 1,870 and 2,013 lines of code respectively (excluding blanks and comments). This is not surprising considering the simplicity of the domain, which precludes the possibility of an "explosion" in analysis time.

6.7.2 Effect of Optimisations

Determinism analysis is an example of an analysis for correctness that also has efficiency benefits. This is because a procedure that can succeed only once can be executed more efficiently than a procedure that can succeed multiple times, as there is no need to consider backtracking. Similarly, a procedure that cannot fail can be executed more efficiently than one that can.

Since HAL compiles to Mercury, we can take advantage of this optimisation performed by the Mercury compiler for free. The Melbourne Mercury Compiler has three different execution algorithms of increasing complexity and decreasing time and space efficiency.

1. A deterministic execution algorithm, for `det` procedures.
2. A semi-deterministic execution algorithm, for `semidet` procedures.
3. A non-deterministic execution algorithm, for `multi` and `nondet` procedures.[5]

[5] There is no dedicated multi-deterministic execution algorithm because `multi` procedures may fail upon

Benchmark	Preds	Lits	TM	TMD	Factor
`aiakl`	7	21	40	30	1.3
`boyer`	14	124	90	50	1.8
`deriv`	1	33	740	500	1.5
`fib`	1	6	20	20	1.0
`hanoiapp`	2	7	610	220	2.8
`hanoidiff`	2	6	80	110	0.7
`mmatrix`	3	7	110	40	2.8
`qsortapp`	3	10	390	190	2.1
`qsortdiff`	3	10	370	210	1.8
`serialize`	5	19	350	270	1.3
`tak`	1	9	100	50	2.0
`warplan`	25	88	330	220	1.5
Total/Average	67	340	3230	1910	1.7

Table 6.7: Effect of determinism optimisations (ms)

See [56] for details of the three execution algorithms.

No new experimental results are presented here, since a careful evaluation of the effect of type, mode and determinism declarations on HAL programs was presented in [11]. It described tests performed on a subset of the standard Prolog benchmarks: `aiakl`, `boyer`, `deriv`, `fib`, `hanoiapp`, `hanoidiff`, `mmatrix`, `qsortapp`, `qsortdiff`, `serialize`, `tak` and `warplan`.

For the determinism results, the benchmarks were first written with precise type and mode declarations, but each procedure was declared as `nondet`, and timings were made. Precise determinism declarations were then added, and timings were made again. The results dealing with determinism declarations from [11] are reproduced in Table 6.7. Columns two and three give the size of each benchmark (number of predicates and literals before normalisation, excluding dead code and the query).[6] Columns four and five give the execution time before and after precise determinism declarations were added (in milliseconds, to the nearest ten). Column six gives the improvement factor.

The maximum improvement was by a factor of 2.8, for `hanoiapp`. The only program that slowed down was `hanoidiff`, which ran at 0.7 times the original speed. The overall improvement found by comparing the total execution time of the benchmark suite was a factor of 1.7; regardless of the exact comparison used, the optimisation is clearly beneficial.[7]

retry, so the non-deterministic execution algorithm's mechanism for handling failure is required.

[6] The number of predicates and literals for `aiakl`, `boyer`, `fib`, `hanoidiff`, `qsortdiff` and `warplan` are different from those given in Tables 5.4 and 6.6. This is due to the query (excluded from this table), and slight differences between the versions used. The common procedures had identical determinism declarations, however.

[7] This experiment was possible because Mercury does not do full determinism inference, but only checks the determinism of procedures with determinism declarations; thus the non-deterministic execution algorithm is used for any procedure declared `nondet`, even if it is actually deterministic or semi-deterministic. By contrast, the HAL compiler performs full determinism inference, and then checks each declaration.

6.7.3 Limitations

As stated in Section 6.1, the determinism inference algorithm is not perfect (and cannot be). Recall one procedure of the `mortgage` program given in Section 2.4.1:

```
:- pred mortgage(cfloat, cfloat, cfloat, cfloat, cfloat).
:- mode mortgage(in, in, in, in, out) is nondet.
mortgage(P, T, I, R, B) :-
    T = 0.0, B = P.
mortgage(P, T, I, R, B) :-
    T >= 1.0,
    mortgage(P + P*I - R, T - 1.0, I, R, B).
```

This procedure is actually `semidet`, as it contains a `can_fail` switch on the variable `T`. Unfortunately, the compiler infers it as `nondet`, because it has no domain knowledge of integers, and cannot determine that the two rules are mutually exclusive.

It would be reasonably straightforward to build in some domain knowledge of numbers that would enable procedures such as this to be analysed correctly, and HAL's switch detection may be extended to do so in the future. But in the general case it is always possible to write procedures that the compiler will infer inaccurately. Typically this can be easily fixed, e.g. by rewriting an undetected switch as an if-then-else:

```
mortgage(P, T, I, R, B) :-
    ( T = 0.0 ->
        B = P
    ;
        T >= 1.0
        mortgage(P + P*I - R, T - 1.0, I, R, B)
    ).
```

Another alternative is to declare the procedure as `nondet`. This is less satisfactory, since it will degrade performance, and is somewhat misleading.

In practice, as with Mercury, the determinism system works extremely well, and cases such as this one are easily worked around.

6.7.4 Effect on Programming Style

HAL's determinism system promotes a very safe, conscientious programming style; many cases that are supposedly impossible due to implicit program invariants must be explicitly handled (usually by throwing an exception or aborting execution). This is a good thing, since all too often such implicit invariants can be broken due to bugs, or subsequent program changes; a program that gives an obvious run-time error is almost always preferable to one that silently continues incorrectly.

For example, imagine that the predicate `count_elements` from Section 6.3.2 is used in a program in which the first argument should never be empty due to a program invariant.

Even though the programmer knows the empty list case will never be called, in order for the procedure to be inferred as `det`, the empty list case must be defined. For example:[8]

```
:- pred count_elements(list(T)::in, int::out) is det.
count_elements([], _) :- error("count_elements: Unexpected empty list").
count_elements([_], 1).
count_elements([_,_|_], 2).
```

The empty list case may seem redundant, but if the program invariant is ever broken and an empty list is passed to `count_elements`, a run-time error will occur immediately, making it very clear exactly what the problem is and where it occurred (assuming the error message is sufficiently informative).

6.8 Conclusion

HAL's determinism system almost completely eliminates the two problems described at the start of this chapter.

1. **Correctness:** It identifies almost all determinism errors, which constitute a significant proportion of all programming errors. It also promotes a robust programming style that provides a good level of protection against the accidental violation of program invariants.[9]

2. **Efficiency:** It allows more efficient code generation. Each procedure uses the most appropriate of three different execution algorithms, of increasing simplicity and efficiency, depending on whether the procedure is non-deterministic, semi-deterministic, or deterministic; the price of non-determinism and semi-determinism is only paid by those procedures that use them. Prior experiments have shown this specialisation increases the speed of HAL programs by a factor of about 1.7. The cost of determinism analysis is quite small, only about 2.5–6.0% of compilation time, which shows the optimisations enabled definitely make it worthwhile from an efficiency standpoint alone.

Thus determinism analysis is an example of an analysis for correctness that also allows performance improvements.

The final benefit of determinism declarations is that they provide a useful form of documentation for each procedure, which is particularly valuable since they are confirmed by the compiler, and thus known to be correct.

[8] The first argument could also be given a mode that ensures it is non-empty, in which case any potential violations would be found at compile-time, and the first clause could be omitted. But for more complicated invariants that the compiler cannot prove, run-time tests must be used.

[9] Of course, this assumes the programmer gives appropriate declarations; determinism errors cannot be found if a programmer abuses the system by declaring every procedure to be `nondet`.

Chapter 7

Inter-module Analysis

Most modern programming languages have a well-defined module system for dividing programs into manageable pieces. This is very important when "programming in the large", as it allows different programmers to work on separate modules. Module systems are also the means by which some language features are implemented. For example, in HAL abstract types are created by exporting a type abstractly from a module using the `export_abstract` declaration.

Most programming languages with a well-defined module system also support *separate compilation*, whereby each module can be compiled by itself. Once all the modules in a program are compiled, they can be linked together into an executable. Separate compilation is vital because in large projects with many modules, it is not reasonable to change a single module and require recompilation of all other modules, as this may take minutes or even hours.

HAL is no exception. The basics of its module system were introduced in Section 2.4.1. It supports separate compilation by generating interface files for each module which provide sufficient information about exported program items (types, modes, predicates, and so on). Before a file can be compiled, the interface files of all the modules it imports must be generated. Once all the modules in a program have been compiled they can be "linked", i.e. the generated Mercury files can be compiled by the Mercury compiler.

Separate compilation is particularly important in HAL because each solver is written in its own module. There are three reasons for this.

1. It is quite sensible from a program design viewpoint.
2. If the representation of a solver type is to be hidden, it must be exported abstractly from its own module.
3. Most solvers have differing "internal" and "external" views of solver types; for example, a constrained floating point variable with instantiation `old` outside the solver may actually be represented as a `ground` index into a simplex tableau inside the solver (see [12, 17] for more details on writing solvers in HAL).

Because HAL solvers are defined as separate modules, accurate inter-module analysis is vital. And because solvers are so much more general than other CLP solvers, and are

written similarly to other HAL modules, generic analysis techniques must be supported. These characteristics drive the need for practical, accurate inter-module analysis.

This chapter describes the compilation model used in the HAL compiler for performing inter-module analysis and optimisation. The compilation model naturally complements the analysis framework presented in Chapter 4, and supports *multi-variant specialisation* — the generation of multiple versions or *variants* of a procedure, each specialised for a particular calling pattern — without any assistance from the programmer (see [51, 36] for descriptions of other systems that also perform multi-variant specialisation).

7.1 Difficulties of Inter-module Analysis

One problem with module systems and separate compilation is that they make context-sensitive program analysis difficult. When compiling a single module, not all the other modules in the program have necessarily been compiled and analysed, so we must work with possibly incomplete information. This leads to two problems.

1. How do we obtain accurate context-sensitive answers for calls to external procedures? Modules containing these procedures may not have been compiled yet, so these answers may not be present. We will call this the *unknown answers* problem.

