Towards Parallel Mercury

Thomas Conway

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

August, 2002

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

Produced on acid-free paper

Abstract

This thesis presents the foundations for extending the implementation of the declarative logic programming language Mercury to support the parallel execution of programs.

The new material in this thesis is in three parts. The first part presents an extension to the existing (sequential) execution model for Mercury that allows programs to be executed in parallel. Programmers can exploit this extension simply by replacing some sequential conjunction operators connecting independent goals with a new parallel conjunction operator. Such changes do not change the declarative semantics of the program, but can improve performance.

The second part of the new material presents a model for explicit threading in Mercury, which is useful when the programmer's goal is not efficiency but the explicit representation of concurrent tasks and control of their interactions. We show how our extended execution model supports the usual operations on threads. We also present a new interpretation for Mercury's system for input and output that provides an intuitive understanding of the interactions between the I/O operations executed by the different threads of a program or by different programs.

The final part of the new material presented in this thesis presents a new technique for obtaining a detailed and accurate picture of the performance of a program. The basis of our technique is associating a complete context with each measurement, rather than approximating the context as conventional profilers do. In order to make our new profiling system feasible we have had to develop sophisticated techniques for reducing the cost of recording complete contexts; in order to make its output tractable, we also had to develop techniques for dealing with interactions between higher order constructs and recursion. We have also developed a tool for helping programmers (and eventually compilers) to digest the large volume of data generated by our profiling technique.

All the ideas presented in this thesis have been implemented in the Melbourne Mercury compiler.

Declaration

This is to certify that:

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

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

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

Thomas Conway

Acknowledgements

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

I would like to thank my supervisor Zoltan Somogyi and Fergus Henderson for including me in the Mercury project right from the start (when it was just “our new language”). Without their many hours of patient help (not to mention many hours of bridge) I would never have achieved the understanding of programming and computers that I have.

Thanks are also due to the rest of the Mercury group: David Jeffery and Peter Ross (who talked all through their 2^{nd} year tutorials that I taught, but I feel that by getting into the Mercury group, I've got my own back); Tyson Dowd who has been a patient listener and good friend; Andrew Bromage who inspired me to nearly become a theoretician; David Overton who is the best friend a man could hope to have and who may one day forgive me for convincing him to become a postgrad; and everyone else in the group who have endured my complaints about life, Mercury, themselves and everyone else with never a bad word.

The staff of the Department of Computer Science and Software Engineering must also be thanked for their encouragement over the years. Particularly, I must thank the following people: Harald Søndergaard, Lee Naish, Peter Stuckey, Peter Schachte, Philip Dart and Julien Reid (without whom our working lives would be a sorry mess).

Thank-you also to my employer and colleagues at *InQuirion* for the last couple of years while I have been writing: Ron Sacks-Davis, Alan Kent, Tim Arnold-Moore, Michael Fuller, Neil Sharman and the rest, who have endured my absences and been most tolerant. Kudos to Brian Lowe who told me “writing a thesis is like eating an elephant: you've got to do it one bite at a time.”

I must also thank my parents (where would we be without them), who have been generous with their time and money to help Kate and I survive the experience. They have always encouraged me as best they knew how.

A huge thank-you is due to Leon Sterling, without whom this thesis would have withered and died (and nearly I along with it)! Thanks for your patience, listening to me rant about the evils of having a thesis to complete, reading my drafts, and especially for writing encouraging comments on them.

Most of all thanks, love and honour to my wife Kate, who ignorant of her peril married an unfinished thesis, and has supported and encouraged me, and still hangs around. Where would I be without you!

S.D.G.

conway `)O+`
Melbourne, February 2002

Contents

List of Figures

Chapter 1

Introduction

In this thesis we will be concerned with the intersection of two subject areas: logic programming, and software engineering. We will be exploring how support for parallelism can be added to the logic programming language Mercury [98], which has been designed to support the engineering of large software systems. The essential design goals of Mercury were that it be a purely declarative language; that it support the construction of large software systems; and that it yield high performance. These goals hold also for adding infrastructure to support parallelism in Mercury, which is what we will be specifically considering.

This thesis assumes that the reader is conversant with Prolog, but does not assume that the reader is familiar with Mercury.

1.1 Logic programming

Logic programming is a powerful programming paradigm because it encourages the separation of the logic of *what* the program should compute from the control which determines *how* it should be computed. As Kowalski observes in [57], this is desirable because this allows the control component of the program to be changed (possibly automatically) to make the program execute faster without altering the meaning of the program. Control strategies include those of Edinburgh Prolog [22] which executes goals in a fixed depth-first, left-to-right manner; NU-Prolog [104] and SICStus Prolog [17] which allow the order in which goals are executed to be constrained to allow predicates to be evaluated only when their arguments have or exceed particular instantiations; and XSB Prolog [87] which has an execution strategy that relies on tabling.

Another important property of logic programs is that they can be interpreted as statements in a formal mathematical logic. The rules of the logic allow us to manipulate programs in a variety of useful ways without altering their semantics. Pure logic programming languages have the property of *referential transparency*, which means that all instances of a particular object in the program are interchangeable, so if the compiler can deduce that two expressions generate the same value, it can eliminate one and just use the value computed by the first expression where the value of the second is needed. This implies that the evaluation of expressions must not have any visible side effects, or the semantics of the program would not be preserved in the presence of such a transformation.

Having an underlying mathematical logic makes it easier for the compiler to prove various properties of programs. The kinds of properties of programs that are interesting proving include type correctness, well modedness, determinacy, and termination. (The first three of these we will discuss in more detail in chapter 2.) They are interesting because these are all properties of programs which are usually implied in the design, and therefore can be used to help validate that the program clauses coincide with the design. In addition to this use, programs that have the properties of being well typed and moded with respect to sufficiently constrained languages of types and modes can be executed much more efficiently than untyped and unmoded ones.

In principle, when using a logic programming language the programmer can specify the logic only, and the language implementation may choose whatever control rules it likes to do the evaluation. In Prolog however the control is implied in the order of clauses and literals within clauses, just as in an imperative programming language like C, the control is implied by the order of statements. But in logic programming languages which have the property that their declarative and operational semantics coincide (unlike Prolog), there is freedom to choose whatever control strategy the language implementation determines is most appropriate for the program. Most logic programming systems use some variation of Robinson's resolution [86] as the basic evaluation mechanism. Within this mechanism there is still quite a lot of flexibility in what order parts of the computation are done (see e.g. Lloyd [59]). There is no particular reason why the steps of the computation need be done sequentially, and there have been many systems described in the literature which utilise some form of parallelism [24, 39]. There are two main ways in which parallel evaluation can take place: multiple disjuncts can be evaluated at once (*OR*-parallelism), or multiple conjuncts can be evaluated at once (*AND*-parallelism). Within the possible AND-parallel evaluation strategies there are two main sub-classes. *Independent* AND-parallel systems only evaluate conjuncts in parallel if there can be no communication (by the binding of shared variables) between them. That is, the evaluation of each conjunct is independent of the evaluation of the others. *Dependent* AND-parallel systems evaluate conjuncts in parallel in the presence of (potential) communication between them. There is a strong connection between independent AND-parallelism and OR-parallelism because in both cases there is no communication between the parallel goals [79]. This has been used in several program transformations (see e.g. [21]) to translate one into the other.

1.2 Programming languages for the real world

Since Gordon Moore first stated in 1965 that the capacity of each new chip was roughly twice that of its predecessor, and each new chip was produced about eighteen months after its predecessor, people have regularly claimed that this exponential growth in the performance of computers must end soon, and so far they have been false. However, to gain more performance computers are increasingly relying on parallelism — the capacity of the hardware to do more than one thing simultaneously. This requires that software be able to utilise this capacity in the hardware, and experience shows that engineering parallel software is generally much more difficult than engineering sequential software. For example where parallel tasks need to share a resource, they must synchronize their behaviour so that they don't corrupt the shared resource. To engineer large parallel software systems, the programmers need support from the programming language to reduce the probability of errors, both in sequential and parallel code.

There are two ways to construct parallel software systems: first construct a sequential software system, and then parallelize it (a process that might possibly be automated); or to use the concept of multiple processes as part of the model for the design of the software system, and construct a *concurrent* program that might be executed either sequentially or in parallel. In this thesis we will consider two alternatives. The first is the addition of a parallel conjunction operator, which has the same declarative semantics as the usual conjunction operator but exploits parallelism, with the compiler taking care of all the synchronization implied by all the data dependencies involved in the parallel conjunction. Programmers can use the new operator explicitly, or they can leave it up to the compiler to find profitable opportunities for its use (which although not implemented yet, is a logical extension of our work). With this alternative, the only effect of parallelism is performance. The second alternative is the development of a framework that makes parallelism explicit and allows (in fact, requires) programmers to manage the issues it raises, such as synchronization and communication between threads.

As stated above, we are interested in constructing a parallel logic programming system subject to the constraints that it be purely declarative, support the construction of large programs and yield high performance.

The design of a purely logical programming language presents some important problems not fully dealt with by logic programming systems before Mercury. In particular, although there have been attempts to model I/O in a purely declarative way, none has been widely adopted in Prolog. Perhaps the most convincing attempt was by Fergus Henderson for NU-Prolog [45], which used an explicit argument pair denoting unique initial and final states of the world. Non-logical predicates were used in the implementation to make sure that the state of the world remained unique and did not get replicated or backtracked-over. Mercury addresses this problem and uses a variation of this solution where uniqueness constraints are enforced at compile time rather than at runtime. In chapter 4, we will develop a new interpretation of this solution that allows us to support explicit concurrency in the presence of I/O.

The construction of large programs presents difficulties that are not usually present in the construction of small programs. Large programs are usually written by more than one person, they take longer to write, and they are more expensive to maintain. We expect the programming language to provide abstraction mechanisms to assist the programmers in structuring the program. The abstraction mechanisms that we consider important include a module system, parametric polymorphism, type classes and existential types.

We also consider static checking to be vital to the construction of large programs because once a program passes a set of static checks, the programmer can be certain that the program satisfies the properties that have been checked. If the static checks are not performed, then the programmer has to rely on other, much less certain methods for establishing the correctness of his programs. These methods include manual debugging and code reviews, both of which are slow, tedious, and expensive. For static checks to be effective, the clauses of the program usually need to be supplemented with declarations describing aspects of the programmers' intended meaning for predicates. As well as aiding static checking of programs, these declarations also provide important documentation for other programmers; and since this documentation is checked by the compiler, it cannot not become incorrect as the program evolves.

Support for the construction of large programs requires a compilation environment that allows the program to be processed piecemeal, rather than requiring that the whole program be processed at once. One reason for this is that large programs take a long time to compile, and if the programmer makes a small change, it is highly desirable to minimize the number of files recompiled. This is another reason why declarations are helpful, because they enable the compiler to avoid doing global analysis of the program, and instead allow it to use analysis on small parts of the program at a time. This is particularly important where the analysis algorithms are super-linear in the size of the program fragment they are working over.

The construction of high performance systems requires the source language to have no frequently used constructs that cannot be implemented efficiently. Moreover, it is important that no language construct require support that has too much impact on the performance of parts of the program that do not use it. In logic programming languages the most obvious example of a language feature that falls in the second proscribed category is variable-to-variable aliasing. Variable-to-variable aliasing requires trailing, and trailing in turn impacts memory management for the entire program, not just the regions that make use of aliasing.

For general purpose high level languages to yield high performance, it is crucial that they are amenable to program analysis and transformation. To allow this, we believe it is highly desirable for the language to be purely declarative — it should not contain any constructs that violate the rules of the underlying logic. In addition, it is important that the program contain sufficient information for precise analysis. To this end, declarations that supplement the program clauses are very important because they restrict (in a well defined way) the possible set of intended interpretations of a predicate (see [69, 72]), and therefore let program analyses avoid having to make unnecessary generalizations.

The Mercury group at the University of Melbourne have developed a logic programming language (Mercury [98]) which aims to satisfy all these requirements. We use the basic Mercury language as the foundation on which we will build some of the infrastructure necessary for constructing a high performance parallel system. The language, and how it is used, have important implications for how we support parallelism, and how we aim to achieve high performance. Thanks to the information in declarations and to the absence of hard-to-analyze features from the language, Mercury programs can be executed very efficiently compared to Prolog programs in most Prolog implementations. (See [96] for some data on the relative performance of some popular Prolog implementations compared to Mercury.) This means that the cost of low-level operations (like creating a thread, or allocating a block of memory for a new stack) are relatively much more expensive. This means that to get good performance from parallelism, we need to ensure that we scrupulously avoid overheads where they are not absolutely necessary, and that we take care when creating parallel programs that we pay attention to the grain-size of the parallel tasks.

Mercury programs have much more constrained dataflow than Prolog programs. This is both a benefit and a limitation. It makes programs much more tractable to analysis. A lot of work has been done to try to extract precise dataflow information from Prolog programs. This kind of information can be extracted much more easily from Mercury programs because it the declarations that accompany the clauses of the program give some information directly, and other information can be derived from the declarations using local analysis. The constrained dataflow in Mercury also reduces the expressiveness of the language. There are common Prolog paradigms (such as difference lists) that cannot be expressed directly in Mercury. Our experience however, is that in many cases where these paradigms are used for efficiency in Prolog they are simply unnecessary in Mercury because Mercury programs can be executed much more efficiently. This is certainly not always the case — there are situations where the aliasing of logic variables allows a faster algorithm than can be easily implemented without using an explicit level of indirection. Even though we note that frequently a Mercury program with the explicit level of indirection is still faster than the Prolog version of the program, the use of logic variables in Prolog is often quite elegant, which cannot always be said of the equivalent Mercury code.

One of the important problems for parallel logic programming implementations is to determine which parts of a given program should be executed in parallel. Several options exist: implementations can leave it to the programmer to figure out which parts of the program should be executed in parallel, and provide either explicit constructs for parallelism or annotations that can be used by the compiler; static analysis can be used to estimate how much work is done by different parts of the program. This analysis can be used to automatically determine which parts of the program should be executed in parallel; profiling techniques can be used to build an empirical model of how much work is done by different parts of the program, and this can be used to determine which parts of the program should be executed in parallel; and a combination of these techniques can be used. In practice, a combination is most likely to be effective since there will be occasions when analysis fails to correctly identify the best candidates for parallelization, when a programmer's intuition for which portions of the program will be expensive will be wrong, and when the profiling data may not be representative of real program loads.

We believe that empirical models of the behaviour of programs are tremendously important for building a practical parallel system. This is true whether the model is used automatically by the compiler to optimize or parallelize the program, or whether the model is used manually by the programmer to do this. Most analysis for program cost (and the related analysis for program termination) is necessarily approximate (see for example [4, 12, 13, 26, 28, 55, 92]), and typically requires mathematical analysis beyond the capabilities of most automated systems. In large systems, it is often impractical to expect programmers to have a good feel for where the the expensive parts of the program will be, and so the automated collection of performance information about programs is extremely beneficial. Profilers for traditional imperative programming languages are not adequate for logic programming languages for three main reasons: they do not scale to deep and complex call graphs, they do not handle recursion with enough precision, and they do not handle higher order constructs well. Therefore, to be able to build accurate models of where programs expend resources we need better profiling tools.

In this thesis we show how infrastructure for supporting parallelism can be added to Mercury in two ways. In chapter 3 we present some infrastructure for allowing multiple parts of Mercury programs to execute in parallel. We will present an operator for independent (AND-)parallel conjunction that uses this infrastructure, and we present a benchmark program which uses this form conjunction. In chapter 4 we show how threads can be supported in a way that lets them interact with each other and with the external world. The implementation of threads uses the infrastructure developed in chapter 3. Chapter 5 will present a new way of profiling Mercury programs to obtain detailed and accurate models of their performance.

We will present an overview of the Mercury language in chapter 2.

In a variety of places we make generalizations about all Mercury programs based on empirical data from the source code of the Mercury compiler itself (which is written in Mercury). We believe these generalizations to be valid for the following reasons: the Mercury compiler is a fairly large program (roughly 240,000 lines of source at the time of writing); it has had many contributing authors with a variety of expertise and experience; and it has been written and maintained over a number of years (work on the compiler began in late 1993 and has been continuous since then). The only caveat on such generalizations, is that the compiler does not represent the only kind of Mercury program that one might write. For example, we have written an XML [15] parser using higher order parser combinators, which makes much more intensive use of Mercury's higher order facilities than the compiler; and we have a graphical image renderer which we wrote for the ICFP 2000 programming contest [66] which makes intensive use of floating-point and vector arithmetic. However, our observation of these other programs is strongly suggestive of the fact that the compiler is indeed sufficiently representative of all Mercury programs that the generalizations that we make are valid.

The Mercury system (including the functionality described in this thesis) is freely available from the Mercury web site at `http://www.mercury.cs.mu.oz.au/`.

Chapter 2

Mercury

Mercury is a purely declarative logic programming language designed at the University of Melbourne. Work on it began in October 1993, and the compiler first bootstrapped in February 1995. It was designed with clear goals that set it apart from other logic programming languages such as Prolog and Gödel [49]. As stated earlier in chapter 1, Mercury was designed with the criteria that it be purely declarative, it support the construction of large programs, and that it yield high performance. We believe we have succeeded fairly well in achieving these aims. Mercury is certainly purely declarative; we have successfully maintained the compiler (which with the Mercury standard library is about 300,000 lines of Mercury code) over more than five years, with an evolving team of between three and fifteen programmers; and as reported in [96], it achieves performance significantly better than any other logic programming system.

Mercury programs consist of Horn clauses giving the definitions of predicates and functions, and declarations. There are four classes of declarations that we will discuss: type, mode, determinism and module declarations. We will first describe the basic syntax of program clauses then describe the declarations in some detail in the following sections.

The Mercury environment consists of a compiler, runtime system, standard user library, and a collection of tools to aid productivity. These include a front end to GNU `make` which takes into account the various auxiliary files used by the Mercury compiler; a profiling tool similar to `gprof`; and a debugging environment. The entire Mercury implementation is structured in a way that allows a variety of optional features to be included or excluded. For example, there are different kinds of garbage collection available (none, a C based conservative collector due to Boehm, et al [11], and an incomplete native collector). As well as optional features, this architecture allows us to support a wide variety of host platforms. Most of the advanced operating system features and C language extensions that we use are factored in such a way as to make it possible to switch their use on or off. This is important because it allows Mercury programs to be executed in an environment that does not require overheads for unused features. A configuration with a specific set of features enabled (or disabled) is referred to as a *grade*.

2.1 Syntax

Mercury syntax is derived from Prolog syntax. It uses the same term structure and almost the same set of operators. The terms that make up the program fall into two classes: declarations and clauses.

The following are examples of declarations:

```
:- module mymodule.
:- interface.
:- implementation.
:- type list(T) ---> [] ; [T|list(T)].
:- pred append(list(T), list(T), list(T)).
```

The declarations in Mercury programs convey various kinds of meta-information to the compiler such as which symbols defined in the current module should be made visible to other modules, the structure of types, the argument types of predicates and so on.

Clauses are either *facts* or *rules*. The following are examples of facts:

```
edge(a, b).
edge(b, c).
append([], Bs, Bs).
is_digit('0').
is_digit('7').
```

The following are examples of clauses:

```
path(X, Z) :-
    edge(X, Y),
    path(Y, Z).

append([A|As], Bs, [A|Cs]) :-
    append(As, Bs, Cs).

term_to_thingy(Term, Thingy) :-
    (
        Term = functor(Atom, Args, _),
        ( Atom = atom("thingy"), Args = [] ->
            Thingy = thingy
        ; Atom = atom("other"), Args = [Other] ->
            Thingy = other(Thingy0),
            term_to_thingy(Other, Thingy0)
        ;
            error("bad term")
        )
    ;
        Term = variable(_),
        error("non-bound term")
    ).
```

In general, a goal (the part to the right of the `:-`) is one of the following:

goal, *goal* which denotes the conjunction of two goals.

goal; *goal* which denotes the disjunction of two goals.

`some` *vars* *goal* which existentially quantifies instances of any variables in *vars* to *goal*.

`all` *vars* *goal* which universally quantifies instances of any variables in *vars* to *goal*.

`not` *goal* which denotes the negation of *goal*.

`if` *cond* `then` *then* `else` *else* (where *cond*, *then*, and *else* are all goals) which denotes if-then-else. It is logically equivalent to the goal

```
(
    cond,
    then
;
    not some Vars cond,
    else
)
```

That is, the disjunction of the conjunction of *cond* and *then* or the conjunction of the negation of the existential closure of *cond* and *else*.

cond `->` *then* `;` *else* which is a synonym for `if` *cond* `then` *then* `else` *else*.

lhs-term `=` *rhs-term* which denotes the unification of *lhs-term* and *rhs-term*.

p`(`*arg1*`, ...,` *argN*`)` which denotes a predicate call to p/N.

Variables in clauses that are not explicitly quantified are implicitly quantified. Head variables are universally quantified, as usual. All other variables are implicitly existentially quantified to their nearest enclosing scope, rather than being existentially quantified to the scope of the clause as in Prolog.

Note also that unlike if-then-else in most Prolog implementations (with the notable exception of NU-Prolog), if-then-else in Mercury does not prune across its condition. Operationally, it prunes away whichever of the *then* arm and the *else* arm is inapplicable, but if the condition has multiple solutions, it will generate them and re-execute the *then* arm multiple times.

When the Mercury compiler processes a program, it first transforms its clauses into a normal form (also referred to in the Mercury documentation as *superhomogeneous* form). In this normal form, all clauses are merged into a single clause with an explicit disjunction, extra unifications are introduced so that all unifications are of the form

Var0 `= f(`*Var1*`, ...,` *VarN*`)`

or

Var0 `=` *Var1*

where each Var_k occurs only once in the unification, and calls are transformed in the same manner (i.e. to make each argument a distinct variable).

Example 2.1.1 *Consider the clauses for* `append/3`

```
append([], Bs, Bs).
append([H|As], Bs, [H|Cs]) :-
    append(As, Bs, Cs).
```

Once transformed into the normal form, this becomes

```
append(V1, V2, V3) :-
    (
        V1 = [],
        V2 = Bs,
        V3 = Bs
    ;
        V1 = [H|As],
        V2 = Bs,
        V3 = [H|Cs],
        append(As, Bs, Cs)
    ).
```

2.2 Types

Mercury uses a many-sorted logic rather than a single-sorted one. The type system that distinguishes the sorts is derived from the Hindley-Milner [50, 65] type system of ML and from Mycroft and O'Keefe's type system for Prolog [67]. It supports parametric polymorphism and some overloading. It also supports typeclasses and existentially quantified types, though we will discuss neither of these features here, as they are not relevant to this thesis. Overall, Mercury's type system is quite similar to the type system of Haskell [51].

There are two parts to the type system: the declaration of types, and the declaration of predicates using the declared types. Each type is declared either as a discriminated union — a set of function symbols and the types of their arguments, or as structural equivalences to another type. We declare types using a language of regular trees. The syntax for type declarations follows.

typedecl ::= `:- type` *typename* `--->` *functors*
        | *typename* `==` *type*
typename ::= atom | functor(*variables*)
functors ::= *functor* | *functor* `;` *functors*
functor ::= atom | functor(*types*)
types ::= *type* | *type* `,` *types*
type ::= atom | functor(*types*)
variables ::= variable | variable `,` *variables*

(The terms "atom", "functor" and "variable" denote terminal symbols representing atoms, function symbols and variables respectively.)

A type declaration describes the (possibly infinite) set of terms that belong to that type. The variables in the *typename* of the type are universally quantified. That is, a type is defined for all values of its type parameters.

Example 2.2.1 *The following are examples of Mercury types.*

```
:- type boolean
    --->    true
    ;       false.

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

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

:- type pair(T1, T2)
    --->    (T1 - T2).

:- type association(Key, Value)
            == list(pair(Key, Value)).

:- type naryTree(T)
    --->    leaf(T)
    ;       node(T, list(naryTree(T))).
```

A type declaration specifies a regular tree denoting the set of terms that belong to the type. When represented as a graph as in figure 2.1, multiple branches denote alternative bindings (e.g. in this example `[]` for the empty list and `[ | ]` for cons), and the upward arcs correspond to recursion in the definition. When we discuss instantiation states and modes, it will be convenient to think in terms of these trees.

As well as declaring the type used in the program, the programmer also declares the argument types for predicates and functions in the program. (These declarations may be inferred for predicates and functions that are local to a module.)

Predicate and function declarations use the following abstract syntax.

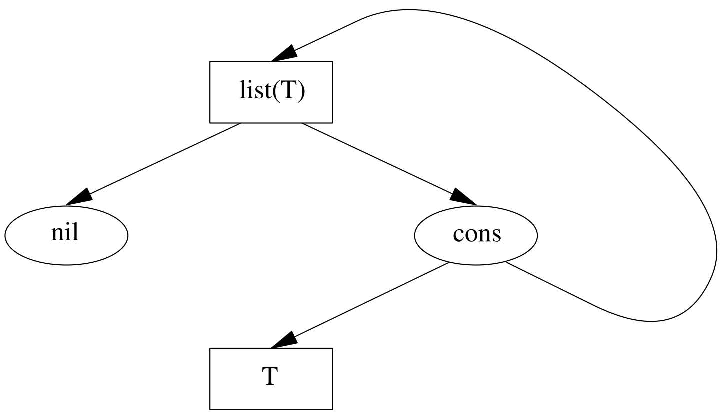

Figure 2.1: A type graph.

preddecl ::= `:- pred` *functor*
funcdecl ::= `:- func` *functor* `=` *type*

These declarations specify that the arguments of the predicate (or function) named in the *functor* rule must have the given types. Functions also have a result type associated with them.

Example 2.2.2 *The following are examples of predicate and function declarations.*

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

:- pred member(T, list(T)).

:- func max(int, int) = int.

:- pred find(association(K, V), K, list(V)).
```

We consider a program to be well typed if each variable of the program in its normal form has a single most general type, and the types of the variables satisfy the constraints imposed by the type, predicate and function declarations (Demoen, et al present a direct use of this strategy in [31]). Currently the Mercury compiler does not implement type checking directly in terms of constraints in this manner, but we have plans to reimplement type checking to do so.

We use the type, predicate and function declarations as the basis for type checking which rejects programs that are not well typed. The type checker is written to give error messages for programs that are not well typed with the goal of helping programmers more easily correct their programs.

2.3 Modes

Mercury programs also contain mode declarations which describe the allowable data flow within predicates and functions. It is possible to declare multiple modes for a predicate or function. The mode system is based on regular trees that describe the instantiation state of a term. Although these regular trees can describe a rich language of partially instantiated terms, we will limit our discussion in this thesis to just two instantiation states: `free` which describes a term which is completely uninstantiated and may not be aliased with any other variable or node, and `ground` which describes a term which is completely instantiated and contains no unbound nodes.

A mode of a predicate associates a pair of instantiation states with each argument of that predicate. The first specifies the instantiation that the argument must have before a call to that predicate can be scheduled (very much like `block` declarations in SICStus Prolog [17] or `when` declarations in NU-Prolog [104]). The second specifies the instantiation that the argument must have on success. This can be viewed as a contract between the predicate and its caller: if the caller provides arguments which satisfy the initial instantiation constraints in the mode declaration, then the predicate guarantees to provide bindings of the arguments that satisfy the final instantiation.

Mercury provides syntax that allows the programmer to provide a name for a pair of instantiation states (i.e. a *mode*). We use this syntax to give names to combinations of the two instantiation states that we will use (`free` and `ground`) to declare the following argument modes.

```
:- mode in = (ground -> ground).
:- mode out = (free -> ground).
:- mode unused = (free -> free).
```

We can then use these argument modes to declare the allowed modes for a predicate:

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

(We will discuss implied modes such as `append(in, in, in)` in section 2.11.) We say that a program is well moded if for each mode of a predicate there is an ordering of each conjunction in that predicate so that a left-to-right execution of the conjunction satisfies the mode constraints.

Example 2.3.1 *Consider the normal form of* `append/3` *presented above. In the* `append(in, in, out)` *mode, the conjunction in the second disjunct must be reordered so that* `V4` *is bound before we use it in the unification with* `V3`*:*

```
append(V1, V2, V3) :-
    (
        V1 = [],
        V3 = V2
    ;
        V1 = [A|As],
        append(As, V2, V4),
        V3 = [A|V4]
    ).
```

By contrast, in the mode `append(out, out, in)`*, the conjunction must have the reverse ordering so that V4 is ground before we make the recursive call to append:*

```
append(V1, V2, V3) :-
    (
        V1 = [],
        V3 = V2
    ;
        V3 = [A|V4],
        append(As, V2, V4),
        V1 = [A|As]
    ).
```

2.4 Determinism

Just as Mercury encourages (and in some cases requires) programmers to declare the allowable dataflow in the predicates that they write, Mercury encourages (and in some cases requires) programmers to declare information about the determinacy of predicates in given modes.

The *determinism* annotation of a mode describes two main aspects of the behaviour of a mode of a predicate (which we denote with the term *procedure*). The first component describes whether

Can fail	Max # solutions	Determinism
yes	0	`failure`
no	0	`erroneous`
yes	1	`semidet`
no	1	`det`
yes	≥ 1	`nondet`
no	≥ 1	`multi`

Table 2.1: The components of the Mercury determinism annotations.

or not it is possible for the procedure to fail without producing a solution. The second component describes whether the procedure is expected to be able to produce exactly zero solutions, exactly one solution, or one or more solutions. The combinations of these give rise to the following possible annotations: `det`, `semidet`, `multi`, `nondet`, `erroneous`, and `failure`. Table 2.1 summarizes the meanings of these annotations.

These annotations are attached to the mode declaration for a predicate by writing `is` *determinism* at the end. For example we annotate the predicate modes we showed earlier in the following way:

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

The first mode of `append/3` is declared to be `det` because it is certain not to fail, and is certain not to have more than one solution because the two clauses are mutually exclusive (i.e. due to the deconstruction unification of the first argument with either `[]` or `[ | ]`). The second mode of `append/3` is declared to be `multi` because it can always produce a solution without failing, and it may produce multiple solutions.

There is an additional component of determinism that describes whether a procedure should use *committed choice* nondeterminism. In many cases, a piece of code may have multiple solutions, but we are only interested in finding a single solution. These cases fall into two categories. The first occurs when a subgoal binds some variables but all of those variables are existentially quantified. In that case, all the surrounding code is interested in is whether the subgoal has any solutions, so computing any solutions beyond the first would be redundant. We call such a subgoal a *committed choice context*. When the subgoal is a call to a predicate that may have multiple solutions, the programmer may wish to declare that the predicate has mode(s) that compute at most one solution. In such modes, the language implementation can usually avoid creating any "choice points", which can lead to huge improvements in speed and memory consumption. If the procedure has at least one solution, we say that the committed choice mode has determinism `cc_multi`; if the procedure may have zero or more solutions, we say that the committed choice mode has determinism `cc_nondet`. Mercury's rules allow procedures with these two determinisms (and in general, code with these determinisms) to be used only in committed choice contexts. (For the details on these rules, see the Mercury reference manual [43].) As we said above, there are only two such contexts. The first is an existential quantification that discards all the bindings made by the quantified goal. The second is the top level predicate of the program, `main/2`. By declaring `main/2` to be `cc_multi`, the programmer is making an assertion: that all the solutions produced by the program are equivalent with respect to the equality theory of the program (even if, as is usual, this is not written down) and the programmer doesn't care which one the program actually computes.

In fact, the benefits of committing to a solution even though others might exist is sufficiently important that several logic programming languages (particularly concurrent logic programming languages including Concurrent Prolog [90], GHC [106] and Parlog [19]) provide only this sort of nondeterminism[1]. Its use is not without drawbacks however, since if not used in a restricted class

[1]Some of the concurrent logic programming languages did provide deep nondeterminism within the *guard* of the

of contexts, it can lead to unsoundness. Mercury's rules (which are similar to the constraints on when negation can be used) for where committed choice nondeterminism may be used ensure that its use never exhibits this problem.

The different determinisms form a lattice in three dimensions: maximum number of solutions, minimum number of solutions, and committed choice context. The first dimension has three values (at most zero solutions, at most one solution, at most many solution); the second has two values (at least one solution, at least zero solutions); and the third has two values (all-solutions context, first-solution-only context). The value of the third dimension makes a difference only when a predicate may have more than one solution; if a predicate has at most one solution, then searching for the all solutions is no different from searching for the first solution.

The vast majority of the predicates in Mercury programs have determinism det, semidet, multi or nondet. The only things that a predicate with a determinism of erroneous can do is loop forever and abort; it cannot succeed without violating the upper bound on the number of its solutions and it cannot fail without violating the lower bound on the number of its solutions. Predicates with a determinism of failure can also fail, but given that all the standard abort predicates have determinism erroneous, the only predicate with determinism failure in almost all programs is the builtin goal `fail`.

There are two kinds of first-solution-only contexts. The first occurs when a goal that may have more than one solution occurs inside an existential quantifier that discards all the variable bindings generated by that goal. In that case, once we know that the goal has at least one solution, computing its remaining solutions serves no purpose. The second kind of first-solution-only context occurs when the top level `main/2` predicate is declared `cc_multi`. We will discuss this kind of context in section 4.1. It is explored in detail by Naish in [70].

It should be noted that the combination of types, modes leads to much higher precision in the determinism information that the compiler can discover. In particular, it leads to much better indexing than is usually found in Prolog implementations. In Prolog, indexing, which is clause based, is nearly always limited to the top-level functor of the first argument of the clause. If deeper indexing or indexing on multiple arguments is required then programmers have to explicitly introduce it by creating separate predicates [76, 112]. The main reason that Prolog implementations do not do more aggressive clause indexing is the absence of mode information. If each node in each argument involved in indexing must first be tested for freeness, the cost-effectiveness of indexing is substantially reduced. Because of the presence of mode information, Mercury implementations can (and do) perform much deeper indexing not just on clause heads but also on any disjunctions that are distinguished by unifications with distinct function symbols. This allows the Mercury implementation to avoid choice point creation which amongst other things avoids memory leaks that result from references to structures in choice points that will never be revisited. The type information present in Mercury programs benefits indexing further by allowing the implementation to determine if all the functors of the type of a term being indexed on are covered by one or more clause or disjunct. Where all the functors are present, then the indexing part of the clause selecting operation is certain to succeed. This leads to substantial performance improvements in many common predicates such as `append/3`.

2.5 Unique modes

As well as describing the dataflow of programs, modes are used to describe the uniqueness properties of terms. A term is *unique* if there is only one live reference to every node in its type tree. Free terms (with an instantiation of `free`) must be unaliased, and so are always constrained to be unique. Ground terms that are constrained to be unique are described by the instantiation `unique`. Although we will not be considering it in any detail in this thesis, it is also possible to describe an instantiation state in which only some nodes in the type tree are unique. For example

clause, but it was discovered (the hard way) that such "deep guards" could not be implemented efficiently. These languages, once restricted to allow only builtin operations such as unification, arithmetic, and comparison, were said to be *flat*.

it is possible to describe a list in which the list cells are all unique, but the elements of the list may be non-unique. The main constraint on type trees that describe partially unique instantiation states is that every ancestor of a unique node must also be unique, but a unique node may have non-unique (shared) descendants.

Terms can also be constrained to have nodes in their type tree with *no* live references. Where all nodes in the type tree are so constrained we use the instantiation `clobbered`. As with unique nodes, it is possible to have a term containing a skeleton of clobbered nodes, though we will not be considering them elsewhere in this thesis.

Using these instantiations, we can declare the following argument modes:

```
:- mode ui = (unique -> unique).    % unique in
:- mode uo = (free -> unique).      % unique out
:- mode di = (unique -> clobbered). % destructive in
```

An intuitive understanding of these is that an argument constrained by the mode `ui` must be unique on call, and unique on success, and therefore can be passed, but may not have new references to it created; an argument constrained by the mode `uo` is an output variable where the argument is the only live reference; and an argument constrained by the mode `di` is unique on call, and cannot be used again after the call succeeds, since any code which may cause backtracking across or into the call is forbidden and leads to a compile-time error.

Example 2.5.1 *We can declare the following unique mode for append:*

```
:- mode append(di, di, uo).
```

We can interpret this intuitively as saying that we can take the only reference to two lists, and "destructively" append them to produce a unique reference to a new list.

There are two main motivations for uniqueness. One which we will discuss in the next section is to allow declarative input and output, and the other is to allow declarative destructive update. If the instantiation state of a node in the type tree for a term goes from unique to clobbered and in a nearby location needs a new cell of the same size, then the compiler can arrange to reuse the old cell rather than discarding it and allocating a new one. At least in principle, in the case of the unique modes of `append/3` this allows the compiler to produce code that does no new allocation, and simply clobbers the tail of the first list with the second. As well as improving the performance of the unique modes of predicates like `append/3`, unique modes also allow a declarative and efficient implementation of $O(1)$ access and update data structures like arrays. By making the skeleton of the array unique, we can update elements of the array without having to make a copy of it. For more details about this use of uniqueness, see for example [45, 63, 64, 96].

2.6 Input and output

Mercury's unique modes and strong determinism give us a system in which we can model input and output (I/O) in a purely declarative way. We model I/O using an explicit state of the external world using an abstract type called the `io__state`. Predicates that perform I/O take a pair of arguments with this type: one representing the state of the world before the I/O took place and one representing the state of the world after. For this to make sense we need to ensure two things. First we need to ensure that there is only one live state of the world at a time. (In spite of parallel-universe theories, duplicating the external world is not a possibility for Mercury programs!) Second, we need to ensure that backtracking never causes us to (try to) undo an operation on the external world.

The unique modes `di` and `uo` do exactly what we want in this situation. The predicate `write_string/3` which writes a string out to the standard output stream (e.g. the terminal) has the following declaration:

```
:- pred write_string(string, io__state, io__state).
:- mode write_string(in, di, uo) is det.
```

That is, it takes a string and the current state of the world, which it destroys, and returns a unique new state of the world in which the string has been written to the standard output stream. Predicates that perform I/O use the mode `di` for their initial state of the world. As a consequence, programs that could possibly backtrack across such a predicate are illegal.

2.7 Modules

Mercury programs are divided into *modules* that consist of *interface* and *implementation* sections (introduced by `interface` and `implementation` declarations respectively). Types, predicates, functions, argument modes, and predicate modes that are declared in an interface section are made visible to other modules, whereas everything in declared in an implementation section is visible only within the module. Program clauses may be put only in implementation sections.

2.8 Higher order

Mercury supports higher order programming. Procedure objects may be constructed with the following syntax:

```
Var = pred(Arg1::mode1, ..., ArgN::modeN)
                is determinism :- goal
```

Which corresponds to the binding of *Var* to a closure with arguments (*Arg1*, ..., *ArgN*), with modes (*mode1*, ..., *modeN*), and which has determinism *determinism*, and has the goal *goal*. Any variables in *goal* that occur in the same scope as the higher order term are added as implicit (input) arguments of the closure.

Closures are invoked with the `call/N` predicate.

2.9 Sequential execution

The following account of the execution algorithm for Mercury programs is rather brief and is intended to convey the essentials rather than be encyclopedic in its detail. Readers who wish to know more than is presented here should read [96] and other papers describing the Mercury implementation (e.g. [23, 95, 97]).

Mercury programs are executed on an abstract machine. The type, mode and determinism information available at compile time means that the compiler can generate much more specific instructions for the virtual machine than is the case for typical Prolog compilers (which usually target the Warren Abstract Machine [112] or one of its variants). The abstract machine is comprised of a set of argument registers, an activation stack for procedures that cannot have more than one solution (the *detstack*), an activation stack for procedures that may have more than one solution (the *nondetstack*), a heap for allocating data terms, and a number of special purpose registers.

The Mercury compiler produces separate code for each declared (or inferred) mode of a predicate (i.e. each procedure). By doing this, it can produce much more specialized code for each procedure that can be executed with much less overhead than if no assumptions about the modes were made (as is the case in most Prolog compilers). Procedures that cannot have more than one solution (those declared `det`, `semidet`, `cc_multi`, `cc_nondet`, `erroneous`, or `failure`) which we will refer to as deterministic, are executed using the *detstack*. Procedures that use the *detstack* use a push-on-entry, pop-on-exit (or fail) stack discipline. We don't maintain a frame pointer, but rather have a single stack pointer (*sp*). The prologue of a procedure increments the stack pointer (`succip`) to reserve space for temporaries used by the procedure, and saves the success continuation onto the stack. The epilogue of a procedure restores the success continuation from the stack and decrements the stack pointer to release the space.

Deterministic code is comprised of the following types of goals: conjunctions, unifications, calls, switches (which are disjunctions comprised of alternatives that are detected to be mutually exclusive because they are distinguished by unifying a given variable with different functors), negations, and in some cases failure. Conjunctions are executed by simply executing the conjuncts in turn. Because each unification has been transformed into a normal form, and because the exact mode of each unification is known, most unifications lead to code that does a functor test (if the combination of type and mode information doesn't yet uniquely determine what the functor *must* be) or constructs a new cell on the heap and instructions to extract or fill in fields. Calls are compiled into instructions to save variables onto the *detstack*, place the appropriate variables in the argument registers, place the return address in the success continuation register, and branch to the called procedure. Switches are handled by determining which functor the switch variable is bound to, and doing a computed goto to the code for the appropriate disjunct (or failing for functors of the type that are not covered by the switch). In deterministic code, failure is handled by keeping track at compile time of where execution should branch to on failure, and generating simple branches at the point of failure. Procedures that can fail use the first argument register to return a boolean flag that indicates whether the procedure succeeded or failed.

Procedures that use the *nondetstack* use a push-on-entry, pop-on-failure stack discipline. We maintain explicit links between stack frames on the *nondetstack*. The *succip* field of the stack frame contains the success continuation address for the procedure using the stack frame. (This corresponds to the saved copy of the success continuation used by deterministic procedures.) The *succfr* field of the stack frame points to the frame of the procedure that will be executed when the procedure using this frame succeeds. The *redoip* field of the stack frame contains the address to which execution should branch when a redo occurs in a context in which this frame is topmost on the *nondetstack*. The *prevfr* field of the stack frame contains a pointer to the stack frame immediately below this one on the *nondetstack*. There are two stack pointers that point into the *nondetstack*: *curf* points to the stack frame for the currently active stack frame and *maxfr* points to the top frame of the *nondetstack*. (Mercury uses a depth-first, left-to-right search strategy, and uses the *nondetstack* as a linearisation of the parts of the search tree that have yet to be completely explored.)

Nondeterministic procedures have the same types of constituent goals as deterministic ones, but may also contain nondeterministic disjunctions. When execution enters a nondeterministic procedure, a new frame is created on the top of the *nondetstack*. When execution enters a disjunct the address of the start of the next disjunct is written into the *redoip* slot of the stack frame. When failure occurs in a nondeterministic context, execution branches to the address pointed to by the *redoip* of the topmost frame on the *nondetstack* (i.e. the value of *maxfr* is assigned to *curfr*). When a second or later disjunction, occurs conjoined with the first, the compiler generates code to produce a new frame on the *nondetstack*, which is manipulated in much the same fashion as the original. (This description is rather simplified — as always, there are many devils in the details, and we refer the reader to [96] and [23] for said details.)

2.10 The all-solutions predicate `solutions/2`

Mercury's determinism system prevents deterministic code from calling nondeterministic code directly. This is a direct result of the way the number-of-solutions component of determinism behaves over conjunction. If we denote the three values as *0* for zero solutions, *1* for one solution and *many* for one-or-more solutions, then we can define the table 2.2 to show how this property behaves over conjunction. If a goal may have many solutions, then a conjunction containing that goal may have many solutions (providing there are no conjuncts which have no solutions). If we wish to include nondeterministic code in our programs we need a mechanism for encapsulating nondeterminism.

Two such mechanisms exist. The first is a result of the fact that a multi-solutioned goal that

det[A]	*det[B]*	*det[(A, B)]*
0	*0*	*0*
0	*1*	*0*
0	*many*	*0*
1	*0*	*0*
1	*1*	*1*
1	*many*	*many*
many	*0*	*0*
many	*1*	*many*
many	*many*	*many*

Table 2.2: The maximum number of solutions component of determinism, and how it behaves over conjunction.

does not bind any non-local variables and terminates is equivalent to `true` or `fail` (if the goal may not fail, then it is equivalent to just `true`). This is of limited use but does crop up in code that uses nondeterministic search to determine properties of relations. For example, consider a predicate `is_ancestor/1` in a family-tree program that is true if the person denoted by its argument has any children:

```
:- pred is_ancestor(person).
:- mode is_ancestor(in) is semidet.
is_ancestor(P) :-
    ancestor(P, _).

