Strong Modes can Change the World!

Fergus Henderson

Department of Computer Science
University of Melbourne
Parkville 3052
Australia
fjh@munta.cs.mu.oz.au

November 8, 1993

Supervisor: Dr. Lee Naish

Abstract

We investigate compile-time garbage collection, structure reuse, and destructive assignment for logic programming languages, using an expressive strong mode system. By attaching groundness and uniqueness information to each node of a variable's type tree, the mode system forms the basis for carrying out these optimizations. Our system allows polymorphic mode definitions to achieve a high level of expressiveness. We implement a simple preprocessor that does structure re-use optimization for Prolog programs annotated with mode declarations. Finally, we show how our mode system can be used to provide declarative I/O for logical programming languages.

Acknowledgments

I'd like to thank Lee Naish (my supervisor), Harald Søndergaard, Jeff Schultz, Alistair Moffat, Will Winsborough, my parents, my fellow honours students, and my flat-mates.

I must apologize to Phil Wadler for the title, shamelessly adapted from his "Linear Types Can Change the World!"[Wad90].

Contents

Chapter 1

Introduction

This honours report explores the use of certain optimization techniques for logic programming languages, using a strong mode system as the basis for these optimizations. We discuss the motivations for investigating these optimizations in detail in Chapter 2; For now, it suffices to say that we consider these to be the key optimizations required to achieve the efficiency of conventional languages.

We assume that the reader has a good general knowledge of logic programming. Some knowledge of the standard techniques used to implement Prolog (such as how terms and variables are stored in memory) would be useful but is not essential.

The optimizations in question have been variously known as "compile-time garbage collection", "structure re-use", and "destructive assignment". These terms are rather ambiguous and over-lapping in general usage; we will take the opportunity now to clearly delineate between the different optimizations, by adopting the following definitions:

- *compile-time garbage collection* refers to detecting when areas of memory become unreferenced, and returning them to the free memory pool.

- *structure reuse* also requires detecting when areas of memory become unreferenced, but instead of returning them to the free memory pool, we instead immediately reuse the memory to build a new term. Structure reuse has also been known simply as *local reuse* [FW91].

- *destructive assignment* works by linking an input argument with an output argument, and replacing these with a single argument whose value is destructively updated to replace the input by the output. Destructive assignment is different from structure reuse because the former reuses whole terms, whereas the latter reuses one memory block at a time.

The distinctions between the different optimizations are discussed further in Chapter 4.

To implement these optimizations requires both data-flow analysis and determinism analysis. Determinism analysis is required to ensure that memory that we wish to reuse will not be needed again on backtracking. This report does not investigate determinism analysis; however, such analysis is quite straight-forward once you are in possession of the necessary data-flow information[1], so it should be easy to extend our results to include

[1]For example, Lee Naish's "nac" preprocessor does determinism analysis using the data-flow information provided by NU-Prolog's "when" declarations [Nai86]

determinism analysis. (For our prototype implementation, we have a “safe” destructive assignment primitive, which trails all destructive assignments to terms that are older than the most recent choice point; such destructive assignments are then undone on backtracking. In the case where trailing is not required, this incurs only a small overhead for the test, but ensures the soundness of our transformations; if trailing is required, then the “optimizations” turn into “pessimizations”, but the code will still function correctly.)

We rely on mode declarations to provide us with most of the data-flow information required. This provides a great simplification and has many other advantages which are discussed further in Chapter 2 and in Section 1 of Chapter 3. The mode system we suggest is more expressive than any we have seen before. It is based on the idea of a mode as a mapping from one state of instantiatedness to another. Following [Som89] we attach mode information to every node of a variable's type tree, and we therefore allow recursive mode declarations. We also allow polymorphic modes, where mode declarations may contain instantiatedness variables in the same way that polymorphic type declarations may contain type variables. Finally, for the purposes of our optimizations, we allow mode declarations to specify uniqueness of references, so that we know when a term may be reused.

The rest of the report is organized as follows. Chapter 2 explains why the particular optimizations in question are so important. Chapter 3 defines our mode system, and Section 2 of Chapter 4 defines a syntax for expressing these modes and gives some examples. The remainder of Chapter 4 discusses implementation of the optimizations; we give an algorithm for mode checking, and show how this can be modified to perform compile-time garbage collection and structure reuse optimization. Finally Chapter 5 looks at an application of our mode system for expressing input/output using logically pure, declarative code.

Chapter 2

Motivation

2.1 Use of Logic Programming Languages

Logic Programming offers many advantages — high-level declarative programming, implicit memory management, executable specifications, etc. — yet remains relatively unsuccessful compared to conventional languages. A major factor preventing more widespread acceptance of logic programming languages is their relative lack of efficiency when compared to conventional procedural languages like C and Pascal. Compilation techniques are improving, and recent developments such as [Tay90, RD90, HSC$^+$90] have shown that for some benchmarks, Prolog can be faster than C; however it remains the case that many common algorithms (for example, hashing) are much more efficient in languages such as C. For these reasons, Prolog and other logic programming languages are still not competitive when it comes to raw speed, and as such, they remain niche programming languages.

One important reason why many Prolog programs are not implemented as efficiently as conventional language programs is the use of the "Logical Variable". Whereas variables in conventional languages represent areas of machine storage, variables in Prolog are a more abstract and powerful notion, but one that is more difficult to map to the underlying hardware; in particular, it is not possible to directly express the low-level notion of destructive assignment assignment in Prolog. Standard programs in C or Pascal will use destructive assignment to modify large data structures, but with current Prolog implementations, equivalent Prolog programs end up copying whole data structures in order to modify them.

However, a sophisticated compiler could take advantage of situations in which structures are copied but the old copy is then discarded, and generate code using destructive assignment as is done for the procedural versions. More generally, the compiler may detect when certain areas of memory will no longer be referenced, and can either re-use these areas immediately, or return them to the pool of unused memory, thus avoiding the need for run-time garbage collection for these areas. These important optimizations provide the motivation for this report.

2.2 Importance of Destructive Assignment

We consider destructive assignment, and the other optimizations that it subsumes, to be some of the most important optimization opportunities which "industrial" Prolog compilers has not yet taken advantage of. It is important for the following reasons:

- It is one of the few optimizations that will give trans-linear speedup. Without destructive assignment, modifying a structure of size N is an order O(N) operation; with destructive assignment, it is O(1).

- When you re-use a structure, you avoid allocating new memory, *and* you avoid the overhead of having to garbage collect the old structure. This can also dramatically reduce memory bandwidth, often the limiting factor in Prolog computations, and improve locality and cache performance.

- It allows arrays and other data structures such as hash tables, binary trees, etc. to be implemented efficiently.

- It avoids the need for "tricky" schemes like open-ended data structures. Such schemes make programs more difficult to comprehend, and do not sit well with type/mode systems. (The worst cases involve explicit calls to non-logical predicates such as `var/1`; programmers using such code have sacrificed declarative semantics in the quest for efficiency.) For example, it means that the use of the ground representation for meta-programming need be no less efficient than using the non-ground representation; we believe that once this efficiency barrier is overcome, there would be more than sufficient incentive for programmers and language implementors to switch to using the logically pure ground representation.

- It means that deterministic code can approach the speed of conventional languages.

- And it means that algorithms from conventional languages can be transliterated into Prolog fairly directly, without having to worry that the resulting program may be un-usably inefficient.

2.3 Previous Approaches

To date there has been a fair amount of research on algorithms for detecting structure sharing and opportunities for structure reuse and compile-time garbage collection, generally involving global analysis via abstract interpretation. Some examples of this sort of research are [MWB90, FW91]. However there have been few if any actual implementations in real Prolog systems. Although we expect that in the future, global analysis of programs will become the norm for optimizing compilers, at the present time it appears that the full complexities of global analysis are a barrier to successful implementations.

The constraint logic programming language "Trilogy" is the only logic programming language that we know of which does allow destructive update. Unfortunately it is done in a what appears to us to be a fairly ad-hoc and limited manner, no doubt for pragmatic reasons. The result is like a mixture of Prolog and Pascal. Trilogy has four possible modes: input, output, input-output and symbolic. Input variables must be ground on procedure entry, while output variables must be ground on procedure exit. Symbolic variables correspond to unrestricted logical variables as in Prolog[1]. Input-output variables actually represent a pair of variables, one input and one output, and may be destructively updated using an assignment operator "`:=`", just as in Pascal. In [Vod88], Voda laments the "inevitable loss of readability" when using these input-output modes, but notes that for large arrays the inefficiency of the alternative is prohibitive.