2. In what contexts should the exported procedures of a module be analysed? Modules that call these procedures may not have been compiled yet, so their calling contexts may not be known. We will call this the *unknown contexts* problem.[1]

These problems are particularly relevant for compilers such as the HAL compiler that perform multi-variant specialisation without any assistance from the programmer.[2] These problems are also relevant for a compiler that generates only a single variant of each procedure that is "as specialised as possible" given the contexts from which it is called.

In the context of the intra-module analysis framework presented in Chapter 4, the two problems correspond directly to these two questions.

1. What answer should Aexternal_proc return for any given calling pattern?

2. What calling patterns should be in the initial set S?

With these questions in mind, let us now consider some previous approaches to inter-module analysis and optimisation.

7.2 Previous Approaches

There are surprisingly few publications on approaches to inter-module analysis. Nonetheless, here is a summary of those found in the literature.

[1] Note that the terms *optimal success problem* and *optimal calls problem* are used in [52].

[2] As opposed to programmer-assisted multi-variant specialisation, such as that performed by the HAL compiler for predicates with multiple procedures.

Sheer pessimism: The simplest approach is to assume the worst and use a strategy that is safe in all circumstances. The unknown answers problem is solved by using $\top$ as the answer for external procedure calls. The unknown contexts problem is solved by analysing each exported procedure P using the calling pattern $P : \top$. Obviously, this approach may give very poor results for multi-module programs.

Abandon context-sensitive analysis: Using goal-independent analysis instead of goal-dependent analysis is a commonly used approach. Unfortunately, the results are not as accurate unless the analysis domain satisfies quite restrictive properties.[3] Also, because procedures are not analysed in context, it is not possible to perform inter-module multi-variant specialisation without programmer declarations or by using a subsequent goal-dependent step which would suffer from the original two problems.

Usage conditions: Mazur *et al.* describe a memory reuse analysis for Mercury [45], and suggest a method for applying it across module boundaries. Two versions of each procedure are generated: one involving the reuse suggested by a so-called *default analysis*, and one involving no reuse. The version using reuse is annotated with the conditions the calling procedure must satisfy to safely call it. This is a reasonable approach for analysis domains that have a suitable default analysis and in which such calling conditions can be expressed, however it results in an imprecise specialisation because the default analysis used is not tailored to individual calls.

Cross-module inlining: A different approach is to use inter-module inlining to reduce the two problems. For example, Mercury allows inter-module optimisation for several of its optimisations. As described in [61], the declarations and code of suitable procedures are written to a generated `.opt` file; when compiling a module that imports a module with a `.opt` file, the `.opt` file is read as if it were part of the source of the module being compiled.

One problem with this approach is that information can only be propagated from imported modules to importing modules — there is no way for a module to "request" information from another module — so it partially solves the unknown answers problem (for procedures included in the `.opt` file), but not the unknown contexts problem. Also, the information propagation is not transitive — it can only cross one module boundary. Page 20 of [61] emphatically demonstrates this problem; when applying a deforestation optimisation to the Mercury compiler itself:

> "Unfortunately this does not result in a measurable change in runtime. The main reason for this is that much of the runtime of the compiler is taken up by predicates manipulating 2-3-4 trees and sets represented by ordered lists. These predicates are hidden under two levels of interfaces to allow easy replacement of the data structures, which hinders inter-module optimisation."

A more complex and powerful approach is *lambda-splitting* [2], used in the SML/NJ compiler [39] for Standard ML [46] which involves breaking modules into inlineable and non-inlineable

[3] It must be *condensing*; see e.g. Section 4 of [40].

parts. Still, to make inter-module analysis information available only via inter-module inlining is clearly unsatisfactory; for example, it is of little use for large procedures that should not be inlined.

Mercury's termination analysis approach: Mercury allows transitive inter-module information propagation for its termination analysis, recorded in `.trans_opt` files. The unknown contexts problem is not relevant for termination analysis, since it is a bottom-up analysis and calling contexts are irrelevant.

The unknown answers problem is solved by compiling according to dependency order when possible, and using "don't know" (i.e. may not terminate) for procedures missing an answer. Each `.trans_opt` file can rely on other `.trans_opt` files, allowing termination information to be propagated across multiple module boundaries. This approach finds the greatest fixpoint of the termination information across modules. One unavoidable consequence of this is that it can never prove that mutually recursive procedures in different modules can terminate.

We will see that this approach has similarities to the approach used by HAL, although HAL's approach is more general, works for top-down analyses, and supports multi-variant specialisation.

7.3 The HAL Approach

The approach used by the HAL compiler was first described in [6]. It has one principal idea: at each stage both the analysis information and the executable program are correct, but recompilation may result in improved generated code. To create a correct executable, each module only needs to be compiled once. To create a maximally optimised executable, each module may need to be compiled more than once. If the code is changed, some anaysis information may no longer be correct, and must be invalidated.

When a module is compiled, it "asks" that information missing from the modules it calls be gathered when they are next compiled, and "tells" its calling modules when previously unavailable or more accurate information is available. We use $\top^*$ as the calling description for a procedure if its context is unknown, and $\top^*$ for unknown answers.[4] Variants are generated for all calling patterns encountered so far, and if a required variant from another module has not been generated yet, we use the best safe variant that has been generated. The variant for the default $P : \top^*$ calling pattern of each procedure is always generated, so there is always a safe variant of each procedure to link with. As compilation proceeds, increasingly accurate information is gathered and more accurate answers and calling descriptions are found until the greatest inter-module analysis fixpoint is reached and an optimal executable can be created.

There are two primary data structures required for this compilation model: the *analysis registry*, and the *inter-module dependency graph*.

[4] Recall that Section 5.3.2 defined $\top^*$ as the top-most description possible with respect to other information, such as mode declarations or previous analyses.

7.3.1 Analysis Registry

The analysis registry (or *registry* for short) is an extended answer table containing the most up-to-date answer obtained for each exported procedure. Basic registry entries have the same form $P : D_P \mapsto Ans$ as entries in the answer table used in the analysis framework (see Section 4.2.2), but they can be augmented with three pieces of extra bookkeeping information.

- An entry with an answer that is still safe, but might be improved through further compilation is *marked* and annotated with the $\bullet$ symbol.
- An entry with an answer that is no longer safe and should not be used is *invalid* and annotated with the $\perp$ symbol.
- An entry for a calling pattern for which no variant has yet been generated is annotated with a *version calling pattern* which indicates the most optimised variant that can be safely used in its stead.

This leads to the following five forms of registry entry.

- $P : D_P \mapsto Ans$ means the answer Ans is up-to-date, and the compiled variant of P is valid and optimal.
- $P : D_P \mapsto^{\bullet} Ans$ means the answer description Ans can be safely used, and the compiled variant of P is valid but possibly sub-optimal. This occurs when a more precise answer is obtained for a calling pattern that $P : D_P$ relies on. Recompilation of P's module may result in a better answer and a more highly optimised variant.
- $P : D_P \mapsto^{\perp} Ans$ means the answer description Ans cannot be safely used and the compiled variant of P is invalid. This occurs when a less precise or incomparable answer is obtained for a calling pattern that $P : D_P$ relies on. The module containing P must be recompiled before the final executable can be created.
- $P : D_P \mapsto^{\bullet} Ans\ (P : D'_P)$ means the calling pattern $P : D_P$ was encountered in another module, but there was no entry for it. Ans was the best answer we could obtain from the existing information, and $P : D'_P$ was the best safe variant we had generated code for. We will continue to use the answer Ans and the $P : D'_P$ variant until the module containing P is recompiled whereupon the exact answer will be found and the dedicated variant will be generated.
- $P : D_P \mapsto^{\perp} Ans\ (P : D'_P)$ means this once was a marked entry with a version calling pattern, but a procedure it relies on changed in a way that rendered it invalid. Ans and the generated $P : D'_P$ variant cannot be safely used.

In addition to the calling pattern entries, the analysis registry also has an overall status for each module. If any of a module's entries are invalid, the overall status is *invalid*; otherwise if any of its entries are marked, the overall status is *marked*; otherwise its status is *ok*. Also, if any previously generated variants of calling patterns not in the analysis registry are

affected by a changed answer in another module, the overall status can be changed. We will see how this can happen in the next section.

The analysis registry serves four distinct purposes.

1. It provides answers for external calling patterns.
2. It indicates which external variants have been generated, and for those that haven't, what is the best safe alternative that can be used.
3. It provides a way to mark/invalidate answers.
4. It provides a way to mark/invalidate previously generated variants.

One important point needs to be made regarding the analysis registry: *not all exported procedures need a registry entry.* Recall the discussion of *perfect calls* from Section 5.3.2 — a perfect call is one with the calling pattern $P : \top^*$ for which there is only one possible, or *innate* answer. Perfect calling patterns do not require a registry entry, for the following reasons corresponding to the four purposes above.

1. Their answers can be obtained immediately from the available information because they are innate.
2. Version calling patterns are never needed because the required $P : \top^*$ variant is always generated and can always be used without loss of accuracy.
3. Their answers cannot change without changing the procedure (and this case is handled separately, see Section 7.4.1), and thus need never be marked/invalidated.
4. Marking/invalidation of perfect call variants can be achieved by changing the module's overall status.

In the following sections, we will use two notational conveniences. Firstly the arrow $\mapsto^?$ will stand for an unmarked, marked, or invalid entry. Secondly, sometimes we will treat all registry entries as having a version calling pattern. Entries without one can be thought of as having the form $P : D_P \mapsto^? Ans\ (P : D_P)$.

7.3.2 Inter-module Dependency Graph

The inter-module dependency graph (IMDG) is a call dependency graph containing entries of the form $Q^{\mathtt{N}} : D_Q \longrightarrow P^{\mathtt{M}} : D_P$, where procedure Q exported from module `N` called with the calling description D_Q calls procedure P exported from module `M` with calling description D_P.[5] It is used to determine which modules should be marked or invalidated when an answer is updated. Note that only exported procedures are included and the calls need not be direct, e.g. $Q^{\mathtt{N}} : D_Q$ may call $P^{\mathtt{M}} : D_P$ indirectly via other (exported or non-exported) procedures in `N`. For example, all three call graphs in Figure 7.1 would result in the IMDG entry $Q^{\mathtt{N}} : D_Q \longrightarrow P^{\mathtt{M}} : D_P$ ('*' indicates the procedure is exported). The third call graph would also result in the additional entry $R^{\mathtt{N}} : D_R \longrightarrow P^{\mathtt{M}} : D_P$.

