ABSTRACT INTERPRETATION FOR
THE COMPILE-TIME OPTIMIZATION OF LOGIC PROGRAMS

by

Thomas Walter Getzinger

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Computer Science)

December 1993

Dedication

To Karen for the love and support, and to Sarah for waiting.

Acknowledgments

It is impossible to complete several years of work without the help and guidance of many. Al Despain provided invaluable direction and advice, as well as all the resources I needed to complete my research. Dean Jacobs introduced me to Al and gave me insights into the theoretical aspects of logic programming and abstract interpretation. I would also like to thank Michel Dubois for his helpful comments.

Many colleagues helped in various ways. Peter Van Roy developed the original Aquarius compiler, which I used in my work. He also answered innumerable questions about this compiler and other topics. I also made use of code and ideas developed by Anno Langen and Gerda Janssens. There are a number of others who helped me with advice, acting as a sounding board for ideas, or merely helped me maintain my sanity, most notably Bart Sano, Jim Su, and the other Trojan Project members at USC. Steve Branch helped make my mathematics understandable. Don Hopp provided support and time away from work to spend on my research.

This research was partially sponsored by a Hughes Aircraft Company Doctoral Fellowship, NSF grant CCR-9012288, and DARPA contract J-FBI-91-194.

Table of Contents

List of Tables

List of Figures

Abstract

Abstract interpretation is a powerful framework for describing data flow analyses of programs. These analyses can provide useful information for determining the applicability of compile-time optimizations. Many different types of analyses and optimizations have been suggested for Prolog. So far, this exploration has been rather ad-hoc. Our goal is to locate the “right” set of analyses for Prolog compilation. Since everyone’s interpretation of what constitutes “right” will vary, we begin by developing a taxonomy of global analyses for Prolog. We then describe the features that must be present in an abstract interpretation framework meant to be used in conjunction with a Prolog compiler. Using this taxonomy and the framework integrated into a compiler, we perform a systematic search, trading off analysis and compilation time for execution time and compiled code size.

This taxonomy and the information we derive during this search should help others to focus and simplify their searches. It should be useful for applications of abstract interpretation in addition to just compilation, for example, program proof of correctness and partial evaluation. It should also provide many insights into abstract interpretation of other languages, such as concurrent logic languages and functional languages.

We demonstrate a wide range of performances, varying by a factor of over 4.6 in code size and 4.4 in execution time. At the same time, the global analysis time varies by almost an order of magnitude, but the compile time only by a factor of two. We demonstrate an absolute improvement in performance over previous Prolog compilers. By using a unified framework to perform global analysis and to maintain descriptions during code generation, we show a 42% reduction in compilation time. At the same time, we show a decrease in code size by 36% and a reduction in execution time by 28%.

Chapter 1: Introduction

1.1 Motivation

Computer languages can be classified according to how abstract they are, that is, how far away from the underlying machine they are. Machine languages and assembly languages are the most concrete. C is a fairly concrete high-level language. Prolog and LISP are examples of abstract languages. Languages like Pascal and Ada fall somewhere in the middle.

Abstract languages have certain advantages. In general, they allow the user to concentrate more on solving a problem and less on the details of controlling the machine. For Prolog, some specific advantages are dynamically typed variables, unification (pattern matching), speculative execution, and automatic dynamic memory allocation and reclamation. Along with these advantages come some disadvantages, typically in terms of performance. For Prolog implementations, this is usually due to tagging of values (to specify dynamic types), dereferencing (caused by the generalized unification algorithm and unbound variable implementation), backtracking and non-deterministic execution (to provide for speculative execution), and run-time garbage collection (for automated dynamic memory reclamation).

A “super-compiler” should be able to overcome these difficulties and provide an efficient low-level implementation from a high-level abstract program. If we consider, for example, an experienced programmer as the “super-compiler”, given enough time he can translate a high-level specification into an efficient assembly language program. But, automating the translation process to this degree is rather ambitious.