:- pred ancestor(person, person).
:- mode ancestor(in, out) is nondet.
....
```

In the declared mode of `is_ancestor`, there are no output bindings, so irrespective of any search that may occur within it, `is_ancestor` can only ever be observed to succeed or fail. The language implementation can therefore prune away any unexplored parts of the search space (i.e. choice points) after the first success of the call to `ancestor`.

The second mechanism is more general and lets programs access the multiple bindings generated by the evaluation of a nondeterministic predicate. It uses a library predicate `solutions/2`, which is not directly implementable in pure Mercury, but which none the less has a pure declarative interpretation. Its signature is:

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

The goal `solutions(Closure, Solns)` is true if and only if `Solns` is a sorted list, free of duplicates, of all the solutions to `Closure`. Constraining the list to be sorted and be free of duplicates simplifies the semantics since (assuming the call to `solutions` terminates) the computed relation is independent of the computation rule. This is desirable since it means that we can retain the commutativity of both conjunction and disjunction which is semantically desirable and gives language implementations more freedom to optimize programs.

This all-solutions predicate is implemented in terms of a generalization called `aggregate` which has the signature (though we show only one of the several modes):

```
:- pred aggregate(pred(T), pred(T, U, U), U, U).
:- mode aggregate(pred(out) is nondet,
          pred(in, in, out) is det, in, out) is cc_multi.
```

by the following code:

```
solutions(Closure, Solns) :-
    promise_only_solution(solutions2(Closure), Solns).

:- pred solutions2(pred(T), list(T)).
:- mode solutions2(pred(out) is multi.
solutions2(Closure, Solns) :-
    aggregate(Closure, cons, [], Solns0),
    sort_and_remove_dups(Solns0, Solns).

:- pred cons(T, list(T), list(T)).
:- mode cons(in, in, out) is det.
cons(T, Ts, [T|Ts]).
```

The predicate `promise_only_solution` is a special builtin predicate (intended for implementors only) with the signature:

```
:- pred promise_only_solution(pred(T), T).
:- pred promise_only_solution(pred(out) is cc_multi, out) is det.
```

which amounts to an assertion by the programmer that all the possible solutions to the given closure (which has a committed choice determinism) are equivalent (that is, are equal under the intended interpretation) and therefore there is no risk to the soundness of the program in treating it as if it were simply deterministic. The interpretation of equivalence here is important, since under the intended interpretation (which may not be the Herbrand interpretation), two values may be syntactically different, but semantically equivalent. This is indeed the case for our present use of `promise_only_solution` where we will get back a list of solutions in any order, possibly containing duplicates, though any list which contains all the solutions that can be computed by the closure is a valid result.

The `aggregate/4` predicate is implemented with a mixture of C code and Mercury code and is executed with the following steps:

- A solution is generated.
- Some code is executed to swap over the heap memory allocation area so that a special dedicated region of memory (the *solutions-heap*) is used instead of the normal one.
- The solution is copied into the solutions heap.
- The *accumulator* closure (passed in the second argument) is called with the current value of the accumulator and the solution.
- The heaps are swapped back.
- The normal heap is truncated back to discard any bindings made during the execution of the solution-generator. (Heap truncation on failure is a common technique in Prolog implementations.)
- Backtracking is initiated to continue generating solutions.
- After all solutions have been generated, the final value of the accumulator is copied from the solutions-heap onto the normal heap.

Nested calls to solutions do not cause problems because the two heaps can be used interchangeably, and the inner call will finish cleaning up after itself before execution returns into the outer one.

This implementation is very similar to the implementations of `findall/3` in some Prolog implementations (see e.g. [73]). The main difference is the use of an explicit accumulator predicate (`aggregate/4`) rather than hard-coding the use of `cons`, and the fact that the usual mode of use is properly declarative, unlike `findall` which depends on the order of evaluation.

2.11 A declarative view of Mercury programs

While it is beyond the scope of this thesis to develop a detailed semantics for Mercury programs, it will be important to have at least an intuition for how we might give a declarative view of Mercury programs. Many aspects of the declarative view we will sketch are closely linked with other current work within the Mercury project to use boolean constraints for mode checking. The most straight-forward way is to express Mercury programs as pure Prolog, and use a semantics for pure Prolog.

Figure 2.2 shows a simple program and figure 2.3 shows the program after it has been transformed in the way we will describe.

First, let us consider definite first order Mercury programs, containing no negation (or if-then-else), higher order or typeclasses. Syntactically, we can transform this class of programs into pure Prolog very simply. Explicit disjunctions must be factored out and new predicate symbols introduced for each, and all the implicit quantification must either be made explicit, or for greater simplicity in the transformed program, all variables that have multiple scopes should be renamed so that each variable in transformed program has exactly one scope.

Types have been dealt with by many authors (for example [77]). From the point of view of a declarative view of Mercury programs, we consider types to simply constrain the set of programs which we will consider legal — we simply don't consider ill-typed programs, and the well-typed programs we can read as if they were untyped. This isn't at all to say that the types are not important, but beyond determining which programs are legal, and which are not, we needn't consider them further. Thus, considering the transformed program (in figure 2.3), we can simply ignore the type declarations.

Unlike types, however, modes are critical in translating Mercury programs into pure Prolog. While modes can be viewed as constraints on the success set (see for example [74]), we use them as meta information which we use transform clauses to allow the compiler to ensure that the operational behaviour corresponds to the declarative view of the program as often as possible. The uniqueness components of modes are like type constraints (in the functional programming world, these same constraints are sometimes attached to the types — see [110] or [75]), and so we needn't consider them in our translation.

We use the mode constraints in the program to define a total ordering of the conjoined literals on the right hand side of a clause (i.e. the body of the clause). Because we wish to allow the programmer to retain as much control as possible over the ordering, we want the order used to be as close as possible to the order in which the literals occurred syntactically. For each declared mode of a predicate, we make a copy of all the clauses, and name them apart (e.g. if `append/3` had three declared modes, we would create three copies of it named `append_0/3`, `append_1/3`, and `append_2/3`). We compute a partial order by using the modes to determine which instances of each variable are *producing* instances, and which instances are *consuming* ones, and constraining consuming instances to occur after producing ones. As part of the computation of the partial order, we select a mode for each call, so all calls are annotated with the specific mode of the call. Where the selected mode of a predicate has an argument with an output mode, and the variable in that argument position is already bound (a situation we call an *implied* mode), we introduce a new variable and add a unification (with the mode `in = in`) after the call. We then linearize this partial ordering by choosing at each step the lowest numbered literal allowed by the partial ordering. (There are many complications in the details of this which are being addressed in the current work in progress of Overton, Stuckey and Somogyi. See [31] for similar work for HAL [32].) If the constraints are unsatisfiable, then we conclude there are mode errors in the program.

The reordering of conjunctions ensures that we don't engage in under-constrained searches that are often unbounded (such as calls to `append/3` in which neither the first or third arguments are bound).

There have been numerous strategies proposed in the literature for integrating higher-order into first order semantics (e.g. [18, 111]). The main problem is that general higher-order unification is undecidable — a very undesirable thing for a practical logic programming language. In Mercury, any unification of a higher-order term with anything other than a free variable is illegal, thus

```
:- pred main(io__state, io__state).
:- mode main(di, uo) is det.
main(IO0, IO) :-
    read(L1, IO0, IO1),
    read(L2, IO1, IO2),
    read(L3, IO2, IO3),
    append3(L1, L2, L3, L4),
    write(L4, IO3, IO4),
    solutions((pred(Lx::out) is nondet :-
        append3(L1, L2, Lx, L3)
    ), Lxs),
    foldl(write, Lxs, IO4, IO).

:- pred append3(list(T), list(T), list(T), list(T)).
:- mode append3(in, in, in, out) is det.
:- mode append3(out, out, out, in) is multi.
append3(As, Bs, Ds, Es) :-
    append(As, Bs, Cs),
    append(Cs, Ds, Es).

:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
append([], Bs, Bs).
append([A|As], Bs, [A|Cs]) :-
    append(As, Bs, Cs).

:- pred foldl(pred(T, U, U), list(T), U, U).
:- mode foldl(pred(in, di, uo) is det, in, di, uo) is det.
foldl(Closure, [], Acc, Acc).
foldl(Closure, [X|Xs], Acc0, Acc) :-
    call(Closure, X, Acc0, Acc1),
    foldl(Closure, Xs, Acc1, Acc).
```

Figure 2.2: A simple Mercury program.

```
main(IO0, IO) :-
    read_0(L1, IO0, IO1),
    read_0(L2, IO1, IO2),
    read_0(L3, IO2, IO3),
    append3_0(L1, L2, L3, L4),
    write_0(L4, IO3, IO4),
    solutions_0(introduced_0(L1, L2, L3), Lxs),
    foldl_0(write_0, Lxs, IO4, IO).

append3_0(As, Bs, Ds, Es) :-
    append_0(As, Bs, Cs),
    append_0(Cs, Ds, Es).

append3_1(As, Bs, Ds, Es) :-
    append_1(Cs, Ds, Es),
    append_1(As, Bs, Cs).

append_0([], Bs, Bs).
append_0([A|As], Bs, [A|Cs]) :-
    append_0(As, Bs, Cs).

append_1([], Bs, Bs).
append_1([A|As], Bs, [A|Cs]) :-
    append_1(As, Bs, Cs).

foldl_0(Closure, [], Acc, Acc).
foldl_0(Closure, [X|Xs], Acc0, Acc) :-
    call(Closure, X, Acc0, Acc1),
    foldl_0(Closure, Xs, Acc1, Acc).

introduced_0(L1, L2, L3, Lx) :-
    append_1(L1a, L2a, Lx, L3),
    L1a = L1,
    L2a = L2.
```

Figure 2.3: The program from figure 2.2 after transformation.

Mercury simply avoids the most problematic aspect of higher-order. The simplest way to translate higher order constructs into pure Prolog is to augment the program with clauses for each instance of `call/N` that could occur in the program (see e.g. [18, 111]). So, for example the program fragment:

```
append([], Bs, Bs).
append([A|As], Bs, [A|Cs]) :-
    append(As, Bs, Cs).
```

would lead to the following extra clauses for `call/N`:

```
call(append, As, Bs, Cs) :- append(As, Bs, Cs).
call(append(As), Bs, Cs) :- append(As, Bs, Cs).
call(append(As, Bs), Cs) :- append(As, Bs, Cs).
call(append(As, Bs, Cs)) :- append(As, Bs, Cs).
```

Given that we have already eliminated any problematic unifications (i.e. ones which are undecidable but would become decidable but incomplete if treated the way we treat them in `call/N`) by declaring that programs containing them are illegal, the resulting program is a fine first-order pure Prolog program corresponding to the original Mercury program.

Typeclasses are transformed away by introducing an extra argument for each typeclass and passing an explicit dictionary structure containing a closure for each method in the typeclass, and then introducing predicates for each method in a typeclass that extract the appropriate closure from the dictionary structure and calling it.

Negation is another problematic area for developing a declarative view of Mercury programs. Lots of work has been done in the area of understanding negation, and we note that negation can be dealt with in reasonable ways. The best approach for Mercury is to follow Naish's treatment [68], which was the basis for Mercury's handling of negation. Somogyi's proposal for Ptah [94] also presents a declarative view of negation which is much the same. The basic constraint in both these approaches is that negations may not bind any non-local variables, and therefore not upset the soundness of the proof procedure.

2.12 Related work

In this thesis we are concerned with adding infrastructure for parallel execution to Mercury. Mercury programs are much more constrained than Prolog programs (e.g. by types and modes), and these constraints have a substantial impact on how we go about exploiting parallel execution. Our basic approach is to build some machinery in the runtime system to support parallel execution and then allow programs to access that machinery either directly or indirectly. We present two way for programs to access the machinery for parallel execution: an independent AND-parallel conjunction and explicit threads. The remainder of this section is devoted to examining the ways parallelism has been exploited in other logic and lazy functional languages.

2.12.1 OR-parallel & AND-parallel Prolog

There has been a large body of work specifically on the parallelization of Prolog. It is highly desirable to be able to take existing Prolog programs, and without the programmer having to do any work at all, make them run faster by compiling and executing them on a parallel machine. Quite a number of systems that aim for this goal have been produced. Very roughly they exploit parallelism in three different ways: those that seek to exploit OR-parallelism by evaluating multiple OR branches of the search space concurrently; those that seek to exploit AND-parallelism by evaluating multiple AND branches of the search space concurrently; and those that seek to exploit both. There are a considerable number of systems that use one or more of these strategies including MUSE [3] (OR-parallel), PNU-Prolog [71] and &-Prolog [47] (AND-parallel) and Andorra [42] (both). All these systems try to execute unmodified Prolog programs in parallel, and assume by default that any two conjuncts or disjuncts may be executed in parallel. This can lead to what is sometimes called *embarrassing parallelism* where very many small parallel tasks are created such that the costs of managing and synchronizing them far outweigh the speedup that results from the parallel execution itself.

Unfortunately, it seems to be the case that for many programs, running them in a parallel Prolog environment makes them slower. The single main reason is that the execution algorithms necessary to handle the general case of Prolog execution have very high overheads compared to the sequential algorithms. One of the main ways in which research into parallel Prolog has progressed is using sophisticated static analysis to avoid having to use a *general* parallel execution algorithm, but instead to use one or more specialized execution algorithms for certain classes of computation (e.g. PEPSys [8, 84, 114], CIAO and its predecessors [47, 48, 93], and ACE [80]). Garcia de la Banda [1] amongst others (e.g. [28, 81, 113]), developed static analysis that tried to determined which goals were independent at compile time, and inserted runtime checks where the static information was insufficient. While this analysis was quite effective, and the resulting system had quite good performance, it did have two significant drawbacks. The analysis cannot always determine at compile time which goals are independent, and therefore has to resort to runtime checks. If the programmer provides to an independent AND-parallel Prolog system the same kind of information that is present in Mercury programs, that Prolog system should not need runtime checks any more than the Mercury implementation needs them. However, many Prolog are not type- or mode-correct (in the Mercury sense), and Prolog programmers rarely provide declarations even for the ones that are. This one reason why analyses to eliminate runtime independence checks can rarely eliminate all such checks. It is very easy to under-estimate the cost of these checks — even if they occur only infrequently, if they have to traverse large data structures to look for variable aliases, they can have an unexpectedly high cost. The second drawback is that the analysis depends on having global information about the program, which means that it is not possible to use separate compilation. Also, like most analysis, its running time is superlinear in the size of the compilation unit (i.e. the whole program), so for very large programs, the analysis becomes infeasible. (This issue is commonly overlooked in the literature, where researchers are generally concerned with devising algorithms and techniques without regard to how they scale beyond quite modest programs. Many Prolog researchers consider 1000 lines to be a large Prolog program. The Mercury compiler contains more than 300,000 lines of Mercury code.)

Closely related to Mercury, is the language Ptah [94] proposed by Somogyi, which used strong type and mode information very similar to that in Mercury. He proposed an execution algorithm that allowed backtracking to occur between concurrent processes, however it stressed *monotonicity* — where computations cannot retract bindings shared with another process by failing [2]. The mode system of Ptah was similar but not quite equivalent to Mercury's mode system. Ptah allowed constrained forms of circular dataflow where Mercury (currently) allows only left-to-right dataflow. As pointed out in section 2.11, Mercury programs are constrained to allow only linear dependencies among conjuncts. By contrast, Ptah allowed a richer set of mode declarations which allowed circular data dependencies subject to the constraint that there was a unique identifiable producer for every variable, and there was a sequence of mode segments (i.e. a mode for a slice through a procedure) that ensured that producers were executed before consumers. There was a construct called *iterated mode segments* that allowed unbounded sequences of mode segments to be used to express common stream operations. A similar system could possibly be implemented for Mercury, but rather than using Somogyi's iterated mode segments, recursively defined modes would probably be more elegant, and would integrate more naturally with the existing system, where types and instantiations are both defined recursively. These recursive definitions give a language of (infinite) regular trees for which many useful algorithms (such as unification) are well understood.

Unfortunately, Ptah was never actually implemented, so there is no empirical data to assess the effectiveness of the techniques proposed.

[2] Reform Prolog [9] had a similar concept, *unconditional bindings*, and used it for similar purposes.

2.12.2 Committed choice languages

The problems of making the parallel execution of full Prolog efficient and the difficulty of reconciling parallel I/O with Prolog's semantics lead to the development of the *committed choice* class of languages. Clauses in committed choice languages contain a *guard* as well as a head and a body. To execute a call, implementations of these languages try the guards of all the clauses whose heads unify with the call. When one succeeds, they commit to that clause and discard the others. If a failure occurs later, then the entire computation fails; once the implementation chooses a clause to commit to, it never backtracks to choose another.

As originally proposed, most of the committed choice languages allowed arbitrary computation in the guard, but without fail discarded general guards in favor of "flat" guards containing only unifications and builtin operations. This was because it was too difficult to execute the general *deep* guards efficiently. Concurrent Prolog [90], Parlog [19], and GHC [106] are all examples of languages that have evolved in this fashion.

Of these, the early versions of Parlog are the most interesting in relation to Mercury because they had mode declarations for predicates (although clause bodies could override these declarations), and (at least in principle) the compiler would verify that the clauses of the program were consistent with the mode declarations. However, Parlog was not strongly typed, and the mode system had only three possible modes: input, output, and a general "unknown" mode. This proved to be insufficiently expressive, and after a brief experimental phase, the Parlog compiler ignored the mode declarations altogether. The core of the problem was that to express the dataflow involved in stream data structures (and the various other parallel paradigms developed for such languages), it is really necessary to capture not just whether the top level node of a data structure is an input, but information about the whole data structure.

An important characteristic of the committed choice languages is that they eschew simple declarative semantics in favor of a simple and (relatively) efficient implementation. By comparison, it has been an important goal for the Mercury project to achieve both simplicity of semantics and efficiency of implementation. Mercury allows committed choice computations, but constrains the use of such computations to contexts where the soundness of the program is preserved. In languages like Parlog, the program will abort if the wrong clause is chosen and failure subsequently occurs.

2.12.3 Threaded Prolog & Erlang

Unlike the parallel Prolog and committed choice logic programming languages discussed in the previous sections, our work involves explicit parallelism, and uses ground structures to communicate between concurrent processes. SICStus-MT [34] is a system that is similar in some respects. It creates threads with a `spawn/2` predicate which does much the same as our `spawn/3`. SICStus-MT provides communication via a pair of predicates `send/2` and `receive/1`. The failure of a thread leads to a runtime exception. The main difference between our system and SICStus-MT, is that we have made sure that we have designed our system in such a way as to ensure that it has a simple declarative interpretation — something that the authors of SICStus-MT have not tried to achieve. Indeed, there is no particular reason that they should, since Prolog does not have a simple declarative interpretation even without threads (because of side-effects, unsound negation, etc). Therefore in our system, we can be sure that the failure of a thread will never occur, and our determinism declarations for threads explicitly capture the notion that the interaction of threads is nondeterministic.

Paul Tarau has done some research on a related line: multi-engined Prolog implementations [25, 102, 103]. The engines communicate deterministically, but an engine can ask another for its next solution, so `findall/3` is implemented at the user-level by creating an engine to execute the nondeterministic goal, and then making repeated calls to the engine to get the next solution. The resulting system is similar to Wadler's technique in lazy functional programming of simulating nondeterminism by returning not a single answer but a list of solutions [109]. Of course, the main difference is that the functional version has a very carefully thought out semantics, where the multi-engine approach does not. In Mercury, it is possible to replicate this kind of functionality with a one solution-at-a-time version of `unsorted_solutions/2`, which is committed choice, since the particular sequence of solutions that are computed depends on the details of the computation rule as well as on the semantics of the closure.

Erlang [5, 6] was created explicitly to support flexible, and robust programming of highly concurrent applications (originally for programming telephone exchanges). It is a derivative of Prolog, in as much as its creators started with Prolog, but discarded full unification, backtracking and added concurrency primitives for creating threads, passing messages between them and allowing them to control one another. Indeed, the resulting language is usually considered a functional language rather than a logic one. It has been quite successful: it has been used to implement high level applications for providing web services, and it has been used to implement low level infrastructure such as IP stacks. It has not tried to be `pure` in the way that some of the parallel logic programming do. It has taken a pragmatic approach, using the ideas of pattern matching, automatic memory management and nondeterministic choice, without trying to construct a system that has a simple declarative semantics closely coinciding with its operational semantics.

2.12.4 Parallel & Concurrent Haskell

Glasgow parallel Haskell (GpH) [105] provides parallel execution for Haskell programs. Like our system, GpH uses explicit parallelism using the `par` and `seq` combinators. The `par` combinator takes two arguments and spawns a task to evaluate the first while returning the second. The `seq` combinator also takes two arguments, evaluates the first (reducing it to weak head-normal form), then returns the second. These two combinators allow the programmer to explicitly parallelize or sequentialize parts of the execution of a program.

We can make an equivalent predicate to the `par` combinator in Mercury by creating a predicate `par/2` that takes a closure of type `pred(T)` as its first argument and returns straight away an object of type `parResult(T)` which represents the result of the computation. This object is essentially similar to a Prolog variable (with locking). The computation itself is started up as a new task which destructively modifies the `parResult` object when it finishes. A function `value/1` takes the `parResult` object and returns the value contained in it, blocking if necessary, which is rather like the `seq` combinator. (Declaratively, we can view the `parResult` object as just being a wrapper around the value, and `val/1` as an access method for extracting the value, ignoring the possibility of blocking altogether.)

The implementation of GpH is very different to our implementation. Rather than using multiple threads within a single address space, GpH is implemented on top of the Parallel Virtual Machine (PVM) library, which uses explicit message passing between processes. This has the advantage that it allows distributed computations across multiple machines rather than relying only on multiple processors within a single machine. It also has the benefit (highlighted by our benchmarking), that garbage collection can run independently for each process. Garbage collection itself is complicated by the fact that external processes can hold roots, however the authors claim that this does not cause too many difficulties. Copying of data between processes is done on demand, so rather than composing a message containing a closure and all its reachable data, only the some of the data is copied, leaving "pointers" back to the spawning process.

When a task is spawned in our system (as we will show in chapter 4), a whole new task structure is created to execute the task regardless of whether the the task can be executed any time soon. One of the limitations of this scheme is that a task consumes a moderate amount of (virtual) memory, so creating task structures that will not be used for a long time wastes resources, and if many such tasks are created, then it is quite conceivable that the process will run out of memory. GpH has a nice way of avoiding this problem. Rather than creating the task structure straight away it creates a "spark" which is effectively just the closure. When a worker process needs some work to do, it takes a spark and creates the necessary execution environment for it before executing it. The sparks are much smaller and consume fewer resources than a full execution environment, so many sparks can exist. Interesting future work would be to adapt this technique to our system. We could create a closure for each task, and when an engine needs more work, it could create a task structure for the closure at that point. An obvious optimisation would be to keep the task structures of tasks that have finished, instead of destroying them. That way, a new task structure would only need to be created when a running task suspends. This would substantially reduce the memory requirements for programs that create many independent tasks.

Another Haskell variant that is relevant to our work is Concurrent Haskell [54]. This work is quite similar to our work on explicit threads. They use the IO monad in the same way we use the IO state to provide a declarative means for communication between threads. The authors describe mechanisms for creating and communicating between threads that are very much like our own.

Chapter 3

A model for parallel execution

In this chapter we present a modification to the Mercury execution model and runtime system which enables multiple goals to be evaluated in parallel. We show how this is used to provide independent AND-parallel conjunction, and we present a benchmark which show some of the performance issues that arise in our implementation.

3.1 One system; Two execution environments

The Mercury system consists of a compiler, library, and runtime environment. The system is organised so that a variety of features can be included in or omitted from the system. These features include different garbage collection options, debugging, trailing to support backtrackable destructive update, profiling and so on. The different features are included or omitted from the runtime system (which is mostly written in C) using conditional compilation. It is a requirement that our support for parallel execution work within the constraints imposed by this architecture. Our support should be as portable as the rest of the system which runs on most Unix and Win32 platforms.

Our overall approach is to split the current execution environment into two parts. The *engine* will contain those parts of the runtime environment that do not associate to the evaluation of any particular goal, but support the evaluation of goals in general. The *task*[1] will contain those parts of the runtime environment that relate to the evaluation of a specific goal. Our overall model for the parallel execution of Mercury programs is to have multiple engines running on separate processors, and have them independently evaluate tasks. Of course, the tasks must be able to communicate and interact, though we will restrict the allowable interactions to those for which we can provide an efficient implementation. The purpose for splitting the execution environment in this way is to allow us to map engines onto machine processors and have them work at evaluating tasks from a pool.

Using this model, and ignoring any other optional features in the Mercury system, four forms of the Mercury system are possible:

- A single engine allowing only one task (i.e. the current system);
- a single engine allowing multiple tasks;
- multiple engines allowing only a single task per engine;
- multiple engines allowing multiple tasks per engine.

As it happens in our implementation, allowing multiple tasks per engine has only a very marginal cost for programs that make use of only a single task, so we end up with two versions of

[1] In the source code for the system the task structure is called a *context* structure. We have used the term *task* in our discussion for clarity.

the system: one that is sequential (possibly concurrent), using only a single engine, and one that is parallel using multiple engines.

Each engine structure contains a pointer to the task it is currently executing. In fact, as we shall show later, this field is only used when switching between tasks; during normal execution, we access the relevant data through virtual machine registers.

3.2 One system; Three implementation layers

For the purposes of exposition we will consider the implementation of support for multiple tasks and parallelism in three layers. The operating system (OS) level provides us with a way of creating processes that can run on the multiple processors that are available on the hardware platform. The OS also provides support for synchronizing those processes, and allowing them to communicate (e.g. via message passing or shared memory). The execution model layer uses the facilities provided by the OS to execute tasks in parallel. The execution model that we will describe is a modification of the basic execution model for Mercury described briefly in section 2.9. The language layer provides mechanisms (either implicit or explicit) for programs to utilise the facilities for parallel execution and multiple tasks provided by the execution model layer. The language layer may constrain the specific types of parallelism that occur (for example independent AND-parallel programs cannot deadlock). As suggested in the previous section, and in accordance with the goals stated in chapter 1, we want an efficient implementation, so to a large extent we will allow choices in the OS and execution model layers to dictate what choices are available at the language layer.

3.3 The OS layer

At the OS level, the availability of portable techniques to access parallel execution is fundamental to making the Mercury implementation support parallelism. The main alternatives on Unix-like platforms are *processes* and *threads*.

Unix processes are the original mechanism for having multiple jobs run, and support for them is pretty uniform amongst the Unix platforms. For our purposes, the important consideration is that processes each run in their own address spaces, so either all communication between them must be explicit, or explicit shared memory segments must be allocated. There are two different paradigms that are used for processes to communicate. From the BSD line of Unix variants, the special file `/dev/zero` could be mapped into memory with the `mmap` system call before the parent process forks to create the child process. This creates a region of shared memory between the parent and its children, and either sockets/pipes or spin-locks are used to perform synchronization. From the System V line of Unix variants, explicit shared memory segments can be created, as can explicit message queues and semaphores. These are propagated to child processes when the parent forks, or they can be explicitly attached to the child process after the fork. (The interested reader should consult [101] for details about both the BSD and System V interprocess communications (IPC) mechanisms.) Although most modern Unix variants support both, there are a number of subtle differences, and processes are relatively expensive to create.

We prefer threads as a mechanism for accessing machine parallelism. POSIX threads [58, 83] are a standardized way of creating light-weight parallel process-like objects. Threads are created within a process, and share the same address space as all the other threads in that process. Synchronization between threads is achieved with mutexes and condition variables which are also part of the POSIX standard. Threads are much cheaper to create than processes, and because they share the same address space, there is no requirement to explicitly communicate data between them either by message passing or copying into shared memory. The main disadvantage of using POSIX threads is that they are a relatively recent standard, and there are still variations in the support amongst Unix platforms[2], and some implementations of pthreads are not (yet) well tuned.

[2]For example one commercial Unix implementation has a missing underscore (_) in just one of the function

Having split the original execution model into the engine and task components, we then wish to harness parallel hardware by using threads to run an engine for each processor. Some versions of Unix allow a thread to be bound to a specific physical processor, but on most systems, we must simply rely on the OS to do this implicitly.

3.4 The execution model layer

We have already stated that our overall execution model is to have a Mercury engine running on each processor in the machine, and to have a pool of available tasks for them to work on. Using threads as our mechanism for achieving this means that it is natural for us to use the shared address space to pass data between the engines and tasks. This effectively enables us to utilise shared memory machines, particularly symmetric multiprocessor (SMP) machines. Since most multiprocessor machines in the world are SMPs, this seems to be a reasonable design decision. It is simple; it is efficient, since shared data do not need to be copied; and the single-assignment nature of the source language means that locking of shared data is unnecessary in almost all cases.

The alternative to using the shared address space would be to use an explicit message passing system which would port more easily to machines with non-uniform memory access, and would scale to distributed environments more readily. The parallel implementation of Haskell described in [105] takes this latter approach. (We note that it would be interesting to try this approach, though probably it would be best done at a higher level using a library to establish connections between processors, and using closures to pass tasks around.)

Implicit in the engine-task model, is that tasks are executed independently. By this we mean that the behaviour of one task shouldn't directly affect the behaviour of another, unless they explicitly communicate. In particular, this means that one task may not, by failing, retract bindings that may be visible to another task. This restriction greatly simplifies the execution model, and allows programs to be executed in parallel with much lower overheads than would otherwise be the case.

The reason that shared data do not need to be locked in most cases is that the Mercury language requires that every variable has a single producer designated at compile time, so in a legal Mercury program, it is guaranteed that only a single point in the code will try to bind a given free variable. The mode system constrains consumers of bound variables to be scheduled after the producers of the binding. There are possible extensions to the mode system that relax these constraints, by allowing free variables to be shared, and therefore to allow unifications whose precise mode is not known until runtime. Other possible extensions to the mode system could guarantee a single producer, but would need to suspend the consumers that try to examine the variable until it was bound. In these cases locks would be necessary, but we have not implemented any such extensions to the mode systems so we need not consider them further.

The Mercury abstract machine was originally defined as a collection of global variables and macro definitions. We have modified it to place each global variable inside either a data structure representing an engine or a data structure representing a task. The engine structure then contains a pointer to the task structure for the task it is currently executing. Broadly, the Mercury abstract machine can be broken up into a few logical groups. There are the two execution stacks (detstack and nondetstack) which store the activation frames of procedures currently being executed, the heap and the solutions-heap from which we allocate Mercury terms, the special virtual machine registers that are used to manage the stacks and the heaps, and the general purpose virtual machine registers we use for argument passing and temporary storage.

The execution stacks, and the virtual machine registers managing them, belong in the task structure since they store information specific to the execution of a particular goal.

We put the heap in the engine for two reasons. The heap is a large object, so we don't wish to replicate it more often than necessary. Also, garbage collection needs to consider all the possible root pointers, not just those in any one task, so its scope is greater than the task. Another possibility would be to make the heap a global structure, shared between all engines. A logical

names!

reason for this would be that all tasks and engines must be considered by the garbage collector to find all the roots, however this has the significant disadvantage (which we will come back to) that it would require obtaining and releasing a lock to prevent two engines from trying to allocate from the heap at the same time. Given that Mercury is an allocation intensive language, this would be a considerable burden, so instead we put one heap in each engine.

From an architectural point of view, the most important thing to ensure with respect to the placement of the general purpose virtual machine registers is that accesses to them be as quick as possible, since almost everything the virtual machine requires accesses to them. We keep the first N (where N depends on the platform) virtual machine registers (both general and special purpose) in real machine registers. We keep all the others in an array called `fake_reg`, which also has room for saved copies of the virtual machine registers that are allocated their own machine registers. Ideally, the address of each element of the `fake_reg` array should be known at compile time, so that the (C) compiler can generate a load from an absolute address if that is the fastest way to access the virtual register. If the compiler doesn't know the address of each virtual register but has a pointer (that we make sure is kept in a machine register) to the start of the array, the compiler can generate a register-indirect load. On platforms with few registers (e.g. x86), loading from a constant address is likely to be faster. On platforms with plenty of registers, an indirect load may even be preferable to a load from a known absolute address since on some RISC platforms (e.g. SPARC) loading from a constant address takes two instructions, whereas a register-indirect load with a small constant offset is a single instruction.

The `fake_reg` array could be a member of either the engine structure or the task structure. As we just discussed, whichever structure it is a member of, we will either need that structure to be a global variable or we need a global register variable pointing to it. Since the number of task structures is dynamic, they cannot be global variables, so effectively we have three choices: the `fake_reg` array is in the engine structure which is a global variable, the `fake_reg` array is in the engine structure which is pointed to by a global variable, or the `fake_reg` array is in the task structure which is pointed to by a global variable. The last alternative requires significantly more memory than the first two, since there are more task structures than engine structures, and the Mercury abstract machine is defined to support 1024 general purpose virtual machine registers. We have therefore chosen to put the `fake_reg` array into the engine structure. In grades that support parallelism, i.e. multiple engine structures, we cannot store the engine structures in global variables, since we do not know at compile time how many engines we may want at runtime (This is because the program may be compiled on a machine with a different number of processors to the one on which we will run). In grades that do not support parallelism, i.e. which have only one engine structure, we do store that structure in a global variable to give the compiler maximum freedom to optimize accesses to its fields. The differences between the two methods of access to the engine structure are hidden from most of the code of the runtime system by a layer of macros.

A potential downside of our choice of storing the `fake_reg` array in the engine structure is that switching an engine from one task to another requires saving any live registers from the `fake_reg` array into the task structure, and then restoring them when the task is next executed. If we wish to allow tasks to be preemptively scheduled, then we need a mechanism for knowing which virtual registers are live at any given instant, because the alternative of assuming that all virtual registers are live and saving them all will be prohibitively expensive. Unfortunately, knowing exactly which registers are live at any given time is rather difficult. One option would be to track the highest numbered register used in a procedure and create a lookup table that maps the code address range of a procedure to the highest numbered register used in that procedure. However, the Mercury compiler cannot construct such a table since it does not know what addresses procedures start at, and constructing the table at runtime is an expense we are not willing to pay.

As a consequence our current implementation supports only cooperative scheduling, not preemptive scheduling of Mercury level threads. Therefore an engine switches contexts only at known points in the code. The compiler arranges things so that at such points, only the special purpose registers are live. Since there are only a small number of these, saving them into the task structure is relatively inexpensive. If we ever wanted to support preemptive scheduling, we would have to put the `fake_reg` array into the task structure (where it logically belongs, since it stores values that are local to the evaluation of a given Mercury goal). To keep accesses to the `fake_reg` array fast, we would make the global register variable that currently points to the engine structure point to the structure of the currently executing task instead. This would require replacing the pointer in the engine structure that points to the structure of the currently executing task with a pointer in the task structure that points to the engine currently supporting its execution.

The engine structure therefore contains the following fields:

`fake_reg` is an array of word sized structures that represent the virtual registers in the abstract machine. As we mentioned above, some of the entries in this array may be only for saved copies of machine registers.

`hp` is a special register that points to the start of the unused area of the heap.

`heap_zone` points to the structure containing the memory management information about the heap.

`sol_hp` is a special register that points to the start of the unused area of the solutions heap.

`solutions_heap_zone` points to the structure containing the memory management information about the solutions heap.

`this_context` contains a pointer to the task currently being executed by the engine.

`owner_thread` contains a value that identifies the thread in which the engine is executing.

`c_depth` contains a counter that tracks the number of times execution has (recursively) passed from C code into Mercury code.

The last three of these fields we have introduced for the parallel execution environment. Their use will be described in section 3.4.2.

The task structure contains three kinds of fields. The first kind are backing storage for the special virtual registers that need to be saved and restored when a task suspends and resumes.

`context_succip` contains the value of the success instruction pointer for this task when the task is not active.

`context_sp` contains the value of the *detstack* pointer for this task when the task is not active.

`context_maxfr` contains the value of the pointer to the topmost frame on the *nondetstack* for this task when the task is not active.

`context_curfr` contains the value of the pointer to the current frame on the *nondetstack* for this task when the task is not active.

The second kind are the data structures that are used for managing the memory used by the stacks. As described in [96] these areas of memory use explicit memory protection (using the `mprotect` Unix system call) to avoid having to check explicitly for overflow or underflow each time a stack frame is created or destroyed. The structures contain the necessary information for the runtime system to identify the bounds of the stack so that in the event of an illegal access, the system can determine if stack underflow or overflow was the cause.

`detstack_zone` points to the structure containing the memory management information about the *detstack* for this execution context.

`nondetstack_zone` points to the structure containing the memory management information about the *nondetstack* for this execution context.

The third kind of field are those related to suspending and resuming tasks.

`resume` contains the address to which execution should branch when this task is next executed.

`owner_thread` contains a value that identifies the thread in which execution entered Mercury code from C code using this task.

`context_hp` contains the value of the pointer to the start of the free space in the heap when this task was last active.

`min_hp_rec` contains the minimum value to which we may truncate the heap when this task next becomes active.

The last two of these fields relate to heap reclamation on failure and will be described in section 3.4.3.

3.4.1 Context switching and scheduling

As we have outlined already, at present all context switching is cooperative, not preemptive. When a task decides either that it needs to suspend (e.g. waiting for a lock, or for I/O) or that it should yield to another task, it needs to save its state, and arrange for another task to be run. To save its state, a task has to copy the values of the stack control registers (for both stacks) and the success continuation register from their virtual machine register locations into its task structure. It also must save the address at which execution should proceed when the task is next run. Conversely, to restore a saved task, the control register values are copied back from the task structure into the virtual machine registers, and execution branches to the continuation address stored in the task structure.

We use a first-in-first-out (FIFO) scheduling policy. Given that we use cooperative scheduling, "fair" scheduling policies are not really feasible, a last-in-first-out (LIFO) approach is the simplest alternative. With the actual implementation that we have, it is quite simple to switch between the two, and in practice we find that a FIFO policy works better. For the scheduling of runnable tasks, we have a run queue: task structures have a `next` member which is used to link them together. The run queue is represented by a global pointer to the start of the linked chain of runnable tasks, a global pointer to the last task on the run queue and a lock for protecting updates to the run queue. When a task becomes runnable, it is put on the end of the queue. When a Mercury engine needs work, it takes the first item off the front of the run queue. Some care is needed in the implementation to make the locking safe. If there are no runnable contexts, then we need to atomically suspend the engine and release the lock on the run queue.

3.4.2 Foreign code

The architecture as we have described it, allows any task to be executed on any Mercury engine. When foreign code calls Mercury code, and that Mercury code suspends, it is important that we ensure that the task being used to execute the Mercury code does not try to return from Mercury code into foreign code in any engine other than the one on which execution entered the Mercury code. This is because the foreign code will (almost certainly) have used the C stack to store data, and since each Mercury engine runs in a different POSIX thread with a different C stack, we need to ensure that when the Mercury code exits, the foreign code picks up the right stack. To do this, we have modified the wrapper code that the foreign code invokes to call Mercury code so that it records in the task structure which engine was being used when execution branched into Mercury code. Thereafter, this task is constrained to use only that engine. When the Mercury code finishes, the field that records which (if any) engine the task structure is required to use (`owner_thread`) is reset to its previous value.

3.4.3 Heap allocation and garbage collection

The Mercury implementation supports two heap allocator/collectors: a conservative collector for C written by Hans Boehm et al [11], and a native garbage collector (which is still incomplete at the time of writing). The conservative collector is a drop-in replacement for `malloc()`, that works

# of words	% of allocations
1	10.61
2	60.03
3	6.27
4	8.53
5	2.50
6	1.14
7	3.90
8	0.12

Table 3.1: The relative frequency of memory allocations of different sizes for the Mercury compiler.

by assuming that any bit-pattern that looks like it could be a pointer into the area of memory that it manages *is* such a pointer, and therefore does not release the block of memory being pointed to. While this may sound dubious, in practice it works very effectively, and gives us a collector that works with both Mercury code, and the C code used to implement some parts of the runtime environment and library. The collector uses several straight-forward but important techniques that substantially improve the precision with which it can reclaim memory. For example, if the collector encounters a bit-pattern that looks like a pointer that points into memory under the collector's control, but which the collector knows to be free (because it has not been allocated), the collector can *black list* the surrounding block of memory, meaning that it will not be allocated. This important technique substantially reduces the likelihood of memory being conservatively retained, even though it is really garbage.

The conservative collector has an important drawback with respect to multi-threaded execution: by default it allocates from a single global pool, so every allocation has to obtain and release a global lock. For an allocation-intensive language like Mercury, this lock quickly becomes a bottleneck for performance. The collector package however, does have some facilities that we can use to ameliorate this problem. It has a function called `malloc_many()` which takes a size parameter and allocates a page-worth of objects of the given size. The objects are linked through their first word to form a free-list. We can arrange to use `malloc_many()` to allocate pages of blocks with various sizes in each engine, so that the engine can then parcel them out without having to obtain a lock. It only needs to obtain a lock inside `malloc_many()` when it gets more blocks from the global pool. Fortunately, as shown in table 3.1, the majority of allocations, are accounted for by just a few sizes. In each engine we have a pool for objects of one or two words (accounting for 70% of allocations), a pool for objects of three or four words (accounting for almost 15% of allocations), and a pool for objects of five, six, seven, or eight words (accounting for another 8% of allocations). These pools together account for over 92% of allocations, thus dramatically reducing the amount of contention for the global allocation lock.

When executing with native garbage collection (or no garbage collection), we can use a common memory management trick from Prolog: heap reclamation on failure. Because no bindings can escape a failed computation, any heap that was allocated during that computation must be garbage and can be reclaimed. Like most Prolog implementations, the native memory allocator for Mercury parcels out heap memory by advancing a heap pointer. When execution reaches a choice point, our execution algorithm saves the value of the heap pointer, and then when execution backtracks, resets its value to the one it had before the failed computation. However, we cannot use this same scheme unchanged in coroutining and parallel grades. If task 1 allocates some heap and suspends, and then task 2 allocates heap and suspends, and then task 1 resumes and fails, it cannot reset the heap pointer in this way because task 2 may have allocated data structures on the heap.

In coroutining and parallel grades, we therefore manage heap reclamation on failure by adding an extra virtual control register for the heap `min_hp_rec`. Together with `hp`, `min_hp_rec` is used to allow the maximum allowable amount of heap to be truncated on backtracking. When a task is saved, the values of both registers are copied into the task structure. When a saved task is being restored, we compare the current value of `hp` and its saved value in the task structure we are about to restore. If the current `hp` is greater than the new task's, then another task has allocated terms on the heap, so the lowest value to which `hp` can be truncated on backtracking is its current value. We therefore set `min_hp_rec` to this value also. If the current `hp` is less than the new task's, then another task must have truncated the heap on failure since this task was last run. This can happen if the last task to allocate any heap was the same one that truncated it on failure, and neither this and nor any other task run between those two events has allocated any heap. If the current `hp` and the new task's are equal, then no task run since the new task was last run has allocated any heap (or if it did, was able to reclaim it on backtracking), so the lowest safe point to which the heap may be truncated is exactly what it was before (and is stored in the `min_hp_rec` register).

When backtracking occurs, rather than unconditionally resetting `hp` to the value saved in the choice point, the saved value is compared with `min_hp_rec`. If the value in the choice point is lower then `hp` should be reset to `min_hp_rec` not the value saved in the choice point, since `min_hp_rec` is the minimum safe value.

3.4.4 Execution stacks

The execution stacks are allocated when a new task is created. The detstack is usually allocated 4Mb of storage, and the nondetstack 128Kb. We not only allocate this storage, but we establish *red-zones* at either end, which are regions of memory which we make unreadable and unwritable using the Unix system call `mprotect()`. This is done to avoid having explicit overflow/underflow checks.

If we are creating many short-lived tasks, then two issues come up: the overheads of creating the stacks becomes a significant cost factor; and the amount of virtual memory that gets used becomes very large.

One way of reducing the amortised startup costs is to cache task structures when they are no longer used, and to then use these rather than allocate new ones. This is a satisfactory solution when the number of live tasks at any one time is fairly small, but does not help if there are a large number of tasks.

Another option is to dispense with the stacks altogether and allocate stack frames on the heap. This makes the size of each task much smaller, but does increase the cost of normal execution, since calls must be made to the heap allocator to create each stack frame, and links must be established between them.

Because the management of heap allocated stack frames is more expensive than the management of normal stack frames, it is desirable to retain a stack for deterministic sub-computations that can be proven to be suspension free. If a sub-computation cannot suspend, then it will not need to store any information in the task to allow it to survive context switches. Instead of using a task based stack, it can use a Mercury engine based stack, which is only allocated once per processor, rather than once per task.

Proving suspension freeness can be done with a straight-forward bottom-up analysis that should be done just before code generation, after other high level transformations such as inlining, specialization, and constant folding. Currently, native Mercury computations themselves cannot cause suspension, so only foreign code calls can possibly cause suspension. We add an annotation to foreign code declarations that allows the programmer to specify whether or not the foreign code may suspend. The conservative default for this is to assume that the code may indeed suspend. Higher order poses a problem to straight forward analysis, because we often lose information about what procedure is being called, and therefore whether the higher order call may suspend or not. Generally speaking, however this is not too much of a problem because higher order calls are relatively expensive, and therefore the overheads of heap allocated stack frames are relatively less important. Moreover, higher order calls are often eliminated by a constant propagation pass that generates specialized copies of procedures that make higher order calls when the closure being called is known.

3.5 The language level

Having constructed the infrastructure necessary for supporting multiple parallel tasks, it remains to show how they can be utilised from the Mercury level. In this section we show how they can be used to support independent AND-parallel execution. (In chapter 4 we will show a more explicit model that uses an explicit thread model too.)

We introduce independent AND-parallelism with an explicit parallel conjunction operator "`&`". Parallel conjunction has the same declarative semantics as sequential conjunction. In both cases the operators correspond to the logical conjunction connective, however, the language accepts only a subset of all possible conjunctions: those that are well moded. For sequential conjunction, the well modedness rules require that there is an ordering of the conjunction such that every consuming occurrence of a variable comes to the right of a producing occurrence of that variable. For parallel conjunction, the well modedness rules are different. A parallel conjunction "`A & B`" is well moded if the goals "`A`" and "`B`" each bind a distinct set of variables, and "`B`" does not depend on any bindings made in "`A`" and "`A`" does not depend on any bindings made in "`B`". We impose the additional requirement that both goals are deterministic (and thus will succeed exactly once). In practice this last restriction is largely cosmetic since we can transform nondeterministic and possibly-failing code into deterministic code. If a parallel conjunct `A` may fail, then we can wrap it in an if-then-else and put its outputs in a *maybe* term (using tuples as necessary). Thus

```
    ....,
    (A & B),
    ....
```

becomes

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

    ....,
    (
        ( A ->
            X = yes({outputs-of-A})
        ;
            X = no
        )
    &
        B
    ),
    X = yes({....}),
    ....
```

The code following the parallel conjunction can then unify `X` and extract `A`'s outputs if `A` succeeded. Of course, it should be noted that in this paradigm, even when `A` fails, `B` is still executed to completion, and the failure of the conjunction occurs only once both have completed. Potentially, this leads to substantial wasted effort[3]. We do not believe this to be a serious problem because if the failure of `A` may lead to a large amount of wasted work on `B`, then the programmer should know this (perhaps with the aid of a profiler), and use sequential conjunction rather than parallel conjunction.