[5] We use the superscripted module name here to distinguish between procedures from different modules in the same IMDG entry.

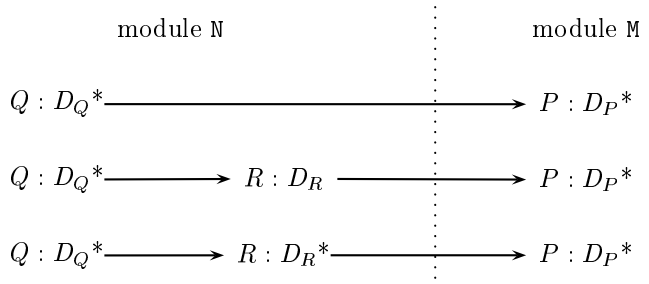


Figure 7.1: Inter-module dependency graph I

As for the registry, perfect calls are handled specially. If a perfect calling pattern is on the right-hand side of a $Q : D_Q \longrightarrow P : D_P$ dependency, the dependency need not be recorded. This is because the answer for $P : D_P$ is innate and thus cannot change without P's signature changing, in which case the analysis information is reset anyway (see Section 7.4.1). For example, consider the inter-module call graph in Figure 7.2, and imagine we are performing a simple groundness analysis. The call to `p` must be a perfect call, since the only

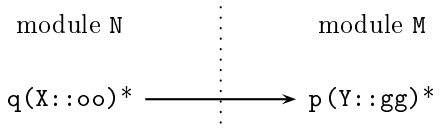


Figure 7.2: Inter-module dependency graph II

possible calling description is $\top^*$ (`Y` is ground), and it has an innate answer (`Y` is ground). So no $\mathtt{q} : D_\mathtt{q} \longrightarrow \mathtt{p} : \{\mathtt{Y}\}$ entry need be recorded.

If a perfect calling pattern is on the left-hand side of a $Q : D_Q \longrightarrow P : D_P$ dependency, or "interrupts" a dependency between exported procedures, the recorded dependencies take the form $(\mathtt{N}) \longrightarrow P : D_P$ in which the left-hand side's calling pattern is anonymous, and only the calling module is recorded. Consider the two call graphs in Figure 7.3. In the first

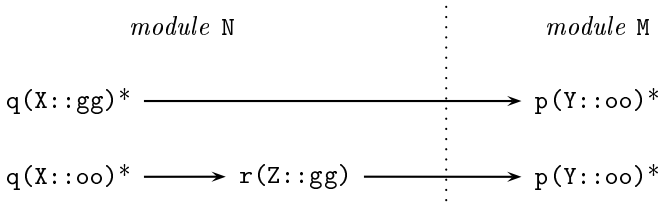


Figure 7.3: Inter-module dependency graph III

graph, `q` is a perfect call, and thus does not have a registry entry to mark/invalidate. If the answer to `p` changes, the answer to `q` will not, although any previously generated variant may be sub-optimal or invalidated. In the second graph `r` is a non-exported perfect call that "interrupts" the dependency between the exported procedures `q` and `p`; `r` does not have a registry entry to mark/invalidate because it is not exported. If the answer to `p` changes,

the answer to `r` will not, although any previously generated variant may be sub-optimal or invalidated.

In both cases, the IMDG dependency has the form $(\mathtt{N}) \longrightarrow \mathtt{p} : D_{\mathtt{p}}$. If $\mathtt{p} : D_{\mathtt{p}}$'s answer changes, `N`'s overall module status entry is updated to indicate that one or more previously generated variants have been marked/invalidated by a changed answer.[6]

7.3.3 Recording Information Between Compilations

Both data structures are divided into multiple pieces, one per module. The analysis registry for the module in `M.hal` is stored in the file `M.reg`; it contains the entries for the procedures exported from that module, as well as a single overall entry indicating whether the module is ok, marked or invalid. The inter-module dependency graph is stored in the file `M.imdg`; it lists the external callers of each of its procedures, i.e. entries of the form $Q^{\mathtt{N}} : D_Q \longrightarrow P^{\mathtt{M}} : D_P$, rather than the form $P^{\mathtt{M}} : D_P \longrightarrow Q^{\mathtt{N}} : D_Q$, so that when a calling pattern's answer is updated, we can easily find and mark/invalidate its callers.

7.3.4 Treatment of Special Modules

There are two kinds of modules that must be treated differently to ordinary program modules. The first is the `system` module, in which HAL built-ins are conceptually defined. We say "conceptually" because there is no actual `system.hal` file; the built-ins are implemented using a mixture of Mercury and C code in a file `system.m`, and `system`'s interface file `system.hint` is composed by hand. There is a `system.reg` registry file which is also composed by hand, and which we do not allow to be changed by compilation.

This is feasible because none of the system predicates call predicates from other modules, so their registry entries need never be marked/invalidated due to answers changing in other modules. The only shortcoming of making `system.reg` read-only is that new entries cannot be added for missing calling patterns — we have chosen to make it the analysis writer's responsibility to ensure `system.reg` contains suitable calling patterns; if any calling patterns not present occur in a program, the analysis results may be sub-optimal (since the answer for a less precise calling pattern will be used). We also assume that answers for built-ins will not change, so there is no need for a `system.imdg` file.

Library and solver modules also need special handling, but the treatment of their `.reg` and `.imdg` files is not so simple as for the `system` module. For the moment, we will treat them like `system`, and assume they have answers for the calling patterns likely to be encountered. In Section 7.5.2 we will consider this issue further.

[6] This overall status marker introduces one further complication — if performing incremental compilation incremental or analysis, when the answer for $\mathtt{p} : D_{\mathtt{p}}$ in Figure 7.3 changes `N`'s registry will be marked/invalidated, but there won't be a direct record of which external calling pattern caused the marking/invalidation, which will preclude incremental compilation/analysis of that module. The solution is to augment the module's overall status with the calling patterns of the external procedures with changed answers (in this case $\mathtt{p} : D_{\mathtt{p}}$). This information, when combined with other information that is required for incremental compilation/analysis, would suffice.

7.4 Compiling a Single Module

When compiling a module `M`, several steps must be performed. These are interleaved with the operations of the analysis framework. The steps are as follows.

1. Decide the initial set of calling patterns S to analyse.
2. Find an answer for an external call when Aexternal_proc is called.
3. Update the analysis registry.
4. Update the inter-module dependency graph.
5. Generate code for all necessary variants, such that each procedure call is made to the most specialised variant possible.

The following sections describe each of these steps in detail. For the moment we will assume that there are no procedure-level cyclic inter-module dependencies; that case is dealt with in Section 7.5.1.

7.4.1 Deciding Initial Contexts

If module `M` has been compiled before, has not changed since its last compilation, has no marked or invalidated entries, and its overall registry status is neither *marked* nor *invalid*, it does not need to be recompiled. Otherwise, we must decide the initial set of calling patterns S to analyse, as follows.

if (`M.hal` has changed since last compiled) **or** (`M.reg` is marked or invalid)
    foreach procedure P exported from `M`
        foreach $(P : D_P \mapsto^? \_\ (\_))$ in `M`'s registry
            add $P : D_P$ to S
        add $P : \top^*$ to S
    `analyse`(S)
else
    halt compilation

If the file needs analysis, we look at each exported procedure, and add the calling patterns for any entries in `M.reg` to S.[7] We also add $P : \top^*$ for each procedure (if already in S, it is not added twice). Note that for a procedure's `M.reg` entry to be added, its type, mode, determinism and number of arguments must exactly match an exported procedure in the source `.hal` file. Any entries for procedures that no longer exist are removed at this time.[8] Finally we call `analyse`, the entry function of the analysis framework.

[7] If performing incremental compilation or analysis, the compiler would only add those calling patterns with marked/invalidated answers or sub-optimal/invalidated previously generated variants, and any other calling patterns affected by these. This is not currently done since the HAL compiler does not do incremental compilation but generates code for all procedures in a module, thus requiring analysis information for all procedures.

[8] This may still leave behind unused contexts, from intermediate analysis results or from modules that are no longer of interest. A global step for removing these might be useful if the number of them becomes a problem.

7.4.2 Obtaining External Answers

When a call to external procedure Q defined in module `N` is encountered, instances of Aexternal_proc can call the function `get_ext_answer` to obtain an answer. If so, they must provide a function glb for computing the greatest lower bound of two answers, and a predicate is_more_precise that determines if its first argument is strictly more precise than its second. The file `N.reg` is read into memory if it has not already been.

```
get_ext_answer(Q : D_Q)
    if (there is a valid entry Q : D_Q ↦? Ans (_) in N's registry)
        return Ans
    else
        D_best := ⊤*;  Ans_best := ⊤*;  Ans_glb := ⊤*
        foreach valid entry Q : D'_Q ↦? Ans' (Q : D''_Q) in N's registry
            if is_more_precise(D_Q, D'_Q)
                Ans_glb := glb(Ans_glb, Ans')
                if is_more_precise(Ans', Ans_best)
                    D_best := D''_Q
                    Ans_best := Ans'
        if (N is not a library/system module)
            add Q : D_Q ↦• Ans_glb (Q : D_best) to N's registry
        return Ans_glb
```

If there is a valid entry that exactly matches the calling pattern, we return its answer. If there are no exactly matching entries, we consider all the entries in the analysis registry whose calling descriptions are strictly less precise than D_Q. As an answer we use the greatest lower bound of the answer descriptions of such entries. Because there is no variant for the requested calling pattern, we must find a version call pattern for a procedure that is safe to use; of the safe matches, we choose the one with the most precise answer description. When it comes to code generation, any call to $Q : D_Q$ can safely use $Q : D_{best}$. Finally, we add a marked entry to the registry, which ensures $Q : D_Q$ will be analysed next time `N` is compiled.

It is worth noting that [6] suggests that if the analysis domain is downwards closed, the calling description D_Q could be included in the greatest lower bound of all the inexactly matching answer descriptions. While this would result in a more accurate answer, in the end it will make no difference, because the answer description will be immediately combined with the call's pre-description which subsumes the information in D_Q.