Although Trilogy's destructive assignment is no doubt a great boon to practical programmers, we believe that a more general approach that blends better with the underlying logic and semantics of the language is both possible and practical. The next chapter outlines the mode system which is the foundation of our more general approach.

[1] Although in addition, since Trilogy is a constraint language, symbolic variables may also have constraints attached to them.

Chapter 3

Strong mode systems

3.1 A New Approach to Structure Re-use

A mode system allows programmers to express more about their programs, by constraining the directions of data-flow; in particular, mode systems generally allow one to specify a predicate's required inputs and sometimes also its outputs. Mode systems have been around since at least as early as 1977 [War77], and have proven particularly useful for coroutining and parallel languages, such as NU-Prolog (whose `?- when` declarations are a form of modes) [TZ86] and Parlog [CG85, CDB+90].

There is a strong analogy between type systems and mode systems which we have found to be a fruitful source of ideas. Strong mode systems share most of the advantages of strong type systems. In particular, strong mode systems offer the following benefits:

- More efficient code: predicates can often be used in several different ways, but this flexibility comes at the cost of efficiency. By generating separate code for the different modes of a predicate we can regain that lost efficiency. Importantly, generating separate code for each mode makes it much more practical to compile directly to machine code. The information provided by mode declarations allows a variety of different optimizations.

- Quicker compilation: mode systems provide information that an optimizing compiler would otherwise have to obtain by possibly lengthy analysis of the program.

- More predictable compiler optimization behaviour: with strong mode systems, it is possible for the programmer to (in effect) suggest to the compiler when to apply certain optimizations, and if the mode system does not reject these suggestions (by complaining that the code contains a mode error) then the programmer can be confident that the compiler will be able to carry out those optimizations. By contrast, techniques such as global analysis via abstract interpretation make it more difficult for the programmer to determine what sort of optimizations the compiler will be able to perform on a given piece of code.

- More readable and maintainable programs: just like types, modes are an important form of documentation. They make it easier to understand the dynamic behaviour of the code, and they make clear a predicate's intended use(s).

- More reliable programs: the use of strong modes, like the use of strong types, eliminates a whole class of programming errors.

Furthermore, an appropriate mode system can help to make it possible to develop efficient programs incrementally. Initially the program can be developed without using modes, or using simple mode declarations that impose few constraints. Later, the programmer can go back and refine the precision of the mode declarations to enable the compiler to generate more efficient code. In this way, the benefits of strong modes are available if you require them, but it is possible to prototype code quickly without worrying about modes[1].

Mode systems are also very useful for providing control information in coroutining and parallel systems, and even for purely sequential systems they can be used to re-order conjunctions (using a mode-dependent computation rule), but in this report we restrict our attention to sequential systems with a left-to-right computation rule.

3.2 Type System

We base our mode system on a simple polymorphic regular-tree style type system as in [MO84]. Types are represented as and/or trees; recursive types are allowed, for which the type tree is conceptually infinite. The type system also allows polymorphic types, parameterized by one or more type variables.

There is still considerable discussion about type systems for logic programming, and we refer the interested reader to [Pfe92] and [Vod88]. However we believe that the ideas that we will present would equally well apply to just about any reasonable type system, so we shall not dwell on this issue further.

3.3 Basic Mode System

In the remainder of this chapter we develop the theoretical foundation for our mode system. We avoid going into details of syntax and implementation here, but the reader may find some of these concepts easier to understand after looking at the examples in Section 4.2.

Following [Som89], we attach mode information to each or-node of the type tree. As we remarked earlier, many researchers have considered the notion of modes before, but we believe that our approach has novel aspects; one key insight is to note that the "mode" of a predicate can be considered as a mapping from the initial state of instantiation of the arguments of the predicate, to their final state of instantiation.

At any given point of execution, each or-node in the type tree for a variable has an *instantiatedness* which is one of

- *unborn*,

[1] In practice, however, the author finds that the extra redundancy and compile-time checking provided by strong typing makes it possible to develop *working* prototypes much more quickly than if optional type declarations are omitted, despite the extra typing involved [no pun intended], and we expect that the situation to be the same for mode declarations.

- *free*, or
- *bound*.

We sometimes refer to these different possibilities for instantiatedness as *instantiatedness tags*.

An *instantiatedness tree* is an assignment of instantiatedness to each or-node of a type tree, which satisfies the following constraints:

- All the children of each unborn or free node are unborn.
- All the children of a bound node are either bound or free.
- The top-level (root) node is not unborn.

We associate instantiatedness trees with variables, not with occurrences of variables.

An instantiatedness tree does not specify the exact instantiation pattern (i.e., binding) of a variable, because at each bound or-node, it leaves the corresponding value unspecified; this then means that the instantiatedness tree must include the instantiatedness of *every* sub-tree of bound or-nodes, not just one of them.

Furthermore, we always allow variables to be *more* instantiated than specified by their instantiatedness trees[2]. This means that specifying the instantiatedness to be "free" actually doesn't provide any information at all — it just means "possibly unbound". In contrast "bound" always means definitely bound. The reason that we need to define "free" in this way is because of sharing between different variables. If we unify two free variables, and then unify one of the variables with a constant, both variables will become bound. Rather than try to keep track of all the possible sharing between variables, we simply adopt a conservative definition for "free".

If a node's instantiatedness is "unborn", then it means that the node's parent node may not yet be bound. Such nodes may have no corresponding node in the variable's current binding, and thus no actual machine storage allocated to them; their existence in that case is purely conceptual. The "unborn" instantiatedness tag is present to ensure a one-to-one correspondence between the nodes of the type tree and the nodes of the instantiatedness tree. Another way of viewing matters is to say that an instantiatedness tree maps onto a subtree of the type-tree; where the leaves of the instantiatedness tree are "free" nodes, the type tree may be deeper than the instantiatedness tree. Choosing between these two alternatives is basically just an aesthetic issue — semantically there is little difference.

For recursive types, the instantiatedness tree will be infinite, since the completely unrolled type tree is infinite. We also introduce *regular instantiatedness trees*, which are the subset of instantiatedness trees that can be defined using a (finite) recursive definition. Practical implementations will of course be restricted to using only regular instantiatedness trees.

As execution progresses, variables may become more instantiated. A *mode* is a mapping from an initial instantiatedness to a final instantiatedness which satisfies the following constraints: As the execution progresses, an unborn node may become free or bound,

[2] The instantiatedness tag free-unique, defined later, is an exception to this rule: free-unique nodes must always be free.

and a free node may become bound, but a bound node cannot become free or unborn, and a free node may not become unborn.

There is some correspondence between these modes and the modes used in the language Ptah [Som89]; we give the following names to our modes, and for those familiar with Ptah we note the corresponding modes in Ptah:

unused:	unborn $\rightarrow$ unborn	
outfree:	unborn $\rightarrow$ free	(Ptah's outfree)
infree:	free $\rightarrow$ free	(Ptah's infree)
outbound:	unborn $\rightarrow$ bound	(Ptah's outbound)
created:	free $\rightarrow$ bound	(Ptah's outbound)
inbound:	bound $\rightarrow$ bound	(Ptah's inbound/infree)

A *possible mode tree* is an assignment of modes to each node of the type tree of a variable. The *initial instantiatedness tree* for a mode tree is the tree corresponding to the initial instantiatedness of each mode in the type tree. Similarly the *final instantiatedness tree* for a mode tree is the tree corresponding to the final instantiatedness of each mode in the type tree. A possible mode tree is a genuine *mode tree* if both its initial instantiatedness tree and also its final instantiatedness tree do indeed satisfy the constraints for instantiatedness trees.

Alternately, a *mode mapping* can be defined as a mapping from an initial instantiatedness tree to a final tree. The only difference between a mode mapping and a mode tree is that the former is a mapping between trees, while the latter is a tree of mappings. (In the syntax we suggest below, we use mode mappings, because they allow us to use a simple notation for instantiatedness trees which doesn't need to refer to "unborn" nodes.)