For parallel conjuncts that are nondeterministic, we can use `solutions/2` to gather up all the success bindings, and use a nondeterministic call to `member/2` to supply them nondeterministically to the context of the parallel conjunction, so a parallel conjunction of `A` and `B` becomes

[3]In Prolog it could also lead to a change in observable program behavior if `B` contains side effects, but this cannot happen in a pure language such as Mercury.

```
....,
(solutions(A, AsSolutions) & B),
member(Soln, AsSolutions),
....,
```

As in the case of parallel conjuncts that may fail, in the case where `A` yields no solutions, this transformation has the potential to waste large amounts of work on evaluating `B`. However, it is actually more likely that this form will reduce the amount of work done for two reasons. The first reason is that `B` gets executed exactly once, whereas in the case of a sequential conjunction it gets executed once for each solution to `A`. The second reason is that the call to `solutions` sorts the solutions to `A` and removes duplicates, the rest of the outer conjunction following the call to `member/2` may be executed fewer times than in the original code (though there is the added $O(nlogn)$ cost of sorting the solutions). In fact, it is not unusual for sequential Mercury programs to contain this sort of code using sequential conjunction rather than parallel conjunction.

The compiler normalizes nested parallel conjunctions so that the internal representation of a parallel conjunction is as a list of parallel conjuncts. The compiler generates code that creates a new task for all but the last parallel conjunct; the last parallel conjunct is executed in the parent task. (At most one conjunct can execute in the parent task, and the last conjunct is as good a choice as any.) The resumption address in each of the new tasks points to start of the generated code for the corresponding parallel conjunct. The parent task communicates bindings to the children by copying its topmost *detstack* frame (which corresponds to the current procedure) onto the *detstack* of each of the children. All parallel conjuncts are generated with the same view of the stack, that is, variables are all stored in the same slot in each child task. The output bindings of the children are communicated back to the parent by initializing a stack slot for each output variable with the address of a stack slot in the parent into which the binding of the variable should be stored. The code for each parallel conjunct other than the last one has an epilogue. The epilogue starts by copying the output bindings to their correct locations (by doing an indirect assignment into the parent's stack frame). It then performs a *join* operation to let its parent know that it has finished its execution, after which it terminates and releases its task structure. In the code for the last conjunct (which executes in the parent's task), the variables will already be in the right stack frame, so we don't need to do anything special for them. The code for the last conjunct also does a join operation, but it uses a variant of the join operation which blocks until all the other conjuncts have performed their join operation. It then continues execution at whatever construct follows the parallel conjunction.

Figure 3.5 shows the source code of a simple parallel conjunction. The code generated for it is shown in 3.5. The two input lists are in (virtual) registers `r1` and `r2` on entry, and the output list will be placed in `r1` on exit. Line 2 shows the creation of the stack frame. The stack frame contains slots for storing the two input arguments (slots 1 and 2), temporary storage for the intermediate result lists `Cs` and `Ds` (slots 3 and 4), a slot for the synchronization datastructure (slot 5), a pointer back to the stack frame (slot 6) and the return continuation (slot 7). The stack pointer `sp` is saved into the stack frame on line 6. This is so that later, when the two child tasks need to give their outputs back to the parent task they can find where to store them (see lines 19 & 29 for where this happens). Lines 7 & 8 allocate and initialize a structure (containing a lock and a counter) that is used for synchronization (on lines 20 & 30). The first argument of the macro call `init_sync_term` is a pointer (which in this case is stored in `r4`) to the structure and the second argument is the number of tasks that need to join before a call to `join_and_continue` on this synchronization term will proceed. The macro call on line 10 to `fork_new_context` creates a new task which will execute code at the address specified by its first argument, and the current context branches to the address specified by the second argument. The third argument gives the number of slots holding variables in the current stack frame. This is used by the macro to copy the stack frame from the current context to the child context.

```
:- module par.

:- interface.

:- import_module list.

:- pred p(list(int), list(int), list(int)).
:- mode p(in, in, out) is det.

:- implementation.

p(As, Bs, Es) :-
    (
        append(As, Bs, Cs)
    &
        append(Bs, As, Ds)
    ),
    append(Cs, Ds, Es).
```

Figure 3.1: An example of a parallel conjunction

3.6 Performance

Having shown how we provide support for parallelism and concurrency at the execution model layer, and one way in which we can access that functionality by using independent parallel conjunction, we now provide some results that give some information about how the system performs. The goal of this section is to show some of the practical performance issues for our execution model, rather than in anyway trying to be an exhaustive performance analysis or comparison with other systems. While we would like to have produced a large suite of benchmarks to give a more comprehensive evaluation of the performance of our system, our efforts to have been severely hampered by the necessity of maintaining the parallel execution machinery in an environment of rapid changes in the Mercury system itself, and in which the only parallel machines we have access to either have old and not fully working pthreads implementations, or which do not have a port of the parallel version of the Boehm colector on which Mercury depends. Moreover, during the timeframe of the production of this thesis, much of the emphasis within the Mercury group has shifted from the compilation to low-level C as described in [96], to a high-level C system [44], in which the details of the parallel execution machinery are quite different.

The main factor that concerns us is how much independent parallel computations affect each other. Independence means that there are no direct interactions, and the benchmark program that we will use is long running, so startup and synchronization issues will not have an impact. Knowing this is important because although within reason we can amortize startup and synchronization costs by selecting an appropriate grain size, if the parallel execution of independent computations has too many indirect interactions, then the capacity for our system to yield good results in practice is severely reduced.

The benchmark program that we will use is a Monte Carlo integrator. We present the program in some detail because it shows some interesting higher order programming techniques that are not generally seen in Prolog. (Although they can be used in Prolog, they are generally held to be too inefficient to be widely useful.) Monte Carlo integration is a numerical integration technique that is used to compute the volume (in any number of dimensions) of a complex shape. If the shape is complex, then analytic techniques are usually intractable, and it can be impossible to write a parametric function that describes the shape to use with other more traditional numerical integration techniques. It is particularly effective for compound shapes that are composed by using union, intersection and difference operators to combine simpler shapes; a technique called

```
/* code for predicate 'p'/3 in mode 0 */
p_3_0:
    incr_sp(7);
    stackvar(7) = succip;
    stackvar(1) = r1;
    stackvar(2) = r2;
    stackvar(6) = sp;
    incr_hp(r4, 8);
    init_sync_term(r4, 2);
    stackvar(5) = r4;
    fork_new_context(p_3_0_i2, p_3_0_i3, 6);
      /* one slot holds return address */

p_3_0_i2:
    r1 = &mercury_data___type_ctor_info_int_0;
    r2 = stackvar(1);
    r3 = stackvar(2);
    call_localret(list__append_3_1, p_3_0_i4);
p_3_0_i4:
    stackvar(3) = r1;
    field(mktag(0), stackvar(6), -3) = stackvar(3);
    join_and_terminate(stackvar(5));

p_3_0_i3:
    r1 = &mercury_data___type_ctor_info_int_0;
    r2 = stackvar(2);
    r3 = stackvar(1);
    call_localret(list__append_3_1, p_3_0_i7);
p_3_0_i7:
    stackvar(4) = r1;
    field(mktag(0), stackvar(6), -4) = stackvar(4);
    join_and_continue(stackvar(5), p_3_0_i8);

p_3_0_i8:
    r1 = &mercury_data___type_ctor_info_int_0;
    r2 = stackvar(3);
    r3 = stackvar(4);
    succip = stackvar(7);
    decr_sp_pop_msg(7);
    tailcall(list__append_3_1);
```

Figure 3.2: The code generated for the parallel conjunction in 3.5. To improve the clarity of the code, we have simplified the automatically generated code slightly by removing some prefixes from the identifiers, and deleting some wrapper macros.

```
:- type shape == pred(float, float, float).
:- inst shape = (pred(in, in, in) is semidet).
....

:- pred monte(box, shape, int, volume).
:- mode monte(in, in(shape), in, out) is det.

monte(Box, Shape, NumPoints, Volume) :-
    rnd__init(17, Rnd0),
    monte2(Box, Shape, NumPoints, Rnd0, 0, Hits),
    Prop = float(Hits) / float(NumPoints),
    BoxVolume = (Box^xmax - Box^xmin) *
                (Box^ymax - Box^ymin) *
                (Box^zmax - Box^zmin),
    Volume = BoxVolume * Prop.

:- pred monte2(box, shape, int, rnd, int, int).
:- mode monte2(in, in(shape), in, in, in, out) is det.

monte2(Box, Shape, N, Rnd0, Hits0, Hits) :-
    ( N > 0 ->
        rnd__frange(Box^xmin, Box^xmax, X, Rnd0, Rnd1),
        rnd__frange(Box^ymin, Box^ymax, Y, Rnd1, Rnd2),
        rnd__frange(Box^zmin, Box^zmax, Z, Rnd2, Rnd3),
        ( call(Shape, X, Y, Z) ->
            Hits1 = Hits0 + 1
        ;
            Hits1 = Hits0
        ),
        monte2(Box, Shape, N - 1, Rnd3, Hits1, Hits)
    ;
        Hits = Hits0
    ).
```

Figure 3.3: Mercury code for a Monte Carlo integrator

constructive solid geometry (CSG). For Monte Carlo integration, all that is needed is a predicate that tests whether a given point lies inside or outside the shape. We generate randomly distributed points within a bounding box and count the number that lie inside and the number that lie outside. Given that we can trivially determine the volume of the bounding box, we can compute the volume of the shape as a proportion of the volume of the bounding box from the ratio of the number of points that lie inside the shape and the number that that lie outside. The precision of the result depends on the number of points that are tested. The accuracy of the result depends on having an even distribution of points throughout the bounding box. The points must be randomly distributed; with any systematic way of traversing the bounding box, it would be possible for the traversal system to interact adversely with the shape and produce arbitrarily bad results.

To ensure an even distribution of points we use a sophisticated random number generator (RNG). The most common library RNG implementations (such as the one in the Mercury standard library) use linear-congruential generators (see e.g. [82]), which although very fast, usually have sequential correlations which mean that the random number sequence is not evenly distributed. Our RNG is a Mercury translation of George Marsaglia's "Mother of all Random Number Generators" [61] which is itself based on work on Mersenne Primes by Matsumoto and Nishimura [62]. It

Figure 3.4: The alchemists' symbol for Mercury (Unicode U+263F)

is a very robust generator and has a cycle length of about 2^{250}, which is to say more than enough for our purposes.

Our Monte Carlo integrator is very simple. It takes a closure for testing whether a given point is inside or outside the shape to be integrated, a bounding box on the shape and an integer specifying the number of points to test, and it returns the volume of the shape. The source code of our Monte Carlo integrator is shown in figure 3.3.

The shape we will use is the solid shape shown in figure 3.4 which is a stylization of the alchemists' symbol for Mercury. The predicate for testing "insideness" uses a library for constructive solid geometry that uses union, intersection, difference, transformation operators for translating, scaling and rotating, and a collection of primitive shapes (semi-spaces, spheres, cones, cubes, cylinders, and toruses). For each of the primitive shapes, there is a predicate that takes three coordinates (corresponding to the x, y and z axes) describing a point and succeeds if the point is inside the shape. Figure 3.5 shows the code for the primitive shapes that we use. The tests are simply derived from the mathematical definitions of the shapes. They are all "unit" shapes which can then be transformed by linear transformations that shift, scale and rotate, to position, size and orient them as desired. Note that the torus primitive has an extra parameter for the minor radius, since that cannot be manipulated (relative to the major radius) by the normal linear transformations. These transformations are provided as higher order combinator-like predicates that take a *shape* (the closure type shown at the top of figure 3.3) and the three coordinates and succeed if the point is inside the transformed shape. So, for example to make a sphere of radius 0.4, centred on the point $(0.5, 0.5, 0.5)$ we need to shrink the unit sphere to 0.4 its original size, and translate it up, across and back by 0.5. This is achieved with the closure expression `translate(scale(sphere, 0.4, 0.4, 0.4), 0.5, 0.5, 0.5)`. This is now an expression of type *shape*.

The code for the transformations that we use is shown in figure 3.6. Note that they actually apply the inverse transformation to the point. This is because we need to achieve the effect of mapping the point from the "world" coordinate system into the "object" coordinate system. More information on the details of the linear transformations can be found in resources such as [36].

Not only do we want to be able to transform shapes, but we also need to be able to combine them. We combine them using the same combinator approach that we used for transformations. The three combinators we provide are a union combinator which takes two shapes and succeeds if the given point is inside either shape; an intersection operator which takes two shapes and succeeds if the given point is inside both shapes; and a difference operator which takes two shapes and succeeds if the given point is inside the first but not inside the second. The code that does this is shown in figure 3.7.

The Mercury shape is made up of the following components:

- A torus for the "head".
- half a torus (constructed as the difference between a torus and a semi-space —

```
:- pred sphere(float, float, float).
:- mode sphere(in, in, in) is semidet.
sphere(X, Y, Z) :-
    X * X + Y * Y + Z * Z =< 1.0.

:- pred torus(float, float, float, float).
:- mode torus(in, in, in, in) is semidet.
torus(R, X, Y, Z) :-
    sqr(sqrt(sqr(X) + sqr(Z)) - 1.0) + sqr(Y) =< sqr(R).

:- pred cylinder(float, float, float).
:- mode cylinder(in, in, in) is semidet.
cylinder(X, Y, Z) :-
    sqr(X) + sqr(Y) =< 1.0,
    0.0 =< Z, Z =< 1.0.

:- pred planeZ(float, float, float).
:- mode planeZ(in, in, in) is semidet.
plane(_, _, Z) :-
    Z >= 0.0.
```

Figure 3.5: The constructive solid geometry primitives used to construct the Mercury shape.

`difference(torus(0.25), planeZ)`) with spheres on the ends for the "winged-helmet"

- a pair of cylinders (one rotated about the y axis 90° so that it is horizontal) capped with spheres for the body and arms.

Figure 3.9 shows the points that lie within the shape after sampling 1000, 10,000 and 100,000 points. It should be obvious that very very many points need to be tested to get a precise answer. This is usually the case with Monte Carlo integration, and is why a parallel integrator is appealing.

We have a top-level driver program that simply calls the `monte/4` predicate to compute the volume of the Mercury symbol by sampling 1,000,000 points and writes out the answer. Figure 3.8 shows the volume computed for the shape after different numbers of points. Our reference parallel machine[4] takes 224 seconds to do a Monte Carlo integration of the Mercury shape with 1,000,000 points using the normal sequential execution model. This figure was arrived at by doing 8 runs, discarding the highest and lowest times and averaging the remaining times (as are all the running times we present).

Assuming that the "insideness" predicate is externally supplied, then the only code that we can optimize is the code shown earlier in figure 3.3 (the random number generator has already been optimized as much as possible). Given that the field accesses to `xmin`, `ymin`, etc, are inlined, as is the arithmetic, there is not much that can be optimized in the integration code itself, it only remains to find the best way to parallelize the code. In this case, the program is simple enough that it isn't really necessary to profile the program to discover appropriate places to introduce parallelism. A pretty clear way is to divide up the points (denoted by `NumPoints` in the program) to sample across the N processors, count the number of hits for each group in parallel, and then sum them at the end. Because the testing of each point is completely independent, we can use the independent AND-parallel conjunction that we introduced in section 3.5. Of course, testing each point in parallel would be a very bad idea, since the kinds of SMP machine we are directing our implementation at have a few processors, maybe as many as a dozen, but certainly not hundreds

[4] This machine is a Sun SPARCserver 1000 with three 50 MHz TI SuperSPARC processors and 256 megabytes of memory running SunOS 5.3 (Solaris 2.3). Each processor is rated at 60.3 SPECint92, and has a 4-way associative 16 Kb I-cache and a 5-way associative 20 Kb D-cache, backed by 1 Mb of unified secondary cache.

```
:- pred translate(shape, float, float, float, float, float, float).
:- mode translate(in(shape), in, in, in, in, in, in) is semidet.
translate(Shape, Dx, Dy, Dz, X, Y, Z) :-
    call(Shape, X - Dx, Y - Dy, Z - Dz).

:- pred scale(shape, float, float, float, float, float, float).
:- mode scale(in(shape), in, in, in, in, in, in) is semidet.
scale(Shape, Sx, Sy, Sz, X, Y, Z) :-
    call(Shape, X / Sx, Y / Sy, Z / Sz).

:- pred rotateY(shape, float, float, float, float).
:- mode rotateY(in(shape), in, in, in, in) is semidet.
rotateY(Shape, Theta, X0, Y0, Z0) :-
    Sin = sin(-Theta),
    Cos = cos(-Theta),
    X = Cos * X0 + Sin * Z0,
    Y = Y0,
    Z = -Sin * X0 + Cos * Z0,
    call(Shape, X, Y, Z).
```

Figure 3.6: The constructive solid geometry combinators for translating, scaling and rotating used for constructing the Mercury shape.

of thousands. In any case, it will be much simpler just to share the points out equally between the processors in the first place. Figures 3.10 and 3.11 show a modified version of our program. The predicate `monte2/6` remains unchanged — we simply invoke a number of instances of it in parallel. For maximum flexibility, rather than creating a four-way parallel conjunction, we use a loop that can split the points between any number of tasks (which will hopefully be mapped on to processors by the operating system).

Running this parallelized version of the program, we do see speedups as we increase the number of processors. Figure 3.6 shows the runtime of the program with 1, 2 and 3 processors and the relative speed, with the garbage collector enabled and disabled. When the garbage collector was disabled, we grouped computations with `solutions/2` to get some limited collection using heap reclamation on failure.

On our reference platform (a SPARC multiprocessor), the program runs about 10% faster when compiled for parallel execution rather than sequential execution. This is because all accesses to components of the virtual machine (i.e. virtual registers which are not mapped onto real registers) are accessed indirectly through the register that points to the the engine structure, rather than directly via an address constant. On the SPARC architecture, this typically means that a virtual register access can be executed in 1 instruction rather than 3 the first time they are used in each basic block (depicted in figure 3.6). For subsequent accesses, the address of the Mercury engine is usually cached in a C compiler controlled register, and so the sequential code needs only 1 instruction for the load (or store).

Because the benchmark program is dominated by large independent parallel computations, we would expect the speed to scale pretty much directly with the number of processors. The timing results show that this is clearly not so, however. The reasons however are quite different for the two ways in which the program was executed.

In the version of the program that uses garbage collection, about half of the time is being spent garbage collecting with using a single thread, and as the number of threads goes up, a higher proportion of the time is spent in the garbage collector. This is mainly due to the fact that when the garbage collector decides to do a collection, it suspends all (POSIX) threads, so that it can scan the stacks for roots. Unfortunately this degrades performance significantly as can

```
:- pred union(shape, shape, float, float, float).
:- mode union(in(shape), in(shape), in, in, in) is semidet.
union(A, B, X, Y, Z) :-
    (
        call(A, X, Y, Z)
    ;
        call(B, X, Y, Z)
    ).

:- pred intersection(shape, shape, float, float, float).
:- mode intersection(in(shape), in(shape),
                in, in, in) is semidet.
intersection(A, B, X, Y, Z) :-
    call(A, X, Y, Z),
    call(B, X, Y, Z).

:- pred difference(shape, shape, float, float, float).
:- mode difference(in(shape), in(shape), in, in, in) is semidet.
difference(A, B, X, Y, Z) :-
    call(A, X, Y, Z),
    not call(B, X, Y, Z).
```

Figure 3.7: The constructive solid geometry combinators for combining shapes.

#points	estimated volume
10	0.000000
100	2.800000
1,000	2.660000
10,000	2.702000
100,000	2.555000
1,000,000	2.654120

Figure 3.8: The computed volume of the Mercury figure

partially be seen by the difference in performance of the version of the program which does not use garbage collection. The garbage itself is created by the use of floating point numbers. Mercury uses C's `double` to represent such numbers and since these do not fit in a single machine word on our target machine (a 32 bit SPARC box), they are represented using a two word heap allocated cell.

Without garbage collection, we use calls to the builtin predicate `solutions/2` (see 2.10) which uses heap truncation after generating each solution. We split the computation of `monte2` into blocks, each inside a call to `solutions/2` so that any garbage generated during that block gets reclaimed. Because it is possible for the goal being evaluated inside a call to `solutions/2` to suspend (although it never does in this program), its implementation contains locking to prevent multiple threads from colliding inside critical regions. This leads to the observed behaviour, that with 3 processors, the program runs twice, not three times as fast as the 1 processor version. Note though, that this is still running more than twice as fast as the garbage collected version.

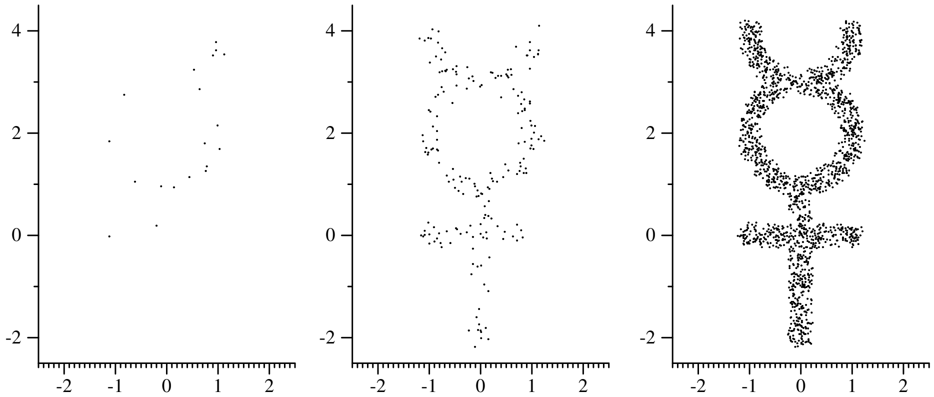

Figure 3.9: The points lying within the volume of the Mercury figure after 1,000, 10,000, and 100,000 samples

```
:- pred monte(box, shape, int, int, volume).
:- mode monte(in, in(shape), in, in, out) is det.

monte(Box, Shape, NumProcs, NumPoints, Volume) :-
    rnd__init(17, Rnd0),
    parMonte(NumProcs, NumProcs, Box, Shape,
            NumPoints, Rnd0, HitsList),
        % add up the ints in HitsList
    sumHits(HitsList, 0, Hits),
    Prop = float(Hits) / float(NumPoints),
    BoxVolume = (Box^xmax - Box^xmin) *
                (Box^ymax - Box^ymin) *
                (Box^zmax - Box^zmin),
    Volume = BoxVolume * Prop.
```

Figure 3.10: A modified version of `monte/4` for harnessing parallel execution.

```
:- pred parMonte(int, int, box, shape, int, rnd, list(int)).
:- mode parMonte(in, in, in, in(shape), in, in, out) is det.

parMonte(N, Np, Box, Shape, Prec, Rnd0, HitsList) :-
    ( N > 1 ->
        rnd__irange(99, 9999999, Seed, Rnd0, Rnd1),
        rnd__init(Seed, NewRnd),
        (
            monte2(Box, Shape, Prec // Np, NewRnd, 0, Hits)
        &
            parMonte(N - 1, Np, Box, Shape, Prec, Rnd1, HitsList0)
        ),
        HitsList = [Hits|HitsList0]
    ; N = 1 ->
            % make sure we catch any extras
        Rest = Prec // Np + (Prec mod Np),
        monte2(Box, Shape, Rest, Rnd0, 0, Hits),
        HitsList = [Hits]
    ;
        error("parMonte: N < 1")
    ).
```

Figure 3.11: Mercury code for distributing the work of the Monte Carlo integration across multiple tasks.

# CPUs	GC enabled		GC disabled	
# CPUs	Time (seconds)	Relative Speed	Time (seconds)	Relative Speed
1	205	1.0	104	1.0
2	155	1.3	81	1.3
3	121	1.7	51	2.0

Figure 3.12: The runtime and relative speedup of the Monte Carlo integrator utilising 1, 2 and 3 processors

```
/* sequential model: */
sethi   %hi(MR_engine_base), %l7
or      %l7, %lo(MR_engine_base), %o5
ld      [%o5+52], %i1
```

```
/* parallel model: */
ld      [%i0+52], %i1
```

Figure 3.13: The SPARC instructions necessary to load a virtual register in the sequential model (left), and the instructions necessary to load a virtual register in the parallel model (right).

Chapter 4

Explicit Mercury threads

It is often useful to capture the notion of multiple tasks happening concurrently as a paradigm for constructing programs. Server-like programs such as web servers, operating systems, and graphical user interfaces can often benefit from the use of explicit concurrency as a way modeling the desired behaviour of the program. For example, in a web server it may be a strongly simplifying design decision to have a separate thread handle each request that comes in from the network. Where a language does not provide support for such a paradigm, it is necessary to use other, much less convenient ways to structure programs that naturally need to do interleaved computations especially those involving I/O.

In chapter 3 we presented the infrastructure necessary to execute multiple (deterministic) goals in an interleaved way, possibly in parallel. Section 3.5 showed one way in which Mercury programs could make use of that infrastructure to execute independent (non-communicating) goals in parallel. This chapter presents an alternative way in which programs can make use of this infrastructure in a way that *does* allow communication between the interleaved tasks.

We present an interface that allows programs to create concurrent tasks and have them synchronize with one another. For consistency with other nomenclature, we will refer to these concurrent tasks as *threads* (using the term Mercury-thread where ambiguity between the Mercury and OS/C levels exists). Unlike parallel conjunction which is a language construct to support parallel execution, we provide threads support as a user level library.

One of our key goals is to have threads communicate with one another. The most common way that this is achieved in other logic programming environments is through the use of shared variables: a free variable is passed to multiple goals one of which progressively binds it, and inspect the parts of the binding made by other goals. There are three main difficulties with this approach. One is that although the programmer usually has in mind that one goal should be the producer of a particular binding and that another goal should be the consumer of that binding, predicate logic has no way of expressing this concept, and so many languages either rely on non-logical constructs (such as `var/1`), or they don't let the programmer express this concept and therefore are very difficult to execute efficiently. The second difficulty is that even when the language allows the programmer to express such producer-consumer relationships (for instance many modern Prolog variants have *delay* or *when* declarations), if the bindings of shared variables may be retracted due to failure, then it also makes it very difficult to execute programs efficiently. A large group of languages avoid this problem by using "don't care" nondeterminism rather than "don't know" nondeterminism, and commit to the first clause whose head unifies with the call and whose guard succeeds. This causes a lack of completeness (when viewing the program as a formula of first order predicate logic) or even unsoundness when combined with negation. The third difficulty is that shared variables used as communication channels between threads (or goals) become quite burdensome if the programmer wishes to store them in a data-structure. Each time a message is communicated between one goal and another, any data structures containing the shared variable usually need to be updated to contain a new shared variable. (Merely binding the variable representing the old state of the channel to a term containing the variable that represents its state would require all references to the current state of the channel to traverse a long and growing list of terms representing obsolete states, which would lead to unacceptable performance.)

For example consider the following NU-Prolog program fragment. Imagine a web server with session support, in which the predicate `server/2` takes all the parsed HTTP requests, and forwards them on to the appropriate session, or if the session is new, creates a new session object:

```
:- type sessions == mapping(session_id, list(session_msg)).
:- pred server((session_id - session_msg), sessions).
:- server([], _) when ever.
:- server([Msg|Msgs], _) when Msg.

server([Id - Msg|Msgs], Sessions0) :-
    ( search(Id, Sessions0, SessionMsgs0) ->
        SessionMsgs0 = [Msg|SessionMsgs],
        replace(Sessions0, Id, SessionMsgs, Sessions1)
    ;
        % start a new session
        session([Msg|SessionMsgs]),
        insert(Sessions0, Id, SessionMsgs, Sessions1)
    ),
    server(Msgs, Sessions1).
```

The main data structure used by the `server` is a mapping from session ids to free variables. The free variable for a given session is the channel that `server` uses to pass messages to that session; it does this by instantiating the variable to a list containing the first message and a free variable representing all future messages. Even in this simple example, the need to update the free variable denoting the remainder of the list of messages (`SessionMsgs`) adds to the complexity of the code (both in terms of understanding and performance). More importantly, it means that it makes it very difficult to embed communication channels inside abstract types, and it is difficult to make efficient channels that have multiple writers. The way Oz/Mozart addresses this issue is to make communication channels ground objects. There is however a semantic issue here, in that if the object is ground then sending a message effectively requires a side effect.

We finesse these problems by combining aspects of both approaches. Communication between Mercury threads is done by sharing a ground object denoting the communication channel, but using the state of the world object (i.e. the `io__state`) as a place to store the (abstract) value representing the current state of the channel. By using the `io__state`, which is unique and does not allow backtrackable updates, we constrain communication between threads to be monotonic (deterministic), which avoids the need for a high-overhead execution algorithm (i.e. we can use the execution algorithm developed in chapter 3).

4.1 Spawn

Rather than use a language construct to denote a goal that should be executed as a thread, we use a closure. As stated earlier, we use the `io__state` to allow the closure to communicate with the outside world and other threads, so the closure must be given a pair of such arguments to be used. This leads to the following type for goals that are to be executed as threads:

```
:- type thread == pred(io__state, io__state).
```

New threads are spawned using such closures using the `spawn/3` procedure. `Spawn/3` has the signature:

```
:- pred spawn(pred(io__state, io__state), io__state, io__state).
:- mode spawn(pred(di, uo) is cc_multi, di, uo) is cc_multi.
:- mode spawn(pred(di, uo) is det, di, uo) is cc_multi.
```

and here is a simple program that uses it:

```
:- pred main(io__state, io__state).
:- mode main(di, uo) is cc_multi.

main(IO0, IO) :-
    spawn(hello, IO0, IO1),
    world(IO1, IO).

:- pred hello(io__state, io__state).
:- mode hello(di, uo) is det.

hello(IO0, IO) :-
    print("hello\n", IO0, IO).

:- pred world(io__state, io__state).
:- mode world(di, uo) is det.

world(IO0, IO) :-
    print("world\n", IO0, IO).
```

This program spawns a thread which will execute the goal `hello(IOinitial, IOfinal)`, where the `io__state` argument `IO1` contains any necessary information to describe any I/O actions done by `hello/2`. It then executes the goal `world(IO1, IO)`. This program has two possible outputs, assuming `print/3` is atomic (though any single execution of it will lead to only a single output). It will yield either of the following two outputs:

```
hello
world
```

or

```
world
hello
```

Recall that the determinism of a goal `cc_multi` denotes that the goal will have at least one solution, that it may have multiple solutions, and that it can only be called from a first-solution-only context, which means that we never backtrack into the goal to seek a second or subsequent solution. Intuitively, the reason that `spawn/3` has this determinism is that the I/O actions executed by the spawned call can be interleaved with the I/O actions of the rest of the program in more than one way, and different interleavings conceptually lead to different final states of the world.

The first declared mode of `spawn` (in which the determinism of the closure is `cc_multi`) is necessary so that spawned threads can in turn call `spawn/3`. The other mode (in which the closure is `det`) is subsumed by the first, but it is necessary because the Mercury compiler requires exact determinism matches for closures [1].

The implementation of `spawn/3` is very short; it is shown in figure 4.1. `Spawn` is implemented in C using the foreign code interface. It creates a new task structure, and pushes the closure representing the goal to be executed in the new task onto the detstack of the new task. It sets the resumption address in the task structure to point to a piece of hand-written Mercury code that arranges for the closure to be called. We put that Mercury code into an if-then-else with a false condition to prevent it from being executed in the spawning task. When an engine is allocated to execute the spawned task, this code will take the closure from its slot in the detstack (now pointed to by the virtual machine's stack pointer register) and give it to a function that will call that closure. (The call to `save_registers` is required because the code calls the closure through

[1] This is arguably a bug in the Mercury implementation, but in practice is rarely much of an inconvenience.

```
:- pragma c_code(spawn(Goal::(pred(di, uo) is cc_multi),
                       IO0::di, IO::uo),
            [will_not_call_mercury, thread_safe],
"{
    Task *task;
    task = create_task();
    task->resume = &&spawn_call_back_to_mercury_cc_multi;
        /* Store the closure on the top of the stack. */
    *(task->task_sp) = Goal;
    task->next = NULL;
    schedule(task);
    if (FALSE) {
        /* Executed via task->resume */
spawn_call_back_to_mercury_cc_multi:
        save_registers();
        call_back_to_mercury_cc_multi(*((Word *) sp));
        destroy_task(ENGINE(this_task));
        runnext();
    }
}").
```

Figure 4.1: The C implementation of `spawn/3`

```
:- pred call_back_to_mercury(
    pred(io__state, io__state),
    io__state, io__state).
:- mode call_back_to_mercury(
    pred(di, uo) is cc_multi, di, uo) is cc_multi.
:- pragma export(
    call_back_to_mercury(pred(di, uo) is cc_multi, di, uo),
    "call_back_to_mercury_cc_multi").

call_back_to_mercury(Closure) -->
    call(Closure).
```

Figure 4.2: The code for calling the closure for the new thread

the C interface — actually, the C version of the Mercury predicate shown in figure 4.2.) When the (indirect) call to the closure returns, the code will deallocate the task structure and release the engine to work on some other task.

4.2 Communication

Threads have an `io__state` available and therefore can do I/O. Even if we provided no other communication mechanism, this would allow threads to communicate using files, pipes, sockets or any number of other ways. On the other hand, all these mechanisms require system calls and are therefore too expensive. A cheap way is to allow threads to store data in the `io__state`. If they have an agreed way of referring to it, then one thread can place a datum in the `io__state` and another can examine it. In effect, we can provide something like global variables. The I/O library provides a way of storing a single item in the `io__state`, and we can use this item to store a dictionary that associates keys of some sort with data to create a global variable library. This

```
:- pred init_globals(io__state, io__state).
:- mode init_globals(di, uo) is det.

:- typeclass global(Key, Data) where [].

:- pred set_global(Key, Data, io__state, io__state)
    <= global(Key, Data).
:- mode set_global(in, in, di, uo) is det.

:- pred get_global(Key, Data, io__state, io__state)
    <= global(Key, Data).
:- mode get_global(in, out, di, uo) is det.
```

Figure 4.3: The interface of the global variable library

library does not provide any locking mechanisms to ensure the serialization of updates. We will discuss locking and synchronization in section 4.3 The interface and the code of the library are shown in figures 4.3 and 4.4.

For convenience, we can let users of the library use values of any type they like as keys. To help with type safety though, we use typeclasses to constrain key types and value types so that they are linked. This adds a burden on the user of the library, since they have to declare their global variable equivalent as an instance of the appropriate typeclass, but it also provides a mechanism that enables programmers to have confidence that their keys will not collide with others in the system. The user can create a type local to his module and use it to construct a key for a global variable. If this type is made private to the module, then it is not possible for other modules to create values of this type to use as keys. For example, the following code fragment shows how we can do this to create a private global for storing a list of integers.

```
:- type myGlobalName ---> myGlobalName.
:- type myGlobalData ---> data(list(int)).
:- instance global(myGlobalName, myGlobalData) where [].
...

    set_global(myGlobalName, data([1,2,3,4]), IOn, IOm),
...
    get_global(myGlobalName, data(Nums), IOp, IOq),
...
```

Another conceptually important class of objects for programs that wish to share data between multiple threads is *pointers*. At an abstract level, pointers are references to data which we can pass around and then "dereference" to yield the data corresponding to the reference. Unlike global variables (as we have shown them so far), pointers are assigned values dynamically, so we can create an unbounded number of them at runtime. We can use global variables to mimic pointers — we can create a unique key for each individually updateable data item, and then use `get_global` and `set_global` to access the pointed-to data. However, this would require a data structure that mapped these unique keys (simulating pointers) to the current values of their associated data items. However, this map would defeat the garbage collector: the pointed-to data will persist in the `io__state` even when there are no more instances of the key live on the stack or elsewhere in the heap. We can address this problem by using a different abstract data type for the simulated pointers. We end up with a very similar interface, except that it is the implementation of the pointer library, rather than its client, that chooses the name. The idea is that the pointer simulation library does not keep references to simulated pointers. When a new data item is created, the library hands back the only reference to the pointer to its client. When the client wants to perform indirection through the pointer, they give these simulated pointers

```
:- implementation.

:- import_module map, require, std_util.

init_globals(IO0, IO) :-
    map__init(Globals),
    io__set_globals(univ(Globals), IO0, IO).

set_global(Key, Data, IO0, IO) :-
    io__get_globals(Univ0, IO0, IO1),
    ( Univ0 = univ(Map0) ->
        map__set(Map0, univ(Key), univ(Data), Map),
        io__set_globals(univ(Map), IO1, IO)
    ;
        error("set_global: globals not initialized")
    ).

get_global(Key, Data, IO0, IO) :-
    io__get_globals(Univ0, IO0, IO),
    ( Univ = univ(Map0) ->
        ( map__search(Map0, univ(Key), DataUniv) ->
            ( DataUniv = univ(Data0) ->
                Data = Data0
            ;
                error("get_global: data has wrong type")
            )
        ;
            error("get_global: no such global")
        )
    ;
        error("get_global: globals not initialized")
    ).
```

Figure 4.4: The code of the global variable library

```
:- type pointer(T).

:- pred pointer__new(T, pointer(T), io__state, io__state).
:- mode pointer__new(in, out, di, uo) is det.

:- pred pointer__get(pointer(T), T, io__state, io__state).
:- mode pointer__get(in, out, di, uo) is det.

:- pred pointer__set(pointer(T), T, io__state, io__state).
:- mode pointer__set(in, in, di, uo) is det.
```

Figure 4.5: The interface for the simulated pointer library

back to the get or set predicate, but they don't retain copies of them either. Therefore when the last copy of the simulated pointer in the client program becomes unreachable, a conservative collector such as Boehm's [10] should be able to recover its storage.

However, we can't implement the simulated pointers as direct pointers either, since a set operation through one copy must be visible on the next operation through another copy. We therefore implement simulated pointers as a pointer to a pointer; the first pointer encodes the identity of the simulated pointer, the second its current value. The predicate that creates a new simulated pointer allocates a cell for the second pointer on the heap, and returns the address of this cell (i.e. the first pointer) as the representation of the simulated pointer. Figures 4.5 and 4.6 give the interface and the code of our simulated pointer library. (This library is part of the Mercury distribution; it is in extras/concurrency/global.m.)

4.3 Synchronization

Where data structures may be modified by more than one thread it is usually necessary to use synchronization primitives to ensure that threads don't interfere with each other. There are numerous techniques for implementing synchronization, including locks, semaphores, and monitors. We use semaphores as our synchronization primitive; figures 4.7 shows the interface of the semaphore library. Semaphores are of course an abstract data type. Their implementation contains the current value of the semaphore and a list of the tasks suspended on the semaphore. The *signal* operation increments the value of the counter, and, when changing the value from 0 to 1, wakes one of the waiting tasks (if there are any). The *wait* operation tests the semaphore's value to see if it is greater than 0. If it is, its decrements the value and proceeds. If the value is 0, then the wait operation (atomically) puts the current task on the list of suspended tasks and then suspends the task (i.e. switches the engine currently executing it to execute some other task).

The program shown in figure 4.8 demonstrates the use of semaphores. We assume the presence of an abstract unbounded buffer to which we can atomically write bytes and from which we can atomically read bytes. (We will show how it is implemented below.) This program has two producer tasks that each read bytes from one of two files and write those bytes to the unbounded buffer. The third, consumer task removes bytes from the buffer and writes them out. Since the two producer tasks are running asynchronously, the sequence of bytes put into the unbounded buffer (and therefore the sequence of bytes written out by the consumer task) may be any interleaving of the bytes in the two files. Note that this program can deadlock. The consumer task relies on a sentinel value (`-1`) to terminate. If the producer tasks terminate without putting a sentinel into the buffer, then the consumer task will eventually block forever.