7.4.3 Updating the Analysis Registry

Once analysis is complete, we update `M`'s registry. This may also require the marking of multiple other modules' registries, if they rely on procedures from `M`.

```
foreach P^M : D_P in S
    if (there exists an entry P^M : D_P ↦? Ans_old (P^M : D'_P) in M's registry)
        let Ans_new := answer for P^M : D_P in answer table
```

if $(Ans_{new} \neq Ans_{old})$ **or** $(D'_P \neq D_P)$
  replace the entry with $P^{\mathtt{M}} : D_P \mapsto Ans_{new}$ in M's registry
  if is_more_precise(Ans_{new}, Ans_{old})
    $Status := \bullet$
  else
    $Status := \bot$
  foreach entry $Q^{\mathtt{N}} : D_Q \longrightarrow P^{\mathtt{M}} : D_P$ in M's IMDG
    if (module N is not a library/system module)
      mark $Q^{\mathtt{N}} : D_Q \mapsto \_\ (\_)$ with *Status* in N's registry
  foreach entry $(\mathtt{N}) \longrightarrow P^{\mathtt{M}} : D_P$ in M's IMDG
    if (module N is not a library/system module)
      update N's registry's overall status with *Status*
elseif ($P^{\mathtt{M}} : D_P$ is not a perfect call)
  insert $P^{\mathtt{M}} : D_P \mapsto Ans_{new}$ in M's registry

For each calling pattern in S, if it had an old entry in M's registry we compare the new answer against the old. If the answer is the same, we don't need to do anything else. If the answer has changed, we replace the entry and change the status of all the procedures from other modules that call $P^{\mathtt{M}} : D_P$. If the new answer is strictly more precise or the entry has a version calling pattern (i.e. no specific variant has been generated for it before) we mark dependent entries; if it's strictly less precise or incomparable, we invalidate dependent entries. When marking/invalidating an individual entry with *Status*, we also update the overall marking of that module's registry by taking the least upper bound of the old status and the new *Status*, where the ordering is $ok \sqsubseteq marked \sqsubseteq invalid$. This is also how the overall status is updated when handling anonymous entries. Note that each dependent module's registry and IMDG are read from file if not already present in memory.

If there was no old answer, and the call is not perfect, we can insert the new answer (perfect calling patterns will never have an old answer); we don't need to worry about marking other modules — there cannot be any external callers of this calling pattern (or none encountered via the compilations performed so far), otherwise we would have at least one registry entry for it.

M's registry may have "legacy" entries in it, for calling patterns that are no longer encountered. Such entries can be removed at this point, as long as they are not of the form $P : \top^*$ — we must always have that entry and generate that variant, even if it's not currently used, so that other modules can use it in the future if necessary.

7.4.4 Updating the Inter-module Dependencies

We must also update the inter-module dependency graphs of all modules that M calls. This ensures that when answers from those modules change, `M.reg` can be marked/invalidated appropriately.

foreach module N imported by M
  if (module N is not a library/system module)

delete all entries of the form $P^{\mathtt{M}} : \_ \longrightarrow \_$ and $(\mathtt{M}) \longrightarrow \_$ in `N`'s IMDG
foreach $P^{\mathtt{M}} : D_P$ in S
  foreach non-perfect $Q^{\mathtt{N}} : D_Q$ reachable from $P^{\mathtt{M}} : D_P$
    if ($P^{\mathtt{M}} : D_P$ is a perfect call) **or** (the $P^{\mathtt{M}} : D_P \longrightarrow Q^{\mathtt{N}} : D_Q$ dependency
        is "interrupted" by a perfect call)
      add $(\mathtt{M}) \longrightarrow Q^{\mathtt{N}} : D_Q$ to `N`'s IMDG
    else
      add $P^{\mathtt{M}} : D_P \longrightarrow Q^{\mathtt{N}} : D_Q$ to `N`'s IMDG

For all modules imported by `M`, we remove any old IMDG entries of the form $P^{\mathtt{M}} : \_ \longrightarrow \_$ or $(\mathtt{M}) \longrightarrow \_$, and replace them with new entries for all calling patterns reachable from `M`.

Finding the reachable calling patterns is a non-trivial exercise. The inter-module call graph is computed by compute_call_graph (see Section 4.3) once analysis is completed. It is then transitively closed and non-exported procedures are removed. The reachable calling patterns of a module can then be easily extracted.

There is one minor difficulty when updating the inter-module dependency graph. If `M` used to import module `N` and call one of its procedures, but no longer does, `N.imdg` will have one or more entries of the form $P^{\mathtt{M}} : D_P \longrightarrow Q^{\mathtt{N}} : D_Q$ in it, and these won't be removed using the algorithm above. Then if the answer to $Q^{\mathtt{N}} : D_Q$ changes in the future, the registry entry for $P^{\mathtt{M}} : D_P$ will be marked/invalidated even though P no longer relies on Q. This problem can simply be ignored — it should occur rarely, and when it does it will only result in some unnecessary compilations without compromising correctness. This is the approach used by the current implementation. Alternatively, the above algorithm could be changed to process every `.imdg` file in the program rather than just the ones imported by `N`. Or the modules imported by `M` could be stored in `M.imdg` and if any are removed between compilations, all entries of the form $P^{\mathtt{M}} : D_P \longrightarrow \_$ could be removed from their `.imdg` files.

7.4.5 Generating Code

Now we must generate code for all the variants encountered during analysis. Only domains whose information can be used to optimise constraints require multi-variant specialisation, such as Freeness^{HAL} (see Chapter 5). By contrast, determinism analysis (see Chapter 6) does not. The following description is for multi-variant specialisation without any restriction on the number of variants generated. Techniques for reducing the number of variants are discussed in Section 7.10.

The HAL compiler uses name mangling to differentiate between different variants. To support this, each abstract domain for which multi-variant specialisation is performed must provide an injective function desc_to_string that converts an abstract description to a string. The strings for each abstract domain the procedure is being specialised for are concatenated and used as a suffix.

The three tasks relating to multi-variant specialisation during code generation are as follows.

1. Determine which variants to generate.

2. Specialise each procedure call and higher-order unification to use the best possible variant.

3. Optimise "constraints" according to annotations.

Determining the variants: After analysis, all variants encountered are recorded by compute_call_graph (Section 4.3). The first step during code generation is to determine the names of the variants to be generated.

foreach P in module M
  foreach combination Ds of calling descriptions, one per analysis domain
    generate code for the variant named `local_variant_name`(P, Ds)

`local_variant_name`(P, Ds)
  Suffix := `""`
  foreach D_P in Ds
    Suffix := *Suffix* ++ `desc_to_suffix`(D_P)
  return *Suffix*

`desc_to_suffix`(D_P)
  if $(D_P = \top^*)$
    return `""`
  else
    return desc_to_string(D_P)

There is one complication when generating variant names: if the calling description is $\top^*$, we do not use any suffix; this corresponds to the "default" variant. The main reason for this is that it allows modules that have undergone multi-variant specialisation to be linked with modules that have not.

For example, if generating the variants for a procedure `p` for which the encountered calling descriptions were $\top^*_A$ and D_A from analysis domain A, and $\top^*_B$ and D_B from analysis domain B, where the strings returned by desc_to_string for D_A and D_B are `"_dA"` and `"_dB"` respectively, we will generate four variants with the following names:

`local_variant_name` (`"p"`, $[\top^*_A, \top^*_B]$) = `"p"`
`local_variant_name` (`"p"`, $[\top^*_A, D_B]$) = `"p_dB"`
`local_variant_name` (`"p"`, $[D_A, \top^*_B]$) = `"p_dA"`
`local_variant_name` (`"p"`, $[D_A, D_B]$) = `"p_dA_dB"`

Specialising procedure calls: When a procedure call P (local or external) is encountered, we inspect its annotations to determine Ds, its calling descriptions for all relevant analysis domains. If P is local we can simply convert each description to a suffix using `local_variant_name` — we know that the appropriate variant will be generated, because it must have been encountered during analysis. If P is external, the exact variant may not have been generated — this is where the version calling pattern is used.

`external_variant_name`(P, Ds)
    Suffix := ""
    foreach D_P in Ds
        if (there exists a valid entry $P : D_P \mapsto^? \_$ in `M.reg`)
            Suffix := *Suffix* ++ `desc_to_suffix`(D_P)
        elseif (there exists a valid entry $P : D_P \mapsto^? \_\ (P : D'_P)$ in `M.reg`)
            Suffix := *Suffix* ++ `desc_to_suffix`(D'_P)
    return *Suffix*

We use the best safe variant possible, as indicated by the version calling pattern $P : D'_P$. If a local call was unreached during analysis, it will have no annotations, so Ds will be empty and the empty suffix will be used. There will always be an entry for an external calling pattern $P : D_P$, as it must have been added when the call was encountered during analysis.

Higher-order unifications are treated similarly so that the closure created uses the best safe variant possible.

Optimising constraints: When an optimisable constraint is encountered, we inspect its annotations to determine if it can be improved. For example, when performing the freeness optimisations discussed in Section 5.2.3, if the literal is a `unify_oo` and the annotations indicate that both its arguments are free, we replace it with `unify_var_var`.

7.4.6 Creating An Executable

Any modules containing invalid entries must be recompiled; once all modules are valid, the generated code files can be linked together into an executable (i.e. passed to the Mercury compiler). If no modules are marked, the executable created will be optimal.

7.4.7 An Example

We are now ready to put everything together with an example. Although the compilation model has a lot of details, in practice the workings are quite straightforward. Consider a simple groundness analysis of a program consisting of the following two modules:

```
%-------------------------%
:- module m_app.
:- herbrand list/1.

:- export pred p(list(int)::oo, list(int)::oo) is nondet.
p(X, Y) :- app([4,5,6], X, Y).

:- export pred app(list(T)::oo, list(T)::oo, list(T)::oo) is nondet <= eq(T).
app([], _, []).
app([X | Xs], Ys, [X | Zs]) :-
    app(Xs, Ys, Zs).
```

```
%-------------------------%
:- module m_app_caller.
:- import m_app.
:- herbrand list/1.

:- export pred app_caller is nondet.
app_caller :- app([1], [2,3], [1,2,3]).
```

The "`<= eq(T)`" annotation on `app/3` indicates that the elements of its arguments must be members of the `eq` type class, which means that they support being tested for equality.