The mode mappings

$$ground \rightarrow ground$$

and

$$free \rightarrow ground$$

correspond to the standard notions of *input* and *output* modes. Here we are using "ground" to indicate a mode tree all of whose or-nodes are bound, and "free" in this context indicates a mode tree whose top-most (root) or-node is free (hence all the other or-nodes must be unborn). On the other hand mode mappings can express many other, more complex data-flow patterns; for example, it is easy to express the equivalent of Parlog's weak input modes, where only the outer-most functor need be bound[CDB+90]. We will give examples later when we discuss the syntax in Chapter 4.

A term is *approximated by* an instantiatedness tree if for every or-node in the type-tree for that term,

- if there is a corresponding node in the term, and that node is a free variable, then the corresponding instantiatedness in the instantiatedness tree must be "free";

- and if there is a corresponding node in the term, and that node is a functor or constant, then the corresponding instantiatedness in the instantiatedness tree must be either "free" or "bound".

A *mode declaration* for a predicate is an assignment of a mode tree for each argument position of that predicate. A mode declaration is considered as an assertion by the programmer that every call of the predicate with argument terms that are approximated by the initial instantiatedness trees of the mode declaration will (if it succeeds) result in a binding of those argument terms that is in turn approximated by the final instantiatedness trees of the mode declaration. We refer to the assertion associated with a mode declaration as the *mode declaration constraint*.

A *mode set* for a predicate is a set of mode declarations for the predicate. (Later we impose some further constraints on the allowable mode sets, to ensure that the mode checking algorithm is linear rather than exponential). A mode set is considered as an assertion by the programmer that the predicate will only be called with argument terms that are approximated by the initial instantiatedness trees of one of the mode declarations in the set; i.e., that the specified mode declarations are the only allowed modes for this predicate. We refer to the assertion associated with a mode set as the *mode set constraint*.

Now we come to defining *well-modedness*. We want to reject programs for which the mode declaration constraints or the mode set constraints are not always satisfied. In general, however, this would be undecidable — a problem for both the compiler and the programmer. We want a definition of well-modedness that is easy for the compiler to determine, and easy for the programmer to understand. Rather than requiring global analysis of the entire program, determining if a predicate is well-moded should require only local analysis using the clauses of the predicate and the mode declarations for any called predicates. This prompts the following definition:

A predicate is *well-moded* with respect to a given mode declaration iff it is possible to prove that

- the corresponding mode declaration constraint is correct, and that
- the mode set constraints for the called predicates are not violated by this predicate

given only

- the code for the predicate
- the assumption that the mode declaration constraints for the called predicates all hold, for those particular calls
- the details of the computation rule

Note that the proof cannot use the definitions of the called predicates; it is only allowed to use their mode sets. This ensures that only local analysis is required.

The reason that this definition is phrased in terms of the existence of a proof, rather than truth of an implication, is simply so that we can make it quite clear what assumptions are allowed.

The definition is not merely theoretical — in Chapter 4, we present a mode-checking algorithm that will determine whether a predicate is well-moded with respect to a given mode, i.e. whether such a proof exists. We do not, however offer proof of soundness or completeness for our algorithm. Soundness (i.e., that the algorithm accepts *only* well-moded predicates) should be fairly obvious from the description of the algorithm, but completeness (i.e., that the algorithm accepts *all* well-moded predicates) is not immediately obvious, despite our careful choice of definition for well-modedness; in any case we later modify the algorithm in ways that definitely eliminate completeness, in order to ensure linear rather than exponential performance.

We say that a predicate is well-moded (without any qualifications) if it is well-moded with respect to all the mode declarations in its mode set.

A program is well-moded if all its predicates are well-moded.

A query is well-moded (with respect to a program) if it is possible to prove that the mode set constraints for the called predicates are not violated by the query. The assumptions allowed in the proof are the same as those allowed for proving a predicate to be well-moded.

3.4 Relationship with Abstract Interpretation

The instantiatedness trees that we have defined form an domain suitable for abstract interpretation. Indeed mode checking corresponds loosely to abstract interpretation, except that since we have mode declarations available, we need only make a single pass over each predicate; it is not necessary to iterate multiple times until a fixpoint is reached. Full abstract interpretation of the type originally described in [Myc81] would yield a *mode inference* algorithm.

3.5 Extended Mode System

Although using strong modes of the form defined in the basic mode system above would have the many advantages which we have previously discussed, our specific interest is in using the mode system to capture the data-flow information that is required for structure reuse and destructive assignment.

We now proceed to expand our mode system to record the number of references to each term (zero, one, or many). Recording such information in the mode system is very closely related to the way Wadler embeds uniqueness information into the type system with his notion of "linear types" for functional languages [Wad91, Wad90][3]. Using these concepts leads us to the following refined definition of instantiatedness.

At any given point of execution, each or-node in the type tree for a for a variable has an *instantiatedness* which is one of

- *unborn*,
- *free*,
- *bound*,
- *free-unique*, indicating that this node of the type-tree is unbound, and that this variable is the only live reference to that node

[3]Note, however, that we became aware of this correspondence only after the event; it was Somogyi's use of modes to represent unique producers in [Som89] that originally led to investigate these ideas.

- *bound-unique*, indicating that this node of the type-tree is bound, and that this variable is the only live reference to that node

- *dead*, indicating a variable which will not be referred to again.

With these instantiatedness tags, it is possible to specify the mode of predicates which may safely re-use some of their inputs, either by returning them to the free memory pool (compile-time garbage collection), or by destructively updating them. For example a node whose mode is initially "bound-unique" may be safely re-used after the last reference (i.e. when it becomes "dead"). The mode mappings

$$ground\text{-}unique \to dead$$

and

$$free\text{-}unique \to ground\text{-}unique$$

correspond to *unique input* arguments and *unique output* argument modes, analogous to the input and output argument modes mentioned earlier, although again it is possible to express much more complicated mode mappings in cases where that is necessary.

In order for the compiler to make use of this mode information to actually generate code that does destructive update, it is necessary to record some additional details. For compile-time garbage collection, it suffices to record just the nodes which are reusable; in this case we would include the following additional instantiatedness tag:

- *dead-reusable*, indicating that this variable is the only reference to this node of the type-tree, and that this variable which will not be referred to again — and hence that the variable may safely be re-used[4].

For structure reuse and destructive update, it is also necessary to record the value to which each bound node is actually bound, for those nodes with zero or one references, when that information is available. To do this we would include two additional instantiatedness tags:

- *bound-unique(Term)*,

- *dead-reusable(Term)*,

The refined definition for instantiatedness used in the extended mode system leads us now to a refined definition of instantiatedness tree. An *instantiatedness tree* is an assignment of instantiatedness to each or-node of the type tree for a variable, which satisfies the following constraints:

- All the children of each unborn, free or free-unique are unborn.

- All the children of each bound-unique or dead-reusable node are either bound, free, dead, bound-unique, free-unique, or dead-reusable (i.e., anything except unborn).

- All the children of a bound node are bound, free, or dead.

- The top-level node is not unborn.

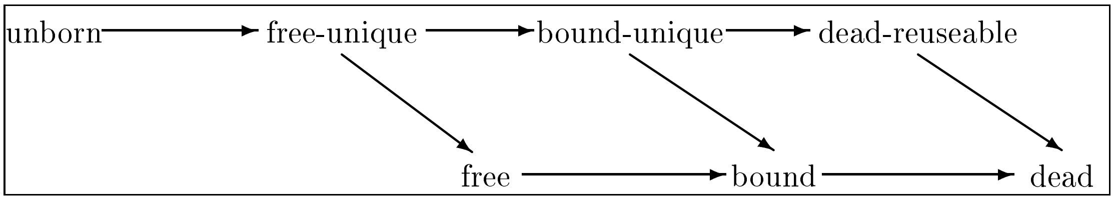

Figure 3.1: Instantiatedness transitions during (forward) execution

As execution progresses, the nodes in a variable's type tree may become *more instantiated*; we use this term in a general sense, although strictly speaking, since our "instantiatedness" tags encode more information than just free/bound, this may be slightly misleading. These possible transitions are summarized in Figure 3.1, which defines the partial order that we are referring to; the arrows go from the least instantiated "free-unique" to the most-instantiated "dead". We define a *mode* as a transition from an initial instantiatedness to a final instantiatedness such that final instantiatedness is not less instantiated than the initial instantiatedness.

3.6 Aliasing

In this section, we introduce another instantiatedness tag "same-as(Var)" to record aliasing information. Like the dead-reusable, dead-reusable(Term), and bound-unique(Term) instantiatedness tags, same-as(Var) is intended to be used internally by the compiler or preprocessor, and is not intended to be used directly by the programmer in mode declarations.