Semaphores can also be used to implement mutual-exclusion locks (*mutexes*). The simplest implementation is as a semaphore whose value is initialized to one, with the *lock* operation doing a wait, and the *unlock* operation doing a signal on the semaphore. However, this implementation may not behave correctly if for some reason the program does an unlock operation on a mutex

```
:- type pointer(T) ---> pointer(c_pointer).

:- pragma c_code(pointer__new(Data::in, Ptr::out, IO0::di, IO::uo),
            [thread_safe, will_not_call_mercury],
"{
    Word *tmp;
    incr_hp((Word) tmp, 1);
    *tmp = Data;
    Ptr = (Word) tmp;
    IO = IO0;
}").

:- pragma c_code(pointer__get(Ptr::in, Data::out, IO0::di, IO::uo),
            [thread_safe, will_not_call_mercury],
"{
    Data = * (Word *) Ptr;
    IO = IO0;
}").

:- pragma c_code(pointer__set(Ptr::in, Thing::in, IO0::di, IO::uo),
            [thread_safe, will_not_call_mercury],
"{
    * ((Word *) Ptr) = Data;
    IO = IO0;
}").
```

Figure 4.6: The code of the simulated pointer library

```
:- type sem.

:- pred new_sem(int, sem, io__state, io__state).
:- mode new_sem(in, out, di, uo) is det.

:- pred sem_wait(sem, io__state, io__state).
:- mode sem_wait(in, di, uo) is det.

:- pred sem_signal(sem, io__state, io__state).
:- mode sem_signal(in, di, uo) is det.
```

Figure 4.7: The interface of the semaphore library

```
main(IO0, IO) :-
    open_file("file1", File1, IO0, IO1),
    open_file("file2", File2, IO1, IO2),
    make_buffer(Buffer, IO2, IO3),
    new_sem(0, Sem, IO3, IO4),
    spawn(producer(File1, Buffer, Sem), IO4, IO5),
    spawn(producer(File2, Buffer, Sem), IO5, IO6),
    spawn(consumer(Buffer, Sem), IO6, IO).

:- pred producer(input_stream, buffer, sem,
                 io__state, io__state).
:- mode producer(in, in, di, uo) is det.

producer(Input, Buffer, Sem, IO0, IO) :-
    read_byte(Input, Res, IO0, IO1),
    ( Res = ok(Byte) ->
        put_item(Buffer, Byte, IO1, IO2),
        sem_signal(Sem, IO2, IO3),
        producer(Input, Buffer, Sem, IO3, IO)
    ;
        put_item(Buffer, -1, IO1, IO)
    ).

:- pred consumer(buffer, sem, io__state, io__state).
:- mode consumer(in, in, di, uo) is det.

consumer(Buffer, Sem, IO0, IO) :-
    sem_wait(Sem, IO0, IO1),
    get_item(Buffer, Byte, IO1, IO2),
    ( Byte \= -1 ->
        write_byte(Byte, IO2, IO3),
        consumer(Buffer, Sem, IO3, IO)
    ;
        IO = IO0
    ).
```

Figure 4.8: A program showing how semaphores are used

```
:- type mutex.

:- pred new_mutex(mutex, io__state, io__state).
:- mode new_mutex(out, di, uo) is det.

:- pred lock_mutex(mutex, io__state, io__state).
:- mode lock_mutex(in, di, uo) is det.

:- pred unlock_mutex(mutex, io__state, io__state).
:- mode unlock_mutex(in, di, uo) is det.
```

Figure 4.9: The interface of the mutex library

that is not in the locked state. A more robust implementation uses a semaphore in this way internally, and uses a second semaphore to protect the internal semaphore against multiple unlock operations. The Mercury code used to implement this follows:

This mutex module works by storing the information about the mutex as a global (of type `mutex_internals`), and using a semaphore (denoted in the code as `OuterSem`) to protect updates to it. The mutex itself consists of a flag which represents whether it is open or closed, and a semaphore (denoted in the code as `InnerSem`) which is used to suspend threads that try to obtain the mutex when it is in a locked state. Notice that if this circumstance occurs, the thread trying to obtain the mutex releases the lock protecting the mutex internals before suspending on the inner semaphore. These two operations must be done in this order to prevent deadlock.

This implementation of mutexes has the desirable property that a lock operation on an already locked mutex will cause the task attempting the lock to suspend, yielding the CPU to another task. Mutexes implemented as spinlocks would not be suitable as the only implementation of mutexes in a concurrency library, because some applications may hold mutexes for long periods. Of course, spinlocks are useful for protecting critical sections that are guaranteed to execute quickly, which are often found in system code (e.g. in our implementation of semaphores — see below).

With our mutex module, we can now create the buffer module we used earlier. A buffer will be a channel that we can put bytes into, and take bytes out of in a first-in-first-out order. We represent it using a structure which contains a mutex and a global which refers to a queue object[2]. Semaphores are implemented using the C interface, as shown in figure 4.13. The C representation of a semaphore is a structure containing a counter, a linked list of task structures that are blocked on the semaphore and a low-level lock. (These low level locks can be implemented either as spinlocks or, more portably, as POSIX mutexes).

The implementation of `sem_wait` examines the counter, and if it is positive, decrements the counter and returns. If the counter is zero, then the engine saves the task and links it onto the list of tasks that are suspended on the semaphore, and does a context switch, i.e. suspends this task and starts executing the next runnable task (if any). The implementation of `sem_signal` increments the counter, and if there were any tasks suspended on the semaphore, it wakes one of them by making it runnable. It also saves the state of the current task and puts it back on the global run queue, and leaves it up to the scheduler in `runnext()` to select which task this engine will be allocated to. (Of course, the scheduler may decide to allocate this engine to a third task, or both tasks may continue to run in different engines.)

All the accesses and updates of the semaphore structures in both `sem_wait` and `sem_signal` are protected by a critical section implemented by LOCK/UNLOCK operations on the low-level mutex of the semaphore. Accesses and updates to the engine structure do not need to be so protected, since only the task currently running on the engine can access and/or modify the engine structure. Accesses and updates to the global run queue are protected by mutexes, but this is hidden inside

[2]A queue abstract data type is provided by the Mercury library. It is implemented using a pair of lists as described in [76].

```
:- type mutex ---> mutex(sem, mutex_internals_ptr).
:- type mutex_internals_ptr == pointer(mutex_internals).
:- type mutex_internals ---> mutex(open_closed, sem).
:- type open_closed --> open ; closed.
:- instance global(mutex_internals_ptr, mutex_internals)
        where [].

new_mutex(Mutex, IO0, IO) :-
    new_sem(1, OuterSem, IO0, IO1),
    new_sem(0, InnerSem, IO1, IO2),
    MutexInternals = mutex(open, InnerSem),
    new(MutexInternals, MutexInternalsPtr, IO2, IO),
    Mutex = mutex(OuterSem, MutexInternalsPtr).

lock_mutex(Mutex, IO0, IO) :-
    Mutex = mutex(OuterSem, MutexInternalsPtr),
    sem_wait(OuterSem, IO0, IO1),
    get(MutexInternalsPtr, MutexInternals0),
    MutexInternals0 = mutex(MutexState, InnerSem),
    (
        MutexState = open,
        MutexInternals = mutex(closed, InnerSem),
        set(MutexInternalsPtr, MutexInternals, IO1, IO2),
        sem_signal(OuterSem, IO2, IO)
    ;
        MutexState = closed,
        sem_signal(OuterSem, IO1, IO2),
        sem_wait(InnerSem, IO2, IO3),
        lock_mutex(Mutex, IO3, IO)
    ).

unlock_mutex(Mutex, IO0, IO) :-
    Mutex = mutex(OuterSem, MutexInternalsPtr),
    sem_wait(OuterSem, IO0, IO1),
    get(MutexInternalsPtr, MutexInternals0),
    MutexInternals0 = mutex(MutexState, InnerSem),
    (
        MutexState = open,
        sem_signal(OuterSem, IO1, IO)
    ;
        MutexState = closed,
        sem_signal(InnerSem, IO1, IO2),
        MutexInternals = mutex(open, InnerSem),
        set(MutexInternalsPtr, MutexInternals, IO2, IO3),
        sem_signal(OuterSem, IO3, IO)
    ).
```

Figure 4.10: The code of the mutex library

```
:- type buffer.

:- pred make_buffer(buffer, io__state, io__state).
:- mode make_buffer(out, di, uo) is det.

:- pred put_item(buffer, byte, io__state, io__state).
:- mode put_item(in, in, di, uo) is det.

:- pred get_item(buffer, byte, io__state, io__state).
:- mode get_item(in, out, di, uo) is det.
```

Figure 4.11: The interface of the unbounded buffer library

```
:- type buffer(T) ---> buffer(mutex, sem, buffer_internals_pointer(T)).
:- type buffer_internals_pointer(T) == pointer(buffer_internals(T)).
:- type buffer_internals(T) ---> buffer(queue(T)).

make_buffer(Buffer, IO0, IO) :-
    Buffer = buffer(Mutex, Sem, BufferInternalsPtr),
    new_mutex(Mutex, IO0, IO1),
    new_sem(Sem, IO1, IO2),
    queue__init(Queue),
    new(buffer(Queue), BufferInternalsPtr, IO2, IO).

put_item(Buffer, Item, IO0, IO) :-
    Buffer = buffer(Mutex, Sem, BufferInternalsPtr),
    lock_mutex(Mutex, IO0, IO1),
    get(BufferInternalsPtr, buffer(Queue0), IO1, IO2),
    queue__put(Queue0, Item, Queue),
    set(BufferInternalsPtr, buffer(Queue), IO2, IO3),
    unlock_mutex(Mutex, IO3, IO4),
    sem_signal(Sem, IO4, IO).

get_item(Buffer, Item, IO0, IO) :-
    Buffer = buffer(Mutex, Sem, BufferInternalsPtr),
    sem_wait(Sem, IO0, IO1),
    lock_mutex(Mutex, IO1, IO2),
    get(BufferInternalsPtr, buffer(Queue0), IO2, IO3),
    ( queue__get(Queue0, Item, Queue1) ->
        Queue = Queue1
    ;
        error("get_item: queue empty")
    ),
    set(BufferInternalsPtr, buffer(Queue), IO3, IO4),
    unlock_mutex(Mutex, IO4, IO).
```

Figure 4.12: The code of the unbounded buffer library

```
:- pragma c_code(sem_wait(Semaphore::in, IO0::di, IO::uo),
            [will_not_call_mercury, thread_safe], "{
    Semaphore    *sem;
    sem = (Semaphore *) Semaphore;
    LOCK(&sem->lock);
    if (sem->count > 0) {
        sem->count--;
        UNLOCK(&sem->lock);
    } else {
        save_task(ENGINE(this_task));
        ENGINE(this_task)->resume = &&wait_skip_to_the_end;
        ENGINE(this_task)->next = sem->suspended;
        sem->suspended = ENGINE(this_task);
        UNLOCK(&sem->lock);
        runnext();
    }
wait_skip_to_the_end:
    IO = IO0;
}").

:- pragma c_code(sem_signal(Semaphore::in, IO0::di, IO::uo),
            [will_not_call_mercury, thread_safe], "{
    Semaphore *sem;
    Task      *task;
    sem = (Semaphore *) Semaphore;
    LOCK(&sem->lock);
    if (sem->count >= 0 && sem->suspended != NULL) {
        task = sem->suspended;
        sem->suspended = task->next;
        UNLOCK(&sem->lock);
        schedule(task);
    } else {
        sem->count++;
        UNLOCK(&sem->lock);
    }

    save_task(ENGINE(this_task));
    ENGINE(this_task)->resume = &&signal_skip_to_the_end;
    schedule(ENGINE(this_task));
    runnext();
signal_skip_to_the_end:
    IO = IO0;
}").
```

Figure 4.13: The C code for implementing semaphores.

the implementations of the `schedule()` and `runnext()` functions.

4.4 A declarative understanding of explicit threads

We have presented a collection of predicates which allow programmers to write programs that have a very different kind of operational behaviour to programs that do not use these predicates. Several of the predicates are implemented in C, so we cannot interpret them directly as statements in predicate logic, and therefore must provide some declarative interpretation for the operational behaviour which they lead to. We will do this by showing how we can implement them in pure Prolog, and by showing a Prolog meta-interpreter that executes threaded programs.

The starting point which we use for constructing this declarative view is to re-examine our interpretation of the `io__state`. As formulated by Henderson in [45] (a paper based on ideas from [110]), the `io__state` is a single monolithic entity which represents the state of the external world. By using the mode system to ensure that it is unique with respect to both forward execution and backtracking, it provides a reasonable model of the external world. In this model, I/O operations transform the world from one state into another, the transformation consisting not just of the I/O action (such as writing a string) that the predicate asks for (if any), but also any other changes in the state of the world caused by external factors since the last I/O action by this program.

The original model [45] talked only about sequential logic programs. We can update this model to account for programs having multiple tasks by including, among the external factors that may affect the final I/O state returned by an I/O action by a given task, the effects of actions executed by all the other tasks in the program. However, this updated model makes it hard to express interactions between the actions of the tasks, and the constraints on these interactions, and this in turn makes it hard to reason about those interactions. We therefore use another model based on streams, which was first used in early work on parallel declarative languages [20, 46, 91].

In this model, all changes to the state of the operating system result from I/O action requests. Some I/O action requests are the work of this program; others are the work of other programs; yet others are the work of physical devices (such as terminals and network interfaces) whose action requests can be modeled as if they were generated by a program. Every program (and entity simulated by a program) generates a stream of action requests that may affect the state of the system. The system non-deterministically merges these streams into a single stream, and applies the changes they call for (if any) to the current state of the operating system.

I/O action requests typically also ask for a response. The response can be returned from the operating system through a separate network of streams that routes responses back to individual programs [46], or by including in each request a free variable that the OS binds to a term representing the response to the request [20, 91].

To express this way of modeling I/O in Mercury, one would need a type declaration like this:

```
:- type io__state == list(io__action).
:- type io__action
        --->    read_int(io__result(int))
        ;       write_int(int)
        ;       create_file(string, io__result(file_descriptor))
        ;       delete_file(string, io__result)
        ....
```

The type `io__action` would have an alternative for every system call supported by the operating system. (For simplicity of exposition, we consider `read_int` and `write_int` to be system calls.) Behind its abstract interface, a predicate like `io__write_int/3` would be implemented with code that adds a `write_int/1` message to the list of I/O actions requests being sent to the OS.

One way of interpreting the pair of arguments would then be that the final `io__state` of `main/2` is the list that results from adding messages for the I/O actions of the program to the list that was given in the first argument. Actually, in this interpretation, there is no particular reason why `main/2` should have two arguments at all — it could simply return a list of interactions. The operating system, as the caller of `main`, could then apply those actions. For this to behave in the way we observe, which is that programs do I/O progressively, not all at once at the end, we need to take a step back and treat the program as pure Prolog and assume that we have a selection rule that will allow us to evaluate a literal from the OS which “writes” the integer, then go back to evaluating literals from our program. With this interpretation of the program, we can also explain the behaviour of I/O actions that return data. A predicate like `io__read_int/3` will also add a message to the list, but the message will not be a ground term; the result part of the term will be left as a free variable by the program, and will be instantiated to a ground term by the OS.

A more powerful way of using the pair of `io__state` arguments is to use them as a difference pair representing a difference list [60, 100]. In this interpretation, the first argument represents the complete list of I/O actions performed by `main/2` and its continuation, and the second argument represents the list of I/O actions performed by the continuation. A predicate with two `io__state` arguments indicates what I/O requests it wishes to make by adding its requests at the front of the list of the continuation's requests, so that its requests will be processed by the OS before the continuation's. Of course, the continuation hasn't started executing yet, and so it hasn't started making any requests; its list of requests will therefore be represented by a free variable. While (the current version of) the Mercury mode system does not allow free variables to be used like this, pure Prolog systems do.

In this interpretation, one can model `io__write_int/3` with pure Prolog code like this:

```
:- pred io__write_int(int, io__state, io__state).
:- mode io__write_int(in, di, uo) is det.
io__write_int(Num, [write_int(Num)|Rest], Rest).
```

The mode declaration here appears to be inconsistent with the code, but isn't really, because the clause for `io__write_int` is in pure Prolog, not Mercury, and the mode declaration therefore doesn't apply to it. The mode declaration merely describes constraints on how this I/O primitive can be used by its callers. In the state of the world model, these constraints ensure that there is only one state of the world at a time; in the stream of I/O requests model, the same constraints ensure that every difference list is appended to exactly once. This is precisely the condition required for the difference list representation of the list of I/O requests to be a correct reflection of the simple list representation [60].

The difference list representation of the stream of requests model allows us to describe concurrency primitives such as spawn and synchronization operations in a natural manner. For example, we can model spawn as if it were in implemented by the following pure Prolog clause:

```
spawn(Goal, AllRequests, ContinuationRequests) :-
    new_thread(call(Goal, SpawnedGoalRequests, [])),
    merge(SpawnedGoalRequests, ContinuationRequests, AllRequests).
```

Basically, the effect of `spawn` is to (nondeterministically) merge the requests that will be made by the spawned goal and the requests that will be made by the computation that called `spawn` after its call to `spawn`. The predicate `new_thread` will be defined in our meta-interpreter, and will execute the goal as an independent thread. We pass the empty list as the second I/O state argument of the spawned goal in order to be able to capture the spawned goal's requests as a simple (non-difference) list. Conceptually, this corresponds to the fact that the spawned goal has no continuation; its completion doesn't cause any other code to start executing and generating I/O requests.

The nondeterministic merge predicate `merge/3` is implemented in the usual way as shown in figure 4.14. The NU-Prolog style when declaration says that merge can be called when either of the first two arguments is bound to a non-variable term. If the first argument is bound to a cons but the second isn't bound, we want merge to apply the second clause; if the second argument is bound to a cons but the first isn't bound, we want merge to apply the fourth clause. However, regardless of which clause is applied first and which interleaving of the two input lists is generated first, merge will on backtracking produce all the possible interleavings of those two lists. On the other

```
:- pred merge(list(T), list(T), list(T)).
:- merge(As, Bs, _) when As or Bs.
merge([], Bs, Bs).
merge([A|As], Bs, [A|Cs]) :-
    merge(As, Bs, Cs).
merge(As, [], As).
merge(As, [B|Bs], [B|Cs]) :-
    merge(As, Bs, Cs).
```

Figure 4.14: The Mercury/NU-Prolog code for `merge/3`

hand, the Mercury determinism system ensures that I/O actions take place only in deterministic contexts and never have to be backtracked over.

The mechanism that the Mercury determinism system uses to ensure this is based on the notion of a first-solution-only context. The `spawn` predicate has determinism `cc_multi`, which denotes that while it may have more than one solution (it may generate more than one interleaving of the requests of the spawned goal and the spawning computation), we only want the first one to be produced; the computation will never backtrack into `spawn` to produce any more solutions. Since `spawn` has determinism `cc_multi`, its body is a first-solution-only context, and since the body of `spawn` cannot fail after returning from the call to `merge`, the call to `merge` is also in a first-solution-only context. This is why a hypothetical implementation of this particular use of the pure Prolog merge predicate would be justified in never creating a choice point.

On the other hand, the caller of `spawn` can call `spawn` only from a first-solution-only context. Given that `spawn` has an output that cannot be discarded (the `io__state` is needed for the output argument of `main`) this first-solution-only context can be created only by the `main/2` predicate being declared `cc_multi`. This reflects the fact that any program that calls `spawn` conceptually generates a list of I/O action requests whose order is not totally determined. By declaring `main/2` to be `cc_multi`, the programmer explicitly states that all interleavings of the streams of actions generates by the various tasks in the program *that the program can generate* are equally acceptable as solutions of the program as a whole.

Representing interactions between processes (i.e. threads within a Mercury program, and the external world), as lists of actions is not a new idea; it was first introduced as *streams* in Parlog [37]. In the committed choice programming languages like Parlog, I/O is sometimes done explicitly by constructing a list of messages. While this was more declarative than simply doing side-effects, the committed choice languages don't have the simple declarative semantics we would like because of guards, and so forth. In Mercury programs, which do have a simple declarative semantics, using streams to do I/O (at least in an abstract sense) combines the elegance of a natural representation for I/O with clear declarative semantics.

4.5 A meta-interpreter for threads

In general, there will be some interleavings of the streams of actions that the programmer will not find acceptable as solutions of the program. In particular, the relative progress of the different threads should be constrained to respect the suspension and resumption of threads that use semaphores.

The suspension and resumption of threads in our model is quite different to that in coroutining extensions to Prolog (e.g. NU-Prolog and SICStus Prolog). In coroutining Prolog systems, when a literal in the current goal is suspended, the interpreter may choose any other literal in the current goal to evaluate instead (usually the left-most literal is chosen). In our case, for our programs to obey the mode constraints imposed by the programmer and the language definition, the whole continuation of the literal causing the suspension should also be suspended.

We can show how Mercury programs with threads can be understood by developing a meta-

```
:- type state
        ---> state(list(thread), id, map(id, nat - list(thread))).

:- type thread == list(goal).

:- type goal
        ---> true
        ;    (goal, goal)
        ;    main(data, data)
        ;    new_thread(goal)
        ;    new_sem(data, data, data)
        ;    signal(data, data, data)
        ;    wait(data, data, data)
        ;    ....
        .

:- type data
        ---> id(id)
        ;    []
        ;    [data|data]
        ;    ...
        .

:- type nat ---> zero ; succ(id).

:- type id == nat.

:- type map(K, V) == list(K - V).

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

:- type (A - B) ---> (A - B).
```

Figure 4.15: The types used by the meta-interpreter.

```
:- pred solve(data).
solve(IOs) :-
    solve(main(IOs, []), [], state([], succ(zero), []), _).

:- pred solve(goal, list(goal), state, state).
solve(true, [], Ts0, Ts) :-
    next(Ts0, Ts).
solve(true, [A|Cont], Ts0, Ts) :-
    solve(A, Cont, Ts0, Ts).
solve((A, B), Cont, Ts0, Ts) :-
    solve(A, [B|Cont], Ts0, Ts).

:- pred next(state, state).
next(state([], N, Ss), state([], N, Ss)).
next(state([Cont|Rs], N, Ss), Ts) :-
    solve(true, Cont, state(Rs, N, Ss), Ts).
```

Figure 4.16: The core of a meta-interpreter for threaded Mercury programs.

```
solve(new_thread(A), Cont, Ts0, Ts) :-
    Ts0 = state(Rs, N, Ss),
    Ts1 = state([[A], Cont|Rs], N, Ss),
    next(Ts1, Ts).
```

Figure 4.17: The code for creating a new thread.

```
solve(new_sem(id(N), IO, IO), Cont, Ts0, Ts) :-
    Ts0 = state(Rs, N, Ss0),
    insert(Ss0, N, zero-[], Ss1),
    Ts1 = state(Rs, succ(N), Ss1).
    solve(true, Cont, Ts1, Ts).
```

Figure 4.18: The code for creating new semaphores.

interpreter in (almost) pure Prolog. Our meta interpreter can be quite simple because we do not need to consider how all of Mercury is interpreted, only the deterministic subset. This is because in our model for threads, all threads are fully deterministic (though they might contain encapsulated failure and non-determinism). Therefore, the only types of goals we need to consider are: `true` as a base case, the conjunction of two goals, and calls. Because we need to suspend the continuation of a goal when a call to `wait` suspends, our meta-interpreter is written in a continuation-style. The main solve predicate takes a goal to solve, the continuation of the goal, and a pair of state arguments. The type `goal` represents this, and (at least notionally) has a constructor for every predicate in the program. A thread is simply the list of goals that need to be executed for the thread to complete (i.e. when considering the execution in terms of a continuation passing meta-interpreter, the thread is just the continuation of the current goal). The state arguments encapsulate a list of runnable threads, a counter for generating new semaphore ids, and a dictionary structure mapping semaphore ids to a pair containing the count associated with the semaphore and a list of threads that are blocked on the semaphore. These types are shown in figure 4.15. The core meta-interpreter, shown in figure 4.16 is very simple. If a continuation is non-empty then it expands the goal from the start of the continuation; if the continuation is empty, it gets the next runnable thread from its state and starts executing that. The top-level `solve/1` predicate yields the list of I/O interactions performed by the program. Note that in the implementation of `solve/1`, there is no check that the final state that results from the call to `solve/4` does not contain any suspended (i.e. deadlocked) threads. In practice, the presence of such threads should be a runtime error for the Mercury program.

We now extend this meta-interpreter with clauses for `solve/4` to handle the predicates that are not implementable in Mercury, but are instead implemented in the meta-interpreter. The first of these is the predicate mentioned earlier in section 4.4: `new_thread`. It takes the goal given as its argument and adds it to the list of threads that are runnable. (Note we insert it ahead of the continuation of the current thread — this is an arbitrary scheduling decision.) The code for handling `new_thread` is shown in figure 4.17.

The code for handling `new_sem` is very simple. It allocates a new semaphore id (we use natural numbers in successor notation for simplicity) which it binds to the appropriate output argument of `new_sem`, and executes the continuation of the current thread. The implementation of this is shown in figure 4.18.

The code for executing `wait` searches in the mapping from the state argument for the given semaphore and then calls `wait/6` to handle the wait operation. If the count associated with the semaphore is zero it adds the current thread to the list of threads that are suspended on this semaphore, and runs the next runnable goal. If the count associated with the semaphore is non-zero then the thread is not suspended, but is scheduled as runnable, and the count associated with the semaphore is decremented The code for this is shown in figure 4.19.

```
    solve(wait(id(X), IO, IO), Cont, Ts0, Ts) :-
        Ts0 = state(_, _, Ss),
        search(Ss, X, V-Gs),
        wait(V, X, Cont, Gs, Ts0, Ts).

    :- pred wait(nat, id, thread, list(thread), state, state).
    wait(zero, X, T, Gs, Ts0, Ts) :-
        Ts0 = state(Rs, N, Ss0),
        insert(Ss0, X, zero-[T|Gs], Ss),
        Ts1 = state(Rs, N, Ss),
        next(Ts1, Ts).
    wait(succ(V), X, T, Gs, Ts0, Ts) :-
        Ts0 = state(Rs, N, Ss0),
        insert(Ss0, X, V-Gs, Ss),
        Ts1 = state(Rs, N, Ss),
solve(true, T, Ts1, Ts).
```

Figure 4.19: The code for performing a wait operation on a semaphore.

```
    solve(signal(id(X), IO, IO), Cont, Ts0, Ts) :-
        Ts0 = state(Rs, N, Ss),
        search(Ss, X, V-Gs),
        Ts1 = state([Cont|Rs], N, Ss),
        signal(V, X, Gs, Ts1, Ts).

    :- pred signal(nat, id, list(thread), state, state).
    signal(zero, X, [], Ts0, Ts) :-
        Ts0 = state(Rs, N, Ss0),
        insert(Ss0, X, succ(zero)-[], Ss),
        Ts1 = state(Rs, N, Ss),
        next(Ts1, Ts).
    signal(zero, X, [Cont|Gs], Ts0, Ts) :-
        Ts0 = state(Rs, N, Ss0),
        insert(Ss0, X, zero-Gs, Ss),
        Ts1 = state(Rs, N, Ss),
        solve(true, Cont, Ts1, Ts).
    signal(succ(V), X, Gs, Ts0, Ts) :-
        % Gs must be [],
        Ts0 = state(Rs, N, Ss0),
        insert(Ss0, X, succ(succ(V))-Gs, Ss),
        Ts1 = state(Rs, N, Ss),
        next(Ts1, Ts).
```

Figure 4.20: The code for performing a signal operation on a semaphore.

```
:- pred search(map(K, V), K, V).
search([K-V|_], K, V).
search([K0-_|Map], K, V) :-
    K0 \= K,
    search(Map, K, V).

:- pred insert(map(K, V), K, V, map(K, V)).
insert([], K, V, [K-V]).
insert([K-_|Map], K, V, [K-V|Map]).
insert([K0-V0|Map0], K, V, [K0-V0|Map]) :-
    K0 \= K,
    insert(Map0, K, V, Map).

:- pred mercury_clause(goal, goal).
```

Figure 4.21: The utility routines used by the meta-interpreter.

Calls to the predicate `signal/3` are handled by looking up the given semaphore in the mapping putting the current thread back on the queue of runnable threads, and calling the meta-interpreter predicate `signal/5`. If the value of the semaphore is zero, and there are no suspended threads, then it increments the value of the semaphore and continues the execution of the next runnable thread. If the value of the semaphore is zero and there are suspended threads, then the first of these is made runnable. If the value of the semaphore is non-zero then there are no suspended threads, and the value of the semaphore is incremented. The code for handling signal operations is shown in figure 4.20.

Figure 4.21 shows some utility predicates used by the meta-interpreter.

Chapter 5

Profiling programs

One of the biggest problems for parallelizing programs in declarative languages like Mercury is avoiding creating too many small parallel tasks, often referred to as “embarrassing parallelism”. When this happens, the cost of creating and managing the tasks very quickly outweighs the benefits from their parallel execution. A great deal of work has been done on developing analyses that build, at compile time, an estimate of the cost of a given procedure call, the point being that an implementation can then choose to execute in parallel only calls whose cost exceeds some threshold [27, 30, 56, 92]. The estimate need only be a lower bound, since we only want to parallelize computations that we are sure will be sufficiently expensive to offset the cost of creating and managing the parallel task.

These techniques have a number of important limitations which, while not rendering them useless, impact their potential to automatically parallelize Mercury programs. The first limitation is that the analysis techniques cannot determine the constant factors involved in the cost models of programs. These are influenced not only by implementation details from the execution model and runtime system (such as how expensive it is to make a procedure call, and how garbage collection is implemented), but also by the characteristics of the underlying hardware (such as what kind of caches the machine has). The second limitation is that these analyses generally compute the cost of a given procedure call as a function of the “size” of the input arguments[1] which can usually only be determined at runtime. Actually computing the size of terms at runtime adds very materially to the overheads in the program, and therefore makes it much more difficult to achieve useful gains from parallelism. (Making the size of a term part of its representation eliminates this overhead but doesn’t really help, since it introduces a distributed space and time overhead.) The third limitation of analysis, as observed elsewhere in this thesis is that for it to achieve reasonable precision it must be done globally. This limits the largest Mercury program to which it can be applied if only because of the memory required to store the entire program, let alone the time needed to perform the analysis.

Using measured rather than computed costs avoids all three of the above limitations. We therefore believe that the choice of what calls to parallelize should be guided by profiling information. This is not to say that analysis should be rejected, but rather that it should at least be augmented with quantitative data from the actual execution of the program. To this end, we need an accurate quantitative picture of how expensive different parts of the program really are. This chapter describes a profiling system we have constructed with the ultimate goal of using it for guiding the automatic parallelization of Mercury programs via profiler feedback to the Mercury compiler.

Over the years, researchers and software engineers have designed and implemented many different profiling systems. The variety of existing profiling systems is driven mainly by the intense conflict at the heart of profiler design: a good profiler should present to the programmer as much information relevant to the performance of the program as possible, but the more information

[1] The “size” of the input arguments can be defined in a variety of ways — see [4, 12–14, 99].

a profiler gathers, the more the gathering process itself interferes with the performance of the program, and thus the less valid the gathered information is. This conflict has driven designers to build profilers that gather only the minimum necessary information, no more. Profiler designs differ because what information is necessary depends on many factors, including programming language, programming style and program size.

Declarative programming languages pose new challenges for profilers because they introduce new programming styles. These include extensive use of recursion including mutual recursion, intensive reliance on various forms of polymorphism, and frequent use of higher order constructs. All these existed or at least could be simulated in imperative or object-oriented languages, but as long they were not typical and were rarely used in performance critical code, profiler designers could mostly ignore them. They could design profilers as if (mutual) recursion, polymorphism and higher order calls did not exist, and make the minimum concession to their existence in the implementation: they could ensure that the profiler does not crash on programs containing such code without making the profiler gather the information that the programmer would need to optimize program fragments containing such code. Designers of profilers for declarative programming languages do not have that option. Since declarative languages do not have iteration, no program fragment can be performance critical without being recursive (possibly mutually recursive), and the pervasive use of polymorphism and higher order code implies that performance critical code will likely use those constructs as well. A profiler for a declarative language must therefore make sure that the information it gathers and presents to the programmer will allow hot spots to be located and diagnosed even in the presence of (mutual) recursion, polymorphism and higher order code.

In this chapter, we present a new profiling technique called *deep profiling* that we have designed to meet this requirement. It gets its name from its chief design feature, the association of a deep context (effectively the entire list of ancestors of the current function or predicate) with each profiling measurement. Recording this much information at an acceptable cost was a considerable engineering challenge, as was finding a way to display this much information without confusing and distracting the programmer. Actually, we have found that the information gathered by the profiler can be used not only for performance improvement, but also for elucidating the typical behaviour of the program to maintainers, programmers who are unfamiliar with the structure of a program either because they were not involved in its initial implementation or because the program has changed significantly since the last time they looked at it. In addition, our profiler (like many others) can be used for coverage testing: finding parts of the program that have not been executed at all and in which therefore undetected bugs may still be lurking.

The structure of this chapter is as follows. Section 5.1 lists the main questions that the designer of a profiling system must answer, together with the answers used in existing profilers. Section 5.2 lists the requirements imposed on a profiler by the design of the Mercury language and by the structure of typical Mercury programs. These issues are shared not only by other logic programming languages but also by functional languages, and, to a lesser extent, by some object-oriented language as well. Sections 5.3 and 5.4 are the core of the paper: they give a detailed description of deep profiling, the profiling system we have designed to meet the requirements listed in section 5.2. It should be adaptable to any language that has the same list of requirements (since laziness adds extra requirements, adaptation to lazy functional languages would require further work). Section 5.3 shows how deep profiling instruments the program to associate a deep context with each measurement at acceptable cost. It also discusses tradeoffs in our chosen approach, and outlines possible approaches we have decided not to take. Section 5.4 shows how data gathered during profiling runs is post-processed and presented to the programmer. Section 5.5 presents experimental data on the costs of deep profiling: the overhead it imposes on the programs being profiled and cost of running the profiling tool itself. Section 5.6 then compares deep profiling with other profiling systems described in the literature, including cost centre stack profiling [52] and call path profiling [40, 41].

5.1 Dimensions of profiler design

Many different profiling systems exist for a variety of languages and platforms. These systems can be usefully compared by examining the following characteristics.

What is being measured. The most common forms of profiling measure how much time is spent in each part of the program. However, profiling may also measure memory-related statistics, such as the number of memory allocations in each part of the program, the amount of memory allocated by each part of the program, or the amount of memory live during the execution of each part of the program. It may also count the number of times each part of the program is executed, either focusing only on some kinds of constructs (e.g. branches taken, exceptions thrown, locks contended for, or goals failed (in logic programming languages)) or on a universal basis, at various levels of granularity (e.g. function calls, statement executions). (Some of this information is as useful for coverage testing as for profiling.) Most profilers measure more than one aspect of the program, although few measure *all* these aspects, at least not at the same time.

How much context each measurement has. In the simplest case, each measurement is independent of context; e.g. the profiler may count the number of times function `f` is called, but will not record where each call came from. The profiler may add context to each measurement by recording what function each call to `f` came from. That context can be refined further by noting not just that a call to function `f` came from function `g`, but also that it came from a specific call site (e.g. line 120, column 10). The context can be made more detailed by recording, for each measurement, not only the identity of the parent caller function or call site, but also the identities of *all* the ancestor caller functions or call sites.

How measurements are taken. The two most common ways of collecting profiling data are to instrument the code (which can be done by modifying the source program, using the compiler to generate instrumented code, or by post-processing the output of the compiler), and to periodically interrupt the program and take samples of the desired data. The sampling approach suffers from the obvious disadvantage that its measurements are not deterministic, but errors resulting from the uncertainty are usually significant only for measurements in infrequently executed parts of the code, and the programmer usually does not care about the performance of such code. The two techniques are often combined in a hybrid approach, in which some measurements are taken by instrumentation and other measurements are taken by sampling. The usual choice is that time is measured by sampling and everything else is measured by instrumentation. (Since most CPU types lack timers with sufficient resolution, which currently means in the region of nanoseconds, a profiler that tried to measure time via instrumentation would not be portable.)

The relative runtime overhead of profiling. Different profiling techniques have different impacts on the running time and memory usage of the program. Some kinds of measurement (e.g. CPU time) are distorted by these overheads. One kind of distortion results from an uneven distribution of the overheads over the measured aspects program execution: for example, the addition of a fixed number of instructions to each function will produce a proportionately bigger increase in the times recorded for small functions than for big functions. Another kind of distortion occurs because profiling overhead alters the interaction of the program with the virtual memory system, with caches, with branch target buffers, etc. In theory, both these kinds of effects can be compensated for. Unfortunately, given the complex pipelined and superscalar architecture and multilevel memory hierarchy of modern machines, and the dearth of published information on their design at the required level of detail, such compensation is usually far from feasible. The only practical way to minimize distortions of the data presented to programmers is to keep the overheads of the profiling process as low as possible.

When the runtime costs of profiling are paid. In many profilers, the costs of profiling are paid only at the events that profiling measures. For example, the cost of counting a procedure call is paid at the procedure call, and the cost of recording a sample for CPU time measurement is paid at the time the sample is taken. However, in profilers that put each measurement in context, the work of setting the contexts may be performed when the contexts change, even if there is no measurement to be made at that point. This reduces overall overhead if the average number of measurements per context change is more than one.

When the information is collated. Before being presented to the user, the raw data collected during the profiling process must be organized and compacted. Some systems just record the raw data during the execution of the profiled data, leaving the data reduction to a substantial post-processing stage; others perform data reduction during program execution, leaving the post-processing stage to do little more than formatting the data. The latter kind of system incur more runtime overhead and thus distort the profile more, but in return they do not require storage of the raw data in toto at any time. Given that the raw data can be quite big (tens or hundreds of megabytes), this is a substantial advantage. Hybrid systems preprocess the raw data during runtime but still require some post-processing. A good tradeoff is to perform processing tasks that reduce the volume of the data at runtime, and to do all other processing tasks after runtime.

When the programmers specify what information they are interested in. In many profilers, programmers never specify this at all; the profiling tool produces a file containing all its data, and programmers look at whatever parts of this file they are interested in. As the number of measurements and the size of the context of each measurement increases, this approach becomes unworkable, because programmers find themselves facing a sea of data they have to summarize themselves. Some profiling tools are therefore interactive; they digest a database of measurements produced by some profiling runs, direct the programmer to the most interesting parts of the database, and then allow programmers to ask queries on the database. Some go even further, and allow or even require programmers to state what parts of the program they are interested in even *before* compiling the program, and perform measurements only on that part. This reduces overhead and therefore distortions in the resulting data, at the cost of requiring recompilation and rerun of the program if the programmers find that they need information about a previously non-profiled part of the program after all.

5.2 Requirements for Mercury profiling

In this section, we list the requirements placed on a profiler by the characteristics of the Mercury language and Mercury programs. We expect most other eager declarative languages and even many imperative object-oriented languages to have the same or very similar requirements, which means that our profiler design should also be well suited for them. Laziness imposes a significant additional requirement, but we believe deep profiling can be adapted to handle lazy languages as well.

Recursion as a basic language construct. The most obvious issue for profiling declarative languages is that they depend far more heavily on recursion than imperative languages do; in most declarative languages (including Mercury), recursion is the only way to implement repetition. A profiler for an imperative language can usually get away with reporting that e.g. "of the 1000 calls to function `f`, 200 were from function `g`, 100 from function `h`, and 700 from function `f` itself" without also reporting e.g. how many recursive calls each call from `g` gives rise to, or how long the average call to `f` *from g* took. The reason is that most functions written in such languages are not recursive, so the probability that such missing information will cause programmers difficulty in speeding up a performance-critical function is often low enough to be acceptable. Profilers for such languages can thus treat recursion as an exception, in whose presence the correct attribution of costs is a nice bonus but not essential. On the other hand, in declarative languages a function or predicate *cannot* be performance critical without being recursive. Profilers for such languages therefore must treat recursive calls properly, which in almost all cases means treating them as belonging to the same profiling unit as their recursive ancestor. (We discuss an important exception below.)

Mutual recursion. Programs in declarative languages often operate on complex data structures, a significant proportion of which are defined in terms of mutually recursive types. The procedures that process such data structures are therefore often also mutually recursive. Mutually recursive procedures act as each other's extensions, with each procedure containing part of a repetitively executed piece of code. Calls between mutually recursive procedures should therefore be treated like self-recursive calls in most ways, the exception being that we do still want to associate costs incurred directly by a procedure with that procedure, and not just with the clique of mutually recursive procedures as a whole.

Polymorphism of predicates. Tables storing key-value pairs are some of the most common data structures. In most imperative programming languages, a program that needs several different kinds of tables would have a different function for searching each kind of table, either written by the programmer or generated automatically by the compiler (e.g. when using C++ templates). When written in a programming language (such as Mercury) that supports and encourages the use of parametric polymorphism, a single function can perform searches on *all* tables of key-value pairs in the program, regardless of the types of the keys and values.

This poses a problem for profilers. Whereas programmers would typically find it useful to know e.g. the average cost of a call to a function that searches a binary search tree containing e.g. environment variable name - value pairs, they typically would *not* find it useful to know the average cost of a call to a function that searches *all* binary search trees in the program. Even though that function appears to be a single unit in the source code, it is actually an amalgam of several distinct units for the programmer, and the profiler must help the programmer decompose the costs reported for the function into the costs of its distinct uses.

Polymorphism of callers. Unfortunately, it is not sufficient for the profiler to assume that two uses of a function are distinct if and only if the function is called from two different sites. To see why not, consider the Mercury compiler, which is written in Mercury itself. The compiler has several hundred data structures that are tables of key-value pairs, many temporary, some permanent. All these tables are implemented as different instances of the same polymorphic type. This type, `map(K,V)` is an abstract type; its actual definition private to the `map` module, to allow its definition to be changed without impacting any other modules. Currently, maps are implemented as 2-3-4 trees, and predicates in the map module just call their counterparts in the `tree234` module. (The Mercury compiler and library contain many such "forwarding" predicates; the abstraction they provide makes it easier to revise old design decisions.)

One of the most expensive predicates in the Mercury compiler is `tree234__search`, which performs lookups in 2-3-4 trees. This predicate is expensive because it performs searches on all those several hundred kinds of tables. Despite this fact, it is called from only one place, `map__search`. Clearly, the profiler must use more detailed context information, not just the immediate caller, if it is to correctly distinguish the different uses of `tree234__search`.

Polymorphism within callers. Most profilers make no distinction between different calls to function `f` from function `g`, i.e. they treat the call graph as having just one arc from `g` to `f`. This effectively makes the assumption that all the different calls from `g` to `f` (if there is more than one) have similar enough performance characteristics that the programmer will not mind having their data amalgamated For programs written in traditional imperative languages, this assumption is usually correct.[2] However, this assumption is much less likely to be correct for programs written in languages that support some form of polymorphism, whether the class-based polymorphism of object oriented languages or the parametric polymorphism of declarative languages.

Consider a program that deals with sets of items (e.g. LR(1) items) with each set assigned a set identifier. Such a program would probably use two maps, one mapping identifiers to sets of items and another one mapping sets of items back to their identifier (so that the program can detect when the result of e.g. a set union is a set that has been seen previously). The procedure that handles insertions into this data structure would call the map insertion function (e.g. `map__insert` in Mercury) twice, once to update each of the two maps. If the maps are implemented as a search tree of some kind, then insertions into the map will require repeated key comparisons, which will be much more expensive for the map in which the keys are sets of items than for the map in which the keys are identifiers. The two calls to `map__insert` will therefore have quite different performance characteristics, which means that the programmer will almost certainly want to be given separate information about each call.

Frequent use of higher order constructs. Programs written in declarative languages use higher order constructs significantly more frequently than programs written in imperative

[2]The main reason for this is that programs written in traditional imperative languages tend to have fairly shallow call graphs, and their functions tend to have a single kind of usage, either because the languages lack polymorphism or (e.g. in the case of C++) because template instantiation creates separate copies of functions for different usages.

languages. It is therefore much more important for profilers of declarative languages to handle such constructs properly. At the minimum, this means that the profiler must track caller-callee relationships even when a particular caller-callee pair does not appear in the source code. Two desirable extensions are distinguishing first order and higher order calls in the profiler output, and, for higher order calls, distinguishing between closures constructed at different sites that happen to call the same function.

Nested use of higher order constructs. In a program that makes extensive use of higher order constructs, it is likely that some calls to a higher order function will be from descendants of that same higher order function. For example, consider a program working with a genealogical database whose task is to print, for each parent in the database, the parent's name along with the names of all their children. The main program may use `foldl` to call `handle_parent` for each parent in the database, while `handle_parent` itself may use `foldl` to print the names of all the parent's children.

A simplistic profiler would consider `foldl` and `handle_parent` to belong to the same strongly connected component of the call graph, since each calls the other. However, programmers are unlikely to view the program that way; they are much more likely to view the two uses of `foldl` as distinct entities. The two uses cannot be distinguished by type, since both loop over lists of people; they can be distinguished only by the identity of the closure passed to `foldl` or by the location of the top-level call to `foldl` (since the code there determines what closure is passed to `foldl`). What the programmer usually wants is for the profiler to treat the program not as it is, but as it would be if all top-level calls to `foldl` were specialized, i.e. if each call to `foldl` was replaced by a call to a distinct new procedure which is renamed apart copy of `foldl`. (Some optimizing compilers, including the Mercury compiler, perform exactly this specialization in an attempt to transform higher order calls into more efficient first order calls.)

Bidirectional foreign language interfaces. Like most other non-mainstream languages, Mercury has a foreign language interface (FLI). This interface is bidirectional. On the one hand, the programmer may write some C code instead of some clauses as the implementation of a Mercury predicate or function. On the other hand, the programmer may define a predicate or function in Mercury code and then request that the Mercury compiler generate a C function whose body just calls that predicate or function. Programmers frequently use the foreign language interface to link Mercury code with existing libraries.

One important class of existing libraries are libraries implementing graphical user interfaces. Most of these libraries (e.g. Tk) are based on an event loop that performs callbacks to code provided by the client. A Mercury program using the Mercury-Tk interface would start out by constructing some Tk objects. These objects would have associated actions, and these actions would be implemented as Mercury predicates and/or functions which are exported to C code using the FLI. The program would then enter Tk's event loop, which is of course written in C. The event loop would then respond to user actions (key strokes, mouse clicks) by invoking the relevant callbacks on the appropriate objects. These callbacks may cause execution to reenter Mercury code.