Rather than using head variables in the descriptions, we will use the numbers 1, 2 and 3 to represent the first, second and third argument of each procedure respectively. Also, we assume the suffixes generated by `desc_to_suffix` are of the form "`_GiGjGk`" if the variables i, j and k are ground.

Compile `m_app`: We begin with no `.reg` or `.imdg` files. Let us arbitrarily choose to compile `m_app` first. The initial set S contains only default entries for `p` : $\top^*$ and `app` : $\top^*$ (where $\top^* = \emptyset$). The compilation generates `m_app.reg`.

`m_app.reg`

Status = *ok*

`p`	: $\emptyset$	$\mapsto$	$\emptyset$
`app`	: $\emptyset$	$\mapsto$	$\emptyset$
`app`	: $\{1\}$	$\mapsto$	$\{1\}$

The entry for `app` : $\{1\}$ arises from the `app([4,5,6], X, Y)` call. No `m_app.imdg` is generated as yet. The variants generated are `p`, `app` and `app_G1`. The `app_G1` variant will be more optimised than `app`, since its first argument is known to be ground.

Compile `m_app_caller`: The initial set S contains only the default entry `app_caller` : $\top^*$ (where $\top^* = \emptyset$). There is no `m_app_caller.reg` generated because `app_caller` is a perfect call and does not require a registry entry. The compilation updates `m_app.reg` and generates `m_app.imdg`.

`m_app.reg`

Status = *marked*

`p`	: $\emptyset$	$\mapsto$	$\emptyset$
`app`	: $\emptyset$	$\mapsto$	$\emptyset$
`app`	: $\{1\}$	$\mapsto$	$\{1\}$
`app`	: $\{1,2,3\}$	$\mapsto^\bullet$	$\{1\}$ (`app` : $\{1\}$)

`m_app.imdg`

(`m_app_caller`) $\longrightarrow$ `app` : $\{1,2,3\}$

The previously unseen calling pattern `app` : $\{1,2,3\}$ from the `app([1],[2,3],[1,2,3])` call is added to `m_app.reg` and marked (this marks the overall status too). For its answer we use the greatest lower bound of the answers of all existing `app` entries with less precise calling

descriptions (in this case `app` : $\{1\} \mapsto \{1\}$ and `app` : $\emptyset \mapsto \emptyset$).[9] Of these entries, we use the calling pattern of the one with the best answer (`app` : $\{1\}$) as the version calling pattern. The generated code for `app_caller` calls `app_G1`, the most highly optimised existing safe variant of `app` that it can.

The compiled modules could now be linked together into a sub-optimal executable; but to obtain an optimal executable, we must do some recompilations. Since `m_app.reg`'s overall status is *marked* we should recompile it next.

Recompile `m_app`: This time the initial set S contains the four entries from `m_app.reg`. Recompiling updates `m_app.reg` and generates a new `m_app_caller.reg`.

`m_app.reg`

Status = *ok*

`p`	:	$\emptyset$	$\mapsto$ $\emptyset$
`app`	:	$\emptyset$	$\mapsto$ $\emptyset$
`app`	:	$\{1\}$	$\mapsto$ $\{1\}$
`app`	:	$\{1,2,3\}$	$\mapsto$ $\{1,2,3\}$

`m_app_caller.reg`

Status = *marked*

When `m_app.reg` is updated, the marked entry for `app` : $\{1,2,3\}$ is unmarked, its answer updated, and its version calling pattern removed. We then look in `m_app.imdg` to determine which other modules are affected. The only entry in `m_app.imdg` has the form (`m_app_caller`) $\longrightarrow$ `app` : $\{1,2,3\}$ which matches the calling pattern, so we change the overall status of `m_app_caller`'s registry to *marked*, creating the `m_app_caller.reg` file in the process. This indicates that the generated code for `m_app_caller` is still safe, but recompilation could improve it. The following variants are generated: `p`, `app`, `app_G1` and `app_G1G2G3`.

Again, the compiled modules could be linked together into a sub-optimal executable — although the variant `app_G1G2G3` has been generated, `m_app_caller` needs to be recompiled to actually use it.

Recompile `m_app_caller`: The initial set S again only contains the default `app_caller` : $\top^*$ entry. The recompilation changes `m_app_caller.reg`.

`m_app_caller.reg`

Status = *ok*

The code for `app_caller` can now call the most optimal variant of `app`, `app_G1G2G3`, and `m_app_caller.reg` is marked as *ok*.[10]

No modules are marked or invalid, so the compiled modules can now be linked together to create the optimal executable. Note that if we had compiled `m_app_caller` first, the optimal executable could have been made with only three compilations.

[9] As mentioned in Section 7.4.2, because groundness is downwards closed, we could include the calling description in the greatest lower bound computation, in which case we would obtain the better answer {1,2,3}. However, this would not improve the analysis since the answer will be combined with the calling description immediately.

[10] The file `m_app_caller.reg` could be deleted at this point.

7.4.8 Use and Update of `.reg` and `.imdg` Files

The actions that accompany compilation of a single module can be quite complex; it is thus worthwhile to provide an overview. Assume we have three modules `L`, `M` and `N`, and that `L` imports `M`, and `M` imports `N`, as shown in Figure 7.4.

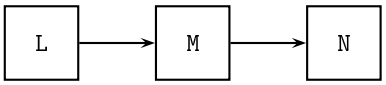


Figure 7.4: Dependencies between modules `L`, `M` and `N`

When compiling module `M`, the `.reg` and `.imdg` files are used as follows.

- `L.reg`, `L.imdg` and `N.imdg`: not used.
- `M.reg`: used to determine the initial set of calling patterns S.
- `M.imdg`: used to determine which variants in `L` are affected by answer changes in `M`.
- `N.reg`: used during analysis for finding exact or approximate answers for external calls. Also used during code generation to determine which of `N`'s variants `M`'s procedures can use.

The changes made to the `.reg` and `.imdg` files are as follows.

- `L.imdg` and `M.imdg`: not changed.
- `L.reg`: entries are marked or invalidated if they rely on calling patterns from `M` whose answers have changed.
- `M.reg`: marked/invalid entries are replaced with unmarked ones. Procedures missing entries have one added for the calling pattern $P : \top^*$.
- `N.reg`: missing entries are added and marked.
- `N.imdg`: old entries of the form $P^{\mathtt{M}} : D_P \longrightarrow Q^{\mathtt{N}} : D_Q$ and $(\mathtt{M}) \longrightarrow Q^{\mathtt{N}} : D_Q$ are removed and replaced with new entries.

It is clear that the use and update of each module's `.reg` and `.imdg` files are quite independent.

7.5 Complications

There are two complications that have been avoided so far: the handling of invalid entries when cyclic dependencies are present in the program call graph, and the treatment of library and solver modules. We discuss them in this section.

7.5.1 Cyclic Dependencies

If there are procedure-level cyclic inter-module dependencies in the program's call graph, care must be taken when recompiling modules so that invalid information is not incorrectly used to obtain new answers. The danger is that we might indirectly use an invalid old answer for a calling pattern to support its new answer.

When an inter-module cycle has one or more registry entries invalidated, the simplest solution is to throw away all possibly invalid information. For each $P : D_P$ entry in the strongly connected component (SCC) containing the invalid entry, we reset the entry to $P : D_P \mapsto^\bullet \top^*$.

For example, consider the cycle in Figure 7.5. Assume the cyclic inter-module dependency is at a procedure level, and that an answer changes in `P`, invalidating an entry in `N`. We would then reset all entries in `L`, `M`, and `N` to $P : D_P \mapsto^\bullet \top^*$. Any module in the SCC can now be safely recompiled.

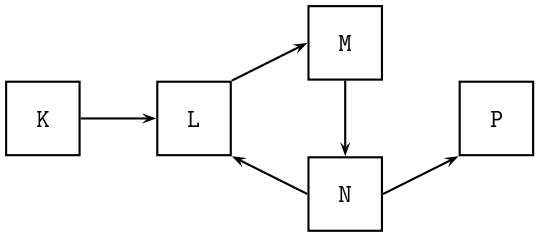


Figure 7.5: A cyclic inter-module dependency graph

Unfortunately, this simple approach may lead to more recompilation than necessary. A better approach is to choose one module with which to break the SCC; registry entries from other modules within the SCC that it depends on are reset as above. Once this is done, that module can then be safely recompiled, as it will not be relying on any invalid information within the SCC. The other modules can then be recompiled in turn.

For example, again consider the cycle in Figure 7.5, and again assume an answer in `P` changes, invalidating an entry in `N`. We can break the cycle at module `N` by resetting all the entries it depends on within the SCC (those in `L`) to $P : D_P \mapsto^\bullet \top^*$. `N` can then be recompiled safely, followed by `M`, `L` and `K` (if necessary).

Note that SCCs can be computed fairly straightforwardly using the dependency information already present in the `.imdg` files.

7.5.2 Library and Solver Modules

There are several problems when dealing with library modules.

- Normal modules have the rest of the program to provide calling contexts. Library and solver modules do not. How should appropriate entries be added to the `.reg` file?
- Library and solver modules may be used by many different programs. If the calling pattern of every procedure that calls the library/solver module is added, the `.imdg` file may grow very large.

- Library/solver modules are not likely to be recompiled very often, if at all.

Let us consider two possible solutions.

Treat like the `system` module: We could treat the library/solver module like the system module, by making the module's `.reg` file read-only, and not having a `.imdg` file. A declaration such as

```
:- library_module <module_name>.
```

could be used to flag a module as being a library or solver module. The appropriate entries could be added to the `.reg` file by hand. If this is difficult, we could instead follow these steps to generate it semi-automatically.

1. Write the library module without the `library_module` declaration.
2. Write a driver program that calls all the appropriate calling patterns.
3. Compile the driver program so that appropriate entries are added to the library module's `.reg` file.
4. Compile the library module so that answers are obtained and variants generated for the appropriate calling patterns.
5. Add the `library_module` declaration to the library.

The library is now ready to be used like the `system` module.

This approach would avoid the overfull `.imdg` problem, although if the answer for any calling patterns changed, there would be no way to determine which modules would be affected. The module also would never require recompilation as long as it didn't change. This solution would be most suitable for highly stable libraries.