As we remarked earlier, we do not attempt to keep track of all possible sharing between variables. This means that in some circumstances, we may lose some information. For example, after

```
X = Y, X = a
```

we do not know that Y is bound.

However this situation occurs too often for us to simply ignore the issue. This is especially true because (to keep other parts of the analysis simple) our mode-checking algorithm often introduces sharing of this sort. To improve the precision of our mode system, we allow the mode system to record *aliases*, which occur when one variable is known to be definitely identical with another. We also allow aliases between some component of a variable (i.e. some node in the variable's type tree other than the root node) and a different variable. For these purposes we introduce another instantiatedness tag "same-as(Var)". If when traversing the mode tree for a variable X, we come across a node with the instantiatedness tag same-as(Y), then it means that the actual instantiatedness for this node and all it's descendants should be obtained by looking up the mode tree for Y. Thus the same-as(Var) tag means simply that Var is an alias for this node.

In most situations this recording of aliases is transparent to the rest of the mode system; we must remember to dereference aliases when necessary, but otherwise it does

[4] A more orthogonal but perhaps less informative name would be "dead-unique"

not require any changes. Aliases are introduced whenever we encounter a unification between two variables, or between a variable and a term containing other variables.

So for example, if the instantiatedness of both X and Y is free, and we see a unification such as

```
X = Y
```

then we set the instantiatedness tag of X to "same-as(Y)" (or vice versa — it makes no difference). Then if we see a unification such as

```
X = a
```

we look up the instantiatedness of X, and when we find it to be "same-as(Y)", we in turn look up the instantiatedness of Y, which turns out to be "free". We are unifying a free variable with a constant, which will bind the variable, so we set the new instantiatedness for Y to be "bound". The instantiatedness for X remains "same-as(Y)". Now we have correctly determined that after the two unifications, both X and Y are bound.

The other case when recording aliases is important is when we unify two variables whose instantiatedness is free-unique or bound-unique. Normally if we unify two unique variables, both of which are referenced again after the unification, then both variables become multiply referenced and lose their "uniqueness". However if instead we record the alias, we can consider these two references as equivalent to a single reference, and so avoid losing uniqueness. Doing this means that we must be slightly more careful about when we consider a variable dead-unique: we must not consider such variables dead-unique until after the last occurrence of *all* of its aliases. Another alternative which we have not explored so far would be to keep a compile-time reference count for these variables.

3.7 Polymorphic Modes

The mode system as given is quite precise but is still fairly restrictive. In particular it is not adequate for representing the modes of predicates which take arguments of polymorphic types. The smallest example which demonstrates the difficulty is the following unification predicate `eq/2`, but the problem is that same for more complicated examples such as append.

```
?- type eq(T,T). % Polymorphic type: T is a type variable
eq(X,X).
```

We would like to be able to provide a small number of mode declarations for this predicate that would suffice for any possible use that we might like to make of it. However this is not possible with the monomorphic mode system that we have defined so far. This is because different calls to `eq/2` such as

```
eq(f(a),f(X))
```

and

```
eq(g(a),g(X))
```

each require their own mode declaration in order for the mode system to determine that X will be bound after the calls. To cater for all such possible calls would require an infinite number of mode declarations for `eq/2`.

To redress the situation, we allow *polymorphic mode declarations.* In the same way that polymorphic types allow one to specify a type leaving unspecified the value of certain type variables, with polymorphic modes one can specify a mode but leave the value of certain *instantiatedness variables* unspecified. For example the following mode for eq/2 will correctly handle all the above cases:

```
?- mode eq(A -> X, B -> X).
```

Here `A`, `B`, and `X` are instantiatedness variables. This mode declaration specifies that `eq/2` may be called with any initial instantiatedness of its arguments, and that the resultant final instantiatedness of its arguments are the same (since they both correspond to the same instantiatedness variable `X`).

Polymorphic modes declarations never specify an impossible mode such as free $\rightarrow$ ground; the instantiatedness parameters to these declarations are implicitly universally quantified variables ranging over the set of instantiatedness trees for which the resulting mode mappings are consistent.

We present some more examples of polymorphic mode declarations in Section 4.2.

Chapter 4

Implementation

4.1 Overview

In an ideal system, strong modes and types would be an integral part of the language, and the mode information would be used by the compiler to perform compile-time garbage collection, structure reuse, and destructive assignment optimization, in addition to a being used for a variety of other purposes as discussed earlier. However in the short term, it is more practical to implement a mode system on top of conventional Prolog, using Prolog's "`?-`" declarations for the mode annotations. Instead of the requiring the compiler to do mode checking, and perform the desired optimizations, we can construct a separate preprocessor to do this task. The compiler need only provide some simple built-in predicates to do the actual deallocation or destructive assignments.

4.2 Syntax for Mode Annotations

The syntax for mode annotations that we suggest is as follows. We use standard Prolog syntax for the clauses; all annotations are in the form of "`?-`" declarations. Mode declarations are in theory *optional*: in the absence of a mode declaration, the system can safely assign the mode $free \rightarrow free$ to each argument; however in practice if you give a mode declaration for a predicate, then you will generally need to give mode declarations for all the predicates that it calls to avoid a mode error.

We start with syntax for types used in the Naish-Mycroft-O'Keefe type checker.

```
?- type list(T) ---> nil | cons(T,  list(T)).
?- type intlist = list(int).
?- type list = list(any). % list of anything
```

We extend this firstly to allow definitions of instantiatedness trees, using "`?- inst`" definitions. The names `free` and `free_unique` are built-in `inst` definitions indicating an instantiatedness tree whose top node is as indicated and whose other nodes are all unborn. The names `ground` and `ground_unique` are built-in `inst` definitions indicating an instantiatedness tree all of whose nodes are bound or bound-unique respectively. In addition we allow explicit declaration for more complicated instantiatedness patterns, for example:

```
?- inst list_skeleton ---> bound( nil | cons(free, list_skeleton) ).
?- inst unique_list_skel --->
        bound_unique( nil | cons(free_unique, unique_list_skel) ).
```

These "inst" definitions can be used in "mode" definitions, to defined mode mappings. For example, the following mappings are used so often that we have made them built-in:

```
?- mode input :: ground -> ground.
?- mode output :: free -> ground.
?- mode uniq_input :: ground_unique -> dead.
?- mode uniq_output :: free_unique -> ground_unique.
```

We allow `inst` and `mode` definitions to be polymorphic. Here are some examples of the use of polymorphic definitions:

```
% another way to do list_skeleton
?- inst list_skel(M) ---> bound(nil | cons(M, list_skel)).
?- inst list_skeleton = list_skel(free).

?- mode output(X) :: free -> X
?- mode input(X) :: X -> X
?- mode output_list_skeleton = output(list_skeleton).
?- mode input_list_skeleton = input(list_skeleton).
```

Finally, we arrive at predicate mode declarations. There are two forms of these: the first is more convenient for predicates with multiple modes,

```
?- pred length(list, int).
?- mode length(output_list_skeleton, input).
?- mode length(input_list_skeleton, output).
```

while the second form is more convenient for predicates with only one mode, since it allows you to combine the mode and type declarations:

```
?- mode square(int:input, int:output). % square a number
?- mode member_chk(T:input, list(T):input). % test membership
```

Predicate mode declarations may also be polymorphic. The parameters to these declarations are implicitly universally quantified variables ranging over the set of instantiatedness trees for which the resulting mode mappings are consistent. We give a longer example leading up to the mode declarations for append.

```
?- inst list_skel(X) ---> bound(nil | cons(X, list_skel)).
?- mode input_list_skel(X, Y) :: list_skel(X) -> list_skel(Y).
?- mode output_list_skel(X) :: free -> list_skel(X).
```

It might be useful to have an abbreviation `uniq(Mode)` meaning `Mode` with all the instantiatedness tags (`free`, `bound`, `ground` and `dead`) replaced with their unique versions. The alternative involves a certain amount of duplication:

```
?- inst uniq_list_skel(X) ---> bound_uniq(nil | cons(X, uniq_list_skel)).
?- mode uniq_input_list_skel(X, Y) ::
        uniq_list_skel(X) -> uniq_list_skel(Y).
?- mode uniq_output_list_skel(X) :: free_uniq -> uniq_list_skel(X).

        % The two basic modes of append
?- pred append(list, list, list).
?- mode append(input_list_skel(X,Y), input_list_skel(X,Y),
               output_list_skel(Y)).
?- mode append(output_list_skel(Y), output_list_skel(Y),
               input_list_skel(X,Y)).

        % A mode for destructive append
?- mode append(uniq_input_list_skel(X,Y), uniq_input_list_skel(X,Y),
               uniq_output_list_skel(Y)).
```

If append is called with all arguments free, then it will generate infinitely many solutions. Generally this indicates a programming error, so we would not give a mode declaration for this use of append. The same effect could be achieved by inserting an explicit call to a list construction predicate if necessary. If we *were* to include a mode declaration for append called with all arguments free, however, then it would look like this:

```
?- mode append(output_list_skeleton, free->free, free->free)
```

Note that final instantiatedness of the third argument is simply `free`, whereas in fact the third argument may be bound to a list with an uninstantiated tail. This is because `free` actually means only "possibly unbound", not "definitely free", as we discussed earlier.