Ideally, the profiler for Mercury would understand C code as well, and would be able to treat the Mercury and C parts of the program in a uniform manner. However, given the large differences between the two languages, even deciding what such a profiler should do would be quite a complex undertaking. Building such a bilingual profiler would not be cost effective for us, given that we prefer to write all our new code in Mercury, and only use the foreign language interface to link to third party libraries that we have no wish to modify or even (in most cases) to understand beyond the interface level.

However, even a profiler that only understands Mercury must do something sensible with the costs incurred by C code. The only sensible thing to do with such costs is to charge them against the nearest enclosing Mercury procedure that used the foreign language interface to enter C code. Similarly, calls to Mercury procedures from C code should be counted as calls made directly by the nearest enclosing Mercury predicate that used the foreign language interface to enter C code. A Mercury predicate implemented in C can thus be viewed as similar to a predicate that makes higher order calls, in that both have descendants in the call graph that do not appear syntactically in their definition.

5.3 The deep profiling transformation

This section describes our design for gathering the data that one needs in order to satisfy the requirements we identified in the previous section. We start with a basic system that meets three of those requirements: polymorphism of predicates, polymorphism of callers and polymorphism within callers, but does not handle recursion properly and does not handle higher order calls and foreign language code at all. Section 5.3.1 describes the data structures of this system, while section 5.3.2 shows how we build them. The later subsections then extend the system to handle more of our requirements. Section 5.3.3 shows a technique we could use to handle self recursion while section 5.3.4 shows a technique we could use to handle mutual recursion within one module. Our profiler doesn't actually use either of those techniques, preferring instead the approach presented in section 5.3.5 which handles all forms of recursion, even mutual recursion across modules. Section 5.3.6 shows how we preserve tail recursion in procedures that are not nondeterministic. Section 5.3.7 then shows how we handle higher order calls, and section 5.3.8 shows how we avoid unwanted interactions between recursion and higher order calls. Section 5.3.9 describes our treatment of foreign language code, while section 5.3.10 describes our handling of exceptions. Section 5.3.11 describes how we gather profiling measurements, and section 5.3.12 shows how we save the resulting profiling data. The final system meets all of the requirements we listed in section 5.2.

5.3.1 Introducing the main data structures

Our initial profiling system is driven by its demanding requirement of resolving all forms of polymorphism. As we have seen in section 5.2, giving programmers enough information to let them measure distinct uses of a procedure separately requires associating a significant amount context with each measurement, more than most profilers do. At the very least, it requires associating the caller procedure with each measurement of the callee, but many circumstances of practical importance require additional context in the form of the specific call site within the caller procedure and/or the identity of more distant ancestors.

In theory, we could use a chain of ancestor call sites as the sole context of each measurement. However, call sites don't call other call sites; they call procedures, which in turn contain other call sites. We have therefore found it significantly simpler to use an alternating chain of call site nodes and procedure nodes as context: call site nodes link to procedure call nodes and vice versa.

Suppose the program contains five functions: `f`, `g`, `h`, `i` and `j`, where `f` calls `g` and `h` (at call sites `f1` and `f2` respectively), `g` and `h` both call `i` (at call sites `g1` and `h1` respectively), and `i` calls `j` (at call site `i1`). We will want to record measurements for the call chain `f->f1->g->g1->i->i1->j` separately from the measurements for the call chain `f->f2->h->h1->i->i1->j`. This means that the profiler will have to build up a data structure that contains more than one instance of the call site `i1`. In most programs, there will be many procedures (and thus call sites within those procedures) that will appear many times in the tree.

There are several ways one can identify a call site. One can specify them by file name, line number, and column number; by file name and byte offset within the file; by file name, procedure name, and call number within the procedure. All these identifiers are big enough that we don't want to avoid storing redundant copies of them. We therefore build a static data structure for each call site: this data structure describes the call site, and its address serves as a compact identifier of the call site. The set of call sites associated with a procedure is also static information, and thus it is also worthwhile to avoid storing it multiple times. We therefore split procedure call nodes into static and dynamic components just as we do with call site nodes.

The main data structure of the profiler is thus a tree (later to be a graph) containing four kinds of nodes, which we call CallSiteStatic, ProcStatic, CallSiteDynamic and ProcDynamic which, for the time being we will give the following definitions.

CallSiteStatic CallSiteStatic structures are created by the compiler. There is one CallSiteStatic structure for each call site in the source code.

`css_id` This field uniquely identifies the call site, and links it back to the source file for the programmer. Any of the representations above will suffice. The choice of representation does not affect the profiler except with respect to details of its user interface, which we do not discuss in that level of detail in this paper.

`css_callee_id` For now, we will consider this field to contain a pointer to the ProcStatic structure of the procedure called at this call site. In later sections we will also allow it to indicate that the call site represents a higher order call or a typeclass method call (section 5.3.7) or entry into foreign code (section 5.3.9).

ProcStatic ProcStatic structures are created by the compiler. There is one ProcStatic structure for each procedure in the source code.

`ps_call_sites` This field is an array containing the CallSiteStatic structures of the call sites within the procedure.

`ps_num_call_sites` This field gives the number of call sites within the procedure, and thus the size of the array in the `ps_call_sites` field.

CallSiteDynamic CallSiteDynamic structures are created by the instrumented program during a profiling run. There will be one or more CallSiteDynamic structures for each call site through which the program actually performs a call during the profiling run. For a given call site, there will be distinct CallSiteDynamic structures for each distinct context in which those invocations take place.

`csd_self_profile` This field contains all the measurements collected for the invocations of the called procedure from the context represented by this CallSiteDynamic structure and all its ancestors, which will be an alternating chain of ProcDynamic and CallSite-Dynamic nodes. This chain is linked together with downward pointers in order to allow the graph to be traversed from the root at the end of execution, when the information it contains is saved to a file.

`csd_callee_ptr` This field contains a pointer to the ProcDynamic node for an invocation of the called procedure, to be used as part of the context for some profiling measurements in the callee's descendants.

ProcDynamic ProcDynamic structures are created by the instrumented program during a profiling run. There will be one or more ProcDynamic structures for each procedure which is called during the profiling run. For a given procedure, there will be distinct ProcDynamic structures for each distinct context in which those calls take place.

`pd_proc_static` This field points to the ProcStatic structure of the procedure that this ProcDynamic structure represents an invocation of.

`pd_call_site_ptrs` This field contains a pointer to an array, that contains a pointer to a CallSiteDynamic structure for each call site in the procedure. The size of the array is given by the `ps_num_call_sites` field of the ProcStatic structure pointed to by the `pd_proc_static` field, and the call site represented by each entry in this array is identified and linked back to the source code by the corresponding entry in the `ps_call_sites` array. If a given procedure invocation does not make a call through a given call site, then the element in the array pointed to by the `pd_call_site_ptrs` field corresponding to that call site will contain a null pointer. Given the precise contexts we use, we expect this will be true for a large fraction of all call sites in each context, which is one reason why we use an array of pointers to structures instead of an array of the structures themselves. However, the main reason is that we want to be able to link to more than one CallSiteDynamic structure from elements of the array that correspond to call sites representing higher order calls, method calls and entry into foreign language code, as we will explain in sections 5.3.7 and 5.3.9.

5.3.2 Building the initial version of the call graph

As we mentioned in section 5.1, profilers make some of their measurements by sampling. For example on Unix, profilers implement this by asking the kernel to send the program a SIGPROF signal periodically, and recording the measurements in the signal handler function. These measurements should be associated with the context of the code that is being executed at the time the signal arrives. Since the program has no control over the arrival of the signal, the program must maintain at all times a data structure that tells the signal handler what the current context is. The data structure we use in the Mercury profiler is a global variable called `current_call_site_dynamic`, which points to the CallSiteDynamic structure associated with the call site of the procedure invocation currently being executed. To minimize the measurement distortions caused by the program instrumentation we are about to describe, the signal handler ignores profiling signals that arrive while the global boolean variable `inside_instrumentation` is true. Our system sets `inside_instrumentation` to true during the execution of all pieces of instrumentation code except those for which setting and resetting `inside_instrumentation` would be more time consuming than the instrumentation code itself. However, to simplify our exposition and save space we will not show the assignments to `inside_instrumentation`. We will also delay discussion of taking the profiling measurements themselves until section 5.3.11.

The `current_call_site_dynamic` global variable must be updated whenever control passes from one procedure to another. The standard box model for the execution of logic programs [16] classifies control transfers into four categories: call, exit, redo and fail. (Exit and fail represent successful and unsuccessful returns respectively, while redo represents backtracking back into a call.) Each of these methods of control transfer is called a *port* in logic programming debuggers. [3] To give the profiling system control at each of these ports, the compiler transforms the bodies of all procedures when profiling is enabled in a fashion very similar to what it does when debugging is enabled, except that for profiling, the form of transformation it uses is source-to-source. (For reasons discussed in [95], the debugging transformation is implemented in the code generator.)

Internally, the Mercury compiler considers each procedure to be defined by a single goal; multiple clauses are converted into explicit disjunctions very early on. The general transformation therefore starts with a procedure definition such as

```
p(Args) :-
    Body.
```

and converts it into code of the following form:

```
p(Args) :-
    call_port_code(p_proc_static_ptr, TopCSD, MiddleCSD),
    (
        Body',
        ( exit_port_code(TopCSD, MiddleCSD)
        ; redo_port_code(TopCSD, MiddleCSD), fail
        )
    ;
        fail_port_code(TopCSD, MiddleCSD), fail
    ).
```

Each procedure invocation is concerned with the CallSiteDynamic structures representing the calls to its parent and to its children as well as the call to itself. Each procedure invocation refers to

[3] Mercury adds a fifth port as well: control can leave a procedure invocation by raising an exception. However, we will delay consideration of exceptions until section 5.3.10.

```
function prepare_for_normal_call(call_site_num) is
    local pd
    pd := current_call_site_dynamic->csd_callee_ptr
    if pd->pd_call_site_ptrs[call_site_num] is null then
        next_call_site_dynamic := allocate_call_site_dynamic()
        next_call_site_dynamic->csd_self_profile := zero_profile()
        next_call_site_dynamic->csd_callee_ptr := null
        pd->pd_call_site_ptrs[call_site_num] := next_call_site_dynamic
    else
        next_call_site_dynamic := pd->pd_call_site_ptrs[call_site_num]
    end
end
```

Figure 5.1: `prepare_for_normal_call`

those CallSiteDynamic structures as TopCSD, BottomCSD and MiddleCSD respectively. However, the identity of the call site that e.g. TopCSD refers to depends on which procedure invocation's perspective we adopt. `call_port_code` and `redo_port_code` each shift the perspective from the caller to the callee, while `exit_port_code` and `fail_port_code` shift the perspective from the callee back to the caller.

Each of `call_port_code`, `exit_port_code`, `redo_port_code` and `fail_port_code` updates the global variable `current_call_site_dynamic`. Since `call_port_code` and `redo_port_code` are executed when control transfers from parent to child, they set `current_call_site_dynamic` to point to MiddleCSD, the CallSiteDynamic structure of the call site which created the child procedure invocation (that call site will be in the parent invocation); since `exit_port_code` and `fail_port_code` are executed when control transfers from child to parent, they set `current_call_site_dynamic` to point to TopCSD, the CallSiteDynamic structure of the call site which created the parent procedure invocation (that call site will be in the grandparent invocation). This of course requires the child invocation to know not only the address of the CallSiteDynamic structure representing its own invocation, but also the address of the CallSiteDynamic structure representing its parent's invocation. Obviously, the child invocation does not know either call site; it must be passed that information by the parent. Our scheme passes these two addresses in two global variables. The first, `next_call_site_dynamic`, is new: it points to the CallSiteDynamic structure of the child invocation, which the caller would call BottomCSD and the callee calls MiddleCSD. The second, `current_call_site_dynamic`, points to the CallSiteDynamic structure of the parent invocation, which the caller calls MiddleCSD and the callee calls TopCSD. We delegate the task of assigning the proper value to `next_call_site_dynamic` to a service function in the Mercury runtime system, `prepare_for_normal_call`, and transform the bodies of all procedures, inserting a call to `prepare_for_normal_call` before every call in the original code. (This is the difference between *Body* and *Body'* above.) The values of these global variables are then picked up by `call_port_code`, which returns their original values in Mercury variables. The values of these local variables will then be stored in the stack frame of the child procedure invocation until they are needed by one of `exit_port_code`, `redo_port_code` and `fail_port_code`. Figure 5.1 shows the code of `prepare_for_normal_call`, while figure 5.2 shows the code of `call_port_code`.

The code of `prepare_for_normal_call` starts by checking whether a call has already been made through this call site in the current ancestor context. If yes, `prepare_for_normal_call` reuses the CallSiteDynamic structure allocated by the previous call for the new procedure invocation. With the scheme we have presented so far, this can only happen in the presence of backtracking, when a call is reexecuted after a goal to its left has succeeded for the second or later time. However, the first time a call is made through this call site in the current context, the call site will not have a CallSiteDynamic structure yet in the current procedure invocation's ProcDynamic structure. `prepare_for_normal_call` will therefore create one, and will link it into its position in the call graph as a child of the invoking procedure's ProcDynamic structure after initializing its fields.

```
function call_port_code(proc_static_ptr, out TopCSD, out MiddleCSD) is
    local pd
    TopCSD := current_call_site_dynamic
    MiddleCSD := next_call_site_dynamic
    current_call_site_dynamic := next_call_site_dynamic
    current_call_site_dynamic->csd_self_profile.call_count += 1
    if current_call_site_dynamic->csd_callee_ptr is null then
        pd := allocate_proc_dynamic()
        pd->pd_proc_static := proc_static_ptr
        pd->pd_call_site_ptrs := allocate_csd_ptr_array(proc_static_ptr->ps_num_call_sites)
        current_call_site_dynamic->csd_callee_ptr := pd
    end
end
```

Figure 5.2: `call_port_code`

`prepare_for_normal_call` initializes the `csd_callee_ptr` field to null to tell the callee that this is the first call to the callee in the current context. The `call_port_code` of the callee (shown in 5.2) will check for whether this is the case and if it is, it will allocate a ProcDynamic structure, fill in its fields, and link it into the call graph in turn. The `pd_proc_static` field will be filled in with a pointer to the procedure's statically allocated ProcStatic structure, and the `pd_call_site_ptrs` field with a pointer to a newly allocated array of pointers, all of whose elements are initialized to null. In the presence of backtracking, `prepare_for_normal_call` will have found that the callee already has a CallSiteDynamic, created by a previous call; that previous call must also have filled in `current_call_site_dynamic->csd_callee_ptr` with a pointer to the ProcDynamic structure of the callee. In such cases, the `call_port_code` of the callee will use that instead of allocating a new ProcDynamic structure.

The general transformation above wrapping procedure bodies is often unnecessarily wasteful. The reason is that in most programs, the overwhelming majority of procedures (e.g. about 99% of the procedures in the Mercury compiler) are guaranteed to succeed *at most* once, and a large fraction of those (e.g. about 78% of the procedures in the Mercury compiler) are guaranteed to succeed *exactly* once. How the compiler transforms each procedure therefore depends on its determinism. If the procedure is nondet or multi (i.e. it can succeed more than once), it uses the general transformation above. If the procedure is semidet (i.e. it can succeed at most once), it doesn't insert code to get control on redo, since a redo will always step *past* and cannot step *into* such procedures:

```
p(Args) :-
    call_port_code(p_proc_static_ptr, TopCSD, MiddleCSD),
    (
        Body',
        exit_port_code(TopCSD, MiddleCSD)
    ;
        fail_port_code(TopCSD, MiddleCSD), fail
    ).
```

This transformation avoids the creation of a nondet stack frame (Mercury's equivalent to Prolog choice points) and thus reduces the overhead of profiling. Even more important, unlike the general transformation, it does not violate the invariant of the Mercury execution algorithm [96] that procedures which cannot succeed more than once do not leave behind any nondet stack frames when they succeed. (Mercury, unlike Prolog, automatically prunes the remaining disjuncts in a semidet disjunction if a disjunct succeeds, so that if *Body'* succeeds, `fail_port_code` will never be executed.)

For det procedures (i.e. those that succeed exactly once), even the code to handle failures is unnecessary, since such procedures cannot fail. We therefore use an even simpler transformation:

```
p(Args) :-
    call_port_code(p_proc_static_ptr, TopCSD, MiddleCSD),
    Body',
    exit_port_code(TopCSD, MiddleCSD).
```

There are three reasons why we use two service functions, `prepare_for_normal_call` and `call_port_code` to handle calls instead of doing everything in `prepare_for_normal_call`. First, we want to update `current_call_site_dynamic` only when execution has entered the callee. If the code to update `current_call_site_dynamic` were in `prepare_for_normal_call`, then any profiling signals arriving between the return from `prepare_for_normal_call` and the call to the callee, i.e. while the program executes the instructions in the caller that move the input arguments of the call to the registers where the callee expects them, would be incorrectly charged against the callee. Second, by avoiding the requirement to know the address of the callee's ProcStatic structure when compiling the caller, the split makes it much easier to handle higher order calls, method calls and entries to foreign code, since in those cases the identity of the callee is not known until runtime. Third, we need to reset `current_call_site_dynamic` when the call returns, and as long as we wrap `call_port_code`, `exit_port_code`, etc around procedure bodies rather than around call sites, we need something like `call_port_code` to pick up TopCSD and MiddleCSD.

Wrapping calls would have the advantage that procedures that don't make nondeterministic calls need only remember the address of their own CallSiteDynamic structure; they need not remember the address of their parent's CallSiteDynamic structure or of their children's. Procedures that do make nondeterministic calls need to remember the addresses of the CallSiteDynamic structures of their nondeterministic children, so that they can reset `current_call_site_dynamic` when execution backtracks into such a child, but this is a small cost given the rarity of such procedures. Wrapping calls can therefore be expected to reduce memory consumption on the stacks. The reason why we wrap procedure bodies instead is that calls outnumber procedure bodies by a significant margin; in the Mercury compiler, the average number of call sites per procedure is about 5.5. Wrapping procedure bodies therefore adds much less code to the program, and we expect the performance gain from the relative improvement in code locality to outweigh the improvement from lower stack usage.

Linking CallSiteDynamic and ProcDynamic structures upward as well as downward would allow us to avoid using any local variables at all to remember CallSiteDynamic addresses. From `current_call_site_dynamic`, a procedure invocation could always find the CallSiteDynamic structure of its parent as well as the CallSiteDynamic structures of all its children, and therefore `call_port_code`, `exit_port_code`, `redo_port_code` and `fail_port_code` would need only the current value of `current_call_site_dynamic` to find out what its next value should be. The cost would be an extra field in both the CallSiteDynamic and ProcDynamic structures. These structures are effectively permanent; once they are allocated, their memory is not reclaimed until the end of execution. Stack space is of course reclaimed much earlier, on the success or failure of a call (depending on the determinism of the procedure in question). This is a sufficient reason for us eschew upward links and accept the cost in stack usage. Another, more compelling reason is that navigating the tree using upward links is incompatible with the technique we use to handle recursion.

5.3.3 Handling self-recursion

As described thus far, the structure we create is a tree, and it has a call site node for almost every call made by the program.[4] This is obviously not practical: the amount of memory needed to store such a tree would be infeasible for all programs but those with trivially short runtimes. Even if we could store such trees, the information in it would be too sparse to be of any use without significant post-processing. The most important post-processing one would want to do

[4] The exceptions are calls that are reexecuted after a goal to their left has succeeded for the second or later time. Given the rarity in most Mercury programs of procedures that can succeed more than once, this will be a small minority of calls.

is to merge together the nodes that result from self-recursive invocations of a procedure. In a language that supports recursion but not iteration, performing this merging at runtime causes a dramatic reduction in the size of the tree. There are two reasons for this. The obvious reason is that the recursion is often quite deep, and using a single node to represent all the recursive calls eliminates the nodes that would have been there for all the recursive calls after the first. The less obvious but far more important reason is that if more than one recursive invocation makes a call through a call site within the recursive procedure, then the subtrees of those call sites will be merged as well. The sizes of those subtrees, and therefore the savings, can be huge, particularly for calls near the top of the call tree.

Merging the nodes of self-recursive calls at runtime requires treating self-recursive calls differently from other calls. The compiler will continue to insert a call to `prepare_for_normal_call` before all other calls. However, before a self-recursive call it will insert a call to `prepare_for_recursive_call` instead. This function behaves almost the same as `prepare_for_normal_call` except that if and when it initializes the `csd_callee_ptr` field of the CallSiteDynamic it has allocated, it will make it point to the ProcDynamic structure of the current invocation instead of filling it with null. This will cause the `call_port_code` of the new recursive invocation to recognize that this call does not create a new context, but is part of an existing context instead.

5.3.4 Handling mutual recursion within a module

As we discussed in section 5.2, self-recursion is not the only kind of recursion a Mercury profiler must handle. For example, in the Mercury compiler many performance-critical procedures, and some of the procedures near the top of the call tree, are members of cliques of mutually recursive procedures. Unless we merge together the nodes corresponding to invocations of procedures within a clique, the call graph constructed during a profiling run will continue to be infeasibly large for many interesting programs.

Some of these cliques consist of procedures that are all in the same module; some contain procedures from more than one module. Originally we thought that merging nodes involved in intra-module mutual recursion without also merging nodes involved in intermodule mutual recursion would make feasible the profiling of a large enough class of programs that we need not complicate the implementation any further. We therefore implemented a scheme that did this.

The basic idea of this scheme is to have the compiler identify groups of mutually recursive procedures within the module it is compiling, and to treat each of those groups as a single profiling unit. Each of these groups corresponds to a strongly connected component of the part of the call graph visible from the module, and we therefore call them SCCs. (Procedures that are not mutually recursive with another procedure in the same module each have their own SCC.)

The scheme replaces ProcStatic structures with SccStatic structures and ProcDynamic structures with SccDynamic structures. Instead of generating a ProcStatic structure for each procedure in the module, the compiler generates an SccStatic structure for each SCC in the module. SccStatic structures have the same fields as ProcStatic structures do, with one difference: the `ps_call_sites` array has an entry for all the call sites in all the procedures in the SCC. Similarly, at runtime procedure prologues fill in `current_call_site_dynamic->csd_callee_ptr` with a pointer to an SccDynamic structure, not a ProcDynamic structure. The `pd_proc_static` field of the SccDynamic structure points to the SccStatic structure of the SCC, while the `pd_call_site_ptrs` field now points to an array which has an element for every call site in the SCC, just like the `ps_call_sites` array it parallels. This time, the compiler will use `prepare_for_recursive_call` to preface not only self-recursive calls but also any call to a procedure within the current SCC. This is sufficient to eliminate the distinction between self-recursion and mutual recursion within a module.

Having produced an implementation that did this, we found that this approach did indeed reduce the size of the call graph quite substantially, reducing the memory requirement for a profiling run of the Mercury compiler to the point where it just fit into the memory of the machine we had available at the time (128 Mb). The Mercury compiler is a pretty big program: it is over a quarter of a million lines of Mercury code, and its runtime is usually five seconds to a minute.

However its normal memory consumption is usually around 20 Mb; that's when the garbage collector kicks in. We felt that having profiling increase memory consumption about five fold was excessive, and we wanted to be able to profile programs that are bigger and longer-running than the Mercury compiler. We therefore decided that we needed a method that could reuse existing nodes of the call graph even in the presence of recursion that crosses module boundaries.

5.3.5 Handling mutual recursion across modules

From the point of view of the profiler, the ideal way to handle recursion across modules would be to give the compiler access to the complete call graph of the program even when it is compiling a single module, and then use the scheme described in section 5.3.4.

Unfortunately, giving the compiler access to the complete call graph of the program is far from trivial. The reason is that the links in the call graph are subject to change as programmers edit modules. Suppose the system has the complete call graph of the program stored somewhere, and the programmer then edits two modules, modules A and B, in ways that leave their interfaces the same. The standard Mercury compilation system simply compiles first one module (say module A), then the other (module B). However, when the compiler compiles module A, it does not have access to an accurate call graph: the links in the call graph that represent calls made in module B are obsolete. The only way to fix this problem is to change the order in which the compiler performs its tasks. First, it should update the two modules' portions of the call graph. Since the identity of the called procedure in a call may depend on type and mode information as e.g. the compiler resolves overloading, this requires performing static analysis on both modules. Only then can the compiler generate code. If we (naively) divide the compiler into a front end which does static analysis and a back end that does everything else, then the old order was front end on module A, back end on module A, front end on module B and back end on module B, while the required new order is front end on module A, front end on module B, back end on module A and back end on module B. Dividing the compiler in this fashion without making compilation much more time consuming would require a considerable amount of re-engineering.

Actually, the situation is even more complicated. Suppose there is one group of mutually recursive procedures in modules A and B, and another group in modules C and D, where some procedures of the second group call procedures in the first group, but not vice versa. A modification to e.g. module A may introduce a call that merges the two previously separate SCCs into a single SCC. This will change the status of some calls in modules C and D: what were previously non-recursive calls now become recursive calls. Since the compiler must insert different code before these two kinds of calls, the change to module A will require modules C and D to be recompiled, even if the interface of module A has not changed.

Given that detecting cross-module cycles at compile time is infeasible, we have no choice but to detect them at runtime. Since our approach detects cycles without regard for whether they are cross-module or not, or even for whether they involve self-recursion or mutual recursion, we do not need the mechanisms we presented in sections 5.3.3 and 5.3.4. The starting point of our implementation is therefore the scheme we presented in section 5.3.2, which handles all calls uniformly. The schemes we presented in section 5.3.3 always reused the context (CallSiteDynamic and ProcDynamic structures) of the outermost invocation in a sequence of self-recursive calls, we now want a scheme that always reuses the context of the outermost invocation in *any* sequence of recursive calls, even if the stack frames of those recursive calls are interspersed with the stack frames of other procedures, which may be from other modules.

Our approach is based on the definition of a recursive cycle: a call is part of a recursive cycle if the called procedure is one of its own ancestors. At the time of a call, we could check whether the called procedure is one its own ancestors by walking the stacks (Mercury has two) upwards, at each step going from callee to caller, looking for the address of the outermost already active stack frame of the called procedure. Since the stacks can be very deep, searches can be quite expensive. This is especially bad for non-recursive calls, which all involve an unsuccessful search which must scan the stacks all the way back to `main/2`. (Successful searches can be short-circuited by adding extra links between stack frames.) However, even the cost of searching a few stack frames would

```
function call_port_code(proc_static_ptr, out TopCSD, out MiddleCSD, out OldOuterMost) is
    local pd
    TopCSD := current_call_site_dynamic
    MiddleCSD := next_call_site_dynamic
    current_call_site_dynamic := next_call_site_dynamic
    current_call_site_dynamic->csd_self_profile.call_count += 1
    OldOuterMost := proc_static_ptr->ps_outermost_proc_dynamic
    if current_call_site_dynamic->csd_callee_ptr is null then
        if proc_static_ptr->ps_outermost_proc_dynamic is null then
            pd := allocate_proc_dynamic()
            pd->pd_proc_static := proc_static_ptr
            pd->pd_call_site_ptrs :=
                allocate_csd_ptr_array(proc_static_ptr->ps_num_call_sites)
            current_call_site_dynamic->csd_callee_ptr := pd
            proc_static_ptr->ps_outermost_proc_dynamic :=
                current_call_site_dynamic->csd_callee_ptr
        else
            current_call_site_dynamic->csd_callee_ptr :=
                proc_static_ptr->ps_outermost_proc_dynamic
        end
    else
        if proc_static_ptr->ps_outermost_proc_dynamic is null then
            proc_static_ptr->ps_outermost_proc_dynamic :=
                current_call_site_dynamic->csd_callee_ptr
    end
end
```

Figure 5.3: `call_port_code` for the save/restore approach

be greater than our objective of keeping overheads small relative to the program itself would allow us to tolerate. We therefore detect cycles *without* searching the stacks. Instead, we add new fields to each ProcStatic structure that tell us whether execution is currently inside a call to the corresponding procedure or not. If so, these fields must also tell us where the ProcDynamic structure of the outermost invocation is.

The reason why we need to know the address of the ProcDynamic structure of the outermost invocation has to do with why we want to detect cycles in the first place. Suppose the `call_port_code` of a procedure finds, in a given invocation, that `current_call_site_dynamic->csd_callee_ptr` is null, i.e. this is the first call to this procedure in this precise context. Previously, `call_port_code` always created a new ProcDynamic structure for the call, even if an invocation of this procedure was already active among the ancestors. From now on we will avoid allocating a new ProcDynamic structure and instead reuse the ProcDynamic structure of the outermost invocation of this procedure if there is one and if `current_call_site_dynamic->csd_callee_ptr` is initially null. This merges the profiling information of this call with the outermost call of this procedure. This is an appropriate thing to do because consumers of profiling information are unlikely to care *which* iteration of a recursive loop incurs a given cost; they require only that the cost be properly charged to the loop. Making *all* calls to a given procedure in a recursive loop use the same ProcDynamic structure even if the loop extends over more than one module reduces the amount of memory needed for the call graph quite substantially, for the reasons we discussed in section 5.3.5.

We obviously need a field in ProcStatic structures (which we call `ps_outermost_proc_dynamic`) that points to the ProcDynamic structure of the outermost existing invocation of the corresponding procedure, if there is one. We also need a mechanism that tells us whether there is such an invocation. There are two obvious mechanisms we could use for detecting this.

The first mechanism (which we call 'save/restore') requires the `ps_outermost_proc_dynamic` field to be valid at all times, containing a null pointer if and only if there is no active in-

```
function call_port_code(proc_static_ptr, out TopCSD, out MiddleCSD) is
    local pd
    TopCSD := current_call_site_dynamic
    MiddleCSD := next_call_site_dynamic
    current_call_site_dynamic := next_call_site_dynamic
    current_call_site_dynamic->csd_self_profile.call_count += 1
    if current_call_site_dynamic->csd_callee_ptr is null then
        if proc_static_ptr->ps_activation_counter = 0 then
            pd := allocate_proc_dynamic()
            pd->pd_proc_static := proc_static_ptr
            pd->pd_call_site_ptrs :=
                allocate_csd_ptr_array(proc_static_ptr->ps_num_call_sites)
            current_call_site_dynamic->csd_callee_ptr := pd
            proc_static_ptr->ps_outermost_proc_dynamic :=
                current_call_site_dynamic->csd_callee_ptr
        else
            current_call_site_dynamic->csd_callee_ptr :=
                proc_static_ptr->ps_outermost_proc_dynamic
        end
    else
        if proc_static_ptr->ps_activation_counter = 0 then
            proc_static_ptr->ps_outermost_proc_dynamic :=
                current_call_site_dynamic->csd_callee_ptr
    end
    proc_static_ptr->ps_activation_counter += 1
end
```

Figure 5.4: `call_port_code` for the activation counting approach

vocation of this procedure in the chain of ancestors, and thus no outermost invocation. The `call_port_code` of each procedure stores the original value of the `ps_outermost_proc_dynamic` field in a stack slot before checking whether this value is null. If it is, then this is the outermost call to this procedure: `call_port_code` therefore allocates a ProcDynamic structure, and overwrites the `ps_outermost_proc_dynamic` field with its address after filling it in. If the initial value of the `ps_outermost_proc_dynamic` field is not null, then `call_port_code` does not allocate a ProcDynamic structure, and uses the one pointed to by the `ps_outermost_proc_dynamic` field instead. If the procedure can succeed more than once, `call_port_code` will then store the final value of the `ps_outermost_proc_dynamic` field in another stack slot. The `exit_port_code` and `fail_port_code` can then restore the field to the value it had before entry to this call, while `redo_port_code` (if it exists) can restore it to the value it had after entry to this call. We can dispense with the second stack slot if the procedure in question can succeed at most once. This scheme therefore requires between one and two stack slots per stack frame; for programs that are very close to deterministic, such as the Mercury compiler, the average number will be very close to one. Figure 5.3 shows `call_port_code` for procedures that succeed at most once; the version for procedures that can succeed more than once differs only in also returning the final value of `proc_static->ps_outermost_proc_dynamic` in an extra output argument, for use at the redo port (which procedures that succeed at most once do not have).

The second mechanism (which we call 'activation counting') involves adding a second field to ProcStatic structures, an activation counter whose value gives the number of times the corresponding procedure appears in the chain of ancestors starting at the current call. Since `call_port_code` can use the value of this counter to check whether the call is an outermost call, the `ps_outermost_proc_dynamic` field need not contain any particular value if this counter contains zero. The activation counter is of course initialized to zero, and is maintained simply by incrementing it in `call_port_code` and in `redo_port_code`, and decrementing it in `exit_port_code` and in `fail_port_code`. The point of this counter is that `exit_port_code` and `fail_port_code` need not restore the contents of the `ps_outermost_proc_dynamic` field. If the call is not an outermost call, then the `call_port_code` will not have changed the value of the `ps_outermost_proc_dynamic` field and therefore there is no change to be undone, while if the call is an outermost call, then the previous contents of the `ps_outermost_proc_dynamic` field was garbage, and the value it has now is as good 'garbage' as any to reset it to. We therefore need not save the old contents of the `ps_outermost_proc_dynamic` field in a stack slot in the first place. For nondeterministic procedures, we do still need a stack slot holding the new value of the `ps_outermost_proc_dynamic` field. To see why, consider a nondeterministic procedure `p` that contains two conjoined calls to a nondeterministic procedure `q`. If the ancestors of a given call to `p` do not include `q`, then both calls to `q` will be outermost calls. When the second call fails and execution backtracks to the first call, the `redo_port_code` in the first call would not only have to increment the activation counter from 0 to 1, it would also have to change the `ps_outermost_proc_dynamic` field of `q`'s ProcStatic structure, to make it point to the ProcDynamic structure of not the second call but the first. This can only be done if every invocation of `q` saves a pointer to the ProcDynamic structure it uses (i.e. the new value of the `ps_outermost_proc_dynamic` field in `q`'s ProcStatic structure) in a stack slot, so that it copy the value in this stack slot back to the `ps_outermost_proc_dynamic` field in the `redo_port_code`. Figure 5.4 shows `call_port_code` for this approach, again for a procedure that succeeds at most once.

Our implementation can use either approach, depending on a configuration option; we will discuss the performance implications of the choice in section 5.5. Until then, we will assume the use of the activation counters.

Regardless of which mechanism the profiler uses to detect recursive calls, it will find such calls only if the program makes them. If the code of procedure `p` contains calls to procedure `q` and vice versa, but the calls to `p` in procedure `q` are in execution paths that happen not to be taken during a particular run of the program, then the nodes of calls to procedures `p` and `q` will not belong to a cycle in the call graph constructed during that run. What the mechanisms we just described detect is therefore dynamic, actual recursion, not the static, potential recursion that the approach in section 5.3.4 was based on.

5.3.6 Preserving tail recursion

The high-level transformation we use to instrument programs we have just described works well for many programs, but adding `exit_port_code` after procedure bodies defeats last call optimization and tail recursion. For procedures that would otherwise be tail-recursive this results in $O(n)$ stack-space usage (where n is the depth of the recursion) rather than $O(1)$. This is clearly a problem for many programs; for example, for programs that use would-be tail recursive procedures to read in large input files, the sizes of the stacks limit the sizes of the input files that can be used during profiling runs.

We therefore propose that for procedures that are initially tail recursive and can succeed at most once, the profiler should use a modified transformation that preserves tail recursion. (Preserving tail recursion in procedures that can succeed more than once is not worthwhile, because the stack frames of such procedures cannot be reclaimed on success anyway; they can only be reclaimed on failure.) We restrict our attention to self-tail-recursion. We have looked into extending the technique to mutual recursion that is visible within a module, but did not find any way to do so without incurring high levels of overheads. This is not a very significant limitation, given that self-tail-recursion is much more common than mutual tail recursion. Also, turning mutual tail recursion into self-tail-recursion is usually an optimization, so performance-conscious programmers will probably not mind doing it if they need to do so in order to be able to profile their programs with realistic input data.

To preserve tail recursion, we must move `exit_port_code` out of its position at the end of a procedure, so that the tail call can be the last thing executed in the procedure body. Instead, we must perform the work of `exit_port_code` somewhere else, and that somewhere else can only be in the caller. However, we do not wish to require the compiler to know, when generating a call to a procedure, whether that procedure is tail recursive or not, since such a requirement would

```
p(...) :-
    (
        ...
    ;
        % call site #j
        p(...), % not tailcall
        ...
    ;
        ...,
        % call site #k
        p(...) % tailcall
    ).
```

Figure 5.5: Doubling transformation: before

```
p(...) :-
    (
        ...
    ;
        % call site #j
        p(...), % not tailcall
        ...
    ;
        ...,
        % call site #k
        inner_p(...) % was tailcall
    ).
```

```
inner_p(...) :-
    (
        ...
    ;
        % call site #j
        p(...), % not tailcall
        ...
    ;
        ...,
        % call site #k
        inner_p(...) % tailcall
    ).
```

Figure 5.6: Doubling transformation: after

conflict with separate compilation.

Our solution to this conflict starts with a "doubling" transformation, whose steps are all guaranteed to preserve the semantics of the program. Given a tail-recursive procedure `p`, this transformation

- invents a new predicate name, say `inner_p`, that does not occur in the rest of the program,
- defines `inner_p` as a new predicate that is identical to `p` in everything but its name,
- changes all the self-tail-recursive calls in the body of `p` to call `inner_p` instead, leaving all other calls (whether to `p` or to other predicates) unchanged,
- updates the body of `inner_p` to be identical to the new body of `p`.

We say that `inner_p` is the inner procedure of `p`, while `p` is the outer procedure of `inner_p`.

The effect of the doubling transformation is shown in figures 5.5 and 5.6. Some of the recursive calls in the original procedure `p`, such as the one at call site k, are tail recursive, and are thus redirected to call `inner_p` instead. Some, such as the one at call site j, are not, and are thus preserved unchanged.[5]

Procedures that are affected by the doubling transformation are handled specially by the deep profiling transformation. The differences from the standard transformation described in the previous sections are as follows.

[5] Since we are assuming that `p` can succeed at most once, the body of `p` is cannot be an arbitrary disjunction but must be either a switch (a disjunction in which each disjunct unifies the same input argument with a distinct function symbol) or a set of nested switches.

```
p(...) :-
    call_port_code(p_proc_static_ptr, TopCSD, MiddleCSD),
    (
        (
            ...
        ;
            ...
            prepare_for_normal_call(j),
            p(...),
            ...
        ;
            ...,
            save_recursion_depth_N(trcsn_1, ..., trcsn_N,
                SavedDepth_i, ..., SavedDepth_N),
            prepare_for_tail_call(k),
            (
                inner_p(...),
                restore_recursion_depth_exit_N(MiddleCSD, trcsn_1, ..., trcsn_N,
                    SavedDepth_i, ..., SavedDepth_N),
            ;
                restore_recursion_depth_fail_N(MiddleCSD, trcsn_1, ..., trcsn_N,
                    SavedDepth_i, ..., SavedDepth_N),
                fail
            )
        ),
        exit_port_code(TopCSD, MiddleCSD)
    ;
        fail_port_code(TopCSD, MiddleCSD), fail
    ).
```

Figure 5.7: `p` after the doubling and deep profiling transformations

- In the inner procedure, the deep profiling transformation does not add calls to `call_port_code`, `exit_port_code` or `fail_port_code`.
- In both the inner and the outer procedures, the deep profiling transformation prefaces calls to the inner procedure with a call to the new service function `prepare_for_tail_call` instead of the usual call to `prepare_for_normal_call`. This compensates for the absence of `call_port_code` in the inner procedure.

```
inner_p(...) :-
    (
        ...
    ;
        ...,
        prepare_for_normal_call(j),
        p(...),
        ...
    ;
        ...,
        prepare_for_tail_call(k),
        inner_p(...)
    ).
```

Figure 5.8: `inner_p` after the doubling and deep profiling transformations

```
function prepare_for_tail_call(call_site_num) is
    local pd, callee_csd
    pd := current_call_site_dynamic->csd_callee_ptr
    if pd->pd_call_site_ptrs[call_site_num] is null then
        callee_csd := allocate_call_site_dynamic()
        callee_csd->csd_self_profile := zero_profile()
        callee_csd->csd_callee_ptr := pd
        callee_csd->tail_recursion_depth := 0
        pd->pd_call_site_ptrs[call_site_num] := callee_csd
    else
        callee_csd := pd->pd_call_site_ptrs[call_site_num]
    end
    callee_csd->tail_recursion_depth++
    current_call_site_dynamic := callee_csd
end
```

Figure 5.9: Service functions for preserving tail recursion: `prepare_for_tail_call`

- In the outer procedure, the deep profiling transformation wraps all calls to the inner procedure in calls to `save_recursion_depth_N`, `restore_recursion_depth_exit_N` and (if the call can fail) `restore_recursion_depth_fail_N` as shown in figure 5.7, with N being the number of self-tail-recursive call sites in the original procedure (and hence the number of calls to `inner_p` in `p` and `inner_p`), and `trcsn_1`, ... `trcsn_N` being the call site numbers of those self-tail-recursive call sites. (Therefore in our example, k will be a member of the set {`trcsn_1`, ... `trcsn_N`}, but j will not be.) This additional code compensates for the absence of `exit_port_code` and `fail_port_code` in the inner procedure.

These are the only differences from the standard transformation. For example, call sites in the bodies of both the inner and outer procedure that are not calls to the inner procedure still have a call to the usual service procedure (`prepare_for_normal_call`, `prepare_for_higher_order_call` etc) inserted before them.

In cases where the original program would have had a call from (say) `s` to `p` followed by m self-tail-recursive calls from `p` to `p`, the transformed program will have a call from `s` to `p`, a call from `p` to `inner_p`, and $m-1$ self-tail-recursive calls from `inner_p` to `inner_p`. This keeps the complexity of stack memory consumption at O(1), although of course the constant is now higher. since the stack has one more frame, and the deep profiling transformation makes each frame bigger too.

Figures 5.9, 5.10 and 5.11 give the code of the three new service routines; the code of `restore_recursion_depth_fail_N` is identical to the code of `restore_recursion_depth_exit_N` except for updating the `fail_count` field instead of `exit_count`. The `prepare_for_tail_call` function performs the work of the `call_port_code`that is missing from `inner_p` as well as the work of `prepare_for_normal_call`. The functions `restore_recursion_depth_exit_N` and `restore_recursion_depth_fail_N` have the task of performing the work of the calls to `exit_port_code` and `fail_port_code`respectively that are missing from `inner_p`. `save_recursion_depth_N` exists to make the arrangement reentrant. All these service routines use a new field of the CallSiteDynamic structure, the `tail_recursion_depth` field, which is initialized to zero when the structure is allocated. Our algorithm only needs this field to be present and initialized in CallSiteDynamic structures corresponding to self-tail-recursive call sites, but of course it is simpler to make it part of all CallSiteDynamic structures.

Let's walk through what happens during a chain of tail recursive calls. Assume for now that

- the chain of tail recursive calls is not interrupted by any non-tail calls,
- `p` has only one self-tail-recursive call site, and

```
function save_recursion_depth_N(MiddleCSD, trcsn_1, ..., trcsn_N,
        out SavedDepth_1, ..., out SavedDepth_N) is
    local pd, callee_csd
    pd := MiddleCSD->csd_callee_ptr
    for i := 1 to N do
        callee_csd := pd->pd_call_site_ptrs[trcsn_i]
        if callee_csd is null then
            SavedDepth_i := 0
        else
            SavedDepth_i := callee_csd->tail_recursion_depth
            callee_csd->tail_recursion_depth := 0
        endif
    done
end
```

Figure 5.10: Service functions for preserving tail recursion: `save_recursion_depth_N`

```
function restore_recursion_depth_exit_N(MiddleCSD, trcsn_1, ..., trcsn_N,
        out SavedDepth_1, ..., out SavedDepth_N) is
    local pd, callee_csd
    pd := MiddleCSD->csd_callee_ptr
    for i := 1 to N do
        callee_csd := pd->pd_call_site_ptrs[trcsn_i]
        if callee_csd is not null then
            callee_csd->csd_self_profile.call_count += callee_csd->tail_recursion_depth
            callee_csd->csd_self_profile.exit_count += callee_csd->tail_recursion_depth
            callee_csd->tail_recursion_depth := SavedDepth_i
        endif
    done
end
```

Figure 5.11: Service functions for preserving tail recursion: `restore_recursion_depth_exit_N`

- this is the first call to `p` in the current ancestor context.