Make local `.reg` and `.imdg` copies: Another possibility would be to create a local copy of the module's `.reg` and `.imdg` files for each program, and it could then be used like a normal program module. Missing calling patterns could be added easily, and local copies of the compiled module containing only the variants used would be generated. The library/solver module would need to be compiled once or more per program, unlike the previous approach. This solution would be most suitable for library/solver modules with procedures that might have many different calling patterns, which are used in very different ways by different programs.

Current approach: Neither of the suggested approaches have actually been tried. This is because the only analyses implemented so far are Herbrand analyses (see Chapter 5). None of the procedures in HAL's standard library use `old` modes, so they all have perfect answers, which means `.reg` and `.imdg` files are not needed!

Once other analyses are written, the two approaches above and others can be tried. Determining the best approach, the best situation for each approach, or the best combination of approaches, is clearly an area for further work.

7.6 Correctness and Accuracy

The correctness of the described approach is quite obvious in the absence of invalidated entries — the answers used for inter-module calls proceed downwards from $\top^*$, becoming increasingly more accurate, until we reach a fixpoint.

The approach's correctness when invalidated answers are involved is less obvious, but can be established by the following intuitive arguments that show we never use invalid information or invalid variants.

- We forbid the creation of an executable when any of its constituent modules have invalid registry entries.
- Invalid answers are never directly used — if an entry is invalidated, we instead use $\top^*$ as its answer.
- When an answer changes in a way that may compromise the correctness of the modules that depend on it (i.e. it is strictly less precise or incomparable with the old answer), we invalidate all entries that directly rely on it.
- When a module is invalid, any module that (directly or indirectly) depends on it may also be invalid. In the absence of procedure-level cyclic inter-module dependencies, this will be all modules "above" the invalid module in the inter-module dependency graph. Recompiling this "bottom" invalid module will re-validate it (it does not rely on any invalid information), although when recompiled one or more of its answers may change, which will spread the invalidation to its calling modules. The newly invalidated modules can then also be recompiled and re-validated. The invalidation propagation will eventually cease, whereupon a correct executable can be created.
- If cyclic dependencies are present, to recover a valid state we choose one module with which to break each cycle; the invalid entries within the cycle that it depends on are then "reset" and the chosen module can be recompiled safely, as it no longer depends on any invalid information from within the cycle.

There is one further simple but important point required for correctness. If a module has *any* compilation errors, analysis is skipped. This ensures the registry and inter-module dependency graph cannot be polluted with incorrect information.

Now let us consider accuracy. Note that the fixpoint found by this technique is *not* the global greatest fixpoint, but much more accurate than that. It is instead a mix of least fixpoint information from intra-module analysis and greatest fixpoint information from inter-module analysis. In general, it is not as accurate as a method that finds the global least fixpoint. However, unless there are procedure-level cyclic inter-module dependencies present in a program, the fixpoint found will be equal to the global least fixpoint. One could argue that the presence of such dependencies is indicative of poor modular structure, and should be avoided anyway. We feel the advantage of our technique — being able to create a safe executable after only compiling each module once — outweighs the occasional accuracy shortcomings.

7.7 Efficiency Considerations

The analysis registry and inter-module dependency graph are accessed and updated in many different ways throughout compilation, which makes it difficult to identify what is the most appropriate structure for them, both in memory and on disk.

The data structures used in the implementation are as follows. Analysis registry entries are addressed via four levels of indexing: by module, then domain, then procedure, then calling description. This representation is used both in memory and in the `.reg` files. Of course, on disk the primary indexing by module is achieved by storing each module's registry in its own file.

Only one module's IMDG needs to be in memory at any time. This is because the IMDG of the module being compiled is used when updating the analysis registry, and the IMDGs of all called modules are updated one at a time at the end of the different domain analyses. Each module's IMDG has two levels of indexing: by domain, then calling module. Each calling pattern is then paired with a list of its callers from the calling module — this list may be empty if all the callers are anonymous.

However, it seems that the exact form of the registry and IMDG in memory is not very important, simply because the number of entries is generally quite small — a typical module exports only a handful of procedures, or even only one. By not storing perfect calls, the number of entries is reduced even further.

Processing times are dominated by the time taken to read and write `.reg` and `.imdg` files. The implementation has three features to reduce the amount of reading and writing.

- The registry and inter-module dependency graphs are broken into pieces, one per module. At first, we stored a module's registry and IMDG together in a single `.info` file. Once it was recognised that a module's `.reg` and `.imdg` are often used (i.e. read from disk) and updated (i.e. written to disk) independently (see Section 7.4.8), the decision was made to use separate `.reg` and `.imdg` files. The increased number of files opened and closed is outweighed by the time saved by avoiding reading and writing unnecessary bytes.

- Each `.reg` and `.imdg` file is only read into memory when necessary. This minimises the memory required to store the structures. Also, `.reg` and `.imdg` files are only written out to disk again if they have changed, to minimise write times.

- The overall status of each module's registry is stored at the very start of its `.reg` file, where it can be quickly accessed. The use of this is described in the next section.

7.8 Controlling Compilation

Let us now consider the order in which modules should be compiled under this compilation model, how to determine if an executable can be created, and if so, whether the executable will be optimal.

If the modules of a program are not edited, they can be compiled in any order and any number of times, and then be linked together to create a valid executable. However, the only

way to determine whether an executable is optimal is by checking that all registry entries are not marked.

If any module is edited, registry entries may be invalidated. An executable cannot be created if any modules have been invalidated. The only way to determine this is by checking that no registry entries are invalid.

If there are invalid entries combined with cyclic dependencies, the cycle must be identified and broken in the manner described in Section 7.5.1.

With the current implementation, these three tasks must be performed manually by the programmer. The long-term plan is for HAL to have some kind of overarching build tool (such as Mercury's `mmake` [21], based on `make` [15]) to control these tasks automatically. Like `mmake` and other build tools, it should recompile modules that have changed. It should also perform the three tasks described above. The first two can be achieved easily by looking at the overall status in each `.reg` file of the program's modules. The third would be more involved, but since the necessary dependency information is already present in `.imdg` files, identifying and breaking cycles should be relatively straightforward.

When developing a program, it is likely that a programmer will need to create a fully optimised executable — which may require each module to be compiled more than once — comparatively rarely. Thus, control over the compilation speed vs. inter-module optimisation level trade-off would be useful. It would also be useful for a build tool to attempt to minimise the number of recompilations by using analysis information present in the `.reg` and `.imdg` files.

This is only intended as a brief discussion of the features that a build tool would need; the interaction between HAL's inter-module analysis system is a topic deserving much further work.

7.9 Experimental Evaluation

This compilation model was designed for analysing large multi-module HAL programs. Currently, the only such program is the HAL compiler itself. However, it is not suitable for analysis by the Herbrand analyses implemented so far, since the small number of procedures it uses with `old` instantiations are not exported from their modules — hence no `.reg` or `.imdg` files would need to be written.

As an alternative, we took a single-module HAL program that uses Herbrand constraint solving extensively, `icomp`, a cut-down version of an interactive BIM compiler by Bart Demoen, and split it into multiple modules in three different ways to evaluate different aspects of the model — the time required to compile a multi-module program, the effect of the module structure on the number of compilations required to reach the fixpoint, and the effect of procedure-level cyclic inter-module dependencies on accuracy.

Note that this is only a preliminary evaluation, although the results are encouraging. When multi-module programs that require full inter-module analysis and optimisation are written, a more thorough evaluation should be done to further assess the technique.

7.9.1 Cost of Compilation

For the first experiment, we split `icomp` into five modules in a logical fashion that minimised the size of interfaces (each module exported only one procedure). There are five strongly connected components containing more than one procedure in `icomp`; four of them contain two procedures, and one contains four procedures. No strongly connected components were split across modules. The module structure is shown in Figure 7.6.

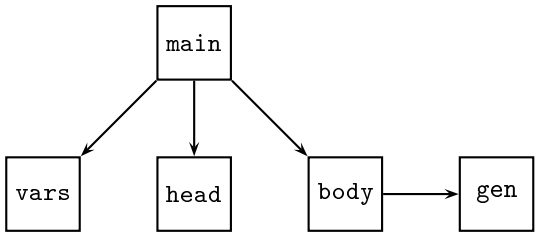


Figure 7.6: A logical module structure for `icomp`

The compilation order chosen was that mentioned at the end of Section 7.4.7 — calling contexts were found and propagated top-down through the inter-module call graph and then missing answers were filled in bottom-up; this was repeated until the fixpoint was reached. This gave the order `main`, `vars`, `head`, `body`, `gen`, `body`, `main`, `vars`, `head`, `body`, `gen`, `body`, `main`, for a total of thirteen compilations.

The analysis and compilation times for each compilation were compared with the times for analysis and compilation of the original single-module version of `icomp`. The experiments were performed under the same conditions as those in Section 5.6.1.

The times results are shown in Table 7.1. Column two and three give the number of predicates and literals (before normalisation) of each module. Column four gives the time for Freeness^{HAL} analysis. Column five gives the overall compilation time. Column six gives the proportion of compile time taken up by Freeness^{HAL} analysis. The first row gives the result for the single-module `icomp`.[11] The remaining rows give the times for the compilation of each module when computing the inter-module analysis fixpoint.

The second and subsequent compilations of each module each took longer than the first compilation, due to the extra calling patterns analysed. The figure of 50430 ms for full compilation of the modularised version is 3.9 times greater than 13030 ms for the single-module version, even though only a little more than twice as much code is compiled. This is largely due to the reading and writing of interface files and the compilation of extra variants. On the other hand, the overall analysis time of 16180 ms is only 2.7 times greater than the 6000 ms for the single-module version. This difference reduces the analysis time proportion down from 46.0% for the single-module version to 32.1%. The accuracy of the analysis is identical to the single-module version, because there are no procedure level cyclic inter-module dependencies. However, unnecessary $P : \top^*$ variants were generated for the single procedure exported from each of `vars`, `head`, `body` and `gen`.

[11] The result is slightly different from that in Table 5.4 because this version contained an extra predicate containing a query.