To bound the problem, therefore, we will start with a program written in Prolog and consider the translation of this program into assembly language for an abstract machine designed for the execution of Prolog.

Prolog is a good choice for “super-compiling” because it is at such a high level and has a simple, underlying mathematics (first-order logic) which allows it to be “understood” and manipulated. Using a dataflow analysis approach known as abstract interpretation [17], it is possible to approximate run-time properties at compile-time. This information can then be used to optimize the generated code. Many researchers have proposed analyses based on this approach and compile-time optimizations based on the results, but most of this work is theoretical, with little practical demonstration of the ideas. It is not clear what benefits can be obtained in realistic benchmarks or what the costs are at compile-time.

1.2 The Thesis

It is with this in mind that I formulate my hypothesis, which is:

> *A significant improvement in logic programming performance can be achieved through:*
>
> 1) *A careful selection of data flow analyses, based on abstract interpretation, and*
> 2) *Advanced compile-time optimizations, based on the results of these analyses.*

In order to attack this problem, I first have a hurdle to overcome. Many researchers have suggested different uses for abstract interpretation, most of them related to compile-time optimization, and, in fact, two compilers have been constructed which use abstract interpretation. Is there room left for a valuable contribution in this area? What is missing from this work is a comprehensive organization and a thorough understanding of the costs and benefits of abstract interpretation.

Therefore, my first goal is to construct a taxonomy of existing and new data flow analyses for Prolog. Once this is developed, and populated with various analyses, the second step is to develop a tool which will enable me to explore the power of these analyses. The next step is to integrate this tool with a complete Prolog system, allowing compile-time optimizations to be included, based on analysis results. The final step is to use these tools to evaluate various analyses and compile-time optimizations, in order to examine the power of abstract interpretation for a wide range of benchmark programs. The goal is to select a collection of analyses that will provide a significant increase in run-time performance with a modest impact in compilation time.

1.3 Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides a review of issues related to Prolog, providing an overview of the language, describing various execution models, and describing Prolog compilation and compile-time optimization. Chapter 3 describes abstract interpretation, in general, and as applied to Prolog and Prolog compilation. Our analysis framework for Prolog compilation is presented in Chapter 4. A taxonomy of analyses for Prolog is provided in Chapter 5. Chapter 6 contains an analysis of the costs and benefits of various analyses from this taxonomy when applied to the problem of optimizing a set of benchmarks at compile time. Chapter 7 provides conclusions and directions for future work.

Appendix A defines the semantics on the Berkeley Abstract Machine, which is the Prolog execution model I used. Appendix B contains a formal definition of the abstract domain used in Van Roy's Prolog compiler [84]; this was used as a point of reference. A review of lattice theory can be found in Appendix C. Source code for benchmarks and the dataflow analysis framework can be found in Appendix D. Appendix E contains the detailed results of my experiments.

Chapter 2: Prolog

This chapter provides an overview of Prolog. It covers the language, execution, compilation, and optimization.

2.1 The Prolog Language

Prolog stands for Programming in Logic (or Programming and Logic [72]). It is an attempt to implement Colmerauer and Kowalski's idea that logic can be used as a programming language [52]. The motivation for Prolog is to separate the specification of *what* the program should do from *how* it should be done. Kowalski phrased this in the form of an equation: Algorithm = Logic + Control [46]. A Prolog program provides a logical specification of what should be done (the logic). The Prolog system then executes this specification (the control). Prolog implements a subset of first order logic known as Horn clause logic [52]. The following paragraphs provide an overview of the important syntax, semantics, common terminology, and some implementation details of Prolog. More depth is provided in [52] and [73]. Figure 1 summarizes the syntax of Prolog.

A *program* in Prolog, also called a *definite* program, consists of a finite set of *definite program clauses*, or simply clauses. Each clause is a logic sentence describing a single consequence (an if-statement). It contains a *head* followed by a *body* consisting of zero or more *goals*. The meaning of the clause is that the head is true if the body can be shown to be true. If the body consists of no terms, the head is always true. For example, the following clause states that a person X is an ancestor of a person Y if X is the ancestor of some person Z who, in turn, is the parent of Y.

ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y).

A *predicate* is identified by its name (predicate symbol) and its *arity* (the number of terms, or arguments, in the head); this is commonly shown as `name/arity` (e.g. `ancestor/2`). The *definition* of a predicate consists of the collection of all clauses with the same predicate symbol and arity in the head. Each predicate defines a logical relation. This relation is considered to be true if any clause in the predicate definition can be shown to be true. For example, the `ancestor/2` predicate can be defined as:

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y).

<definite program> ::= { <definite program clause> }

<definite program clause> ::= <head> [:- { <literal> <operator> } <literal>] .

<head> ::= <predicate symbol> [({ <term> , } <term>)]

<operator> ::= , | ; | ->

<literal> ::= [\+] <goal>

<goal> ::= <predicate symbol> [({ <term> , } <term>)]

<term> ::= <variable> | <constant> | <function symbol> ({ <term> , } <term>)

<predicate symbol> ::= a string of alphanumeric characters, starting with a lower case letter

<function symbol> ::= a string of alphanumeric characters, starting with a lower case letter

<variable> ::= a string of alphanumeric characters, starting with an upper case letter

<constant> ::= a string of alphanumeric characters, starting with a lower case letter
| a string of numeric characters

Figure 1: Prolog Syntax

Showing that a body is true depends on the *operators* in the body. A positive literal (a goal) is true if the predicate can be shown to be true for some set of values for the variables in the goal; this is a predicate call. A negative literal (a goal preceded by \+) is not true (i.e., it fails) if the goal can be shown to be true; otherwise, the negative literal is considered to be true. This is known as the *negation as failure* rule [52]. Two goals connected by a comma (,) are true if both goals are true (and). Two goals connected by an arrow (->) are true if the second goal is true or the first goal is not (implies). Two goals connected by a semi-colon (;) are true if either goal is true (or). These operators have been described in order of precedence.

A variable in Prolog is very similar to a variable in first order logic; it refers to some (possibly unknown) object. A variable starts out *uninstantiated* or unbound; that is, nothing is known about it. As more is known about the object to which a variable refers, the variable is further instantiated. For example, the variable `Fact1` might have the value `has_pet(Tom,Pet)`, which means that some person, referred to as `Tom`, has a pet, referred to as `Pet`. Later, it might be determined that Tom's pet is a golden retriever, named Boomer, thereby further instantiating the value of `Fact1` to `has_pet(Tom,dog(boomer,golden_retriever))`.

Instantiation occurs through *unification* of two terms. For example, unifying `dog(boomer,Breed)` with `dog(Name,golden_retriever)` sets `Name` to `boomer` and `Breed` to `golden_retriever`. When the value of a variable is further instantiated, it changes everywhere it is used. For example, if `Pet` is unified with `dog(boomer,golden_retriever)`, `Fact1` will then have the value `has_pet(Tom,dog(boomer,golden_retriever))`. Unification is basically a form of pattern matching. Therefore, given the following variable instantiations:

V1:x(A,y(B,t),s)

V2:x(B,y(u,C),D)

unification of `V1` and `V2` finds the most general (the least instantiated) pattern which matches these two values, and instantiates both variables to this value. Unification is recursive: two structures are unified by unifying their arguments. Therefore, unification of `V1` and `V2` would proceed as follows:

1: V1 = V2

2: x(A,y(B,t),s) = x(B,y(u,C),D)

3: A = B, y(B,t) = y(u,C), s = D

4: B = u, t = C

The resulting value for `V1` and `V2` would be `x(u,y(u,t),s)`. If two values can't be unified (e.g., they are instantiated to different functors or their arities don't match), the unification fails.