The chain will start with a call to `p` from some other procedure, e.g. `s`. The `call_port_code` of `p` will update `current_call_site_dynamic` to point to the call site from which `s` called `p`. This will still be the value of `current_call_site_dynamic` when execution reaches the call to `save_recursion_depth_N`. As per our assumption, `N` will be 1, and `trcsn_1` will be k, the call site number of the only self-tail-recursive call site in the original `p`. Since `pd->pd_call_site_ptrs[k]` is initially null, `save_recursion_depth_N` will return zero in `SavedDepth_1`. `prepare_for_tail_call` will then allocate a new CallSiteDynamic structure, and make `pd->pd_call_site_ptrs[k]` point to it, just as `prepare_for_normal_call` would. `prepare_for_normal_call` would initialize the `csd_callee_ptr` field of the new CallSiteDynamic to null, and let `call_port_code` fill it in with a valid pointer; since `inner_p` has no `call_port_code`, `prepare_for_tail_call` initializes the field with the value `call_port_code` would have assigned, the address of the ProcDynamic structure of `p` itself. (Reusing `p`'s ProcDynamic structure for `inner_p` is OK because inner/outer procedure pairs have identical structures.) Future tail recursive calls to `inner_p` will also all reuse this same ProcDynamic structure.

Every call to `inner_p` will increment the `tail_recursion_depth` field of the CallSiteDynamic structure at this ProcDynamic's k'th call site. Eventually, when the chain of tail recursive calls comes to an end, that `tail_recursion_depth` field will contain a count of all the calls to `inner_p`, the first call being from `p` and the rest from `inner_p` itself. Given a sequence of tail calls, either they all succeed or they all fail; having the caller succeed while the callee fails or vice versa is possible only for non-tail calls. This is why `restore_recursion_depth_exit_N`, which is called if the call from `p` to `inner_p` succeeds, is justified in incrementing the exit count of the call site by the number of calls through that call site, and why `restore_recursion_depth_fail_N` is similarly justified in doing the same to the fail count field. The `exit_port_code` or `fail_port_code` of `p` itself will then record the success or failure of the call to `p`.

However, when `restore_recursion_depth_exit_N`, `restore_recursion_depth_fail_N`, `exit_port_code` and `fail_port_code` are called, `current_call_site_dynamic` still points to the call site of the deepest call to `inner_p`, since `inner_p` never resets `current_call_site_dynamic`. This is why `p` passes MiddleCSD explicitly to both `restore_recursion_depth_exit_N` and `restore_recursion_depth_fail_N`, and why `restore_recursion_depth_exit_N`, `restore_recursion_depth_fail_N`, `exit_port_code` and `fail_port_code` all use MiddleCSD rather than `current_call_site_dynamic` to locate the CallSiteDynamic structure of the current call. The `exit_port_code` or `fail_port_code` of `p` will also reset `current_call_site_dynamic` to TopCSD, so the only time when `current_call_site_dynamic` does not point to the call site of the currently executing procedure invocation is during the interval from the exit or failure of the innermost call to `inner_p` and the `exit_port_code` or `fail_port_code` of `p`. The difference affects only which CallSiteDynamic structure is charged with a clock tick if and when a profiling interrupt arrives during the interval and does not affect which user-visible *procedure* is charged with the clock tick. Since which invocation of `p` or `inner_p` the profiling interrupt interrupts during a sequence of tail calls to `inner_p` is effectively random anyway, the difference does not distort the profile in any significant way.

What happens if we relax the third assumption, and suppose that this is not the first call to `p` in the current ancestor context? Mostly the same thing. The main difference is that the previous invocations of `p` will have filled in some of the slots in the array of `p`'s ProcDynamic structure, and this may have included the call site of the call to `inner_p`. If it has, then `prepare_for_tail_call` does not need to allocate a new CallSiteDynamic structure, and can reuse the existing one instead. Its fields will contain the right values: the `csd_callee_ptr` field still points to `p`'s ProcDynamic structure, and while the `tail_recursion_depth` field has been updated since its initialization, it has been reset to contain zero by `restore_recursion_depth_exit_N` or `restore_recursion_depth_fail_N`. The `csd_self_profile` field may of course have different contents, but the service functions just increment its various subfields; they do not otherwise depend on its initial content.

What happens if we relax the second assumption, and suppose that `p` (and thus `inner_p`) contains more than one call site calling `inner_p`? Then it is possible that the calls to `inner_p` in the tail-recursive chain involved more than one call site and thus more than one slot in the ProcDynamic structure's array. The various calls to `prepare_for_tail_call` will ensure that every call site that `inner_p` was called through will have non-null CallSiteDynamic pointers in their slots, and that the `tail_recursion_depth` field of each such structure records the number of times the corresponding call site was called through. `restore_recursion_depth_exit_N` or `restore_recursion_depth_fail_N` will then increment the `exit_count` or `fail_count` fields of these structures by the values of their respective `tail_recursion_depth` fields. They will then reset the `tail_recursion_depth` fields of all the filled-in slots to zero to prepare for other calls to `p`.

What happens if we relax the first assumption, and suppose that `inner_p` makes some non-tail calls? If none of these non-tail calls call `p` or `inner_p`, then the operation of `p` and `inner_p` is not affected. Since all the calls to `inner_p` are in `p` and `inner_p` by construction and all calls from `inner_p` to `inner_p` are tail calls, the only potential problem is a call in `inner_p` either directly to `p` or to another procedure, say `q`, which then calls `p`. Since the original call to `p` from `s` is still active, that call to `p` from `inner_p` or `q` will reuse the ProcDynamic structure of the call to `p` from `s`. However, if `p` calls `inner_p` through *any* of its call sites, the calls to `save_recursion_depth_N` and either `restore_recursion_depth_exit_N` or `restore_recursion_depth_fail_N` will together ensure that when this recursive invocation of `p` returns to its caller, the `tail_recursion_depth` fields of *all* the call sites that call `inner_p` in the ProcDynamic structure shared by `p` and `inner_p` are restored to the values they had at the time of the recursive call to `p`. The only change that the call to `p` can do to that array is to replace a null pointer with a pointer to a CallSiteDynamic

```
function prepare_for_higher_order_call(closure, call_site_num) is
    local pd, key, list_ptr
    pd := current_call_site_dynamic->csd_callee_ptr
    key := extract_key_from_closure(closure)
    list_ptr := pd->pd_call_site_ptrs[call_site_num]
    while list_ptr is not null do
        if list_ptr->key = key then
            next_call_site_dynamic := list_ptr->value
            return
        end
        list_ptr := list_ptr->next
    end

    next_call_site_dynamic := allocate_call_site_dynamic()
    next_call_site_dynamic->csd_self_profile := zero_profile()
    next_call_site_dynamic->csd_callee_ptr := null
    list_ptr := allocate_list_node()
    list_ptr->key := key
    list_ptr->value := next_call_site_dynamic
    list_ptr->next := pd->pd_call_site_ptrs[call_site_num]
    pd->pd_call_site_ptrs[call_site_num] := list_ptr
end
```

Figure 5.12: `prepare_for_higher_order_call`

structure whose `tail_recursion_depth`field (at the time of return) contains zero. Since a null pointer is just an optimized representation of such a CallSiteDynamic structure, this is not a semantic change at all. Therefore calls to `p` from within `inner_p` are invisible to the service functions.

5.3.7 Handling higher order calls

So far we have assumed that all calls through a given call site are to the same procedure. While this is true for call sites that correspond to first order calls, which are the vast majority of call sites in most programs, it is not true for call sites that correspond to higher order calls. The Mercury profiler supports three kinds of higher order calls: calling through procedure-valued variable, calling a typeclass method, and calling the generic unification and comparison procedures.[6]

The `pd_call_site_ptrs` field of each ProcDynamic structure points to an array that has one element for each call site in the procedure. We said in section 5.3.1 that the elements are pointers to CallSiteDynamic structures. For first order call sites, this is true. For higher order call sites, it is not. The element for a higher order call site conceptually contains a table mapping the identity of the called procedure to a CallSiteDynamic structure. In practice, we currently implement this table as a linked list of key-value pairs, with the values being pointers to CallSiteDynamic structures.

As its name suggests, we insert a call to `prepare_for_normal_call` only before first order calls. Before calls through procedure-valued variables, we insert a call to the procedure `prepare_for_higher_order_call` (see figure 5.12) instead. It is `prepare_for_higher_order_call` that searches the table for a CallSiteDynamic structure that is appropriate for the current call, and creates one if it does not exist yet.

This scheme takes advantage of the weakly typed nature of C (the language in which the runtime system of Mercury is written) to store pointers to CallSiteDynamic structures and to

[6] The fourth kind of higher order call supported by the Mercury compiler, calling a procedure implemented in the Aditi2 deductive database system (the successor of Aditi [107]), is not supported by the Mercury profiler, since that would require profiling the C++ code of Aditi2 itself.

list nodes in different elements of the same array. Each element of the `ps_call_sites` field of a ProcStatic structure has a component that specifies which kind of pointer the corresponding element in the `csd_callee_ptr` field of a ProcDynamic structure for the same procedure contains, so that the code that traverses the call graph at the end of execution in order to save its information content in a file will know what to do.

There are several reasonable choices for what one may use as the key when calling through a procedure-valued variable, which is implemented as a closure:

- The most obvious choice is the address of the called procedure. Since the program is about to call through it, using it as the key requires no additional overhead.

- We can also use the source location of the code that constructed the closure as the key, since this also uniquely identifies the called procedure.[7] If a particular higher order call site happens to be performance critical because a procedure called from it is expensive, the user may discover that more than one place in the program creates closures that, invoked at this call site, call the same expensive procedure. He or she may then want to know whether the performance characteristics of the calls resulting from the different closures differ, and if so, what those performance characteristics are. If it turns out that calls using closures constructed in one source location are always expensive but calls using closures constructed somewhere else are not, then the first source location is worth investigating but the second is not. Using source location as the key provides the information programmers need to do this.

  Closures in the Mercury implementation already include a pointer to a compiler generated static structure (called the closure's *layout* structure) that describes the types of the values stored inside the closure, since without that information the closure could not be copied. Adding an extra field to closure layout structures that uniquely identifies the location in the source where the closure was created costs a trivial amount of memory and no time. Using the address of the layout structure (which now identifies the source location that constructed the closure) as the key costs an extra load instruction, but since Mercury stores the pointer to the layout structure right next to the address of the called procedure, the probability that this load will cause an extra cache miss is reasonably small.

- The most specific choice is the address of the closure itself. This choice allows the profiler to distinguish between different closures constructed at the same source location. However, since there is no obvious method that the programmer who looks at a profile can usefully optimize some but not all closures constructed at a source code site or can even distinguish them in a meaningful fashion, we have not found the extra prevision worth the cost. The cost itself is substantial, because using closure addresses as keys puts pointers to the Mercury heap into the call graph. This means that the garbage collector must scan the call graph, which grows to be quite a large data structure; it can increase the peak resident set size of the Mercury compiler from around 20 Mb to about 34 Mb (see section 5.5.1). If we don't use closure addresses as keys, then the call graph does not contain any pointers into the Mercury heap, which means that we can put it into memory that is not scanned by the collector. Since this speeds up garbage collection by a large factor, doing without this optimization is a heavy cost.

The Mercury profiler uses the source location of the code that constructed the closure as the key, since this option gives the best tradeoff between preserving the most *useful* information and incurring the least overhead.

We handle calls to type class methods very similarly to calls through procedure-valued variables. The only difference is that the procedure `prepare_for_method_call`, to which we insert a

[7]This requires us to disable a small and rarely used optimization which reuses the contents of an old closure when constructing a new one. The non-optimized code in such cases wraps up the old closure as a hidden argument in a new closure that calls a new procedure created specifically for that closure construction site.

call before type class method calls, gets passed the address not of a closure, but of a *dictionary*, a data structure that contains pointers to the entry points of the procedures implementing the methods of the type class [53]. The code of `prepare_for_method_call` itself is identical to the code of `prepare_for_higher_order_call`, except that it uses different code to extract a key from its first argument.

The choices for keys for method calls are very similar to the choices for keys for calls through procedure-valued variables: the address of the called procedure, the id of the source location that constructed the dictionary, and the address of the dictionary itself. The issues are very similar as well, despite the more complex structure of dictionaries (they contain pointers to the dictionaries of superclasses, something for which there is no equivalent in closures), and our choice is therefore the same: we use the id of the source location that constructed the dictionary as the key.

While in Mercury, unification and comparison are defined on values of every type by default, in languages such as Haskell, they require an explicit "deriving (Eq,Ord)". Therefore it is not surprising that we handle calls to the generic unification and comparison procedures the same way as we handle typeclass method calls. Their service function, `prepare_for_special_call`, works the same way as `prepare_for_method_call`, the only difference is that it extracts the key from a dictionary structure that happens to have different format from ordinary typeclass dictionaries.

5.3.8 Handling recursion through higher order calls

Higher order calls often add an arc to the call graph that was not visible at compile time. These arcs can lead to cycles, which we handle in the way we described in section 5.3.5. The following program fragment contains a cycle that includes an arc created by a higher order call. We call such a cycle a *higher order cycle*.

```
:- type tree ---> leaf(leaf_data) ; interior(list(tree)).

:- pred traverse(tree, T, T).
:- mode traverse(in, in, out) is det.

traverse(leaf(LeafData), Acc0, Acc) :-
    process_leaf(LeafData, Acc0, Acc).
traverse(interior(Children), Acc0, Acc) :-
    foldl(traverse, Children, Acc0, Acc).

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

foldl(_, [], Acc, Acc).
foldl(Pred, [Head | Tail], Acc0, Acc) :-
    Pred(Head, Acc0, Acc1),
    foldl(Pred, Tail, Acc1, Acc).
```

The higher order cycle exists because `traverse` calls `foldl` at a first order call site while `foldl` calls `traverse` at a higher order call site. The self-recursive call in `foldl` creates a second cycle in this program fragment, but this second cycle is not a higher order cycle because it does not involve any higher order call sites. Both cycles are visible in figure 5.13, which contains the call graph created for the above program fragment by the profiler mechanisms we have described in sections 5.3.5 and 5.3.7. This graph corresponds to the programmer's expectations of what the call graph should look like.

Unfortunately, this is not always the case for the call graphs created by the profiler mechanisms we have described so far. For some programs, these mechanisms create higher order cycles where the programmer doesn't want them. As an example, consider the following program fragment, which is a simplified version of a part of the Mercury code generator.

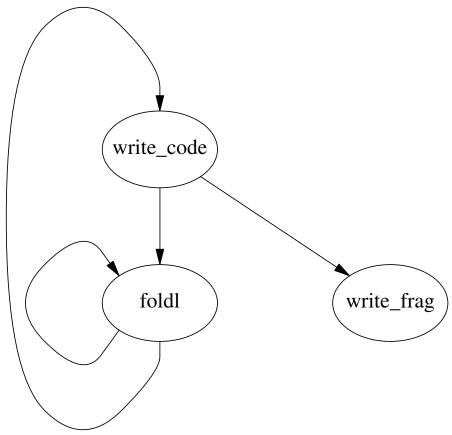

Figure 5.13: A higher order cycle

```
code_gen(conj(Goals), ...) :-
    foldl(goal_gen, Goals, ...).
code_gen(call(Call), ...) :-
    call_gen(Call, ...).

call_gen(Call, ...) :-
    ...
    foldl(save_var, Vars, ...),
    ...
```

The programmer undoubtedly meant the call to `foldl` in `call_gen` to be just a convenient way of avoiding having to write yet another predicate to loop over a list, such as `save_vars`:

```
call_gen(Call, ...) :-
    ...
    save_vars(Vars, ...),
    ...

save_vars([], ...)
save_vars([Var | Vars], ...) :-
    save_var(Var, ...),
    save_vars(Vars, ...).
```

In the programmer's mind, the version of `call_gen` calling `foldl` and the version of `call_gen` calling `save_vars` are semantically identical, and, modulo the overhead of the higher order calls made by `foldl`, they should have the same performance and result in the same profile. However, with the mechanisms we have described so far, the version calling `foldl` will result in the call graph shown in the left half of figure 5.14, while the version calling `save_vars` will result in the call graph shown in the right half of that figure. The figure on the left is clearly not what the programmer would expect.

In general, the profiler cannot be sure what the programmer would expect, since it is possible to come up with programs for which two programmers may reasonably disagree on what they expect. However, we believe that the profiler and its designers may reasonably expect consensus in a fairly large and important set of cases: the cases where higher order predicates are being used as parameterized control structures. In these cases, the programmer uses a single higher order predicate with a procedure-valued argument as a parameter instead of writing a whole collection of first-order predicates that each do the same thing for one particular value of the parameter. The

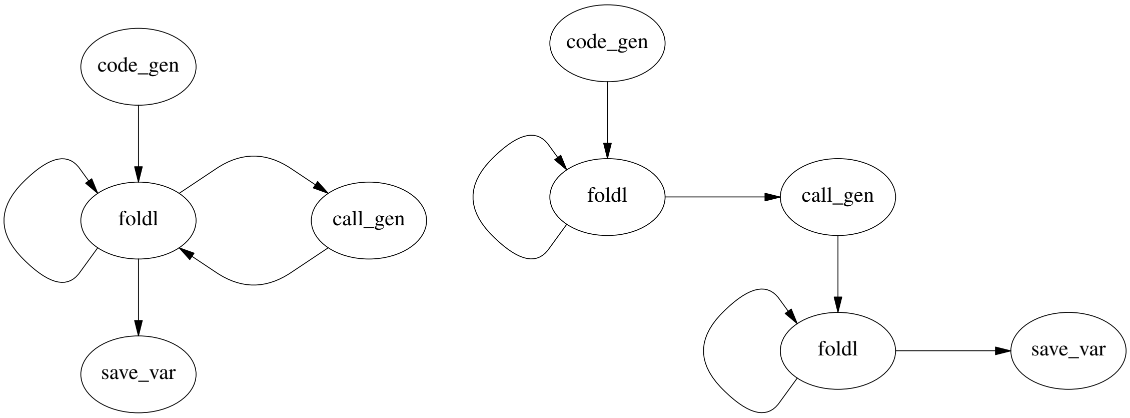

Figure 5.14: A spurious higher order cycle (*left*) and the programmer's intended interpretation (*right*)

`traverse` and `code_gen` program fragments above are both examples of this; both use a higher order predicate as a parameterized control structure. In fact, `foldl` and `map` over lists, and many other higher order predicates and functions that are commonly found in the libraries of declarative languages, are almost always used as parameterized control structures. Calls to predicates and functions such as these account for a large majority of higher order calls in most programs, so it is important for the profiler to do what programmers expect in these cases. This is to handle the program as if each instantiation of each parameterized control structure were implemented as a distinct predicate. If the transformed program fragment has a cycle, as is the case for the `traverse` example above, then we say that it has *intrinsic* recursion through a higher order call; while if the transformed program fragment does not have a cycle, as is the case for the `code_gen` example above, then we say that it has *accidental* recursion through a higher order call. We want the profiler to produce a cycle for program fragments exhibiting intrinsic recursion but not for program fragments exhibiting accidental recursion.

In fact, the Mercury implementation has long had a transformation that, given the call `foldl(save_var, Vars, ...)`, would replace it with a call such as `save_vars(Vars, ...)` after defining `save_vars` as above (although the automatically generated name of the new predicate is not as meaningful as `save_vars`). The original purpose of this optimization was to eliminate the overhead of a higher order call: whereas the original code had a higher order call to `save_var`, the code after higher order specialization has a first order call to `save_var`. However, by switching on this transformation, and one other, when generating profiling executables, we can make the call graph constructed during a profiling run contain cycles only for cases of intrinsic and not accidental recursion, at least in the absence of some rarely used programming techniques (returning higher order values from procedures and putting higher order values into data structures). The other transformation pushes the code following a branched control structure (if-then-else, disjunction or switch) into each branch if the branch instantiates a higher order variable. This turns code such as

```
p(...) :-
    ( ... ->
        P = union
    ;
        P = intersection
    ),
    <code using P>.
```

into

```
p(...) :-
    ( ... ->
        P = union,
        <code using P>
    ;
        P = intersection,
        <code using P>
    ).
```

in which each copy of the "code using P" can be effectively specialized for the relevant value of P.

One practical problem with this approach to discriminating between intrinsic and accidental recursion is that the compiler can perform the required transformation, i.e. the creation of a specialized version of the higher order predicate (which is `foldl` in both the `traverse` and `code_gen` examples) only if it has access to the source code of the higher order predicate. This is a problem, because the definition of the higher order predicate and the call to that predicate that we want to specialize are very unlikely to be in the same module, and the Mercury compiler normally has access to the source code of only one module at a time. We do have a mechanism, which we originally designed for supporting intermodule optimization, that saves some parts of the source code of a module in an optimization file associated with the module, and makes the information in that file available to the compiler when it is compiling another module. The parts of a module that are selected in this way consist of predicates that can be profitably inlined as well as higher order predicates that can be profitably specialized. Normally, a higher order predicate whose definition is above a certain threshold in size cannot be specialized *profitably*, because the increase in runtime from instruction cache misses outweighs the decrease in runtime from the elimination of the higher-order call's overhead. Since deep profiling wants to specialize higher order predicates in order to get the right call graph, it does not matter if the specialization is profitable in the sense of reducing the program's runtime, so when deep profiling is enabled, the optimization file of a module will contain the source code of *all* the higher order predicates in the module, not just the ones below the given size threshold. Doing this effectively solves the problem of distinguishing intrinsic from accidental higher order cycles for higher order predicates being used as parameterized control structures.

The problem with this solution is that it interferes with separate compilation. Normally, a change to a module requires the programmer to recompile only that module, unless the change alters the interface of the module, in which case any other module that depends on that interface must be recompiled as well. When intermodule optimization is switched on, the contents of a module's optimization file effectively becomes part of its externally visible interface. Since changes to module interfaces are relatively rare but changes to the bodies of predicates whose source code appears in optimization files are reasonably frequent, this greatly increases the average time required to rebuild the executable after a change.

Whether this is a problem depends on how many times you need to rebuild the executable with deep profiling enabled. Normally, programmers concern themselves with profiling only after the program is fairly stable, so if profiling reveals only a small number of possibilities for improvement then it is reasonably likely that the recompilation cost is acceptable. (The number of opportunities for performance improvement identified by profiling during initial development tends to have an inverse relationship with the thoroughness with which the program's designer(s) considered performance during the program's design phase.) On the other hand, if profiling reveals many possibilities for improvement, which is likely to be the case during the maintenance phase even if the original design was sufficiently performance-aware, then the cost of recompilation is likely to prove burdensome. We have therefore looked into other methods for distinguishing intrinsic and accidental higher order cycles.

One method we have looked at, which we call the signature method, aims to simulate at runtime the effect that higher order specialization would have on the call graph. The idea is that if a procedure has higher-order arguments, then we want to have separate copies of the fields of its ProcStatic structure that are used for cycle detection for each distinct combination of values of the higher-order arguments; we call each of these distinct combinations a *signature*. That way, a call to `foldl` that gives `goal_gen` as its first argument and a call to `foldl` that gives `save_var` as its first argument are guaranteed to use different ProcDynamic structures even if one call occurs inside the other.

This method incurs the overhead of searching the list of signatures on every call to a procedure with a higher-order argument which finds `current_call_site_dynamic->csd_callee_ptr` to be null. The good news about this is that recursive calls will find `current_call_site_dynamic->csd_callee_ptr` to be non-null. Unfortunately, there are two items of bad news. The first is that this cost must be incurred even for predicates that make no higher order calls but only pass around higher-order data. This is necessary to create a precise enough context to avoid the creation of unwanted cycles in the call graph; if it isn't done, then the calls `foldl(goal_gen, ...)` and `foldl(save_var, ...)` will use the same ProcDynamic structure, which will cause the profiler to build the call graph shown on the left side of figure 5.14. The second, more serious problem is that the cost of checking which signature, if any, in the ProcStatic structure corresponds to the signature of the current call can be quite high, because one cannot usefully construct a signature for a call using only the outermost code address in each closure; one must take into account all the code addresses inside the hidden arguments of closures as well.

Consider programs that use parser combinators [35]. Some of the closures that such programs often pass to e.g. the Kleene star combinator (which is similar to `foldl`) are simple closures with no higher order hidden arguments, e.g. a simple closure that parses a given terminal symbol. More often, though, they pass closures that do contain higher order hidden arguments. For example, the parser combinator for alternation takes two closures (parsers that parse e.g. different nonterminals), and yields a closure that tries first one closure and then the other. The closures of those two parsers are hidden higher order arguments in the closure representing their alternation. One call to the Kleene star combinator may have as its higher order argument a closure representing the alternation of nonterminals NT1 and NT2, while for another call the argument may be a closure representing the alternation of nonterminals NT3 and NT4. If the signature comparison looks at only the outermost code address in each higher order argument, then these two calls will be considered to have the same signature and will therefore share the same copy of the ProcStatic fields for detecting cycles. If one of those calls is inside the other, this means the two calls will be considered to be mutually recursive, which is obviously undesirable if in reality they are not. For example, an XML parser we have written using parser combinators contains about two dozen calls to the Kleene star combinator, of which about half use the same combinator as its higher order argument, but only three of these represent real cycles in the program. If we want to avoid creating artificial cycles for all the other cases, we must consider a signature to be defined by *all* the code addresses in higher order arguments, even the ones inside their hidden arguments. The problem of course is that if signature extraction requires scanning hidden arguments in their entirety, it will obviously take an unbounded and often large amount of time.

A potential solution to this problem is to store the signature of each closure in the closure itself. The signature will be a kind of cryptographic hash that is a function of all the code addresses in the closure and their positions within the closure, since we want to distinguish the closure `p(q, r)` from the closure `p(r, q)`. Associating each copy of the cycle detection fields in ProcStatic structures with a list of these signatures, one from each higher order argument would reduce the cost of searching for the right copy to a manageable level. On the other hand, it would increase the cost of constructing the closures in the first place. This is a step in the right direction, because each closure is constructed just once but tends to be passed around and called more than once. Whether it is enough to make the approach practical, on the other hand, depends on the size of the extra cost imposed on closure construction, which is proportional to the sizes of hidden arguments. As we have said before, these are unbounded in size and often large, and therefore cost is as well, which means that this "solution" is not sufficient. (We could distribute the cost throughout the program by recording a signature in *every* term of the program except first-order constants, a signature-canonicalizing instead of address-canonicalizing variant of hashconsing [33], but this would change the performance characteristics of the program being profiled beyond the bounds we consider acceptable.)

Another potential solution is to limit the scan of hidden arguments to their outermost higher-order layers. By this we mean checking the type of each hidden argument. If it is not a higher order value, then ignore it; if it is a typeclass dictionary or typeinfo structure [53], then consider its code addresses to be part of the signature; and if it is a closure, then consider its code address to be part of the signature and scan *its* hidden arguments recursively. In many programs, including the XML parser, this technique will find all the code addresses in closures that a full scan would find, but at a much lower cost. This technique can be combined with the previous one by using this algorithm to create a signature for each closure when that closure is constructed. However, for programs that create deeply nested higher order terms (the nesting of closures in our XML parser is more than 150 levels deep) and/or have lots of procedures that pass around higher order terms, the cost of even this combined method is likely to be too high.

We have therefore developed a third method, which also does not yield the right call graph for all programs, but has guaranteed low overhead and avoids interference with separate compilation. We call it the zeroing approach. The approach is based on taking the heuristic "a higher order call is very likely be to a lower SCC" seriously. Taking the heuristic as a hard-and-fast rule would require zeroing the activation counters of the current procedure and all the ancestors when making a higher order call, and restoring all those activation counters after the higher order call. The requirement for unbounded memory to store the saved activation counters and for a means to traverse the call graph upward (see section 5.3.2) makes this difficult to implement and definitely not low overhead. However, zeroing the activation counter of the current procedure is sufficient to produce the right call graph in many cases, and zeroing the activation counter of the current procedure and its immediate parent will produce the right call graph in even more cases. The reason is that zeroing the activation counter of a procedure has an effect only if the higher order call can lead to call to that procedure, and in many programs the only cycles that go through higher order calls involve only the procedure that makes the higher order call and possibly its parent but not its more remote ancestors. Both the "zeroing the caller" and the "zeroing the caller and parent" variants require only fixed amounts of storage per higher-order call, and neither requires upward links in the call graph: the caller's ProcStatic structure is reachable through the `current_call_site_dynamic` global variable in `call_port_code`. One can also imagine 'zeroing the callee" and "zeroing the callee and caller" variants as well, but these would require a simple and fast mechanism for finding the ProcStatic structure of the procedure being called, and at the moment closures and typeclass dictionaries do not contain that information.

Zeroing is compatible with the save/restore approach to recursion detection as well as the activation counter method, although in that case the field of the ProcStatic structure being saved, zeroed and restored is the `ps_outermost_proc_dynamic` field, not the `ps_activation_counter` field.

Of the three methods to handle accidental recursion: forced cross-module higher-order specialization, signature-canonicalization and zeroing, we have implemented the first and the third. Zeroing is the default because it does not change the program being profiled.

5.3.9 Handling foreign language code

As we discussed in section 5.2, we want to charge all the costs incurred by foreign language code against the nearest enclosing Mercury procedure that used the foreign language interface to enter the foreign language. To achieve this objective, we don't have to do anything special; the mechanisms we have discussed so far will do the right thing.

The Mercury-calls-foreign part of the bidirectional Mercury foreign language interface allows a programmer to define a Mercury procedure by giving its code in an imperative language supported by the Mercury implementation, typically C, like this:

```
:- pragma foreign_proc("C", math__sin(X::in) = (Sin::out),
                [will_not_call_mercury, thread_safe],
"
        Sin = sin(X);
").
```

The first three arguments of the foreign_proc declaration describe the language in which the definition is written, the procedure being defined and its arguments, and some attributes of the definition; the last gives the code that implements the procedure.

The Mercury compiler does not handle procedures defined by a foreign_proc declaration particularly specially, beyond requiring that they must not be also defined by other such declarations or by clauses. Instead, the Mercury compiler considers foreign language code fragments such as the one above to be an atomic goal type, just as unifications and calls are. Foreign language code goals may occur inside any procedure, not just those defined by foreign_proc declarations. Of course, only procedures defined by foreign_proc declarations start out containing such goals; indeed, the compiler considers foreign_proc declarations to be a kind of clause whose right hand side contains a single foreign language code goal. However, foreign language code goals may then migrate into other procedures as well through inlining and similar source-to-source transformations. Most of these transformations do not care what kinds of goals procedure bodies contain.

The transformations we have described for wrapping `call_port_code`, `exit_port_code` and possibly `fail_port_code` and `redo_port_code` around a procedure body are also oblivious to whether the procedure body contains foreign language goals or not. They set up `current_call_site_dynamic` to point to the call site from which the currently executing procedure was called, regardless of the language in which that procedure is implemented. If the procedure is implemented in foreign language code, this will cause that call site to be charged for all the costs incurred by the foreign language procedure body and by any other foreign language code called from the procedure body.

Most procedures implemented in foreign language code do not call back into Mercury code. A few, however, do. As we discussed in section 5.2, we must handle such callbacks by considering them to be calls made by the nearest enclosing procedure that used the foreign language interface to enter foreign language code; in other words, by the procedure whose call site `current_call_site_dynamic` points to when the callbacks are made. Such callbacks resemble higher order calls in that the identity of the called procedure is not apparent from the "call site", which in this case is not a higher order call but a foreign language goal. However, they differ from higher order calls in that the Mercury compiler cannot insert a call to `prepare_for_higher_order_call` before them. Not only is the callback made by foreign code which the Mercury compiler cannot meaningfully modify, but the foreign language code doing the callback may not even occur inside the Mercury source files seen by the Mercury compiler; typically, it is in another source file (e.g. tcl.c) seen only by the compiler for the foreign language. Callbacks also differ from higher order calls in that a single foreign language code goal can make more than one call to Mercury procedures before it finishes executing.

We therefore handle callbacks slightly differently from higher order calls, and split the task of updating the call graph for callbacks into two parts. The first part consists of inserting a call to a new service function before every foreign language code goal that may cause callbacks to Mercury code. (The annotations on each such goal say whether this may happen or not.) This service function, `prepare_for_callback`, set the new global variable `parent_call_site_dynamic` to point to the element in the current procedure's `pd_call_site_ptrs` array reserved for the "call site" represented by the foreign language code goal, but does not modify `current_call_site_dynamic` itself. That is left for the second part of our approach, the wrapper around the Mercury procedure called from foreign language code.

Such wrappers are an integral component of the foreign-calls-Mercury part of the bidirectional Mercury foreign language interface. This allow a programmer to take a Mercury procedure, e.g.

```
:- pred tcl_status_ok(tcl_status::in) is semidet.
```

```
tcl_status_ok(tcl_ok).
```

and make it accessible from foreign language code by providing an *export declaration* such as the following. (Currently, these are allowed only for procedures that cannot succeed more than once.)

```
:- pragma export(tcl_status_ok(in), "mtcltk_tcl_status_ok").
```

Export declarations cause the Mercury compiler to define a wrapper function in a foreign language (currently C) whose name is given by the second argument of the export declaration and whose code consists of a call to the Mercury procedure identified by the first argument of the export declaration. This wrapper function handles the marshalling of the input and output arguments, compensating for the different argument passing conventions of the languages.

The second part of our approach to handling callbacks is to get the Mercury compiler to insert extra code into wrapper functions when deep profiling is enabled. The extra code we insert before the call to the Mercury procedure searches the list of CallSiteDynamic nodes starting at `parent_call_site_dynamic` for a CallSiteDynamic node that corresponds to callbacks to this Mercury procedure. If it finds such a CallSiteDynamic node, it sets `next_call_site_dynamic` to point to it; if it doesn't, it creates a new CallSiteDynamic node, initializes it, links it into the `parent_call_site_dynamic` list, and then sets `next_call_site_dynamic` to point to it. This code is very similar to the code of `prepare_for_higher_order_call`, differing only in that we use the identity of the procedure being called as the key in the list of key-value pairs, and in using the global variable `parent_call_site_dynamic` instead of `current_call_site_dynamic->csd_callee_ptr->pd_call_site_ptrs[call_site_num]` to point to the memory location where that list is stored.

If a Mercury procedure called from C code itself contains calls to C code, then it can modify the value of `parent_call_site_dynamic`. The wrapper function therefore starts and ends with code that saves and restores the value of `parent_call_site_dynamic`.

5.3.10 Handling exceptions

Exceptions represent an unplanned return from a series of calls, starting from the procedure that throws the exception and continuing through zero or more other procedures until we come to a procedure that catches the exception. In Mercury, there is only one procedure that ever catches exceptions: the procedure `builtin_catch`. This procedure is the basic building block upon which the Mercury standard library procedure `try` and its variants (e.g. `try_io`) are built; user programs in turn must call `try` or its variants if they wish to make a call and catch any exceptions thrown by that call.

When a program compiled with debugging enabled throws an exception, the debugger synthesizes exception events for all the procedure calls that are aborted by the exception. The debugger knows what calls to synthesize exception events for by walking the stacks and creating such an event for each stack frame until it comes to the stack frame of `builtin_catch`, which it recognizes by the known return address in its callee. The deep profiler can do something similar. For each call aborted by the exception, the runtime system can do whatever `exit_port_code` or `fail_port_code` would do to restore the value `current_call_site_dynamic` had before the call and to undo whatever change `call_port_code` may have made in the called procedure's ProcStatic structure. However, instead of incrementing the counter for the number of exits (as `exit_port_code` would do) or the counter for the number of failures (as `fail_port_code` would do), we increment a counter for the number of times that control has left the procedure via an exception.

However, simulating the wrapping of procedure bodies by the deep profiling transformation is not enough; one must simulate the wrapping up of individual calls as well, for goals which are wrapped by the deep profiling transformation. This means calls that are the entry points of tail-recursive procedures (as shown in figure 5.7), and higher order calls when their implementation uses zeroing.

5.3.11 Gathering statistics

So far, we have talked exclusively about the infrastructure needed to collect profiling information during a profiling run. In this subsection we will describe what profiling information we collect.

The Mercury profiler attaches a lot more context to each measurement than most other profilers. The measurements themselves, though, are mostly conventional: counts of program counter samples, counts of memory cells allocated, counts of words of memory allocated, and they are collected by conventional means: recording program counter samples in the signal handler for the profiling signal delivered periodically to the process by the OS, and instrumenting the memory allocation mechanism. The only aspect that may be novel to some readers is that the Mercury profiler can count not only the number of times the `call_port_code` of procedure is executed in a given context, but also the number of executions of `exit_port_code`, `fail_port_code`, and `redo_port_code` as well. The reason is of course that in Mercury, some procedures can fail and some can succeed more than once, so the number of calls and the number of successful exits are not necessarily equal. The Mercury version of the invariant stating that the number of times execution enters a procedure in a given context is the same as the number of times execution leaves that procedure in that context is somewhat more complicated:

```
#call + #redo = #exit + #fail + #excp
```

Each of those five variables gives the number of times that control passes through the corresponding port. For procedures whose determinism indicates they cannot fail, `#fail` will be zero, while for procedures whose determinism indicates they cannot succeed more than once, `#redo` will be zero. (At the moment, `#excp` will *always* be zero, because the runtime system does not yet handle exceptions properly when deep profiling is enabled, and thus any program that throws an exception will get a runtime abort.)

Actually, given that the above equation is guaranteed to hold, any one of the five counts in it can be computed from the other four. Our implementation takes advantage of this fact and does not count calls; the lines of code to this effect in e.g. figure 5.4 were only for exposition.

`exit_port_code`, `fail_port_code` and `redo_port_code` do each increment their own counter in `current_call_site_dynamic->csd_self_profile`, and when implemented, the counter for exceptions will be incremented as well when control leaves a procedure via an exception. When deep profiling is enabled, the memory allocator increments two fields in `current_call_site_dynamic->csd_self_profile` at each memory allocation: it increments the number of cells allocated by one, and it increments the number of words allocated by the size of the cell in words. (The Mercury implementation never allocates fractions of words.) The signal handler for the profiling timer also increments its own field in `current_call_site_dynamic->csd_self_profile`. Thus none of the profiling measurements incurs any significant cost beyond the cost of maintaining the call graph itself.

Both the port counts and the memory allocation accounting are performed synchronously and precisely; multiple runs of the program with the same input will yield exactly the same results for them. By contrast, in our profiler as in most others, the timing data is collected asynchronously and the precise location of the timing interrupts will depend on factors external to the program. As a result, multiple runs of the program that gather timing statistics will produce slightly different results. However, provided that the program runs long enough, these variations will not be significant.

5.3.12 Creating the profiling data file

At the end of a profiling run, just before the program exits, the runtime system turns off profiling interrupts and writes out the call graph to a file. This means all the CallSiteDynamic and ProcDynamic nodes in the graph constructing during the run, and the ProcStatic and CallSiteStatic nodes corresponding to all the procedures and call sites in the program. During the profiling run, all these nodes referred to other nodes by their addresses. Since those addresses won't be the same in the profiling tool that will read in the generated profiling data file, in the process of writing out all those nodes the runtime system assigns an integer node id to all nodes and writes out pointers as the corresponding node ids. Each kind of node has its own kind of node id and a table mapping addresses to node ids, which the runtime system also uses to ensure that each node is written out only once. Actually, CallSiteStatic nodes do not need an address map table; since each CallSiteStatic structure is reachable only from exactly one ProcStatic structure, we can assign them ids that are unique by construction. Each CallSiteDynamic structure is also reachable only from exactly one ProcDynamic structure, but we have an address map table for CallSiteDynamic structures anyway in order to let us write out the nodes of the graph in a convenient order (depth-first, modified to check whether the current node has already been traversed in order to handle cliques).

Many of the fields of the nodes in the call graph contain non-negative integers. We write these integers out using a multi-byte representation in which the least significant 7 bits of each byte are data and the most significant bit is a continuation bit. With this representation, numbers up to 2^7 are represented in one byte, numbers up to 2^{14} in two bytes, and so on. As we will see in section 5.5.1, this simple form of compression is reasonably effective: node ids are dense allocated small numbers, the port counts and memory allocation counts in many CallSiteDynamic structures are small, and the vast majority of fields counting clock ticks contain zero.

We consider the root of the tree of ProcDynamic and CallSiteDynamic structures that we write out to the profiling data file to be an artificial procedure representing the Mercury runtime system, which has a single "callback" call site. Programs in which `main` is written in Mercury will execute three callbacks to Mercury code through this call site: one to initialize the Mercury standard library, one to call `main`, and one to finalize the Mercury standard library. In programs in which `main` is written in another language, the call to a Mercury `main` will be replaced by other calls to Mercury code made by the foreign language program. The same code in the Mercury runtime system finalization routine handles both kind of programs.

If we want a program to generate a consistent profiling data file, we must ensure that all exceptions are caught, even if the program itself does not catch the exception. To this end, the runtime system should catch any exceptions that propagate out of the toplevel calls it makes to Mercury code. If it did not, the profiling data structures may not be consistent when the Mercury standard library finalization code executes.

5.4 The profiling tool

5.4.1 Processing profile information

The information saved in the profiling data file is in a raw form, and much of the information it contains is implicit. For example, the CallSiteDynamic structures it contains give you how much time was spent inside the code of a given procedure when called from a given ancestor context, but they do not tell you give you how much time was spent inside the code of a given procedure *and its descendants* when called from a given ancestor context. The contents of the data file must therefore be processed before being shown to the programmer. This is done by a profiling tool.

The profiling tool starts by reading a data file created by a profiling run, and recreating a slightly modified variant of the call graph that existed in the profiled executable. Some of the modifications are omissions; the tool's version of ProcDynamic nodes have no `ps_outermost_proc_dynamic` or `ps_activation_counter` fields, since these are needed only to detect recursion in a running program. Some of the modifications are variations: nodes refer to each other using integers, not pointers. And some of the modifications are additions. In a profiled executable, only CallSiteDynamic nodes contain profiling measurements, and those measurements cover only the called procedure, not its descendants. Inside the profiling tool, both CallSiteDynamic and ProcDynamic nodes have a pair of fields whose values are derived from profiling measurements. These pairs consist of the `csd_self_profile` and `csd_descendants_profile` fields in CallSiteDynamic nodes and the `pd_self_profile` and `pd_descendants_profile` fields in ProcDynamic nodes.

The `csd_self_profile` field of a CallSiteDynamic structure `csd` contains the measurements derived from invocations of a given procedure when called from a given call site in a given ancestor context. The "given procedure" is `csd->csd_callee_ptr`. The "given call site" is the one identified by the element of the caller procedure's `ps_call_sites` array corresponding to the position of `csd` in the caller invocation's `pd_call_site_ptrs` array. The "given ancestor context" is identified by the chain of ProcDynamic ancestors of `csd` modulo SCCs. What we mean by that is the following. Start with the chain of procedure invocations that are the ancestors of the procedure invocation created at the call site represented by a given CallSiteDynamic structure. Convert this to a chain of ProcDynamic structures by replacing each procedure invocation by the ProcDynamic structure that represents that procedure invocation in the call graph. Find which groups of these ProcDynamic structures form SCCs; each SCC will correspond to a contiguous subsequence of the chain. Then replace each of these subsequences with its first element (the one closest to the root of the call graph), which represents the unique entry point to the SCC. The resulting filtered sequence is what we mean when we say "the ancestor context of the CallSiteDynamic structure".

The `pd_self_profile` field of a ProcDynamic structure contains the measurements derived from invocations of that procedure in a given ancestor context when called from *any* call site in that context. For recursive calls, the call site's CallSiteDynamic structure will be in the same ancestor context as the ProcDynamic structure; for non-recursive calls, the call site's CallSiteDynamic structure will be in a different ancestor context. In each SCC, there will be exactly one ProcDynamic structure that has a non-recursive call made to it; the CallSiteDynamic structure of this call is the *entry point* of the SCC.

The `csd_self_profile` fields of CallSiteDynamic structures contain two kinds of subfields. One kind consists of the counts giving e.g. the number of times the called procedure was called from the call site and ancestor context represented by the CallSiteDynamic structure. The other kind consists of the amount time spent and the numbers of cells and words allocated in the body of the called procedure when called from that call site and ancestor context. The `csd_descendants_profile` fields of CallSiteDynamic structures, which are intended to hold the total costs incurred by all the descendants of the called procedure, contain only the second set of subfields: sums of call counts, exit counts etc derived from many different procedures are not very meaningful, and they don't help programmers either understand or optimize their programs. Alternatively, one can view `csd_descendants_profile` fields as having their call, exit, fail, redo and excp counts hardwired to zero and unaffected by updates.