Module	Preds	Lits	F	All	Prop
`icomp`	31	120	6000	13030	46.0%
`main`	3	16	210	2630	8%
`head`	6	18	650	2740	23%
`vars`	5	15	400	1640	24%
`body`	12	52	1100	4010	27%
`gen`	5	19	1570	4300	36%
`body`			1310	4170	31%
`main`			280	2790	10%
`head`			930	3290	33%
`vars`			510	1860	27%
`body`			2260	5900	38%
`gen`			1700	4490	37%
`body`			2270	5810	39%
`main`			270	2650	10%
Total/Average			16180	50430	32.1%

Table 7.1: Multi-module analysis times (ms)

These results do not hold any great surprises, but they do support our claims that the compilation model is practical.

7.9.2 Effect of Module Structure

For the second experiment, we split `icomp` into four modules in a totally random fashion. The only restriction was that we did not split any strongly connected components across modules. Every module called at least one procedure from every other, which meant that 24 of the program's 31 predicates had to be exported from their module.

The most significant effect of this random module structure was the high number of compilations required to reach the fixpoint — both module compilation orders attempted required thirty compilations to do so. A number of unnecessary variants were also generated to reach the fixpoint. This indicates that the model's behaviour does degrade in the presence of unclean module structure, however the effect is not overwhelming, especially considering that this is a pathological case and in practice module structures are not nearly so random. Also, `icomp` is unusual in that it contains no perfect calls to hasten the fixpoint computation.

7.9.3 Effect of Cyclic Dependencies

In our third experiment, we kept all procedures in a single module except for those involved in strongly connected components of size greater than one; the procedures in these SCCs were split across different modules.

The effect on accuracy was as expected. While the answers for some calling patterns from the SCCs containing more than one procedure were the same, every such SCC had at least one answer that was less accurate than that obtained for the single-module version. The differences in answers were fairly minor, mostly being that the multi-module version answers had more sharing pairs than the single module version answers.

7.10 Conclusion

The compilation model used by HAL for inter-module analysis and optimisation, based on finding the greatest fixpoint of inter-module analysis information, is both practical and accurate, and more satisfactory than a number of other approaches used by other programming language implementations. It solves the two main problems of inter-module analysis described in Section 7.1.

1. The unknown answers problem is solved by storing calling pattern answers in the analysis registry, in the form of `.reg` files. When an answer is obtained for a calling pattern, the registry is updated. Answers can be marked or invalidated to indicate that they are sub-optimal or no longer safe.

2. The unknown contexts problem is solved by using $\top^*$ as the default calling pattern, and then allowing modules to mark the registries of other modules (in effect "telling" them) when a new calling context is encountered. Once an answer is obtained for the new context, the module containing the procedure notifies the calling module by marking its registry.

The compilation model also supports multi-variant specialisation, by providing version calling patterns that indicate whether specific variants exist, and if not, the best variants that can be safely used instead.

The implementation described in this chapter differs somewhat from the original version of the compilation model presented described in [6]. The main differences are: it supports perfect calls; provides more detail about the storage and handling of the registry and inter-module dependency graph; provides more algorithmic detail; and considers the special treatment of system, library and solver modules. These differences mostly arise because this is a description of a specific implementation, rather than a general presentation.

One topic we have not considered is the number of variants generated per procedure by multi-variant specialisation. If there are too many, the number could be restricted to the $P : \top^*$ variant plus a limited number of more specialised variants. This could be done by using unmarked registry entries of the form $P : D_P \mapsto Ans\ (P : D'_P)$ (which were not considered in Section 7.3.1). In this case, any call to $P : D_P$ would actually use the variant $P : D'_P$; however, because the entry is not marked, the more specialised $P : D_P$ variant would not be generated when the module containing P was recompiled.

Similarly, different variants of a procedure may have identical code if their calling descriptions are similar. While this has not been a problem in our experience thus far, it may become an issue for different abstract domains. It might be worth investigating ways to identify variants that are equivalent from an optimisation point of view and then "collapse" them together in a similar way to the variant capping described above (as in e.g. [51, 36]).

The preliminary experimental evaluation showed that the model is quite practical, and that while its performance does degrade as module structures become messier, the effect is not overwhelming and unlikely to be a problem in practice. Once suitable multi-module programs are written, the model should be evaluated more thoroughly to identify possible improvements.

Finally, we have identified two important areas for further work. The first is the testing of different approaches to handling library and solver modules. The second and more important task is the development of a proper build tool to automate compilation; as well as the standard `make`-like tasks of recompiling changed modules, it should be able to recognise whether further compilation could result in a more optimised executable, identify invalidated modules that need to be recompiled, and be able to break invalid cyclic inter-module dependencies. Once these two shortcomings are addressed, this compilation model should provide all the necessary support for practical and accurate inter-module analysis and optimisation.

Chapter 8

Conclusion

In this thesis we have presented the HAL compiler's generic analysis framework. In Chapter 3 we gave it a solid theoretical foundation by defining semantics for top-down and bottom-up analysis. In Chapter 4 we described in detail how the framework accommodates different analysis domains, and the algorithm and data structures it uses to efficiently compute a module's least analysis fixpoint. In Chapter 5 we defined three analyses for optimisation — groundness, sharing and freeness. We also described and evaluated the Herbrand constraint solving optimisations made possible by the gathered analysis information. In Chapter 6 we described HAL's determinism analysis, an analysis for correctness that also allows the use of more efficient execution algorithms for procedures that are deterministic or semi-deterministic. Finally in Chapter 7 we presented the compilation model used in the HAL compiler that allows accurate inter-module analysis in the presence of separate compilation.

Let us consider how well we have achieved the goals we set ourselves in Chapter 1.

- *A clear justification of the form of the framework.* In Section 3.1, we identified necessary and desirable analyses for HAL, and decided that a generic abstract interpretation-based framework capable of performing top-down and bottom-up analyses was most appropriate for HAL.

- *A sound theoretical basis for the framework.* Section 3.3 introduced abstract interpretation, and defined top-down and bottom-up semantics for the framework.

- *A comprehensive description of the implementation of the framework.* Chapter 4 thoroughly described the data structures and operations of the framework, including detailed pseudocode of its main algorithm.

- *Detailed descriptions of analyses implemented within the framework, and any enabled optimisations.* Chapter 5 described in detail groundness, sharing and freeness analyses, and the optimisation of Herbrand constraint solving. Chapter 6 described determinism analysis, and briefly covered the improved execution algorithms it enables.

- *Empirical evaluation of the costs and benefits of these analyses.* Chapters 5 and 6 provided experimental results for both the cost and benefits of these analyses.

We also identified several awkward details for our presentation to tackle.

- *Analysis of full programs, without any restrictions on language features used.* The framework handles full HAL programs containing arbitrarily nested compound bodies including explicit disjunctions and if-then-elses. Advanced language features such as dynamic scheduling and constraint handling rules did not require special treatment, since they are transformed into ordinary language constructs before the framework is invoked.

- *The use of higher-order programming.* Higher-order unifications and calls were considered throughout; for example, procedures only reachable through higher-order calls are recognised and analysed by the framework. Unfortunately, accurate analysis of programs that use higher-order is intrinsically difficult. For example, often the predicate called via a higher-order argument cannot be determined at compile-time. Our approach was pragmatic; we used what information we could (e.g. by keeping downwards closed information from higher-order unifications), but did not go to great lengths to improve upon this.

- *Practical and accurate inter-module analysis.* This goal was very nearly achieved in full. The compilation model described in Chapter 7 is as accurate as a global analysis in the absence of procedure-level cyclic inter-module dependencies, which should be rare.

  The model has the potential to be very practical. Although it may require modules to be compiled more than once to achieve full optimisation, less optimised executables can be created after each module is compiled only once. Since a fully optimised executable is likely to be needed only occasionally, compilation times should not be affected excessively. The generated `.reg` and `.imdg` files are typically small, as is the cost of reading, updating and writing them. The compilation model also robustly handles arbitrary changes to modules by invalidating unsafe analysis information. However, in order for the model to become fully practical, its workings should be completely transparent to the programmer; this will require the development of a `make`-like build tool to control compilation and find the necessary global analysis fixpoint.

- *Appropriate treatment of library modules.* Two possible approaches to handling library modules when performing inter-module analysis were discussed in Section 7.5.2.

- *The use of programmer declarations to improve the accuracy and efficiency of analysis.* The three analysis domains Def^{HAL}, ASub^{HAL} and $\mathsf{Freeness}^{HAL}$ presented in Chapter 5 all used groundness information from mode declarations to augment the information inferred. This was particularly important for efficiency, as the cost of Herbrand analysis jumped considerably when `oo` modes were used frequently.

- *Efficiency considerations, to minimise the cost of analysis.* A strong emphasis was put on efficiency. Section 4.9 discussed six separate implemented optimisations to the basic framework; some made consistent improvements, others made drastic improvements for certain cases. The $\mathrm{DBCF}_{\mathsf{Def}}$ representation was chosen for Def^{HAL} descriptions in

Section 5.3.1 because previous research had shown it to be the most efficient representation of Def. The add_{ss} operation for structure sharing from Section 5.4.2 was specialised so it was not defined in terms of the inefficient $conj_{ss}$ operation. Section 7.7 discussed efficiency considerations of inter-module analysis, the most important of which was that the structure of `.reg` and `.imdg` files were chosen to minimise disk reading and writing.

In addition to these points, we were careful to distinguish between those techniques that have been implemented and those that have not.

This thesis has made two primary contributions. The first is that it provides a comprehensive description of a generic analysis framework and analysis domains as they are implemented for a real compiler, without ignoring any details such as explicit disjunctions, if-then-elses, higher-order unifications and applications, and so on. The second contribution is that it provides the first detailed description of an implementation of the compilation model used for inter-module analysis and optimisation, including details such as perfect calls, the use of `.reg` and `.imdg` files, how to handle modules that require special treatment, and other important details.

Finally, we have identified several further areas for future work. To improve the accuracy of Herbrand analyses, a Pos based groundness analysis could be implemented; also, type-based analysis [37] might give better results. To improve the accuracy of determinism analysis, sharing and freeness information could be used; if multi-variant specialisation performed for Herbrand optimisations was performed before determinism analysis, different variants of one procedure could have different determinisms and thus use different execution algorithms.

As for further optimisations, an analysis that uses sharing and determinism information to identify when `notrail` unifications can be used would be worth implementing since we found them to be 2.9–4.5 times faster than the normal versions.

HAL was designed to support easy experimentation with different solvers and constraint solving methods. With a robust, efficient generic analysis framework in place, it should provide an equally useful platform for experimentation with different compile-time analyses and optimisations.