We plan to use this syntax for our prototype preprocessor, although at the moment for simplicity it only accepts a subset of these constructs.

4.3 Mode Checking Algorithm

Here we give a simple algorithm for mode checking using the basic mode system.

mode-check: Check that a program is well-moded.

for each predicate
  for each mode declaration in the predicate's mode set
    for each clause of the predicate
      mode-check-clause

mode-check-clause: Check that a clause is well-moded.

1. move the head unification into the body of the clause

2. initialize the instantiatedness trees for the variables in the head of the clause to be the initial instantiatedness trees of this mode declaration of the predicate

3. **interpret-goal** the body of the clause

4. check that the instantiatedness trees for the variables in the head of the clause are **compatible** with the final instantiatedness trees of this mode declaration of the predicate

interpret-goal: Given a goal, and a given the instantiatedness trees for the non-local variables in the goal, determine what the instantiatedness of the variables will be after executing the goal. This is done by abstract interpretation of the goal, although unlike full abstract interpretation, since we know the mode declarations for the called predicates we can do this locally using a single pass instead of globally, iterating until a fixpoint is reached.

1. if the goal is:

   (a) a conjunction: **interpret-goal** each half of the conjunction in succession

   (b) a disjunction: **interpret-goal** each half of the disjunction, and let the new instantiatedness trees be found by taking the **greatest-lower-bound** of the instantiatedness trees resulting from each half (handling of disjunctions is discussed further below)

   (c) a unification:

      i. first initialize the instantiatedness trees for any new variables introduced in this unification

      ii. then **abstractly unify** the instantiatedness trees of the terms being unified

   (d) a predicate call:

      i. firstly separate out the argument unification by splitting the call into some unifications followed by a predicate call with all arguments being variables

      ii. **interpret-goal** the unifications

      iii. non-deterministically choose a mode declaration of the predicate being called

      iv. check that the instantiatedness trees of the argument variables are **compatible** with the initial instantiatedness trees of the mode declaration of the predicate being called

      v. replace the instantiatedness trees of the argument variables with the final instantiatedness trees of the mode declaration of the predicate being called

   In a real system we would also handle cases such as if-then-else, cut, various special built-in predicates, etc.

Note that because of the way we handle disjunctions by taking the greatest-lower-bound, we may lose some information and this may actually cause us to reject clauses that are well-moded by the definition in Chapter 3. The alternative would be to mode-check any part of a clause which follows a disjunction twice, once for each possible mode of the disjunction (assuming that the modes were indeed different). However we avoid this for two reasons: firstly, this would make the mode checking algorithm exponential in the number of disjunctions, and secondly, since the generated code might depend on the modes, this could also cause the size of the code generated by the compiler to be exponential in the number of disjunctions. We do not feel that rejecting such clauses imposes any great restriction on the language; if necessary, the programmer may achieve the same effect by transforming clause bodies of the form

$$(A; B), C$$

into

$$(A, C; B, C)$$

which makes the resultant code duplication explicit.

If the mode check fails, then we may have to back-track to a different choice for the mode declaration of one of the called predicates. Again, this may cause the mode checking algorithm to be exponential in the length of the clause body. We can avoid this by requiring that given the initial instantiatedness for the arguments (and for the extended mode system, given also the knowledge of which variables will become (textually) dead after this call), it should be possibly to uniquely determine the resultant final instantiatedness for the arguments. This is in fact not too difficult to ensure, as we will discuss.

Suppose that there are two different mode declarations for which the initial instantiatedness trees are compatible with the current instantiatedness of the arguments. There are several cases to cover. Firstly, if the initial instantiatedness trees of one mode are all more strict than the initial instantiatedness trees of the other, then just choose the former mode. For example, we always choose

```
append(ground -> ..., ground -> ..., free -> ...)
```

over

```
append(free -> ..., free -> ..., free -> ...)
```

in cases where both mode declarations could be used.

Secondly, if that is not the case, then we must check whether the of final instantiatedness trees are different for the different modes. In most cases, the final instantiatedness trees will be the same for all the matching mode declarations anyway. For example, if the arguments have instantiatedness

```
append(ground, ground, ground)
```

and the two matching mode declarations are

```
append(ground->ground, ground->ground, free->ground)
```

and

```
append(free->ground, free->ground, ground->ground)
```

then the final instantiatedness for the arguments is definitely going to be

```
append(ground, ground, ground)
```

If neither of the above two cases holds, then the situation is more difficult. One approach would be to let the final instantiatedness trees be the greatest-lower-bound of the final instantiatedness trees of the matching mode declarations, although we would want to additionally ensure that the final instantiatedness trees are never less instantiated than the initial instantiatedness trees. In some very rare cases we might lose some information by taking the greatest-lower-bound which would result in a mode error later on in the predicate. However these cases are rare enough that we have not encountered any examples where this problem would occur, and in any case it would be easy for the programmer to fix the problem by simply adding a new mode declaration for the called predicate which specifies the correct final instantiatedness.

In return for these additional minor restrictions, we can guarantee that the mode-checking algorithm's worst-case time and space usage are *linear* in the number of modes per predicate times the size of the program times the size of the instantiatedness trees required (no proof is offered, but we believe this to be clear from the description of the algorithm). In theory the number of modes for a predicate could be exponential in the number of arguments, but in practice tends to be quite small; we suspect that having an abnormally large number of modes for a predicate is probably an indication that the predicate actually performs several unrelated functions which should be separated into distinct predicates. We also believe that the instantiatedness trees required will typically be small, and thus that the performance of the algorithm will be effectively linear in the size of the program.

So far we have left three things unspecified: the notion of **compatible** instantiatedness trees, the **greatest-lower-bound** of two instantiatedness trees, and the details of **abstract unification** of instantiatedness trees.

The first is quite simple: an instantiatedness tree X is compatible with an instantiatedness tree Y iff every term approximated by X is also approximated by Y. To compute this, obviously we cannot test every possible term. Instead we compare the nodes of the trees, using the partial order given in Figure 4.1. The arrows in that diagram go from least information to most information. If every node of X specifies at least as much information as the corresponding node in Y, then X is compatible with Y; the phrase "at least as much information" that we are using is defined as follows: A node A specifies at least as much information as a node B iff there is a path from B to A (following the arrows) in the partial order diagram.

Observant readers will note that the diagram does not include "unborn" anywhere; this is because it is not actually necessary to explore the unborn nodes to determine compatibility. Strictly speaking, we should have spoken of "every node of X whose instantiatedness is not unborn" rather than just "every node of X" in the previous paragraph. It should be understood that a practical implementation would probably not actually store these unborn nodes of the instantiatedness trees.

This notion of compatibility, which was computed using a partial order on instantiatedness tags, in turn defines a partial order on instantiatedness trees. It is with respect to this partial order that we take the greatest-lower-bound.