Variables are typically implemented as tagged pointers to their values. Uninstantiated variables are identified using special tag values or as pointers to themselves. Instantiated values are pointers to data structures describing the function symbol and arguments to which the variable is instantiated. The arguments are then recursively implemented as pointers. Tags are used to help in identifying values. One example, uninstantiated values, has been given. Others might include integers, floating point numbers, lists, or structures. When two uninstantiated variables are unified, they are made to refer to a single value by having one variable point to (reference) the other. This is referred to as variable *aliasing*. After this is done a number of time, variables can end up as chains of pointers to values. Therefore, to get at the actual value, this pointer chain must first be traversed. This is referred to as *dereferencing*. Figure 2 shows an example of this for the unification of `V1` and `V2` from above; the chain of dark arrows must be followed to obtain the value of the second argument of the structure to which V1 is bound.

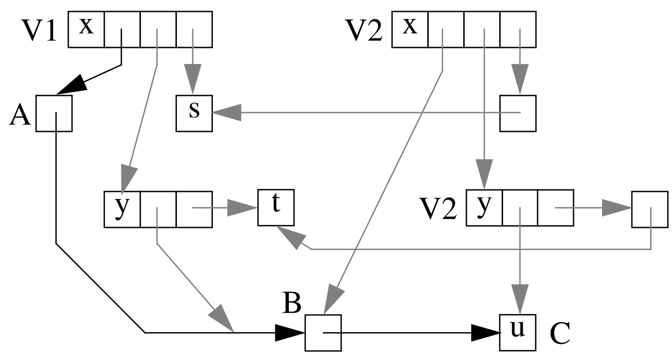


Figure 2: Pointer Chains from Unification

Given an initial body, or query, to prove, Prolog begins by trying to prove each goal in sequence. Proving a goal is done by calling the predicate named in the goal. To call a predicate, the terms in the call are unified with the terms in the head of the first predicate clause. This provides further instantiation of the arguments of the clause. If any terms can't be unified, the clause fails, and Prolog moves on to the next clause. Once head unification is successful, Prolog attempts to show that the body of the clause is true based on this information. If it fails, Prolog again moves on to the next clause, until all clauses have been tried. This is referred to as *backtracking*. Prolog's predicate call and backtracking strategy results in a depth-first traversal of the logical statements in the program, attempting to prove the initial query. In the following example, three binary relations, `p`, `q`, and `r`, are defined:

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

q(a,e). q(a,f). r(e,f). r(f,b).

The following query checks to see if the relation, `p`, holds for two specific values:

:- p(a,b).

To test the query, Prolog first unifies the query with the head of the clause for `p/2`. It then tries to show that the subgoals in `p/2` (`q` and `r`) are true. Figure 3 shows the and-or *resolution* tree [81] for this query, and the order (left-to-right, depth-first with backtracking) in which Prolog will traverse this tree.