Bibliography

[1] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Set-sharing is redundant for pair-sharing. *Theoretical Computer Science*, 2001. To appear.

[2] Matthias Blume and Andrew W. Appel. Lambda-splitting: A higher-order approach to cross-module optimizations. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 112–124. ACM Press, June 1997.

[3] Maurice Bruynooghe. A practical framework for the abstract intepretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, 1991.

[4] Maurice Bruynooghe and Michael Codish. Freeness, sharing, linearity and correctness — all at once. In *Proceedings of the Third International Workshop on Static Analysis*, pages 153–164, Padova, Italy, 1993. Springer-Verlag.

[5] Randal Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[6] Francisco Bueno, María García de la Banda, Manuel Hermenegildo, Kim Marriott, Germán Puebla, and Peter J. Stuckey. A model for inter-module analysis and optimizing compilation. In *Tenth International Workshop on Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, July 2000.

[7] 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.

[8] Michael Codish, Dennis Dams, Gilberto Filé, and Maurice Bruynooghe. On the design of a correct freeness analysis for logic programs. *Journal of Logic Programming*, 28(3):181–206, 1996.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[10] Philip Dart. *Dependency Analysis and Query Interfaces for Deductive Databases.* PhD thesis, Department of Computer Science, University of Melbourne, Australia, 1988.

[11] Bart Demoen, María García de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. Herbrand constraint solving in HAL. In D. De Schreye, editor, *Logic Programming: Proceedings of the 16th International Conference*, pages 260–274. MIT Press, 1999.

[12] Bart Demoen, María García de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. An overview of HAL. In Joxan Jaffar, editor, *Proceedings of the Fourth International Conference on Principles and Practices of Constraint Programming*, number 1713 in LNCS, pages 174–188, Alexandria, Virginia, October 1999. Springer-Verlag.

[13] Bart Demoen, María García de la Banda, and Peter J. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In *Proceedings of the 22nd Australasian Computer Science Conference, ACSC'99*, pages 217–228, Auckland, New Zealand, January 1999. Springer-Verlag.

[14] Daniel Diaz and Philippe Codognet. A minimal extension of the WAM for `clp(FD)`. In *Proceedings of the 10th International Conference on Logic Programming (ICLP'93)*, pages 774–790, Budapest, Hungary, 1993. MIT Press.

[15] Stuart I. Feldman. make — a program for maintaining computer programs. *Software: Practice & Experience*, 9(4):255–265, April 1979.

[16] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October–December 1998.

[17] María García de la Banda, David Jeffery, Kim Marriott, Nicholas Nethercote, Peter J. Stuckey, and Christian Holzbaur. Building constraint solvers with HAL. In *Proceedings of the 17th International Conference on Logic Programming (ICLP'01)*, November 2001. To appear.

[18] María García de la Banda, Kim Marriott, Harald Søndergaard, and Peter J. Stuckey. Differential methods in logic program analysis. *Journal of Logic Programming*, 35(1):1–37, 1998.

[19] María García de la Banda, Peter J. Stuckey, Warwick Harvey, and Kim Marriott. Mode checking in HAL. In J. Lloyd et al., editors, *Proceedings of the First International Conference on Computational Logic*, number 1861 in LNAI, pages 1270–1284, London, United Kingdom, July 2000. Springer-Verlag.

[20] Nevin C. Heintze, Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) programmer's manual version 1.2. Technical Report 92-24, Department of Computer Science, University of Melbourne, Australia, September 1992.

[21] Fergus Henderson, Thomas Conway, Zoltan Somogyi, Peter Ross, and Tyson Dowd. The Mercury user's guide. `http://www.cs.mu.oz.au/mercury/information/doc/user_guide_toc.html`.

[22] 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.

[23] Manuel Hermenegildo, Germán Puebla, Kim Marriott, and Peter J. Stuckey. Incremental analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[24] Manuel Hermenegildo and Francesca Rossi. Non-strict independent and-parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[25] Christian Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of PLILP'93*, number 631 in LNCS, pages 260–268, Budapest, Hungary, 1993. Springer-Verlag.

[26] Christian Holzbaur. OFAI clp(Q,R) manual version 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, December 1995.

[27] Christian Holzbaur, Peter J. Stuckey, María García de la Banda, and David Jeffery. Optimizing compilation of constraint handling rules. In P. Codognet, editor, *Logic Programming: Proceedings of the 17th International Conference*, LNCS. Springer-Verlag, 2001. To appear.

[28] Dean Jacobs and Anno Langen. Accurate and efficient approximation of variable aliasing in logic programs. In *Proceedings of the North American Conference on Logic Programming*, pages 154–165, Cambridge, Mass., October 1989. MIT Press.

[29] Dean Jacobs and Anno Langen. Static analysis of logic programs for independent AND parallelism. *Journal of Logic Programming*, 13(2&3):291–314, 1992.

[30] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM Press.

[31] Joxan Jaffar and Spiro Michaylov. Methodology and implementation of a CLP system. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the 4th International Conference*, pages 196–218, Melbourne, Australia, May 1987. MIT Press.

[32] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[33] Gerda Janssens, Maurice Bruynooghe, and Veroniek Dumortier. A blueprint for an abstract machine for abstract interpretation of (constraint) logic programs. In J. W. Lloyd, editor, *Proceedings of the International Symposium on Logic Programming (ILPS)*, pages 336–351, Portland, Oregon, December 1995. MIT Press.

[34] David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in Mercury. Technical Report 98-13, Department of Computer Science, University of Melbourne, Australia, September 1998.

[35] Simon B. Jones and Daniel Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture '89*, pages 54–74, Imperial College, London, 1989. ACM.

[36] Andrew Kelly, Andrew Macdonald, Kim Marriott, Harald Søndergaard, and Peter J. Stuckey. Optimizing compilation for CLP($\mathcal{R}$). *ACM Transactions on Programming Languages and Systems*, 20(6):1223–1250, 1998.

[37] Vitaly Lagoon and Peter J. Stuckey. A framework for analysis of typed logic programs. In *Proceedings of the Fifth International Symposium on Functional and Logic Programming*, number 2024 in LNCS, pages 296–310. Springer-Verlag, 2001.

[38] Anno Langen. *Advanced Techniques for Approximating Variable Aliasing in Logic Programs*. PhD thesis, University of Southern California, March 1991.

[39] Lucent Technologies. Standard ML of New Jersey. `http://cm.bell-labs.com/cm/cs/what/smlnj/index.html`.

[40] Kim Marriot and Harald Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.

[41] Kim Marriott. Algebraic and logical semantics for CLP languages with dynamic scheduling. *Journal of Logic Programming*, 31(1):71–84, July 1997.

[42] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

[43] Kimbal Marriott and Peter J. Stuckey. Approximating interaction between linear arithmetic constraints. In M. Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, pages 571–585, Ithaca, New York, November 1994. MIT Press.

[44] Nancy Mazur, Gerda Janssens, and Maurice Bruynooghe. Towards memory reuse in Mercury. In *Proceedings of the International Workshop on Implementation of Declarative Languages (IDL'99)*, Paris, France, September 1999.

[45] Nancy Mazur, Gerda Janssens, and Maurice Bruynooghe. A module based analysis for memory reuse in Mercury. In J. Lloyd et al., editors, *Proceedings of the First International Conference on Computational Logic*, number 1861 in LNAI, pages 1255–1269, London, United Kingdom, July 2000. Springer-Verlag.

[46] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[47] K. Muthukumar and Manuel Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In *Proceedings of the Eighth International Conference on Logic Programming (ICLP'91)*, pages 49–63, Paris, France, 1991. MIT Press.

[48] K. Muthukumar and Manuel Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[49] Germán Puebla, María García de la Banda, Kim Marriott, and Peter J. Stuckey. Optimisation of logic programs with dynamic scheduling. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 93–107. MIT Press, 1997.

[50] Germán Puebla and Manuel Hermenegildo. Optimized algorithms for incremental analysis of logic programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[51] Germán Puebla and Manuel Hermenegildo. Abstract multiple specialization and its application to program parallelization. *Journal of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

[52] Germán Puebla and Manuel Hermenegildo. Some issues in analysis and specialization of modular Ciao-Prolog programs. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier – North Holland, March 2000.

[53] Jean-Francois Puget. A C++ implementation of CLP. In *Proceedings of SPICIS'94*, Singapore, November 1994.

[54] 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 (LOPSTR'99)*, number 1817 in LNCS, pages 196–215, Venice, Italy, September 1999. Springer-Verlag.

[55] Peter Schachte. *Precise and Efficient Static Analysis of Logic Programs*. PhD thesis, Department of Computer Science, University of Melbourne, Australia, July 1999.

[56] 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, October–December 1996.

[57] Harald Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In B. Robinet and R. Wilhelm, editors, *Proceedings of the European Symposium on Programming 86 (ESOP 86)*, number 213 in LNCS, pages 327–338. Springer-Verlag, 1986.

[58] Swedish Institute of Computer Science. SICStus Prolog home page. `http://www.sics.se/ps/sicstus.html`.

[59] Andrew Taylor. Removal of dereferencing and trailing in Prolog compilation. In G. Levi and M. Martelli, editors, *Logic Programming: Proceedings of the Sixth International Conference*, pages 48–60, Portugal, Lisbon, June 1989. MIT Press.

[60] Andrew Taylor. PARMA — bridging the performance gap between imperative and logic programming. *Journal of Logic Programming*, 29(1–3):5–16, October–December 1996.

[61] Simon Taylor. Optimization of Mercury programs. Honour's Report, Department of Computer Science, University of Melbourne, November 1998.

[62] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society, series 2*, volume 42, pages 230–265, 1936–7. Corrections, Ibid, volume 48, pages 544–546, 1937.

[63] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 60–76, Austin, Texas, January 1989. ACM Press.

[64] Mark Wallace, Stefano Novello, and Joachim Schimpf. ECLiPSe: A platform for constraint logic programming. `http://www.icparc.ic.ac.uk/eclipse/reports/eclipse/eclipse.html`, August 1997.

[65] Will Winsborough and Annika Waern. Transparent and-parallelism in the presence of shared free variables. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 749–764, Cambridge, Massachusetts, 1991. MIT Press.