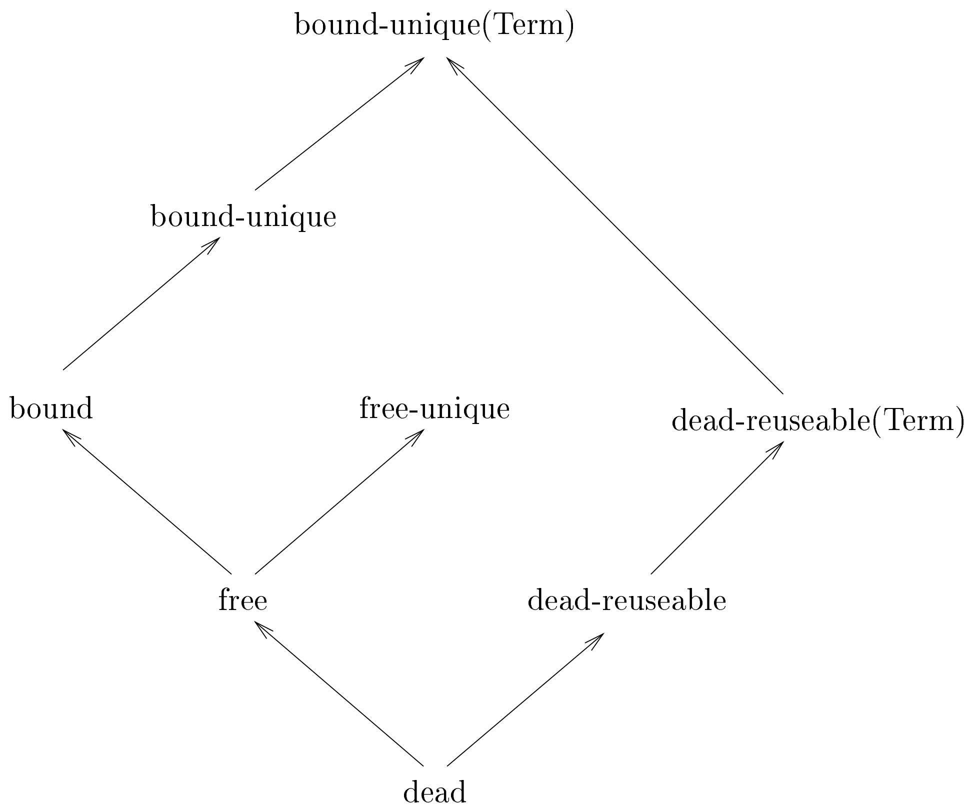

Figure 4.1: Information-Content Partial Order for Instantiatedness

The **greatest-lower-bound** operation combines the information in two instantiatedness trees, in such a way that the resulting instantiatedness tree is compatible with both of the original trees. We compute it by taking traversing both of the original original trees simultaneously, and constructing the nodes of the new tree as we go; the instantiatedness tag of a node of the new tree is the greatest-lower-bound of the instantiatedness tags of the original trees, where the greatest-lower-bound of two instantiatedness tags is defined in terms of the partial order shown in Figure 4.1. Again there is no need to continue traversing the trees past "unborn" nodes.

Note that in fact the algorithm is slightly more complex than this, because we want to preserve any aliases that were the same in both of the original trees; aliases that are present in only one of the two trees must of course be discarded.

The **abstract unification operation** for the basic mode system traverses the two trees in a similar manner to the greatest-lower-bound operation, but the instantiatedness tags for the nodes of the new tree are obtained using the least-upper-bound of the partial order defined by Figure 3.1. This partial order corresponds to "more instantiated" as opposed to "more information".

However for the extended mode system, we must also handle uniqueness of variables and death of variables.

4.4 Compile-time Garbage Collection

Once we have have the mode checking algorithm, implementing compile-time garbage collection and structure reuse is relatively simple.

For compile-time garbage collection, we extend the mode system to record when one of the nodes of a variable becomes dead-reusable, as discussed in Chapter 3. It is then simple to modify the mode checking algorithm so that it inserts a call to the deallocator (to return the memory to the free list) immediately after the instantiatedness of the node becomes dead-reusable, at least in the case that the node is the root node.

This case commonly occurs after head-unification with an argument whose mode mapping is unique-input. For example, consider the following code.

```
?- mode sum(list(int) : bound_unique->dead, int:input, int:output).
sum([],S,S).
sum(X.Xs,S0,S) :-
        S1 is S0 + X,
        sum(Xs,S1,S).
```

The mode checking algorithm moves the head unification into the body of the clause, and then abstractly interprets the body, determining the instantiatedness trees for the variables as it goes. Here we have annotated the second clause to show how the algorithm progresses.

```
sum(A,S0,S) :-
    % A is ground-unique, S0 is bound, S is free
        A = X.Xs,
    % X is bound-unique, Xs is ground-unique
    % The root node for A is now dead-reusable,
```

```
    % and the other nodes for A are dead; we can
    % safely deallocate the root node for A
        deallocate(A),
        S1 is S0 + X,
    % S1 is bound
        sum(Xs,S1,S).
    % S is bound
```

Probably the simplest way to deal with the more complex cases where the node that becomes dead-reusable is not the root node is to split complex unifications into the smallest possible units, by introducing some fresh variables; thus a unification such as

```
        X = f(g(Y))
```

would be replaced by

```
        X = f(A), A = g(Y)
```

This ensures that the `g(Y)` node is the root node of A, so that it can easily be reused when it becomes dead-reusable.

Our investigations of compile-time garbage collection have been purely theoretical; unfortunately our underlying implementation NU-Prolog does not have even run-time garbage collection, so we have not had the opportunity to try our ideas out in practice.

4.5 Difficulties with Lost Indexing

One potential problem with code which results from applying this optimization is that since the head unification has been moved into the body of the clause, we may lose clause indexing. That is, the compiler may no longer be able to detect that when the first argument to sum is bound, only one of the clauses can succeed, so the code is deterministic. If the compiler does fail to detect this determinism, it will generate much less efficient code.

There are two ways around the problem. The first is to make the compiler smarter: it should examine unifications with head variables that occur in the body of the clause, not just the implicit head unifications, when it is analyzing the code to determine clause indexing. (Of course, the compiler may do this already, in which case the problem won't occur in the first place).

The second method is to generate an auxiliary predicate which passes an additional reference to the arguments in question. Using this method for the previous example would result in the following code:

```
sum(A,B,C) :- sum2(A,B,C,A).

sum2([],S,S,_).
sum2(X.Xs,S0,S,A) :-
        deallocate(A),
        S1 is S0 + X,
        sum(Xs,S1,S). % could unfold this to sum2(Xs,S1,S,Xs)
```

The disadvantage of this method is both increased complexity of the compile-time garbage collection optimizer, and additional overhead when executing the code. We would only recommend this method when the first method was impractical, for example if the compile-time garbage collection optimizer was implemented as a stand-alone preprocessor.

4.6 Structure Reuse

Structure reuse optimization is similar to compile-time garbage collection, except that instead of deallocating the unused memory (returning it to the memory management system), we reuse it.

Whenever a free-unique variable is unified with a constant functor, then we know that this is an output unification. The mode checking algorithm will already have to cover this case as part of the **abstract-unify** operation; we can modify the algorithm, so that whenever it encounters such a unification, it checks to see whether the memory required for the output unification can be obtained by reusing already allocated memory. It does does this by searching through the current instantiatedness trees for the variables in the clause, searching for a dead-reusable(Term) node which can be reused for this output unification. In theory any node that occupies the right amount of memory can be reused, but currently our prototype will only reuse nodes that have the same functor as the term being output, because the primitive that we are using for destructive assignment will only allow us to update the arguments of a compound term, not the functor.

If we do find a node that can be reused, then we determine which of the arguments of the functor need to be replaced, and insert code to destructively assign those arguments; we then replace the output unification with an input unification that unifies the free-unique variable with the term we have just reused. Input unifications are much cheaper operations than output unifications, and they avoid allocating any new heap space.

Most of the output unifications that do occur in Prolog programs actually occur as part of the head unification. To increase the opportunity for structure reuse optimization, we delay output unifications by moving them to the end of the clause. Delaying output unifications in this way is an important optimization in its own right: if the clause fails, then we don't have to execute the output unifications (of course, this is only an improvement for deterministic code, but as stated in the introduction, this report is not concerned with determinism analysis). On the other hand, moving output unifications to the very end of the clause means that we lose last-call optimization, which is something that we definitely don't want to do. As a compromise, if the clause does end in a predicate call, then we mode output unifications to just before that call instead of the the very end of the clause.