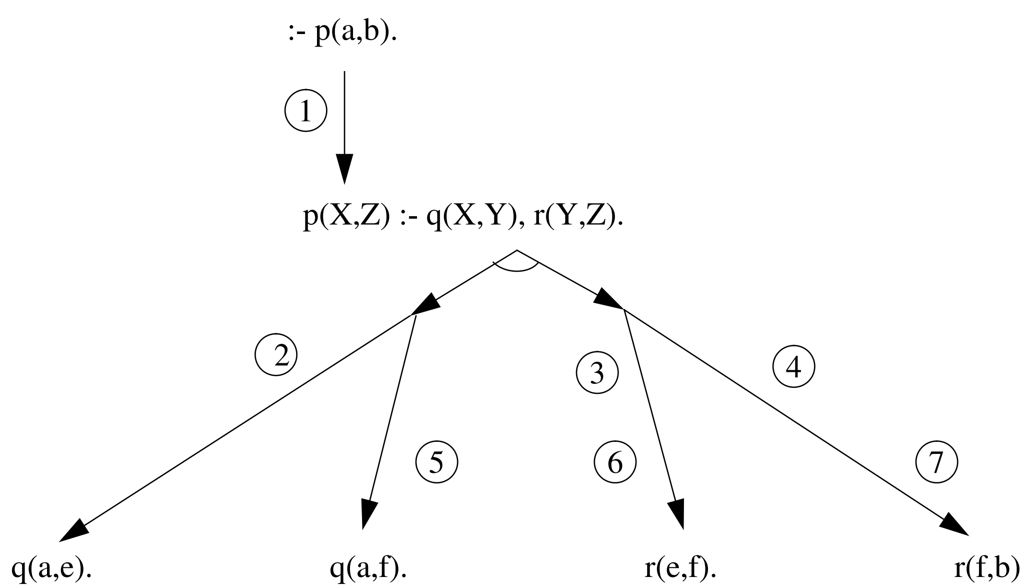


Figure 3: Prolog Control Flow

When backtracking, Prolog returns to the last point where head unification was successful (where it found a clause that it chose to pursue) and continues trying other clauses. This point is referred to as a *choicepoint* (in fact, the most recent choicepoint). In Figure 3, the unification at point 4 fails, causing backtracking to point 5. An important feature of Prolog is that when returning to a choicepoint, Prolog undoes all unification that occurred after the choicepoint. This returns the program to the state it was in before attempting some clause (since it has now decided that this clause was a bad choice). In order to do this, most Prolog implementations perform an operation known as *trailing*. Whenever a variable receives a value, the variable is trailed, that is, a note is made of the fact that the variable used to be uninstantiated. When backtracking, all variables trailed since the most recent choicepoint are set back to uninstantiated. This can create a large amount of garbage (no longer accessed structures) in memory, which will hopefully be reclaimed by the garbage collector for later use. In Figure 3, Y is trailed at point 2, so that when backtracking occurs at point 4, the value of Y will be returned to uninstantiated. This allows the unification at point 5 to succeed (otherwise, it would fail).

Most Prolog implementations provide a collection of built-in predicates and syntactic conventions for such things as arithmetic operations and list manipulation. Operator symbols (e.g., '+') are allowed for predicate and function symbols and can be used infix for an arity of two or pre- or postfix for an arity of one. The functor `./2` is used to describe a list, with an empty list consisting of the constant `nil`. A list containing the first 3 positive integers would appear as `.(1,.(2,.(3,nil)))`. In infix notation, this would be `1.2.3.nil`. For readability, another syntactic representation is provided for lists. `[A|B]` represents the list `A.B`. The previous integer list would be `[1,2,3]` and an empty list would be `[]`.

The Prolog program given in Figure 4 provides a simple, yet fairly complete example of these concepts. It consists of four clauses, which form two predicates: `nreverse/2` and `append/3`. There are also two queries, one for each example predicate. The predicate `append/3` takes two lists and returns a third list which contains all items from the first list followed by all items in the second list. This is defined by declaring that if the first input list is empty, the output list is the same as the second input list; otherwise, it is obtained by building a list containing the first item in the first input list, followed by appending the rest of the first list with the second list. The query `query2` shows an example of this. The predicate `nreverse/2` takes a list and returns a list containing the items from the first list, in reverse order. By inductive description, similar to that for `append/3`, if the input list is empty, the reversed list is empty; otherwise, it is obtained by reversing all but the first item in the input list and appending this to the first item (which is added to the end). The query `query1` shows an example of this.

```
nreverse([],[]). query1 :- nreverse([a,b,c],X).
nreverse([E|L],RL) :-result:
        nreverse(L,L1),X = [c,b,a]
        append(L1,[E],RL).

append([],L,L).query2 :- append([a,b],[c,d],L).
append([E|L1],L2,[E|L3]) :-result:
        append(L1,L2,L3).L = [a,b,c,d]
```

Figure 4: Sample Prolog Program

Because of Prolog's logical nature, however, predicates can be used in many different ways, depending on the *modes* (instantiation) of their arguments. For example, `append/3` succeeds, in general, when the arguments can be instantiated such that the third argument is the concatenation of the first two. If the first two arguments are instantiated as lists and the third argument is uninstantiated (variable), this concatenates the two input lists and returns the result in the third argument (as shown in `query2`). If only the second argument is variable and the third list starts with the same elements that are contained in the first list, it returns the remaining elements in the second argument. If the first two arguments are variable, then, through backtracking, it will successively return all possible ways of splitting the third list into two smaller lists. This has been referred to as the multi-directional nature of Prolog [37].