The `pd_self_profile` and `pd_descendants_profile` fields of ProcDynamic structures are analogous to the `csd_self_profile` and `csd_descendants_profile` fields of CallSiteDynamic structures. The `pd_self_profile`, and `pd_descendants_profile` and `csd_descendants_profile` fields all have their subfields initialized to zero before being filled in by the profiling tool's propagation algorithm, which we show in figure 5.15.

The first step of the algorithm fills in the `pd_self_profile` field of each ProcDynamic structure with the sum of the `csd_self_profile` fields of all the CallSiteDynamic structure that point to that ProcDynamic structure; these are all the call sites that call the procedure in the ancestor context represented by the ProcDynamic structure. Once all the `pd_self_profile` fields have been computed, they effectively become read-only.

The rest of the algorithm is concerned with filling in the `csd_descendants_profile` and `pd_descendants_profile` fields in the call graph. Since we are propagating information from descendants to ancestors, we must process ProcDynamic structures in a bottom-up order. To help us find the bottom-up order, we build a graph G whose vertices are ProcDynamic structures, and which contains an arc from one ProcDynamic structure to another (e.g. p1 to p2) if a procedure invocation whose ProcDynamic structure was p1 made a call to another procedure invocation whose ProcDynamic structure was p2. (The inner foreach ignores slots in the `pd->pd_call_site_ptrs` array that contain null pointers, but it loops over all the ProcDynamic structures that may be pointed to by slots that correspond to higher order call sites, typeclass method call sites, generic unifications or comparisons or foreign language code.) We represent G as a set of these directed arcs, and use a topological sort to find the cliques in the graph. Each clique is a set containing the ProcDynamic structures of mutually recursive procedure invocations. The topological sort

```
local G : set(arc(ProcDynamic, ProcDynamic))
local T : sequence(set(ProcDynamic))
local sum : profile_info

foreach csd: csd is a CallSiteDynamic
    csd->csd_callee_ptr->pd_self_profile += csd->csd_self_profile
done

G := {}
foreach pd : pd is a ProcDynamic
    foreach csd : csd is a CallSiteDynamic in pd->pd_call_site_ptrs
        insert arc from pd to csd->csd_callee_ptr into G
    done
done
T := topological_sort(G)

foreach clique in T, in bottom-up order
    let e be the CallSiteDynamic that is the entry point to clique
    sum := 0
    foreach pd in clique
        foreach csd : csd is a CallSiteDynamic in pd->pd_call_site_ptrs
            if csd->csd_callee_ptr is in a lower clique than pd then
                pd->pd_descendants_profile +=
                    csd->csd_self_profile + csd->csd_descendants_profile
            fi
        done
        sum += pd->pd_self_profile + pd->pd_descendants_profile
    done
    e->csd_descendants_profile := sum - e->csd_self_profile
done
```

Figure 5.15: Profile measurement propagation algorithm

returns these cliques in bottom-up order, which means that the ProcDynamic structures in each clique may contain arcs to ProcDynamic structures in earlier cliques but not in later cliques. This means that the cliques of leaf procedures will be near the front of the sequence, while cliques close to the program entry point will be near the tail of the sequence.

The main loop of the algorithm processes each clique in bottom-up order. Its job is to transmit profiling information from lower cliques to higher ones through the `csd_descendants_profile` fields of the CallSiteDynamic structures representing calls from higher cliques to lower ones. The `csd_descendants_profile` fields of CallSiteDynamic structures representing recursive calls (where the caller and the callee use ProcDynamic structures that are in the same clique) are not touched by the algorithm and they remain zero. This is because we don't want to propagate e.g. times from one ProcDynamic in a clique to another, because we don't maintain records about ancestor/descendant relationships within cliques. (The whole point of detecting recursion is to discard this information and thereby reduce the size of the call graph to a manageable level.) Instead, for each procedure in the clique, we record the time spent in its body (in the `pd_self_profile` field) and inside the call trees of the calls it makes to procedures in lower cliques (in the `pd_descendants_profile` field). We ignore the time spent inside the call trees of calls to procedures in the same clique, since that time will be accounted for when we add up the `pd_self_profile` and `pd_descendants_profile` fields of all the ProcDynamic structures in the clique. (We do the same thing for memory allocation information, but we will focus on times in our exposition.)

The innermost loop computes, for each ProcDynamic structure in the clique, the time spent inside the call trees of calls to procedures in lower cliques from that procedure. The middle loop and the two assignments flanking it are there to compute the time spent in the clique as a whole. This consists of the sum of all the times spent in the call trees of calls to lower cliques by procedures in this clique, and the sum of all the times spent in the bodies of procedures in this clique. This is the cost we want to transmit through the CallSiteDynamic structure of the entry point of this clique (represented by the variable `e`) to the higher level clique containing the procedure that made that call. However, a part of this cost (usually a small part) is already accounted for by the `csd_self_profile` field of `e`, which contains the time spent inside the body of the entry procedure when invoked from the entry point call site. This time, `e->csd_self_profile`, has already been included in `e->csd_callee_ptr->pd_self_profile`, which is `pd->pd_self_profile` for the entry procedure in the clique and which thus has already been included in `sum`. To maintain the invariant that all measurements below a cross-clique call site are included exactly once in the profiling information associated with that call site, we must therefore subtract `e->csd_self_profile` from `sum` before assigning it to `e->csd_descendants_profile`.

The cliques that this algorithm deals with are cliques in a graph of ProcDynamic structures, not cliques in a graph of procedures. This difference can affect what the programmer sees, because in some rare cases, a clique may contain two or more ProcDynamic structures that represent the same procedure. In the absence of recursion through higher-order procedures and zeroing, this procedure cannot be the entry procedure; that procedure is marked as being active for the entire duration that control resides in the clique, and thus any call to that procedure inside its own call tree will reuse the ProcDynamic structure of the entry call. However, in a clique containing e.g. procedures `p` and `q` in which the entry procedure is `p`, it is possible that `p` calls `q` from more than one call site, and that all calls to `q` from one call site have returned by the time `p` calls `q` from another call site. Since `q` would not be marked as active at that time, the algorithms of section 5.3 would create a new ProcDynamic structure for it.

Programmers may be interested in differentiating the performance measurements of `q`'s two call sites only in some cases, but not all. Therefore it would be useful for the profiling tool to optionally run a pass that merges the information in the two ProcDynamic structures. This would require not only replacing the two ProcDynamic structures with one containing the sum of the profiling measurements in the two originals, but also similarly merging all their corresponding descendants, down to the bottom of the graph. This may require not just merging the cliques of the ProcDynamic structures being merged, but also turning across-clique calls into within-clique calls. This is because our cliques represent actual, not potential recursion. If we want to merge two different ProcDynamic structures representing calls to procedure `p`, it may be that both invocations of `p` called `q`, but only one of those calls to `q` resulted in a call back to `p`. The merged ProcDynamic structure of the two calls to `p` will be in the same clique as the merged ProcDynamic structure of the two calls to `q`, which leaves one of the two calls from `p` to `q` within-clique but changes the other from an across-clique call into a within-clique call.

5.4.2 Viewing profile information

The annotated call graph produced by this algorithm is usually very big: hundreds, thousands or even tens of thousands of pages. Writing it to a file and letting the programmer look around in that file would not help programmers very much, since the call graph is useful to programmers only if they can navigate around in it easily. This is exactly what web browsers are for. We have therefore implemented our profiling tool as a server accessed from the Web through a CGI script.

Programmers can explore the information contained in a profiling data file by giving their Web browser a URL of the form http://*servername*/cgi-bin/mdprof?*datafilepathname*. This URL will cause the web server to execute the `mdprof` CGI script. The script will check whether an `mdprof` server process already exists for the named data file, and if not, it will start one. It will then send a request to the server process, relay the response to the web server, and exit.

The server process starts by reading in the named profiling data file, propagating times from descendants to ancestors using the algorithm we have just discussed and creating several kinds of data summaries (see below). It then goes into a loop waiting for and servicing requests of various kinds and returning responses as HTML pages with embedded links that, when followed,

Source	Procedure	Calls	Exits	Fails	Redos
Mercury runtime:0	mercury_compile:main/2-0	1	1	0	0
mercury_compile.m:132	mercury_compile:process_all_args/4-0	1	1	0	0
mercury_compile.m:256	mercury_compile:process_args/4-0	1	1	0	0
mercury_compile.m:383	mercury_compile:process_arg_list_2/4-0	1	1	0	0
mercury_compile.m:392	mercury_compile:process_arg/4-0	1	1	0	0
mercury_compile.m:466	mercury_compile:process_module/7-0	1	1	0	0
mercury_compile.m:528	mercury_compile:compile_all_submodules/5-0	1	1	0	0
mercury_compile.m:549	list:foldl/4-0	1	1	0	0
list.m:1144	mercury_compile:compile/4-0	1	1	0	0
mercury_compile.m:603	mercury_compile:mercury_compile/3-0	1	1	0	0
mercury_compile.m:744	mercury_compile:backend_pass/7-0	1	1	0	0
mercury_compile.m:1541	mercury_compile:backend_pass_by_preds_2/8-0	1	1	0	0
mercury_compile.m:1606	mercury_compile:backend_pass_by_preds_3/10-0	220	220	0	0
mercury_compile.m:1630	mercury_compile:backend_pass_by_preds_4/11-0	220	220	0	0
mercury_compile.m:1697	optimize:proc/5-0	220	220	0	0
optimize.m:59	optimize:repeat/11-0	220	220	0	0
optimize:repeat/11-0		529	529	0	0
optimize.m:169	optimize:repeated/12-0	526	526	0	0
optimize.m:172	optimize:repeat/11-0	309	309	0	0

Figure 5.16: A typical clique's web page: port counts

generate other requests. It exits when it hasn't received a request in a while. (The timeout period is configurable; the default is half an hour.) Any request received after that time will cause the server to be restarted, which incurs a small but noticeable delay (see section 5.5.2 below).

The CGI script and the server program communicate via a pair of named pipes. The names of these named pipes contain the (encoded) name of the profiling data file that the server process is serving requests for, and so do the links embedded in the HTML pages returned by the server. This allows a single web server to support several programmers viewing profiles at the same time, whether they are viewing different profile data files or the same ones.

The first page of a profile viewing session, the one identified by the starting URL above, gives the programmer a menu of choices. Some of the choices available include lists of the most expensive procedures when measured by several criteria, including time taken by the body of the procedure, time taken by the call tree of the procedure (self and descendants), memory allocated taken by the body of the procedure, and memory allocated taken by the call tree of the procedure (self and descendants). These tell the programmer where the hot spots of the program are. However, the choice that is most characteristic of the deep profiler is exploration of the call graph one clique at a time. Figures 5.17 and 5.16 show what the web page of a clique looks like. In this case, the profiled program was the Mercury compiler itself, and the page contains the clique of a call to `optimize:repeat/11-0` [8], the procedure in the optimizer of the Mercury compiler whose job it is to iterate the application of a set of low-level code improvement algorithms to the generated code until the iteration reaches either a fixpoint or a limit on the number of iterations permitted. The iteration is useful because some optimizations introduce code sequences that can be further improved by other optimizations.

Besides the headers and column headings, a clique's page contains two parts. The top part gives all or part of the ancestor context: the filtered chain of ancestor call sites, i.e. the entry call sites of all the ancestor cliques of this clique, in order. each line gives the location of a call and the procedure being called. In this case, the top part is most of the page. Most of the

[8] In this name, `repeat` is the name of the predicate or function, `optimize` is the module name, 11 is the arity and 0 is the mode number.

Source	Procedure	Self quanta	% of root	Total quanta	% of root
Mercury runtime:0	mercury_compile:main/2-0	0	0.00	1,384	100.00
mercury_compile.m:132	mercury_compile:process_all_args/4-0	0	0.00	1,382	99.86
mercury_compile.m:256	mercury_compile:process_args/4-0	0	0.00	1,382	99.86
mercury_compile.m:383	mercury_compile:process_arg_list_2/4-0	0	0.00	1,382	99.86
mercury_compile.m:392	mercury_compile:process_arg/4-0	0	0.00	1,382	99.86
mercury_compile.m:466	mercury_compile:process_module/7-0	0	0.00	1,361	98.34
mercury_compile.m:528	mercury_compile:compile_all_submodules/5-0	0	0.00	1,361	98.34
mercury_compile.m:549	list:foldl/4-0	0	0.00	1,361	98.34
list.m:1144	mercury_compile:compile/4-0	0	0.00	1,361	98.34
mercury_compile.m:603	mercury_compile:mercury_compile/3-0	0	0.00	1,276	92.20
mercury_compile.m:744	mercury_compile:backend_pass/7-0	0	0.00	467	33.74
mercury_compile.m:1541	mercury_compile:backend_pass_by_preds_2/8-0	0	0.00	461	33.31
mercury_compile.m:1606	mercury_compile:backend_pass_by_preds_3/10-0	1	0.07	460	33.24
mercury_compile.m:1630	mercury_compile:backend_pass_by_preds_4/11-0	0	0.00	459	33.16
mercury_compile.m:1697	optimize:proc/5-0	0	0.00	77	5.56
optimize.m:59	optimize:repeat/11-0	0	0.00	63	4.55
optimize:repeat/11-0		0	0.00	63	4.55
optimize.m:169	optimize:repeated/12-0	1	0.07	63	4.55
optimize.m:172	optimize:repeat/11-0	0	0.00	0	0.00

Figure 5.17: A typical clique's web page: clock ticks

ancestors of `optimize:repeat` are in the source file mercury_compile.m. The top-level ancestor of `optimize:repeat` is course `main`, which calls `process_all_args` at line 132, which calls `process_args` at line 256, and so on. Eventually, `optimize:proc` calls `optimize:repeat` at line 59 of optimize.m; this is the entry point of the clique. Each line in the ancestor list represents of an ancestor clique, the first two columns giving the location of the call that represents entry to the clique and the name of the procedure called from there. In live web pages, the procedure name itself is a link leading to the web page of that clique.

The bottom part of a clique's page contains the actual profiling information about the clique. This part contains as many blocks as there are procedures in the clique; usually one, but occasionally more. The block of the entry procedure is always first. Each block starts with a summary line, giving the cost of the procedure in this ancestor context. In this case, that procedure is `optimize:proc`. The rest of the block lists all the call sites in the procedure that were actually called through in this ancestor context. For first order call sites, there will be a single line giving the costs of the call site. For call sites corresponding to higher order calls, method calls and callbacks, there will be one line giving the cost of the call site as a whole, and one or more lines giving a breakdown of this cost based on the identity of the called procedure. In this case, there are only two call sites, both first order. The first is to a `optimize:repeated`, a procedure in a lower clique, while the second call is self-recursive. In the web version of this page, the name `optimize:repeated/12-0` is a link to the page of the lower clique, while the second one is not a link at all. We could have made it a link that leads back to the same page, but such links could confuse the person viewing the profile, and in any case we have found the distinction between the appearance of links and non-links to be a useful indication of the distinction between non-recursive and recursive calls. The name `optimize:repeat/11-0` in the summary line, where it spans the first two columns, is a link, but not to a clique page; it leads to the page summarizing the costs of *all* calls to `optimize:repeat/11-0`, not just the one from this call site and ancestor context. At the moment, these pages look like the bottom parts of clique pages; eventually we plan to add to them a mechanism to allow viewers to browse all the cliques containing the summarized procedure.

Source	Procedure	Calls	Exits	Fails	Redos
Mercury runtime:0	mercury_compile:main/2-0	1	1	0	0
mercury_compile.m:132	mercury_compile:process_all_args/4-0	1	1	0	0
mercury_compile.m:256	mercury_compile:process_args/4-0	1	1	0	0
mercury_compile.m:383	mercury_compile:process_arg_list_2/4-0	1	1	0	0
mercury_compile.m:392	mercury_compile:process_arg/4-0	1	1	0	0
mercury_compile.m:466	mercury_compile:process_module/7-0	1	1	0	0
mercury_compile.m:528	mercury_compile:compile_all_submodules/5-0	1	1	0	0
mercury_compile.m:549	list:foldl/4-0	1	1	0	0
list.m:1144	mercury_compile:compile/4-0	1	1	0	0
mercury_compile.m:603	mercury_compile:mercury_compile/3-0	1	1	0	0
mercury_compile.m:620	mercury_compile:frontend_pass/8-0	1	1	0	0
mercury_compile.m:1097	mercury_compile:frontend_pass_2_by_phases/5-0	1	1	0	0
mercury_compile.m:1239	mercury_compile:detect_cse/6-0	1	1	0	0
mercury_compile.m:1771	cse_detection:detect_cse/4-0	1	1	0	0
cse_detection.m:50	cse_detection:detect_cse_in_preds/5-0	1	1	0	0
cse_detection.m:69	cse_detection:detect_cse_in_procs/6-0	1,978	1,978	0	0
cse_detection.m:77	cse_detection:detect_cse_in_proc/6-0	286	286	0	0
cse_detection.m:192	cse_detection:detect_cse_in_goal_1/7-0	293	293	0	0
cse_detection.m:479	cse_detection:common_deconstruct_2/7-0	25	13	12	0
cse_detection.m:491	switch_detection:find_bind_var/9-0	40	40	0	0
switch_detection.m:455	switch_detection:find_bind_var/11-0	40	40	0	0
switch_detection:find_bind_var/11-0		72	72	0	0
switch_detection.m:488	switch_detection:conj_find_bind_var/11-0	29	29	0	0
switch_detection.m:497	term:apply_rec_substitution/3-0	34	34	0	0
switch_detection.m:500	term:apply_rec_substitution/3-0	34	34	0	0
switch_detection.m:503	Unify for term:var/1	34	28	6	0
switch_detection.m:505	higher_order_call	28	28	0	0
	cse_detection:find_bind_var_for_cse/7-0	28	28	0	0
switch_detection.m:507	hlds_goal:conj_list_to_goal/3-0	28	28	0	0
switch_detection.m:516	det_util:interpret_unify/4-0	10	10	0	0
switch_detection:conj_find_bind_var/11-0		32	32	0	0
switch_detection.m:542	switch_detection:find_bind_var/11-0	32	32	0	0
switch_detection.m:546	switch_detection:conj_find_bind_var/11-0	3	3	0	0

Figure 5.18: A more complex clique's web page: port counts

Figure 5.18 shows the web page of a clique that is more complex in several respects. First of all, the clique contains two procedures, not one. `find_bind_var/11` has 40 calls from `find_bind_var/9` in the parent clique and 32 mutually recursive calls from `conj_find_bind_var/11`, for a total of 72; `conj_find_bind_var/11` has 29 mutually recursive calls from `find_bind_var/11` and 3 self-recursive calls from itself, for a total of 32. `conj_find_bind_var/11` contains a higher order call site, although in this case all the closures that it invokes were constructed at the same source location and call the same procedure, so the call site's summary line doesn't help much. The call counts of the various call sites in `find_bind_var/11` differ; if one wanted to, one could use the actual call counts of the procedures in the conditions and in the then-parts and else-parts of if-then-elses to find out how often the conditions of those if-then-elses succeed. This case also shows that `find_bind_var/11` calls `apply_rec_substitution/3` from two sites, one on line 497 and one on line 500, and that the deep profiling keeps separate statistics for the two call sites (although in this case the port counts of the two call sites happen to be identical.)

Figure 5.19 shows the web page of the 'clique' of the artificial "procedure" representing the Mercury runtime system, in its role as the parent of `main/2`. This "procedure" is the root of the

```
Source     Procedure                   Calls Exits Fails Redos    Self  % of       Total  % of
                                                                 words  root       words  root

<Mercury runtime>                          0     0     0     0       0   0.0 82,906,281 100.0

runtime:0 callback                         3     3     0     0       0   0.0 82,906,281 100.0
          io:init_state/2-0                1     1     0     0       0   0.0         23   0.0
          mercury_compile:main/2-0         1     1     0     0       0   0.0 82,906,258 100.0
          io:finalize_state/2-0            1     1     0     0       0   0.0          0   0.0
```

Figure 5.19: The web page of the call to main/2: port counts and memory words

call graph; it therefore has no ancestors and has zero counts for all its ports. As we discussed in section 5.3.9, it contains a single callback call site through which it makes its calls, not just to `main/2` but also to the library initialization and finalization predicates. The profile shows that it calls each of these three procedures once, that all those calls are successful, that none of the called procedures allocate any memory themselves, that the descendants of the library initialization predicate allocate very little memory and the descendants of the library finalization predicate none at all.

Our system can display five different kinds of measurements for each procedure and call site. These are

- port counts (counts of calls, exits, fails and redos, and eventually exceptions)
- clock ticks
- time (clock ticks multiplied by the time taken by a clock tick)
- memory allocations
- memory words allocated

For each measurement, our system will display four columns. The port counts take four columns because there are (at the moment) four ports, and each has a column for its count. For clock ticks, the four columns are the number of clock ticks in the procedure body (Self), the number of clock ticks in the procedure body and its descendants (Total), and those two numbers expressed as percentages of the total number of clock ticks taken by the program. The columns of the last three measurements are analogous to those of clock ticks. The profiling data we display are taken directly either from the relevant CallSiteDynamic structure's `csd_self_profile` and `csd_descendants_profile` fields or from the relevant ProcDynamic structure's `pd_self_profile` and `pd_descendants_profile` fields, the only difference being that instead of displaying e.g. the time spent in descendants alone we display the time spent in self and descendants, since this is what programmers are interested in most of the time.

Twenty columns of profiling information is a lot: it usually makes web pages too wide to fit on the screen, especially if preceded by some long procedure names. Therefore we have made the set of measurements displayed configurable, with the default set being port counts, clock ticks and words allocated. (Displaying clock ticks instead of times helps remind programmers about the granular nature of timing measurements: it is much harder for a programmer to mistakenly attribute undue precision to "one clock tick" than to "ten milliseconds".) Programmers can toggle the display of each measurement via links in a footer at the bottom of each page. These work by asking the server to send a variant of the current page with a different set of measurements: the set of measurements to display is encoded in the URL. The URLs embedded in the body of each page generated by the server pointing to other pages (e.g. to other cliques) preserve the URL component controlling measurement selection, so that programmers' expressions of preference are persistent: having once chosen what set of measurements they want to see, they see that set of measurements in all the pages they call for until they change their preferences again. We also this mechanism to let users give a limit on the number of ancestor lines displayed on a page. Figure 5.17 shows what a page looks like when displaying only clock ticks and showing all the ancestors; figure 5.16 shows what a page looks like when displaying only port counts and showing at most ancestors (the ... indicates the existence of more ancestors). A typical page would display more than one set of measurements, but while it would fit on a display, it wouldn't fit on a paper page this wide.

As happens quite often, the port counts of figure 5.16 tell a useful tale, even though all the procedures involved are deterministic and therefore have exactly one exit for each call. Since `optimize:proc` is called exactly once on each procedure in the module being compiled and there are 220 calls to `optimize:proc`, the module being compiled must contain 220 procedures. There are also 220 calls to `optimize:repeat` from `optimize:proc` at line 59 in the source file optimize.m. The last line of the page shows that `optimize:repeat` also calls itself recursively 309 times from optimize.m:172, which is why the summary line reports a total of 529 calls to `optimize:repeat`. The number of calls to `optimize:repeated` is 526. Since the body of `optimize:repeat` invokes `optimize:repeated` unless the iteration has reached the limit on the number of iterations permitted, the difference between 529 and 526 tells us that we reached this limit for only three of the 220 procedures in the module being compiled. Increasing the iteration limit would therefore increase the runtime of the compiler only minimally. Reducing the limit, on the other hand, may reduce the compiler's runtime by some fraction of the 4.6% that `optimize:repeat` now accounts for. The call counts imply that each invocation of `optimize:repeat` performs an average of 529/220 = 2.4 iterations. Reducing the maximum number of iterations to 2 would reduce the time spent in `optimize:repeat` by at least the fraction 0.4/2.4 = 0.17, which would reduce the runtime of the compiler by 0.17 * 4.5% = 0.8%. (The average number of iterations cannot be higher than the maximum number, but it can be lower, since some procedures' code cannot be improved at all; in such cases `optimize:repeat` reaches fixpoint after one iteration.) Whether a 0.8% saving in compilation time is worthwhile depends on how much one values compilation time compared to the speed of the generated object code, but the deep profiler gives programmers the data they need to make an informed choice without (in this case) the need to run experiments.

5.4.3 Security considerations

While we expect that most users of the Mercury deep profiler will use the web server on their own machine, the nature of the web means that the `mdprof` server running on one machine can be accessed from anywhere on the internet. This fact requires us as the designers of `mdprof` to ensure that it cannot be used to subvert the security of the system it is running on. In our design of `mdprof`, we addressed the three most important aspects of security: unauthorized alteration of data, unauthorized disclosure of data, and denials of service.

We ensure that `mdprof` cannot be used to modify the data stored on the machine it is running on in any way that would be detrimental to its operation by not opening any file for writing, in either the CGI script or the server process, except the named pipes that the two programs use to communicate with each other. These reside in a directory designated for public use (/var/tmp), whose contents therefore should not be depended on by any security-critical system service. The names of the named pipes themselves include the string "mdprof", which makes it highly unlikely that the act of creating the named pipes will accidentally overwrite another program's data.

The CGI script and the server also do not open any file for reading, except the named pipes and the data file named in the URL. If the data file does not start with the magic cookie that profiled executables put at the start of profile data files, the server exits after printing an error message without revealing any of the file's contents. We generate the same error message if the named file does not exist, so one cannot use the text of the error message to check for the existence of files. The server process will not be able to open the file if its permissions, or the permissions of its parent directories, deny access to the user id under which the web server executes CGI scripts, which in practice means that the server can open only world-readable files. An outsider can therefore use `mdprof` to gain access to the content of a file only if that file is a world-readable profiling data file and he or she already knows its pathname.

The upper bound on the amount of computation that an invocation of the CGI script can do is very small; the script contains no loops (recursive predicates) at all, except those that iterate over (parts of) the URL. The only way to mount a denial of service attack by causing the CGI script to consume machine resources is therefore saturating the network over which requests for web services arrive. Since whether an attacker can perform such a saturation is independent of whether the machine in question has `mdprof` installed, the CGI script does not increase the risk of a denial of service attack.

The server process is a different question. Once it has been created, it stays around until its timeout expires. A dormant server process does not consume CPU time, but it does consume significant amounts of virtual memory. An attacker who can create server processes at will would therefore be able to run the target machine out of virtual memory. There are two defenses against such attacks. First, the server process will exit immediately if cannot open the specified file during its initialization, or if the file's format does not match the format expected of profiling data files, thus tying down only minimal VM resources for a minimal time. Since the CGI script ensures that at most one server process at a time exists for each profiling data file, the number of server processes an attacker can create is limited by the number of profiling data files already on the filesystem of the target machine whose filenames the attacker can discover or guess. In the absence of a previous penetration of security, this will be a very small number. If this is not sufficient reassurance, one can also employ a second line of defense, which is to give the CGI script a limit of `mdprof` N servers, and make it refuse to create new ones if N servers are already active.

5.5 Performance evaluation

There are two aspects of performance one must consider in the design of a profiling system. First, how does enabling profiling affect the performance of user programs? Second, what is the performance of the profiling tool that analyzes profiling information and presents it to the programmer? We answer these two questions in the two subsections of this section. All the benchmark results we report were obtained on a Gateway 5300 XL laptop with a 700 MHz Pentium III CPU and 256 Mb of memory, running Linux 2.2.18pre21.

5.5.1 Performance of profiled executables

To gauge the performance impact of enabling deep profiling, we have compiled the same Mercury program with eleven different sets of different compilation options. The results are reported in table 5.1. Each row of table 5.1 reports on one set of compilation options and gives the speed and size of the resulting executables. The Mercury program happens to be the Mercury compiler itself; during the timing runs, its task was to compile to C code the largest Mercury module we have access to, make_hlds.m (a part of the Mercury compiler), which contains 7958 lines.

The meanings of the columns specifying the compilation options are as follows.

grade The Mercury runtime system and execution algorithm have several optional features, such as debugging, gprof style profiling and deep profiling. The grade selects a subset of these features, giving the executable access to the features it needs without incurring overhead to support the features it does not need. The basic grade is asm_fast.gc. It enables the use of the Boehm garbage collector but no other optional features, and is optimized for speed. The grade asm_fast.gc.debug allows the program to be debugged (similar to the -g switch on many C Unix compilers). The grade asm_fast.gc.prof prepares the program for time profiling using the old Mercury profiler `mprof`, which is a clone of gprof. The grade asm_fast.gc.memprof prepares the program for memory profiling, making the executable record, at every memory allocation, the identity of the procedure that performs the allocation. The grade asm_fast.gc.profdeep prepares the program for deep profiling using the algorithms discussed in this paper.

#	grade	recursion detection	preserve tail rec.	h.o. zero	h.o. spec.	run time	run time no write	executable size
1	asm_fast.gc	N/A	N/A	N/A	default	16.6s	16.7s	6,009 kb
2	asm_fast.gc	N/A	N/A	N/A	maximum	16.5s	16.5s	6,095 kb
3	asm_fast.gc.debug	N/A	N/A	N/A	default	38.8s	39.0s	34,266 kb
4	asm_fast.gc.prof	N/A	N/A	N/A	default	28.4s	28.5s	9,501 kb
5	asm_fast.gc.memprof	N/A	N/A	N/A	default	88.7s	88.8s	9,099 kb
6	asm_fast.gc.profdeep	save/restore	yes	yes	default	44.7s	40.4s	13,719 kb
7	asm_fast.gc.profdeep	save/restore	no	yes	default	45.0s	40.4s	12,615 kb
8	asm_fast.gc.profdeep	activation count	yes	yes	default	45.3s	40.7s	13,381 kb
9	asm_fast.gc.profdeep	activation count	no	yes	default	45.3s	40.7s	12,269 kb
10	asm_fast.gc.profdeep	save/restore	yes	yes	maximum	44.5s	40.0s	13,935 kb
11	asm_fast.gc.profdeep	save/restore	yes	no	maximum	44.3s	39.8s	13,901 kb

Table 5.1: Benchmark results

recursion detection Specifies which of the two methods described in section 5.3.5, the save/restore mechanism or the activation counting mechanism, the executable uses to detect recursion. This option is meaningful only if deep profiling is enabled.

preserve tail recursion Specifies whether the compiler uses the modified transformation presented in section 5.3.6 to preserve self-tail-recursion. This option is meaningful only if deep profiling is enabled.

h.o. zero Specifies whether the compiler generates code to zero the activation counter of the caller's ProcDynamic structure (or, equivalently, to set its outermost ProcDynamic pointer field to null if using the save/restore mechanism for recursion detection) when making higher order calls, as described in section 5.3.8. This option is meaningful only if deep profiling is enabled.

h.o. specialization Specifies how much specialization of higher order predicates the compiler performs. By default, there are limits on the size of the higher order predicates the compiler will specialize, in order to limit the cost of the specialization in terms of both memory consumption and instruction-cache misses. However, as described in section 5.3.8, forcing the specialization of even large predicates may yield better profiling information.

Apart from the options listed above, there was no difference in how the various versions of the program were generated. In particular, they were all compiled with the default optimization level. The options listed therefore account for all the differences one can see in the table.

The times shown were obtained by obtained by executing each test eight times, discarding the lowest and highest times for each test, and taking the average of the remaining six times. For the times shown in the column marked "no write", we set an environment variable that disabled the writing of the profiling data files at the end of execution. This should of course affect only rows 4 to 11; the differences in the first three rows are a result of the limited reproducibility of timing measurements.

The timing results show that deep profiling increases the runtime of the program from less than 17 seconds to about 45 seconds, a factor of about 2.7. However, those roughly 45 seconds drop to about 40 seconds if we disable the saving of profiling data to disk. The code of the program being profiled itself is therefore slowed down only by about a factor of 2.4 (40.4/16.7). Given that gprof-style profiling slows down the executable by about a factor of 1.7 (28.5/16.7) while more accurate profiling systems, including cost centre profiling and Quantify, slow down executables by roughly a factor of two [89], this is not bad, since the extra overhead enables deep profiling to gather significantly more detailed profiling information. In fact, the deep profiler has substantially less overhead than the existing Mercury memory profiler, which performs two hash table lookups on every memory allocation.

The performance of deep profiling is not very sensitive to the options used to control it. The data shows that the save/restore approach to recursion leads to slightly faster code than the activation counter approach, but the difference is not great. Similarly, switching off the use of the technique for preserving tail recursion that we have presented in section 5.3.6 leads only to a slight slowdown, because the extra overhead of saving and restoring the old recursion depth counters almost outweighs the speedups from preserving tail recursion and from halving the number of service functions called. Lifting the limits on the size of procedures that are subject to higher order specialization produces only a small effect (in both the basic and deep profiling grades) because only a small number of procedures that can be usefully specialized are above the default size limit. The similarly limited effect of switching off zeroing is due to the small number of calls to procedure-valued variables (only 1.6% of all calls). The default set of options for deep profiling therefore uses the save/restore approach to recursion detection (for speed), enables tail recursion preservation (mostly to limit stack usage, but also for speed), uses the normal level of higher order specialization (to minimize differences between the profiled and unprofiled programs) and uses zeroing (to yield intuitive call graphs in the presence of accidental higher order recursion).

The options used also have only a minor impact on the sizes of executables compiled with deep profiling. These executables are roughly 2.3 times the size of executables compiled with the default grade, and 1.4 times the size of executables compiled with call graph profiling, due to the code of the extra calls to the service functions and the bulk of the static ProcStatic and CallSite structures. However, they are much smaller than debuggable executables. Debuggable executables contain much more static data than deep profiled executables, they contain slightly more calls to service functions, and (for reasons explained in [95]) each such call is followed by extra instructions.

As we mentioned in section 5.3.2, we set a flag during the execution of profiling primitives, and we charge profiling interrupts to `current_call_site_dynamic` only if this flag is not set. During most profiling runs, this mechanism causes somewhere between 25% and 30% of profiling interrupts to be discarded. A mechanism that could discard all profiling interrupts that arrive during the execution of profiling overhead, leaving only the profiling interrupts that the original, unprofiled program would have received, would have to discard about 58%, leaving only 42% (mirroring the execution time ratio of roughly 140% overhead to 100% original code).

There are two reasons why our mechanism compensates only for about half of the profiling overhead. The first is that much of the overhead is in the form of storing values returned by profiling primitives on the stack and later putting them back into registers for calls to other profiling primitives. Such data shuffling is carried out by the same mechanisms that perform data movement on behalf of the user program, and the deep profiling transformation cannot tag the extra data shuffling it causes because data shuffling of any kind is invisible at the level of source-to-source transformations. (In fact, even notions such as stack slots do not meaningfully exist at that level.) An implementation of deep profiling that operates at the level of the code generator could avoid this problem, by setting/clearing the flag not at entry to or exit from profiling primitives but before the first load and after the last store around the call to the profiling primitive.

The second reason is much more intractable. The presence of profiling increases the sizes of stack frames and procedure bodies and forces values that could otherwise remain in registers to be saved/restored in stack slots across calls to service functions. It thus increases the stresses on the cache system and (in extreme cases) on the virtual memory system. It also changes (usually for the worse) the effectiveness of pipelines, branch prediction mechanisms, and other architecture features designed to speed execution. With modern CPUs, such Heisenberg effects are unavoidable in any profiler that relies on code instrumentation (a requirement that only profilers using specialized hardware can escape). One consequence of this fact is that when programmers modify the program in an effort to speed it up, they must check that the change reduces runtime not only with profiling enabled, but also with profiling disabled. As long as this is so, the cost/benefit ratios of efforts to address the first reason are not very favourable.

Overall, deep profiling has about the same level of time overhead as debugging. This is not very surprising, since the implementations of both consist mainly of adding calls to service functions at various points in procedure bodies. Both transformations add calls at procedure call, exit, redo and fail. Debugging also inserts code at entries to the branches of branched control structures (if-then-elses, disjunctions, switches), while deep profiling inserts code at calls of all kinds.

In the case of debugging, all the inserted calls call the same service function, which returns immediately if (as is the case in the test runs from which row 3 in the table is derived) the program is not being run under the Mercury debugger. In the case of deep profiling, the inserted calls call the service functions listed in table 5.2. The various subsections of section 5.3 have discussed all these service functions, with the exception of `builtin_call_port_code`, which relates to the special status of unification and comparison. In Mercury, unification and comparison are defined on values of every type, even in the absence of an explicit declaration such as Haskell's "deriving (Eq,Ord)". Unification and comparison on user-defined types are implemented as calls to compiler-generated Mercury procedures, which deep profiling transforms just as it transforms all other procedures. Unification and comparison on builtin types are implemented in the Mercury runtime system. When polymorphic code tries to unify or compare two values whose type (actually, whose top type constructor) it does not know, it calls the generic unification or comparison code in the runtime system, which forwards the call to the unification/comparison routine appropriate to the actual type of the two values. Therefore, while the rest of the compiler treats calls to the generic unification and comparison procedures as ordinary calls, deep profiling treats them as a special kind of method call, and prefaces them with a call to the `prepare_for_special_call` service function. When the generic unification or comparison procedure steers the call to the compiler-generated type-specific unify or compare procedure, the call to `prepare_for_special_call` will be followed by the `call_port_code` of that procedure. When the call is steered to the unify or compare code in the runtime system for a builtin type, it will be followed by an "call" to the service routine `builtin_call_port_code`. The "call" is in quotes because the call is in fact inlined in the hand-written code of the runtime system; this additional code is enabled only in deep profiling grades. Similar code also records the success or failure of the call.

Table 5.2 presents statistics about the behavior of each service function. These statistics, and those in tables 5.4 and 5.5, were derived from a test run that is identical to the test run reported in row 6 of table 5.1 except that the executable being tested had compiled into it extra code to collect statistics.

The test run included 116,708,303 calls of various types. Of these, 22,460,518 of these were self-tail-recursive calls, which involved only a call to `prepare_for_tail_call`. The other calls involved both a call to one of the other `prepare` functions in the caller and a call to either `call_port_code` or `builtin_call_port_code` in the callee.

The table tells us that in almost all cases, the various `prepare` functions can reuse an existing CallSiteDynamic structure left in their slot by a previous call, and that `call_port_code` and `builtin_call_port_code` can similarly reuse an existing ProcDynamic structure left in their parent CallSiteDynamic structure by a previous call. The fraction of service functions that require memory allocation is well below 1%: only 0.24% of calls to service functions that can allocate a CallSiteDynamic structure do so, and only 0.18% of calls to service functions that can allocate a ProcDynamic structure do so. These proportions are very similar to the analogous measurement reported for cost centre stack profiling [52]. This is not surprising. Numbers substantially higher than these (say in the whole percent range) would require a program whose activity is much more evenly spread than it is in most typical programs, not just through the code of the program but also through the possible set of ancestor contexts.

Table 5.4 gives the memory overhead of deep profiling. The program being profiled normally has a working set around 20 Mb; deep profiling adds 14 Mb to this. The bulk of this 14 Mb consists of CallSiteDynamic nodes, and much of the rest is `pd_call_site_ptrs` arrays. Only about 40% of the slots in those arrays contain valid pointers to CallSiteDynamic nodes. A small minority (at non-first-order call sites) contain pointers to list nodes; the rest contain null. Had we made the arrays contain CallSiteDynamic structures themselves instead of pointers to them, the call graph would be significantly bigger.

Table 5.5 shows that the simple compression system we have described in section 5.3.12 is fairly effective; it reduces the size of the call graph by almost a factor of two in the process of writing it out to the profiling data file. In the future, we intend to see whether we can use a compression

`prepare_for_normal_call`	allocate new CallSiteDynamic	220,553
	reuse existing CallSiteDynamic	78,224,779
	total	78,445,332
`prepare_for_higher_order_call`	allocate new CallSiteDynamic	2,339
	reuse existing CallSiteDynamic	1,903,738
	total	1,906,077
`prepare_for_method_call`	allocate new CallSiteDynamic	481
	reuse existing CallSiteDynamic	923,534
	total	924,015
`prepare_for_special_call`	allocate new CallSiteDynamic	33,091
	reuse existing CallSiteDynamic	12,939,267
	total	12,972,358
`prepare_for_callback`	allocate new CallSiteDynamic	3
	reuse existing CallSiteDynamic	0
	total	3
`prepare_for_tail_call`	allocate new CallSiteDynamic	26,106
	reuse existing CallSiteDynamic	22,434,412
	total	22,460,518
`call_port_code`	allocate new ProcDynamic	214,437
	reuse outermost ProcDynamic	37,827
	reuse ProcDynamic in call site	92,455,370
	total	92,707,634
`builtin_call_port_code`	allocate new ProcDynamic	4,203
	reuse ProcDynamic in call site	1,535,948
	total	1,549,151

Table 5.2: Node reuse statistics

library to reduce the size of the data file still further. The data shows that this could compress the file by more than a further factor of two. The question is whether the use of more sophisticated compression techniques would produce excessive slow down when the writing out the data file in the profiled executable and when reading it back in the `mdprof` server process.

The prepare functions for non-first-order calls include code to search a list of procedures called from the same call site in the caller's ProcDynamic structure. Our instrumentation shows that these list searches are *very* short. During the test run above, none of the list searches in the `prepare_for_higher_order_call`, `prepare_for_method_call`, and `prepare_for_callback` functions ever searched a list that contained more than one element, which means e.g. all calls through a given higher order call site in a given ancestor context always called the same procedure. (Two calls from the same call site have called different procedures, but they have done so from different ancestor contexts.) Even within the `prepare_for_special_call` procedure, the maximum search length is only five, with 95% of searches stopping after the first test, and 4% stopping after the second. It is therefore not surprising that a simple linear search is best; our experiments with using move-to-front lists instead have produced no detectable timing changes. The same is true for our experiments with the use of different keys.

We have tried quite hard to reduce the overhead of deep profiling to as low a level as possible. The area that superficially looks most promising for further reductions is our use of two calls to service functions, one in the caller and one in the callee, for each (non-self-tail-recursive) call in the original program. Changing this to use only a single call to a service function in the caller would require several changes. Most obviously, the representation of closures would need to be modified to include a pointer to the ProcStatic structure of the procedure they encapsulate. The typeinfo structures from which the generic unify and compare routines extract the addresses of the type-specific unification and comparison procedures, and the typeclassinfo structures from

call graph in memory				14,008,936 bytes
consisting of	CSD nodes	282,573 nodes	32 bytes per node	9,042,336 bytes
	PD nodes	218,641 nodes	8 bytes per node	1,749,128 bytes
	PD array slots	696,626 slots	4 bytes per slot	2,786,504 bytes
	list nodes	35,914 nodes	12 bytes per node	430,968 bytes
call graph on disk				8,196,174 bytes
call graph on disk, gzipped				4,067,527 bytes

Table 5.3: Call graph size statistics

node type	number of nodes	size per node	total size
CSD nodes	282,573	32 bytes	9,042,336 bytes
PD nodes	218,641	8 bytes	1,749,128 bytes
PD array slots	696,626	4 bytes	2,786,504 bytes
list nodes	35,914	12 bytes	430,968 bytes
total	1,233,754		14,008,936 bytes

Table 5.4: Call graph size breakdown

which method calls extract the addresses of the methods would need similar changes. However, a more important problem is that eliminating `call_port_code`also eliminates the ability to pick up TopCSD, which deprives `exit_port_code`, `fail_port_code` and `redo_port_code` of their main input. We could fix this by adding upward links to the call graph, or by doing the work of `exit_port_code`, `fail_port_code`and `redo_port_code` in the caller rather than in the callee. Unfortunately, both of these fixes have their own costs, and are likely to be at least as expensive as the scheme we have presented.

Going the other way, moving the functionality of `prepare_for_normal_call` into `call_port_code` looks more likely to yield speedups. It would require passing an extra argument with every call to identify the location containing the pointer to the call's CallSiteDynamic structure, if it has one. For first order calls, this would mean passing the address of `current_call_site_dynamic->csd_callee_ptr->pd_call_site_ptrs[call_site_num]`, which could be done by the code generator. For higher order calls, we would still need something like `prepare_for_higher_order_call`, `prepare_for_method_call` etc to search the list of callees, to create a new list node if necessary, and to return the address of the CallSiteDynamic pointer inside the right list node to be passed as the extra argument, since the callee can't know whether it was called from a first order call site. However, eliminating the overhead of the second service function call even from just first order calls may yield useful speedups. The downside is that this scheme would require integrating deep profiling into the code generator.