For example, consider a predicate which squares all the elements in a list:

```
?- mode square_list(list(int):uniq_input, list(int):uniq_output).
square_list([],[]).
square_list(X.Xs,Y.Ys) :-
        Y is X * X,
        square_list(Xs, Ys).
```

After we have performed the mode analysis inherent in the mode-checking algorithm, and moved the output unification, we obtain something like this.

```
square_list(A,B) :- A = [], B = A.
square_list(A,B) :-
                % A ground-unique, B free-unique
        A = X.Xs,        % input unification
                % X ground-unique, Xs ground-unique,
                % A dead-reusable('.'(X,Xs))
        Y is X * X,
                % Y ground-unique, X read-reusable
        B = Y.Ys,        % output unification
                % Ys free-unique,
                % B bound-unique('.'(ground-unique,same_as(Ys)))
        square_list(Xs, Ys).
                % Xs dead, Ys ground-unique
```

We can now reuse the storage occupied by A for the output unification:

```
square_list(A,B) :- A = [], B = A.
square_list(A,B) :-
        A = X.Xs,
        Y is X * X,
        setarg(1,A,Y),
        setarg(2,A,Ys),
        B = A,
        square_list(Xs, Ys).
```

Here `setarg/3` is the built-in predicate which actually does the destructive assignment; the call `setarg(1,A,Y)` has the effect of replacing the first argument of the structure A with Y.

Just as with compile-time garbage collection, the situation is more complicated if the node being reused is not the root node, so we decompose complex unifications into their simplest possible parts by introducing fresh variables, to ensure that dead-reusable(Term) nodes are always the root node for some variable.

In [FW91], Foster and Winsborough note that structures that contain variables cannot be safely reused, even if there are no more live references to the structure, because there might still be live references to the variables. Constants or references to nested structures do not pose this problem because they are always copied into registers when decomposing structures. Variables on the other hand cannot be copied in this way, and instead a reference to the variable is placed in the register. They solution that Foster and Winsborough adopt to this problem is to test at runtime whether a structure contains variables, and only reuse it if it doesn't.

However, with the information provided by our mode system, we are able to be more selective. First, we only have to be concerned with the structure components that we do actually replace; it does not matter if the arguments of the functor which remain unchanged have other references. Second, by examining the instantiatedness tags for the components which do have to be replaced, we can determine at compile time whether any of these may still have live references. We only reuse the structure when we know that the the replacement will be safe, i.e. when the instantiatedness of the node being replaced is "bound-unique" or "dead-reusable".

When combining structure reuse with compile-time garbage collection, we must of course be careful to ensure that nodes that can be reused by the former optimization are not deallocated by the latter!

4.7 Destructive Assignment

Structure reuse certainly helps, but there is still room for improvement. We are still passing as pair of arguments for each structure that must be modified. What we really want to do is to generate code that passes a single argument and destructively updates that argument, in the same way that we would write code in a procedural language. As it turns out, this is really not so difficult after all.

If a predicate contains one argument whose initial instantiatedness is free-unique, and another argument whose final instantiatedness is dead, and the predicate always unifies these two variables, then it is possible to use just a single argument position for the two variables. All that is required is to replace all occurrences of the free-unique argument with the other argument, both in the definition of the predicate and in any calls to the predicate. This will make some unifications and assignments unnecessary, so we can then eliminate the unnecessary code.

The `square_list` predicate discussed in the previous section is an example of this. After destructive assignment optimization, it looks like this:

```
square_list(A) :- A = [].
square_list(A) :-
        A = X.Xs,
        Y is X * X,
        setarg(1,A,Y),
                % setarg(2,A,Xs)  (optimized out)
                % Xs = Xs         (optimized out)
        square_list(Xs).
```

An important example is destructive append. Unfortunately the standard code for append is not immediately amenable to destructive assignment optimization.

```
        % The original source code
?- mode append(list:uniq_input, list:uniq_input, list:uniq_output).
append([],Xs,Xs).
append(X.Xs,Ys,X.Zs) :- append(Xs,Ys,Zs).

        % After structure re-use optimization
append(A,B,C) :- A = [], C = B.
append(A,Ys,C) :-
        A = X.Xs,
```

```
        setarg(2,A,Zs),
        C = A,
        append(Xs,Ys,Zs).
```

The problem is that the base case clause unifies the second and third arguments, whereas the recursive case (after structure reuse optimization) unifies the first and third arguments. If we could change the first clause to be

```
append(A,B,C) :-
        A = [],
        A := B,         % destructive assignment
        C = A.
```

then things would work out fine, but the difficulty is that the assignment `A := B` does not work, because `A` is passed by value, not by reference; the original value of A is not changed, so it is not possible to combine the first and third arguments into a single argument.

However it is possible to write append in such a way that we can apply destructive assignment optimization. We can't optimize the case when the first argument is an empty list, but we can allow the other case to be optimized by adding an auxiliary predicate. This auxiliary predicate is obtained by unfolding the recursive call in the second clause.

```
append([],X,X).
append(A,B,C) :-
        A = _._,
        append_2(A, B, C).

append_2(X.[], Ys, X.Ys).
append_2(X1.X2.Xs, Ys, X1.Zs) :-
        append_2(X2.Xs, Ys, Zs).
```

After structure reuse optimization, the auxiliary predicate looks like this:

```
append_2(A, Ys, C) :-
        A = X.[],
        setarg(2,A,Ys),
        C = A.
append_2(A, Ys, C) :-
        A = X1.B,
        B = X2.Xs,
        setarg(2,A,Zs),
        C = A,
        D = X2.Xs,
        append_2(D, Ys, Zs).
```

Now both clauses unify the first and third arguments, and these arguments have modes unique-input and unique-output respectively, so we can apply destructive assignment optimization. We would also want to apply common sub-expression elimination to avoid the output unification for D.

Note that common sub-expression elimination can quite easily be done at the same time as structure reuse optimization. The two optimizations are actually very similar: the former replaces an output unification with input unification, while the latter replaces an output unification with one or more destructive updates followed by an input unification.

The final code for append after applying these optimization follows.

```
append([], X, X).
append(A, B, A) :-
        A = _._,
        append_2(A, B).

append_2(A, Ys) :-
        A = _.[],
        setarg(2,A,Ys).
                % A = A (optimized out)
append_2(A, Ys) :-
        A = _.B,
        B = _._,
                % setarg(2,A,Zs) (optimized out)
                % A = A (optimized out)
        append_2(B, Ys).
```

Note that this does require a fairly smart compiler to avoid losing indexing.

The code for append now corresponds almost directly with the following C code; the main difference is that our code needs an auxiliary predicate for the loop whereas the C code just has a `while` statement inside the `if` statement. (A smart Prolog compiler would recognize that the auxiliary predicate was only called from one other point in the program, and would therefore generate the code for that predicate inline, instead of generating a subroutine call.)

```
list append(list a, list b) {
    if (a == NULL) {
        return b;
    } else {
list tmp = a;
        while (tmp->tail != NULL) {
            tmp = tmp->tail;
        }
        tmp->tail = b;
        return a;
    }
}
```

The main drawback of our approach is that the programmer has to carefully unfold the code for append or similar predicates so that the compiler can apply destructive assignment optimization. It would be desirable to have a declaration or compiler directive which instructed the compiler to generate the auxiliary predicate and unfold the code automatically.

4.8 Some Results

We are in the process of implementing a prototype preprocessor that does structure re-use optimization for sequential NU-Prolog. The preprocessor performs a source-level transformation that introduces calls to a NU-Prolog built-in primitive which will perform the actual destructive updates; it should be simple to convert the preprocessor to work for any other Prolog which provides appropriate primitives. Unfortunately the prototype is not complete at this stage, so we cannot provide statistics for the time taken by the mode-checking algorithm; however we have performed the transformation manually for some common algorithms, so we can provide some statistics for the execution time and memory usage of the transformed programs.

The programs we present here are two versions of quicksort (the simple version using append plus the difference-list version) and naive reverse (chosen simply as a benchmark for the use of append). We performed structure reuse (s.r.) optimization following the algorithm described earlier, and we also measured the effect of performing destructive assignment (d.a.) optimization for `append/3`. Quicksort itself is not amenable to destructive assignment optimization, since most of the work is done in the partition procedure. This procedure partitions one unique-input argument into two unique-output arguments, rather than having a matching pair of arguments, one unique-input and the other unique-output.

Algorithm	Method	CPU Time (ms)	Heap Space (bytes)
Naive quicksort		209	180304
	s.r.	410	0
	d.a.	360	0
Difference-list quicksort		170	111296
	s.r.	330	0
Naive reverse		2660	4004000
	s.r.	6220	0
	d.a.	40	0

Table 4.1: Performance Measurements for some Simple Programs

These statistics are for sorting/reversing a list of 1000 random integers. They use a version of the built-in destructive update predicate that omits the run-time determism tests. Due to space limitations, we cannot include the code used to obtain these statistics in an appendix, but the code is available from the author on request.