2.2 Prolog Execution Models

In order to obtain efficient execution of Prolog, a number of execution models have been proposed. These models provide a bridge between the source language, Prolog, and the machine which executes the Prolog programs. Quite often, these execution models are referred to as *abstract machines*, since they have many characteristics of real machines: memories, registers, and instruction sets.

2.2.1 Warren's Abstract Machine (WAM)

The first such abstract machine for Prolog was described by Warren and is referred to, appropriately enough, as the Warren Abstract Machine, or WAM [88]. Although many refinements have been suggested to this model, the WAM remains the baseline against which other models are compared.

The WAM defines six classes of instructions, five different memory regions, and numerous registers. Aït-Kaci describes this model by starting with a simple execution model and slowly adding features needed to implement Prolog, until he ends up with the full WAM [5]. He adds a seventh instruction class, the set instructions, which we ignore, since they are only an optimization.

The five memory regions in the WAM are the code area, the heap, the stack, the trail, and the push down list (PDL); these memory regions are illustrated in Figure 5. The code area contains the WAM instructions for the compiled program. The heap contains the Prolog terms that have been created during the execution of the program; this is similar to the heap in C, which is used for dynamic memory allocation. The stack contains *environments* and *choicepoints*. Environments are created within a predicate to save variables across calls to subordinate predicates. Choicepoints are created to allow sequencing through multiple clauses of a predicate (if this can't be done deterministically). The trail is used to record all variable bindings in order to be able to undo these bindings when backtracking. The PDL is used in the unification operation. Environments are found in other languages. Choicepoints, the trail and the PDL are fairly specific to Prolog, or at least, to logic programming languages.

The six classes of WAM instructions are get, put, unify, procedural, indexing, and cut instructions. They are summarized in Table 1. The get instructions are used to perform argument matching against the head of a clause. The put instructions load the argument registers prior to calling a subordinate predicate. The unify instructions are used along with the get and put instructions for unifying with existing structures or creating new structures. The procedural instructions are responsible for transfer of control and environment management associated with procedure calling. The indexing instructions are used to implement deterministic and non-deterministic clause selection.

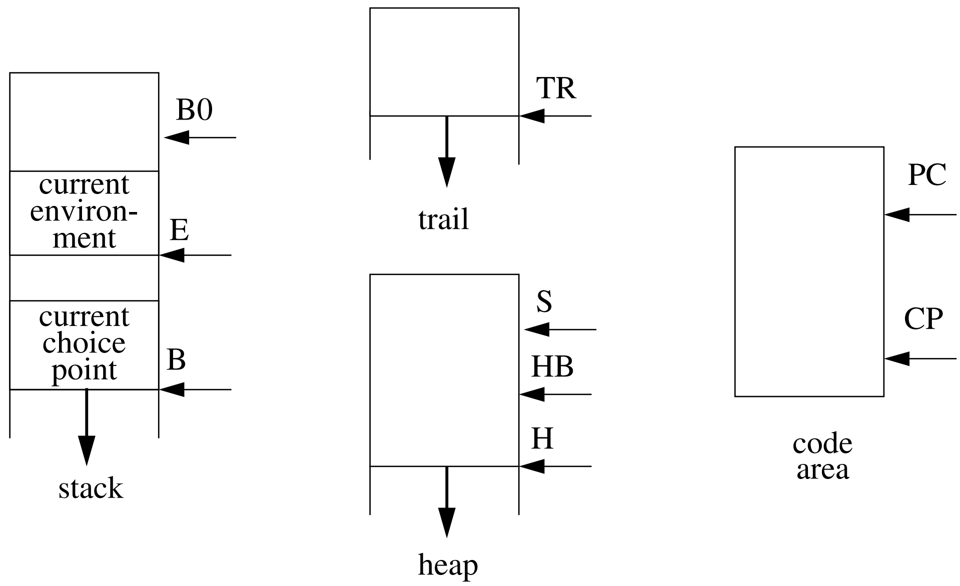


Figure 5: WAM Memory Regions and Registers

The cut instructions are used to implement the cut built-in, which is also important in the implementation of such constructs as if-then-else (X->Y;Z) and negation (\+X).