5.5.2 Performance of the profiling tool

The execution of the profiling tool can be divided into two phases: the execution of startup code and the execution of the request processing loop. In our implementation, the request processing loop is not a problem for performance: each request gets only about a pageful of data, and that pageful does not need a significant amount of computation; it needs only to be looked up in data structures that have already been constructed by the startup code, and formatted. The reason why we do things this way instead of summarizing data on demand has to do with the pages at which the programmer may start his or her exploration of the call graph: the page showing the clique of the top-level call to `main/2`, the page showing the most expensive procedures and the page showing the most expensive call sites. Displaying any of these pages requires work (propagating profiling information all the way to the top) that subsumes the work required by other page requests. If the work has to be done before displaying the first page anyway, the system may as well save the

call graph in memory	14,008,936 bytes
call graph on disk	8,196,174 bytes
call graph on disk, gzipped	4,067,527 bytes

Table 5.5: Call graph size statistics

results of the work instead of repeating the work later.

The time taken by the startup phase has five main components:

- the time taken to read in the profiling data file,
- the time taken to post-process the call graph, explicitly recording relationships that the profiling data file left implicit in the interest of compactness,
- the time taken to find the cliques of the call graph and to link cliques to procedures and vice versa,
- the time taken to propagate profiling data up the call graph using the algorithm of figure 5.15,
- and the time taken to summarize the data in the call graph in various ways, e.g. to group together the ProcDynamic structures that represent calls to the same procedure and thus find the total cost of that procedure.

With our current implementation, all this takes about 22 seconds for the roughly eight megabyte profiling data files produced by the test runs of section 5.5.1. We consider this performance to be eminently acceptable; even profiling tools that generate much less useful profiling information, such as gprof, take significantly longer to digest their raw profiling data.

Reading in the data file accounts for about a third of the startup time, while the other four components account for roughly a sixth each. The reason why ancillary functions such as post-processing consume so much time relative to e.g. the propagation algorithm is that the startup code was not our main focus in optimizing the server. After its startup phase, the server process spends most of its lifetime waiting for requests. Given all the precomputation in the startup phase, the amount of work the server has to do for each request is so small that request processing is not speed critical; even a slow implementation would be fast enough for the human viewer. On the other hand, the server occupies space in main memory or on disk for its entire lifetime. We have therefore designed the main persistent data structures of the server process (those which are needed after the startup phase) for compactness more than for speed. For example, the profiling information associated with each CallSiteDynamic structure contains seven integers: four port counts (call, exit, redo, and fail), a count of clock ticks, a count of memory blocks allocated, and a count of memory words allocated. Many of these integers are often zero, for several reasons.

- Only a tiny minority of CallSiteDynamic structures contain a nonzero count of clock ticks.
- A large fraction of procedures do not contain code to allocate memory, and thus their memory blocks and memory words fields will be zero.
- In most programs, only a small minority of procedures can succeed more than once. Since only procedures that can succeed more than once can be backtracked into, almost all redo fields will contain zero.
- In most programs, only a (reasonably large) minority of procedures can fail, and thus most fail fields will contain zero.

Our representation of profiling measurements therefore is not a simple tuple of seven integers. Instead, it is a data type with several alternative function symbols. Each function symbol implies that a particular set of measurements is zero, and its arguments include only the remaining measurements. Using the normal Mercury representation, this means that each set of measurements is implemented as a tagged pointer to a block of integers, the tag specifying the alternative and thus both the size of the block of integers and the meaning of each integer in the block. The average number of integers per block is much smaller than seven, especially given that all alternatives represent the number of calls implicitly (i.e. computed from the other port counts).

Even using this space optimization, the server process needs around than 110 Mb of memory when viewing e.g. the 8 Mb profile generated by the test runs of section 5.5.1. This is not very surprising, since those 8 Mb were compressed from a 14 Mb original, and the server's data structures record explicitly, for quick access, many pieces of information that were only implicit in the 14 Mb original, including propagated times, the list of CallSiteDynamic nodes that all originate at the same call site in different ancestor contexts, the list of CallSiteDynamic structures that call the same procedure, etc. However, 110 Mb is still more than we would like. We will therefore look into compressing the server's data structures even further.

5.6 Comparison to previous work

Flat profilers such as the original Unix profiler "prof" measure the amount of time spent in each function by sampling the program counter, and use instrumentation to count the number of times each function is called. (A flat profiler for a logic programming language such as Prolog would count exits, redos and fails as well [29].) Such profilers have low overheads, and they are also very effective at what they try to do: telling programmers where the hot spots are, so they can be speeded up. Their main drawback is that they do not tell programmers where the hot spots are usually called from, so they do not have any guidance as to which calls to hot spot functions are worth eliminating or specializing, and which are not.

Graph profilers such gprof [38] try to supply that information as well. Rather than using one call counter per function, it has a call counter per caller-callee pair. It therefore knows not only how many times a function was called, but also how many times it was called from each of its parents. However, it still has only a single counter for the time spent inside each function. This gives accurate results for a flat time profile, which answers the question "how much time was spent in the code of this function?", but leads gprof astray when it tries to squeeze out of these same counters the answer the question "how much time was spent in the code of this function and its descendants?". The reason is that `gprof` attempts to answer the second question by propagating times from the lowest levels of the call graph to the highest, distributing the time spent in each function and its descendants (if any) amongst all the callers of that function by assuming that on average, all calls to a given function take the same time. [9] Unfortunately, different calls to a function take different times, and for many programs this causes gprof to produce incorrect and misleading answers to the second question. Our experience with the first Mercury profiler, `mprof`, which is a clone of gprof, bears this out, and suggests that the more frequently polymorphism, recursion and higher-order code appears in a program, the less useful the output of a gprof-style profiler will be in answering the second question. Since in declarative languages, all interesting programs use recursion and large programs are very likely to use polymorphism and higher-order code, this problem makes gprof-style profilers almost unusable for such languages.

Ponder and Fateman [78] describe a modification to graph profilers that avoids this problem and allows accurate answers to the second question. Their approach associates with each function in the program a variable that represents the time spent inside that function and its descendants. They then bracket each procedure body with code that subtracts the current time from the variable

[9] Actually, gprof's propagation algorithm resembles ours in that its basic units are not functions but groups of mutually recursive functions, and there is no propagation of times between mutually recursive functions. It differs from ours in that the cliques handled by gprof's algorithm are not ancestor-context-specific and do not have a unique entry point, as well as in the use of that unjustified assumption.

at call and adds it back at return. The total effect is of course to increment the variable by the time spent inside the function and its descendants. In order to avoid double-counting the time spent inside recursive calls to the same function, the decrement/increment code in the function prolog/epilog is executed only for top-level (i.e. non-recursive) calls to the function.

The first two questions are not the only ones that programmers may wish to have answered by a profiling tool. Two others are "how much time was spent in the code of this function when called from this call site?", and "how much time was spent in the code of this function and its descendants when called from this call site?". The latter (fourth) question can be answered by a simple adaptation of the Ponder and Fateman approach, which has a counter for each call site and brackets each call site with code to decrement/increment the call site's counter. In a language without higher order calls, such a system can dispense with the per-function counters, since the time spent inside a function and its descendants can be computed by simply summing the counters associated with all the call sites that call that function. In a language that has higher order calls, a single call site can call many different functions. One can then either use separate a counter for each callee (similar to the technique we presented in section 5.3.7) or simply retain the per-function counters. Since retaining per-function counters slows down all functions while using per-callee counters at higher order call sites only slows down functions that make higher order calls, the former approach is preferable. Although we have not been able find publications that give details about the algorithms used by commercial profilers, we believe that some (e.g. Rational Quantify) use this kind of approach.

Deep profiling of course has all the desirable properties of Ponder and Fateman's technique and its variants, and more, at the cost of higher overhead.

Cost centre profiling [89], is a profiling technique developed by Sansom and Peyton Jones, for use with lazy functional languages. Their idea is that expressions, the basic units of functional programs, may be annotated with a construct that identifies the expression as a cost centre. The cost of any evaluation is credited to the closest enclosing cost centre. By doing this, the system successfully distinguishes between calls to a given function from different cost centers. This is particularly important for lazy functional languages such as Haskell because the prevalence of higher order code means that some functions get used very frequently in a wide variety of contexts, and in ways that will vary greatly in cost.

A significant complication with cost centre profiling is that the cost of a function call is paid not in a single block of time bounded by the function call and return, but in dribs and drabs as the thunks created by the function call are evaluated over time on demand from the rest of the program. Cost centre profiling therefore has a field in each thunk that says what *cost centre* should be charged with the cost of evaluating the thunk. All inner thunks created by the evaluation of a thunk will inherit its cost centre, with the exception of expressions that create their own cost centre.

Another complication of cost centre profiling is that while it looks like it answers the fourth question, it doesn't quite do so. If you associate a cost centre with a call site, some of the cost of the call at that call site (including the called function and its descendants) will be associated with that cost centre, but some of it may be associated with other cost centers in the call tree. This means that the answer to the fourth question can be recovered only if each cost centre has a unique parent cost centre. If programmers choose which expressions are cost centers, they can try to ensure this condition. However, the condition cannot be ensured if the compiler makes every call or every function body a cost centre, which is a compilation option in the available implementations of cost centre profiling. This is a significant drawback, because in many cases, asking the compiler to make everything a cost centre is the only practical alternative to an extensive series of recompilations as one tries to find the set of function bodies or call sites that must be made cost centers in order to get useful profiling information [52].

Morgan and Jarvis developed **cost centre stack profiling** [52] as a solution to these problems. As its name implies, cost centre stack profiling associates with each measurement not just a single cost centre but a stack of the enclosing cost centers as well. The stack of cost centers is very similar to our implicit stack of ProcDynamic structures (if the cost centre expressions are entire function bodies) or to our implicit stack of CallSiteDynamic structures (if the cost centre expressions are individual function calls). However, cost centre stack profiling and deep profiling also have major differences in both the data gathered and the implementation.

The deep profiler handles call sites and procedures as different kinds of entities during a profiling run. Cost centre profiling makes no such distinction. Its cost centers can represent either function bodies or function calls (or any expression between those two size extremes), but it has no knowledge of the syntactic relationships between different cost centers. It thus cannot use those relationships to make the output reflect the programmer's expectations more closely, e.g. by presenting a function body and apportioning the cost of the function body among the code of the function body itself and the various calls occurring in it. Also, in its usual mode of use, which designates function bodies as cost centers, cost centre stack profiling does not distinguish different call sites to the same function in a given function body and thus does not help much with the profiling of programs that exploit polymorphism within callers (one of our requirements from section 5.2). The deep profiler does.

The deep profiler and the cost centre stack profiler both detect recursion, but they do so differently and do different things when they find it. When entering a new cost centre, the cost centre stack profiler searches the list of cost centers that have been previously entered from the current cost centre stack configuration. Usually, the answer will be yes, and the system can then simply reuse the extended cost centre stack configuration allocated previously. The deep profiler, by contrast, does a similar search only at higher order call sites; at first order call sites, which are the vast majority, it does an array lookup instead, which is faster.

If the search shows that the program has not previously entered the new cost centre from the current cost centre stack configuration, then the cost centre stack profiler will search the cost centre stack to see whether it already contains the cost centre being entered. If yes, it will create a new configuration by effectively deleting the previous occurrence of the cost centre from the stack. This deletion destroys the call centre stack profiler's record of the entry point of the current cost centre clique, which contains all the cost centers on the stack more recent than the one being deleted. By contrast, the deep profiler preserves this information for presentation to the programmer, and it implements the recursion check in constant time. Since the cost centre stack can be quite deep even after the removal of duplicate entries, this is a significant advantage. (One could argue that our constant time check depends on setting the values of the `ps_outermost_proc_dynamic` fields in ProcStatic structures on the way down from the root, and that this is just as linear as searching the stack up to the root. However, that argument would be wrong: given the branching factors of the call trees of most programs, the time taken by an invocation of `call_port_code` to set up a `ps_outermost_proc_dynamic` field is amortized over a large number of checks by other invocations of `call_port_code`.)

On the other hand, there is one significant cost that deep profiling incurs that cost centre stack profiling does not: the creation and initialization of the `csd_callee_ptr` arrays. However, this happens only when a procedure is entered from a given ancestor context for the first time, and measurements of both deep profiling and cost centre stack profiling show that this occurs on less than 1% of calls. Overall, we expect deep profiling to have significantly lower time overheads than cost centre stack profiling. Unfortunately, [52] does not have figures directly comparing the speeds of executables compiled with cost centre stack profiling and speeds of executables compiled without any form of profiling; it gives figures only for the overhead of cost centre stack profiling over plain cost centre profiling, these overheads ranging from 2% to 384%. For the overhead of cost centre profiling over unprofiled programs [52] directs the reader to Sansom's thesis [88]. However, this contains only a single statement that the time overhead of cost centre profiling is 64% for time profiling only and 133% for both time and heap profiling, and does not identify the program on which those measurements were taken. A fair comparison of the Mercury deep profiler and the Haskell cost centre stack profiler would in any event be significantly complicated by the major differences in the efficiencies of the underlying execution algorithms: due in large part to the absence of laziness in Mercury, Mercury programs are often much faster than the corresponding Haskell programs. (For a rough, non-scientific test, see [85].) This implies that a level of overhead that slows down a Mercury program by e.g. a factor of 2.4 would slow down an equivalent Haskell program by a much smaller factor.

A final difference between the two profiling methods is that the cost centre stack profiler does not even try to break accidental cycles through higher-order predicates. The resulting merging of what to the programmer seem to be unconnected cliques can make the presented profiling information quite confusing.

Of course, our solutions to the problem of programs containing recursion through higher order predicates are not perfect either. The only approach that we know of that can avoid distorting the profile of such programs is *call path profiling* [40, 41], which records not just the current function but also all its ancestors at each measurement. The down side of this approach is the cost of traversing the stack all the way to the root. This is not a problem for gathering timing information via profiling interrupts, since on most machines profiling interrupts occur only sixty or a hundred times a second. The cost *is* a problem for gathering memory allocation statistics, since memory allocations may occur millions of times a second. This makes traversing the stack at each memory allocation impractical, particularly since programs written in declarative languages tend to allocate memory more often and to have deeper nesting of function calls (partly due to the much greater use of recursion) than imperative programs. Hall's proposed solution is to make the profiling of memory allocation statistical as well: instead of recording every memory allocation, record only every Nth allocation, choosing a value for N that makes the overhead acceptable (e.g. N=1000). Unfortunately, statistical memory allocation information is significantly less useful than accurate memory allocation information, not so much when looking for hot spots as when exploring program behavior: for example, one cannot infer from the absence of a recorded memory allocation for a function that e.g. the arm of an if-then-else in that function that causes memory allocation was never taken. The same argument applies with much greater force to recording counts of calls, exits, and failures.

Several systems, including MLWorks [7] and IBM's Java Virtual Machine profiler interface [108] (which is not a profiler itself but rather a piece of infrastructure on which profilers can be built), adopted another remedy. This is to record with each measurement only a fixed number of functions; in IBM's JVM, the default is the current function and its three nearest ancestors. This makes recording a context cheap enough to be done at every measurement, including every memory allocation and every method entry. However, the cost in usefulness is great. The contexts are not connected to the root of the call tree, which means that programmers cannot explore the call tree as they can with deep profiling. An even more serious problem is that the data gathered in this way is not sufficient to compute the full costs (self plus descendants) of either functions or call sites; to get that information, one must use technique as well (e.g. Ponder and Fateman's), incurring two sets of overheads.

One cannot really compare the Mercury deep profiler with basic block profilers such as pixie and QP, with hardware profilers such as Intel's VTune, with profilers that measure issues arising from threading (e.g. lock contention), or with profilers that record which functions hold the root pointers that cause memory to be alive at particular points in program execution, since their objectives are disjoint. However, exploring how some of those objectives could be reached in a system based on deep profiling technology would be interesting future work.

5.7 Discussion

We can report that the detailed contexts associated with each measurement by our new profiler has already led us to a performance problem we probably would not have found any other way; we certainly did not find it using the old, gprof-style Mercury profiler. The performance problem consisted of a code fragment that checked two copies of a large data structure for equality, even though the two copies were guaranteed to be identical. (The Mercury determinism system did not catch the error because the extra unification was in the condition of an if-then-else, and therefore it was allowed both to succeed and to fail.) The time reported by the old Mercury profiler for the procedure containing the extra unification was not large enough to draw attention to itself, being based on average and not actual times spent inside the children of that procedure; the average times of such unifications were much lower than the actual times because most other comparisons of such data structures failed almost immediately. Even if we did decide to speed up that procedure, the profile would not have lead us to the redundant unification, since several other children of the procedure had higher average times recorded for them. However, on a deep profile the problem jumped right out: the time recorded for the procedure was large enough to draw our attention, and the time recorded for the redundant unification accounted for the vast majority of that time. Eliminating the redundant unification sped up the program involved (the Mercury compiler) by about 10%; not bad for a one-line change.

A practical profiler needs to have a way to merge profiling data files from different runs, since finding a representative *set* of sample inputs is usually much easier than finding a single representative sample input. Our deep profiler does not yet have a way to do this, but we are in the process of providing such a mechanism; it should be a relatively simple extension of the algorithm we use to recursively merge two or more ProcDynamic structures in a given clique that represent the same procedure, and their descendants.

Chapter 6

Conclusion

Our contributions in this thesis fall into three categories: our enhancements to the Mercury execution model to enable it to support coroutining and parallel execution, our addition of explicit threading to the tool-box of Mercury programmers, and our work on profiling with a view to automatic granularity analysis.

The main objective of our work on the execution model was to keep overheads as low as possible. The existing, sequential Mercury implementation is much faster than all the Prolog implementations we have measured; Mercury versions of our benchmark programs usually run 2 to 20 times as fast as their Prolog equivalents. This poses significant problems for us as implementors of new extensions. A given absolute amount of overhead that would slow down a Prolog system by a small fraction (say 5%) would slow down Mercury by an unacceptable fraction (say 50%).

The guiding principle of all our modifications to the Mercury execution model is to impose overheads only on programs and program components that use the facility whose presence requires those overheads. In the case of the modifications presented in this thesis, we wanted to preserve the speed of sequential code, and add overheads only to the parts of the program that use coroutining and/or parallelism. As a result, we were willing to accept somewhat higher overheads for thread creation, synchronization and destruction than some other parallel logic programming systems. However, as a result we were able to keep the extra overheads on sequential code with coroutining and/or parallel programs to a minimum. These overheads consist only of the following.

- In coroutining and/or parallel programs, the runtime system has to manage more than one task, and we currently use a slightly different organization of the runtime's engine and task structures to support this. The impact of this different organization on speed is small; on some platforms (e.g. x86) it leads to a slight slowdown, on others (e.g. SPARC) it actually leads to a small speedup.

- When using the Boehm memory allocator and garbage collector, we have per-engine freelists for the most frequently used block sizes. However, when one of these freelists is exhausted or when the program wants to allocate a block of a non-popular size, the allocation incurs the overhead of acquiring and releasing a lock. Increasing the number of block sizes with per-engine freelists reduces this overhead at the cost of greater memory consumption.

- When not using the Boehm memory allocator and garbage collector (i.e. when running without a garbage collector or when using a future native collector for Mercury), recovering memory across a failed computation requires an extra comparison for coroutining and/or parallel programs.

- Some pieces of library code which are implemented in C require locking to prevent multiple threads executing them at the same time. Most standard C library calls made from Mercury are threadsafe (including the I/O functions), but some parts of the Mercury standard library which are implemented in C are not.

The benchmarking we were able to perform was limited by the fact that the parallel machines we have access to either do not have versions of the Boehm collector that work with POSIX threads, or are old and have few processors (such as the SPARCserver 1000 we used for our tests). Nevertheless, our benchmarking suggests that we were successful in keeping down the overheads that our parallel execution environment imposes on sequential code. It also shows that adding a second or third CPU can give substantial speedups. In the benchmarking we did, we focussed on large grain sizes for parallel execution.

The AND-parallel environment we have built has low overheads compared not just with other AND-parallel systems but also compared to OR-parallel systems. Given a compiler transformation along the lines of [21] that turns OR-parallel code into independent AND-parallel code, we would expect our system to be quite competitive with most existing OR-parallel logic programming systems, at least with respect to large granularity programs.

It would be interesting to explore some areas of the design space that we have not looked at in this thesis. The first such area is the use of message passing rather than shared memory. This would be interesting because it would allow us to scale our implementation to distributed computations within the same framework. As noted in section 2.12, Glasgow Parallel Haskell uses this approach, it would be interesting to explore it in the context of Mercury. Our execution model currently requires that a whole task structure be created whenever a new concurrent computation is created. It would be useful have a representation of tasks that have not yet begun to be evaluated that is more space efficient. For example, representing the new computation as a closure would require only a few words of storage.

The second area is support for programs that have very many small concurrent tasks; it would be nice to be able to execute them efficiently. In section 3.4.4 we suggest the use of heap allocated activation frames and suspension analysis to allow tasks that do not suspend to execute efficiently using a stack, and tasks that do suspend to execute in a space efficient manner using heap allocated activation frames. At the present time, this is not fully integrated with the compiler and runtime environment. It would be instructive to complete this and measure its impact on sequential performance (which will be determined largely by the precision of the suspension analysis), and on the memory footprint of concurrent and parallel programs.

Programmers can access the parallel and concurrent execution machinery in a limited way with the independent parallel conjunction that we presented in 3.5. With further work in the mode system, we could enable programmers to create dependent AND-parallel computations, subject to the constraint that the binding of shared variables be deterministic (i.e. no shared variable binding should be retracted due to failure). We have a fairly good idea about how to do this. Currently Mercury mode declarations give a pair of instantiation trees representing the binding state of a variable at a call and on the success of that call. The third area of potential improvement would be to extend this to have modes describe a set of potential intermediate instantiation states (e.g. as a list is progressively bound) as well as initial and final states. Since we wish to be able to represent sets of instantiation states containing an unbounded number of intermediate states (i.e. we don't know in advance how long the list will be), we can use recursive mode definitions. Currently Mercury requires that all free variables be unique. We would like to relax this and allow shared free variables. However, we don't want unrestricted sharing, since we would no longer be able to perform sufficiently precise mode checking at compile time. Instead, we attach to each node in the type tree corresponding to a free shared variable, an annotation describing whether it is a producing occurrence or a consuming one. Mode analysis can then make sure that there is exactly one goal promising to bind that node if there are any goals that might try to consume it. Unfortunately the low level details of the current mode analysis pass in the Mercury compiler does not keep track of the necessary aliasing of variables within each procedure to enable such an extension to the mode system to be implemented. Current work in the Mercury group at the University of Melbourne includes rewriting the mode analysis pass to allow it to track this kind of aliasing. Once this work is complete, we would be able to implement our proposed extensions to the mode system and write programs that use communication via free variables. The existing runtime system should be able to handle dependent AND-parallel computations with only very minor changes, provided that each parallel goal instantiates its output variables monotonically.

Our solution of the problem of managing accesses to shared variables would be very similar to the approaches adopted in most other parallel logic programming systems, including Reform Prolog [9], &-Prolog [47], ACE [80], and GHC [106]; the main difference is in the program analyses required to ensure monotonicity.

We have developed an interface to our execution machinery using explicit (Mercury level) threads. We have shown how we can provide synchronization primitives such as semaphores and mutexes in a way which is declarative as well as efficient. We have also shown how threads and synchronization can be explained in terms of pure Prolog to justify the declarativeness of our approach. Although we have specified a very simple model of threads, there is no technical reason why we could not implement a more sophisticated model (more like the POSIX thread model) that supports a notion of priority for different threads, and allows threads to interrupt or kill each other. Each of these can be expressed in the pure Prolog model that we described.

We have not gone into any detail about the pragmatics of handling blocking I/O in our model. Currently if a call to an I/O predicate blocks on a read or write call to the operating system, the engine executing the call blocks. The correct thing to do is to test to see if the call will block before making it, and to make the call only if it would not block. If the call would block, then the engine should make the task block, and execute another task. If there are no other tasks that are runnable, then rather than waiting on the runqueue alone, the engine should use the Unix system call `select` (or its equivalent on other operating systems, or a busy-waiting loop) so that it will wake when any of the tasks blocked on I/O may proceed. We have written code in the runtime system for Mercury that allows this, however the Mercury standard library code will still require substantial modification to work in this way. For a detailed discussion of the issues involved, see [34].

The main contributions of the material on profiling presented in chapter 5 are the following:

- The design, implementation and validation of a reasonably simple, cheap and non-distorting method for associating detailed contexts with all profiling measurements. These detailed contexts are essential for meaningful profiling information in the presence of any of several forms of polymorphism.
- The identification of accidental recursion through higher order calls as a tough but important problem for profilers for declarative languages.
- The design of three techniques (specialization, signatures and zeroing, two of which we have implemented) that each yield meaningful profiles for large classes of programs that contain recursion through higher order calls.

These contributions are important because they allow us to gather detailed information about the behaviour of the program that can be used either manually or automatically to improve the performance of programs. Improvements can be made not only by sequential optimization techniques such as "reduction in strength" (moving computations out of loops, see [2]), but also by using the data to identify opportunities for parallelization.

Many compilers have a mechanism that allows them to take one or more profiling data files and use the information they contain about the typical behavior of the program to help them optimize the program more effectively. (For example, the SPEC CPU benchmark suite available from www.spec.org specifies training data as well as test data for each benchmark program, and most SPEC submissions exploit it for profiling feedback.) This could mean adjusting the optimization level for each procedure depending on the time cost of that procedure: e.g. optimizing time-consuming procedures for speed and other procedures for space. With a profiler that collects reliable information about the costs of call sites as well as procedures, it could also mean inlining a procedure in some of its call sites but not in others. However, these and other opportunities for tailoring optimization more closely to the program are worth exploiting only if the cost in compilation time is not too high. Since the cost of reading and processing a multi-megabyte profiling data file on every compiler invocation *would* be too high, we have extended the profile viewing tool with a command to write out a summary file containing just the information the compiler needs.

One use of profiling feedback that we expect will be particularly useful in a purely declarative language such as Mercury is automatic parallelization. In imperative languages and logic languages that allow unrestricted variable aliasing, automatic parallelization has traditionally been dominated by the struggle to prove that e.g. two function calls are independent and can therefore be executed in parallel. In declarative languages with such as Mercury which make data flow explicit, proving calls independent is usually trivial (all it requires is checking that neither consumes the other's output), and most programs contain many parallelization opportunities. The problem is that many, even most of those opportunities are too small; the overhead of creating a work package from a call so that it can be given to another CPU often takes more time than evaluating the call would have taken. Therefore in the absence of profiling information, parallelization can (and often does) slow down the program. With profiling information about the average costs of call sites, the compiler can avoid spawning off work packages that are *consistently* too small. However, as we discussed in section 5.2, in many cases the cost of a call through a given call site will vary by several orders of magnitude depending on the ancestor context in which it is invoked. Deep profiling will tell the compiler in which contexts a call site is worth parallelizing. If a call site turns out to be worth parallelizing in some contexts but not others, the compiler may, if it wishes, create two versions of the procedure containing the call, only one of which spawns off the call to be executed in parallel. (It may also have to clone some of the ancestors of that procedure to make sure that the right version is called in each ancestor context.) We expect that this capability will significantly improve the speedups one can get from automatic parallelization, and, probably even more important, reduce the probability that automatic parallelization leads to a slowdown.

Bibliography

[1] M. García de la Banda. *Independence, Global Analysis and Parallelism in Dynamic Scheduled Constraint Logic Programming*. PhD thesis, Universidad Politécnica de Madrid (UPM), July 1994.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, tools*. Addison-Wesley Publishers, 1985.

[3] Khayri A. M. Ali and Roland Karlsson. The Muse approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, 1990.

[4] K.R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. In *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.

[5] J. L. Armstrong and R. Virding. Erlang - an experimental telephony switching language. In *XIII International Switching Symposium, Stockholm, Sweden*. 1991.

[6] J. L. Armstrong, R. H. Virding, and M. C. Williams. *Erlang User's Guide and Reference Manual Version 3.2*. Ellemtel Utvecklings AB, Sweden, 1991.

[7] Nick Barnes. GC and profiling. In *Garbage collection mailing list*. April 2001.

[8] Uri Baron, Jacques Chassin de Kergommeaux, and Jean-Claude Syre. The parallel ECRC Prolog system PEPSys: an overview and evaluation results. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 841–850, Tokyo, Japan, December 1988.

[9] Johan Bevemyr, Thomas Lindgren, and Håkan Millroth. Reform prolog: The language and its implementation. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 283–298, Budapest, Hungary, June 1993. MIT Press.

[10] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, 1988.

[11] Hans-Juergen Boehm. Space efficient conservative garbage collection. *ACM SIGPLAN Notices*, 28(6):197–206, June 1993.

[12] Roland N. Bol. *Loop Checking in Logic Programming*. PhD thesis, University of Amsterdam, The Netherlands, 1991.

[13] A. Bossi, N. Cocco, and M. Fabris. Proving termination of logic programs by exploiting term properties. In S. Abramsky and T. S. E. Maibaum, editors, *Proc. TAPSOFT'91, vol. 2*, Lecture Notes in Computer Science 494, pages 153–180. Springer-Verlag, 1991.

[14] A. Bossi, N. Cocco, and M. Fabris. Typed norms. In B. Krieg-Brückner, editor, *Proc. ESOP'92*, Lecture Notes in Computer Science 582, pages 73–92. Springer-Verlag, 1992.

[15] T. Bray, J. Paoli, and C. Sperberg-MacQueen. Extensible markup language, 1998. Available from `<http://www.w3.org/TR/REC-xml>`.

[16] Lawrence Byrd. Understanding the control flow of Prolog programs. In *Proceedings of the 1980 Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, July 1980.

[17] Mats Carlsson and Johan Widen. SICStus-Prolog user's manual. Technical Report 88007B, Swedish Institute of Computer Science, Kista, Sweden, 1988.

[18] W. Chen, M. Kifer, and D. S. Warren. HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 1090–1114, Cleveland, Ohio, USA, 1989.

[19] Keith Clark and Steve Gregory. Parlog: a parallel logic programming language. Technical Report DOC 83/5, Department of Computing, Imperial College of Science and Technology, London, England, May 1983.

[20] Keith L. Clark and Steve Gregory. Notes on systems programming in Parlog. In *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, pages 299–306, Tokyo, Japan, November 1984.

[21] M. Codish and E. Shapiro. Compiling or-parallelism into and-parallelism. In E. Shapiro, editor, *Third International Conference on Logic Programming, London, United Kingdom. Lecture Notes in Computer Science, vol. 225*, pages 283–297. Springer-Verlag, 1986. Also in New Generation Computing 5 (1987) 45-61, Ohmsha Ltd. and Springer-Verlag.

[22] A. Colmerauer, H. Kahoui, and M. van Caneghem. Un systéme de communication homme-machine en français. Technical report, Groupe de Rescherche en Intelligence Artificielle, Université Aix-Marseille II, 1973.

[23] Thomas Conway, Fergus Henderson, and Zoltan Somogyi. Code generation for Mercury. In *Proceedings of the Twelfth International Conference on Logic Programming*, Portland, Oregon, December 1995.

[24] J. Chassin de Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1994.

[25] K. DeBosschere and P. Tarau. Blackboard-based Extensions in Prolog. *Software: Practice and Experience*, 26(1):49–69, January 1996.

[26] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Language and Systems*, 15(5):826–875, 1993.

[27] S. K. Debray and N.-W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(6):599–613, 1993.

[28] Saumya Debray. Automatic cost analysis of logic programs. In *Proceedings of the NACLP '90 Workshop on Parallel Execution of Logic Programs*, Austin, Texas, November 1990.

[29] Saumya K. Debray. Profiling Prolog programs. *Software Practice and Experience*, 18(9):821–858, September 1988.

[30] S.K. Debray, P. Lopez-Garcia, M.V Hermenegildo, and N.W. Lin. Lower-bound cost estimation for logic programs. In *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 291–306. MIT Press, July 1997.

[31] B. Demoen, M. García de la Banda, and P.J. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In J. Edwards, editor, *Proceedings of the 22nd Australian Computer Science Conference*, pages 217–228. Springer-Verlag, January 1999.

[32] Bart Demoen, Maria Garcia de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. Herbrand constraint solving in HAL. In *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 260–274, 1999.

[33] A.P. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–6, August 1958.

[34] Y. Eskilson and M. Carlsson. SICStus MT — A multithreaded execution environment for SICStus prolog. *Lecture Notes in Computer Science*, 1490, 1998.

[35] Jeroen Fokker. Functional parsers. In *Proceedings of the First International Spring School on Advanced Functional Programming Techniques (LNCS 925)*, pages 1–23, Bastad, Sweden, May 1995.

[36] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics — Principles and Practice (2nd Ed.).* Addison-Wesley, 1990.

[37] I. Foster. *PARLOG as a Systems Programming Language*. Ph.D. thesis, Department of Computing, Imperial College of Science and Technology, London, 1988.

[38] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126, June 1982.

[39] G. Gupta, E. Pontelli, M. Carlsson, M. Hermenegildo, and K.M Ali. Parallel execution of prolog programs. *ACM Transactions on Programming Languages and Systems*, 23(4), 2001.

[40] Robert J. Hall. Call path refinement profiles. *IEEE Transactions on Software Engineering*, 21(6):481–496, June 1995.

[41] Robert J. Hall and Aaron J. Goldberg. Call path profiling of monotonic program resources in UNIX. In USENIX Association, editor, *Proceedings of the Summer 1993 USENIX Conference: June 21–25, 1993, Cincinnati, Ohio, USA*, pages 1–13, Berkeley, CA, USA, Summer 1993. USENIX.

[42] S. Haridi and P. Brand. Andorra Prolog: An integration of Prolog and committed choice languages. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 745–754, Tokyo, Japan, 1988.

[43] Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, and Chris Speirs. The Mercury language reference manual. Available from `<http://www.cs.mu.oz.au/mercury/>`, 2000.

[44] Fergus Henderson and Zoltan Somogyi. Compiling mercury to high-level c code. In *Proceedings of the 2002 International Conference on Compiler Construction*, Grenoble, France, 2002.

[45] Fergus J. Henderson. Strong modes can change the world! Technical Report 93/25, Department of Computer Science, University of Melbourne, Melbourne, Australia, October 1993.

[46] Peter Henderson. Purely functional operating systems. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional programming and its applications*, pages 177–192. Cambridge University Press, 1982.

[47] M. V. Hermenegildo and K. J. Greene. &-Prolog and its performance: Exploiting independent and-parallelism. In D. H. D. Warren and P. Szeredi, editors, *Logic Programming: Proc. of the Seventh International Conference*, pages 253–268. MIT Press, Cambridge, MA, 1990.

[48] Manuel V. Hermenegildo. An abstract machine for restricted and-parallel execution of logic programs. In E. Shapiro, editor, *Third International Conference on Logic Programming, London, United Kingdom. Lecture Notes in Computer Science, vol. 225*, pages 25–39. Springer-Verlag, 1986.

[49] Pat M. Hill and John W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[50] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[51] P. Hudak, S. L. Peyton Jones, and P. Wadler. Report on the programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5), May 1986.

[52] Stephen A. Jarvis and Richard G. Morgan. The results of profiling large-scale lazy functional programs. In *Proceedings of the Eighth International Workshop on Implementations of Functional Languages*, pages 200–221, Bad Godesberg, Germany, September 1996.

[53] David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in Mercury. In *Proceedings of the Twentythird Australasian Computer Science Conference*, pages 128–137, Canberra, Australia, January/February 2000.

[54] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 January 1996.

[55] A. King, K. Shen, and F. Benoy. Lower-bound time-complexity analysis of logic programs. In *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 261–276. MIT Press, July 1997.

[56] A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 478–492, Washington, USA, November 1992. The MIT Press.

[57] Robert A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, July 1979.

[58] Bil Lewis and Daniel J. Berg. *Multithreaded programming with pthreads*. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, USA, 1998.

[59] J. W. Lloyd. *Foundations of Logic Programming, Second Extended Edition*. Springer-Verlag, 1993.

[60] K. Marriott and H. Søndergaard. Difference-list transformation for Prolog. *New Generation Computing*, 11(2):125–157, 1993.

[61] George Marsaglia. The mother of all random generators, 1994. Posted by Bob Wheeler to sci.stat.consult and sci.math.num-analysis on behalf of George Marsaglia on October 28, 1994.

[62] Matsumoto and Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACMTMCS: ACM Transactions on Modeling and Computer Simulation*, 8, 1998.

[63] Nancy Mazur, Gerda Janssens, and Maurice Bruynooghe. Towards memory reuse for Mercury. In *Proceedings of the International Workshop on Implementation of Declarative Languages*, Paris, France, October 1999.

[64] Nancy Mazur, Peter Ross, Gerda Janssens, and Maurice Bruynooghe. Practical aspects for a working compile time garbage collection system for Mercury. Technical Report Technical Report CW310, Departement Computerwetenschappen, Katholieke Universiteit Leuven, ”http://www.mercury.cs.mu.oz.au/information/papers/CW310_ctgc.ps.gz”, 1999.

[65] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[66] Greg Morrisett and John Reppy. The third annual ICFP programming contest, 2000. Available from `<http://www.cs.cornell.edu/icfp/>`.

[67] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[68] Lee Naish. *Negation and control in Prolog.* Number 238 in Lecture Notes in Computer Science. Springer-Verlag, New York, 1986.

[69] Lee Naish. Types and the intended meaning of logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, pages 189–216. MIT Press, Cambridge, Massachusetts, 1992.

[70] Lee Naish. Pruning in logic programming. Technical Report TR 95/16, Department of Computer Science and Software Engineering, The University of Melbourne, ”http://www.cs.mu.oz.au/ lee/papers/prune/”, 1995.

[71] Lee Naish. Parallelizing NU-Prolog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, pages 1546–1564, Seattle, Washington, August 1988.

[72] Lee Naish. Specification = Program + Types. In *Proceedings of the 7th Conference on Foundations of Software Technology and Theoretical Computer Science*, Pune, India, December 1987.

[73] Lee Naish. All solutions predicates in Prolog. In *Proceedings of the Second IEEE Symposium on Logic Programming*, pages 73–77, Boston, Massachusetts, July 1985.

[74] Lee Naish. A declarative view of modes. In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 185–199, Bonn, Germany, September, 1996.

[75] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In *Proceedings of the Conference on Parallel Architectures and Languages Europe*, pages 202–219, Eindhoven, The Netherlands, June 1991.

[76] Richard O'Keefe. *The Craft of Prolog.* The MIT Press, Cambridge, MA, 1990.

[77] Frank Pfenning. Types in logic programming. In *International Conference on Logic Programming*, page 786, 1990.

[78] C. Ponder and R. J. Fateman. Inaccuracies in program profilers. *Software Practice and Experience*, 18(5):459–467, May 188.

[79] E. Pontelli and G. Gupta. On the duality between and-parallelism and or-parallelism. In *Proceedings of the Euro Conference on Parallel Processing '95*, pages 43–54, Stockholm, Sweden, 1995.

[80] E. Pontelli and G. Gupta. Parallel symbolic computation in ACE. In C. Baral, V. S. Kreinovich, V. Lifschitz, and M. Gelfond, editors, *Logic programming, non-monotonic reasoning and reasoning about actions: Symposium — November 1995, El Paso, TX*, volume 21(2) of *Annals of Mathematics and Artificial Intelligence*, pages 359–396, Basel, Switzerland, 1997. Baltzer Science.

[81] E. Pontelli, G. Gupta, F. Pulvirenti, and A. Ferro. Automatic compile-time parallelization of Prolog programs for dependent And-parallelism. In *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 108–122. MIT Press, July 1997.

[82] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipies in C: The Art of Scientific Programming.* Cambridge University Press, Cambridge, England, 1988.

[83] POSIX pthreads. Threads extension for portable operating systems, 1992.

[84] Michael Ratcliffe and Jean-Claude Syre. A parallel logic programming language for pepsys. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 48–55, Milan, Italy, August 1987.

[85] Richard M. Reid. Transforming Mercury to Haskell. Technical Report http://www.dcs.gla.ac.uk/˜reidrm/project/root.html, University of Glasgow, Glasgow, Scotland, April 1996.

[86] J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[87] Konstantinos Sagonas and Terrance Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, May 1998.

[88] Patrick M. Sansom. *Execution profiling for non-strict functional languages.* PhD thesis, University of Glasgow, Scotland, 1994.

[89] P.M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher order functional languages. In *Proceedings of the Twentysecond Symposium on Principles of Programming Languages*, pages 355–366, San Francisco, California, January 1995.

[90] E. Shapiro. A Subset of Concurrent Prolog and its Interpreter. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers (Volume I)*, pages 27–83. MIT Press, London, 1987.

[91] Ehud Shapiro. Systems programming in Concurrent Prolog. In Michael van Caneghem and David H. D. Warren, editors, *Logic Programming and its Applications*, pages 50–74. Ablex, 1986.

[92] K. Shen, V. Santos Costa, and A. King. Distance: A new metric for controlling granularity for parallel execution. In Joxan Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP-98)*, pages 85–99, Cambridge, June 15–19 1998. MIT Press.

[93] Kish Shen and Manuel V. Hermenegildo. High-level characteristics of OR- and Independent AND-parallelism in Prolog. *International Journal of Parallel Programming*, 24(5):433–478, October 1996.

[94] Zoltan Somogyi. *A parallel logic programming system based on strong and precise modes.* PhD thesis, University of Melbourne, Australia, January 1989.

[95] Zoltan Somogyi and Fergus Henderson. The implementation technology of the mercury debugger. In *Proceedings of the Tenth Workshop on Logic Programming Environments*, pages 35–49, Las Cruces, New Mexico, November 1999.

[96] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 26(1-3):17–64, October-December 1996.

[97] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Determinism analysis in the Mercury compiler. In *Proceedings of the Nineteenth Australian Computer Science Conference*, Melbourne, Australia, February 1996.

[98] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The implementation of Mercury, an efficient purely declarative logic programming language. In *Proceedings of the ILPS '94 Postconference Workshop on Implementation Techniques for Logic Programming Languages*, Syracuse, New York, November 1994.

[99] Chris Speirs, Zoltan Somogyi, and Harald Søndergaard. Termination analysis for Mercury. Technical Report 97/9, Department of Computer Science, University of Melbourne, Melbourne, Australia, June 1997.

[100] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., second edition, 1994.

[101] Richard W Stevens. *UNIX Network Programming*. Software Series. Prentice Hall PTR, 1990.

[102] Paul Tarau. Inference and Computation Mobility with Jinni. In K.R. Apt, V.W. Marek, and M. Truszczynski, editors, *The Logic Programming Paradigm: a 25 Year Perspective*, pages 33–48. Springer, 1999. ISBN 3-540-65463-1.

[103] Paul Tarau and Veronica Dahl. A Logic Programming Infrastructure for Internet Programming. In M. J. Wooldridge and M. Veloso, editors, *Artificial Intelligence Today – Recent Trends and Developments*, pages 431–456. Springer Verlag, LNAI 1600, 1999.

[104] James Thom, Justin Zobel, and Editors. NU-Prolog reference manual. Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.

[105] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadephia, USA, May 1996.

[106] K. Ueda. Guarded Horn Clauses. In *Proc. Int. Conf. on Logic Programming*, volume 221, pages 168–179. Springer-Verlag, Berlin, 1986.

[107] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim S. Leask, and James Harland. The Aditi deductive database system. *VLDB Journal: Very Large Data Bases*, 3(2):245–288, April 1994. Electronic edition.

[108] D. Viswanathan and S. Liang. Java virtal machine profiler interface. *IBM Systems Journal*, 39(1):82–95, 2000.

[109] P. L. Wadler. How to replace failure by a list of successes. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 113–128. Springer-Verlag, Berlin, DE, 1985. Lecture Notes in Computer Science 201Proceedings of. Conference at Nancy.

[110] Philip Wadler. Linear types can change the world! In *Proceedings of the IFIP TC2 Conference on Programming Concepts and Methods*, pages 547–566, April 1990.

[111] David H. D. Warren. Higher-order extensions to Prolog: are they needed? In J. E. Hayes, Donald Michie, and Y-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood, 1982.

[112] David H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, California, October 1983.

[113] Richard Warren, Manuel Hermenegildo, and Saumya K. Debray. On the practicality of global flow analysis of logic programs. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 684–699, Seattle, Washington, August 1988.

[114] Harald Westphal, Philippe Robert, Jacques Chassin, and Jean-Claude Syre. The pepsys model: combining backtracking, and- and or-parallelism. In *Proceedings of the Fourth IEEE Symposium on Logic Programming*, pages 436–448, San Francisco, California, August 1987.