The results show that for these cases, structure reuse optimization actually slows the code down by a factor of about two, but saves dramatically on heap space usage: quicksort and reverse can reuse *all* of the memory occupied by their input arguments. The results also show that destructive assignment optimization for append can make a huge difference in speed when appending many long lists, as occurs in the naive reverse benchmark.

One reason for the slow-down for structure reuse is that we are not working at a low-enough level; the NU-Prolog built-in predicate for destructive assignment is not implemented as efficiently as it could be. If we were working at the compiler level, then we could expect an improvement in these times. Also, the NU-Prolog compiler compiles to byte-code which is then interpreted; it would be more useful to make our measurements with a compiler that compiled down to machine code — it is certainly the case that destructive assignment can be implemented efficiently at the machine level.

However, the difference in speed is exaggerated by these measurements. Unlike genuine "industrial" Prolog systems, NU-Prolog does not do run-time garbage collection, so the time for the original versions does not include the time required to garbage collect the heap space consumed. Unfortunately this makes the time comparisons basically not very useful or meaningful.

We would like to be able to compare the speed of the linked-list version of quicksort with an array version, but of course Prolog doesn't have arrays. (It might be interesting to simulate arrays with functors using `arg/1` for array indexing, but we have not done this as yet). We would also like to compare speeds with programs written in C, but as NU-Prolog uses a byte-code interpreter it would be an unfair comparison; for a fair comparison, we would need a Prolog compiler that compiles down to machine code.

Chapter 5

Logically Sound Declarative I/O using Unique Modes

One of the major problems with Prolog is the widespread use of side-effects and non-logical code. The two major reasons for using such code in Prolog are for efficiency, and for input/output. Destructive assignment optimization negates the need for using side-effects for efficiency, and we are about to show how unique modes can provide a clean, logical alternative to the traditional technique of using side-effects for input/output.

The basic idea is quite simple. We model the change of state that is implicit in the traditional approach to input/output by providing an explicit "state of the world" variable which will be passed around by the different predicates in the program. Since there is only one real world, we must not allow references to previous states to linger on in the program. We model this by making the mode of the "state of the world" variable have a unique mode wherever it occurs in the program. (Because changes to this state cannot be undone on backtracking — it is not possible to "trail" input-output! — we would *have* to do the determinism analysis mentioned in the introduction). The program entry-predicate would have the mode

```
?- mode main(list(string):uniq_input,
             io_state:uniq_input, io_state:uniq_output).
```

The `uniq_input` and `uniq_output` modes are predefined modes as explained in Chapter 4. The first argument, a list of strings, is the command line passed to the program. The last two arguments represent the "state of the world" before and after execution of the program. These are of type `io_state`, which would be a built-in data type[1].

Input-output predicates could have modes like the following:[2]

```
?- type char_or_eof ---> c(char) | eof.
?- type getchar(char_or_eof:uniq_output,
                io_state:uniq_input, io_state:uniq_output).
```

[1] Actually, it would be preferable to make io-state a data type exported by the standard input/output module, but this would of course require the language to include some sort of module system.

[2] Note that although these predicate names are based on the I/O functions in C, we have departed from C's use of `int` to represent "either a character or end-of-file". Logically, this should be represented as a disjoint union as we have done above. A smart compiler would notice that this type could be represented in a single machine word and would thus generate the same code anyway.

```
?- type putchar(char, io_state, io_state).
?- mode putchar(input, uniq_input, uniq_output).
```

Thus a simple "cat" program which copies its input to its output might look like this:

```
main(_, Input, Output) :-
        cat(Input, Output).

?- mode cat(io_state:uniq_input, io_state:uniq_output).
cat(State0, State) :-
        getchar(Char_or_eof, State0, State1),
        cat_2(Char_or_eof, State1, State).

?- mode cat_2(char_or_eof:input,
              io_state:uniq_input, io_state:uniq_output).
cat_2(eof, State, State).
cat_2(c(Char), State0, State) :-
        putchar(Char, State0, State1),
        cat(State1, State).
```

This is logically pure, but does require some extra work shuffling all the state arguments around. It is not difficult to get the compiler to optimize these `io_state` variables; this could be done either as a special case, or more generally we could ensure that the compiler always optimizes out variables of unit types (types with only one value) — this optimization is commonly done for many functional languages — and define `io_state` to be a unit type. On the other hand, the argument shuffling is also a burden on the programmer, and this would at first seem more difficult to cleanly avoid. However, we wave our wand of magical syntactic tricks, and let Prolog's Definite Clause Grammar notation do all the extra work for us!

```
main(_) --> cat.

?- iomode cat.
cat --> getchar(Char_or_eof), cat_2(Char_or_eof).

?- iomode cat_2(char_or_eof:input).
cat_2(eof) --> [].
cat_2(c(Char)) --> putchar(Char), cat.
```

The `iomode` declarations here are abbreviations for the longer `?- mode` declarations above. This is very similar to the continuation-passing style of input/output for functional languages, except that the ability to use output modes for examples such as the argument to getchar, which is not present in functional languages, gives us an even more pleasant syntax. The result is code that superficially looks very much like conventional languages, but which is built on a genuinely pure logical foundation.

Chapter 6

Conclusions

We believe that the use of strong type and strong mode systems, combined with type and mode inference could herald a new era in the use of logic programming languages; compared with current languages they hold out the promise of greatly improved efficiency, greater reliability, more expressiveness, and an elimination of the need to use logically impure code and side-effects. We have shown how a simple, practical method for structure-reuse and destructive assignment optimization can be built on top of a strong mode system. We have discussed a new paradigm for logical I/O using unique modes. Finally, we are in the process of implementing a simple prototype for these techniques as a preprocessor for sequential NU-Prolog.

Bibliography

[CDB$^+$90] Jim Crammond, Andrew Davison, Alastair Burt, Matthew Huntbach, and Mellisa Lam. *The Parallel Parlog User Manual.* Parlog Group, Department of Computing, Imperial College, London, June 1990.

[CG85] Keith Clark and Steve Gregory. Notes on the implementation of Parlog. *The Journal of Logic Programming*, 2(1):17–42, April 1985,.

[FW91] Ian Foster and Will Winsborough. Copy avoidance through compile-time analysis and local reuse. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 455–469, San Diego, USA, 1991. The MIT Press.

[HSC$^+$90] Bruce K. Holmer, Barton Sano, Michael Carlton, Peter Van Roy, Ralph Haygood, William R. Bush, Alvin M. Despain, Joan M. Pendleton, and Tep Dobry. Fast Prolog with an extended general purpose architecture. *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*, pages 282–291, May 1990.

[MO84] A. Mycroft and Richard O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[MWB90] Anne Mulkers, William Winsborough, and Maurice Bruynooghe. Analysis of shared data structures for compile-time garbage collection in logic programming languages. *Proceedings of the Seventh International Conference on Logic Programming*, pages 747–762, June 1990.

[Myc81] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, Scotland, 1981.

[Nai86] Lee Naish. *Negation and control in Prolog.* Number 238 in Lecture Notes in Computer Science. Springer-Verlag, New York, 1986.

[Pfe92] Frank Pfenning, editor. *Types in Logic Programming.* The MIT Press, 1992.

[RD90] P. Van Roy and A.M. Despain. The benefits of global flow analysis for an optimizing Prolog compiler. *Proceedings of the Second North American Conference on Logic Programming*, October 1990.

[Som89] Zoltan Somogyi. A parallel logic programming system based on strong and precise modes. Technical Report 89/4, University of Melbourne, January 1989.

[Tay90] Andrew Taylor. LIPS on a MIPS: results from a Prolog compiler for a risc. *Proceedings of the Seventh International Conference on Logic Programming*, June 1990.

[TZ86] James Thom and Justin Zobel. NU-Prolog reference manual, version 1.0. Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.

[Vod88] Paul J. Voda. Types of Trilogy. *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 580–589, 1988.

[Wad90] Philip Wadler. Linear types can change the world! *Proceedings of the IFIP TC2 Conference on Programming Concepts and Methods*, pages 547–566, April 1990.

[Wad91] Philip Wadler. Is there a use for linear logic? *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 26(9):255–273, 1991.

[War77] D. H. D. Warren. Implementing Prolog. DAI Research Report 39, University of Edinburgh, Scotland, 1977.