The registers can be classified, loosely, by the memory region into which they point, as illustrated in Figure 5. There are two registers that point into the code area: the program counter (PC) and the continuation, or return address, pointer (CP). Three registers point into the heap: the structure pointer (S), used during structure unification; the heap pointer (H), indicating the top of, or next available location in, the heap; and the heap backtrack pointer (HB), used to restore the H register during backtracking. Three registers point into the stack: the current environment pointer (E); the current choicepoint pointer (B); and the cut pointer (B0), used in implementing the cut built-in. The only register pointing into the trail is the trail pointer (TR). In addition to these, there are a collection of numbered argument registers (Ai) used for passing arguments to procedures and for temporary storage. These may contain tagged constants, tagged pointers into the heap, or tagged pointers into the stack.

Table 1: WAM Instructions

get instructions	Read mode unification:
get_variable Rn,Ai	Move the value in Ai into Rn
get_value Rn,Ai	Unify Ai with the value in Rn
get_constant C,Ai	Unify Ai with the atomic value C
get_structure F,Ai	Unify Ai with the structure with functor F
get_list Ai	Unify Ai with a list
put instructions	Write mode unification:
put_variable R,Ai	Create new variable in R and Ai
put_value R,Ai	Copy value from R to Ai
put_unsafe_value Yn,Ai	Copy value from R to Ai, and make it global
put_constant C,Ai	Put constant C into Ai
put_structure F,Ai	Create structure with functor F in Ai
put_list Ai	Create list and put pointer in Ai
unify instructions	Compound argument unification:
unify_void N	Unify next N arguments with void variables
unify_variable Rn	Unify next argument with variable in Rn
unify_value Rn	Unify next argument with value in Rn
unify_local_value Rn	Unify next argument with local value in Rn
unify_constant C	Unify next argument with atomic value C
procedural instructions	Procedural control:
proceed	Return from predicate
execute P	Jump to predicate P
call P,N	Call predicate P, keeping Y1-YN in environment
allocate	Create a new environment
deallocate	Remove topmost environment
indexing instructions	Clause indexing:
try_me_else L try L	Create a choicepoint
retry_me_else L retry L	Update the retry address
trust_me_else fail trust L	Remove topmost choicepoint
switch_on_term Lv,Lc,Ll,Ls	Jump based on tag of value in X1
switch_on_constant N,Table	Jump based on atomic value in X1
switch_on_structure N,Table	Jump based on functor of structure in X1
cut instructions	Support for cut built-in (!)
neck_cut	Remove choicepoints later than B0
get_level Yn	Save B0 in register Yn
cut Yn	Remove choicepoints later than Yn

2.2.2 Berkeley Abstract Machine (BAM)

The Berkeley Abstract Machine, or BAM, incorporated many of the suggested improvements to the WAM [84]. The main motivation in the development of the BAM was to provide a model which allowed dataflow analysis-driven optimizations and efficient translation to target machine code. The major differences between the BAM and the WAM which allowed this are summarized below:

- The WAM instructions are complex, restricting the opportunities for optimization. In the BAM, these operations have been broken into smaller pieces, allowing parts of them to be eliminated.
- The types of determinism supported by the WAM were limited by the indexing instructions. The BAM added other instructions, such as arithmetic comparison, to allow more optimization in the presence of deterministic predicates.
- Unification is used extensively in Prolog. There are many special cases of the unification algorithm which can be implemented efficiently on a target processor. The WAM unification instructions allow a compiler to express some, but not all, of these cases. For example, on the WAM, arguments are always dereferenced, even when this is not required. The BAM instructions provide the compiler with more choices, allowing optimized code to be generated for numerous special cases of the unification algorithm.

The memory regions, registers, and classes of instructions remain similar to those of the WAM. Details are provided in Appendix A. Following is a description of some of the differences.

The memory model of the BAM is very similar to that of the WAM, as can be seen in Figure 50. The BAM has six memory regions: the code area, the heap, the trail, the choicepoint stack, the environment stack, and the SDA queue. The choicepoint and environment stacks replace the combined stack from the WAM. Actually, all known implementations of the BAM recombine these into a single physical stack, anyway. The SDA queue is a memory region added to the BAM to support a seldom-used featured referred to as stepped destructive assignment (SDA). Since this is outside the realm of standard Prolog, the BAM can execute without it. General unification is implemented by calling a procedure written in BAM code. Any information that must be saved for the recursive nature of this operation is saved in environments, thereby removing the need for the PDL.

The registers in the BAM are also very similar to those in the WAM, aside from name changes (e.g., r(pc) instead of PC).[1] Since general unification is written as a BAM procedure, the S register is not needed. The B0 register is not needed; pointers to cut points on the trail stack are kept in the argument registers or are saved in an environment. Two registers have been added for stepped destructive assignment: r(sda_queue) and r(sda_queue_next). Also, there is an additional code area pointer, r(tmp_cp). This register is used for saving the return address when calling leaf procedures. This allows the creation of an environment to be avoided when its sole purpose is merely to save the return address to the current procedure when calling low-level procedures (e.g., general unification) or simple built-in predicates (e.g., ==/2).

The major different between the BAM and the WAM is in the instructions. The BAM instructions are grouped into four categories: procedural control flow, conditional control flow, unification, and arithmetic.

The procedural control flow instructions are responsible for implementing procedure calling and environment management.[2] These correspond fairly closely to the WAM's procedural instructions. In addition to the standard procedure calling instructions (procedure, call, and return), the BAM adds instructions for leaf procedures (simple_procedure, simple_call, and simple_return), which don't require saving of the return address register. There are two other procedural instructions on the BAM which don't exist on the WAM: entry and jump_ind. The entry instruction indicates points in execution where garbage collection can occur. Unlike the WAM, which makes atomic changes to the machine state, the BAM makes changes in small increments. To prevent the garbage collector from examining partial data structures in memory, it is only invoked during the execution of an entry instruction. This could have been done as part

1. Table 35 in Appendix A provides a description of the BAM registers.
2. See Table 38 in Appendix A.

of a procedure (pseudo-)instruction, but last call optimization can turn a tail-recursive predicate into a tight loop which never jumps back to the procedure instruction; in this case, it is important that the loop contain at least one entry instruction. The jump_ind instruction performs an indirect jump and is used in the implementation of some built-ins (e.g.call/1 and hash table lookup), not as part of normal predicate compilation.

The conditional control flow instructions support deterministic and non-deterministic clause selection.[3] They correspond to the indexing and cut instructions from the WAM. The BAM choice instruction, which replaces the WAM try, retry, trust, try_me_else, retry_me_else, and trust_me instructions, provides similar functionality, but is more efficient, since it allows a subset of the argument registers to be saved when a choicepoint is created (only those needed by subsequent choices), and a smaller subset to be restored when backtracking (only those needed by a single choice). The indexing instructions, hash and switch, which replace the WAM switch_on_term, switch_on_constant, and switch_on_structure, are more general since they allow indexing on any register or memory word, whereas the WAM instructions only allow indexing on the first argument register. The BAM cut instructions are similar to those of the WAM, except that the choicepoint pointer (r(b)) can be accessed just like any other register. Therefore, there is no need for a special instruction to read this register (WAM's get_level instruction); instead, it is accessed with a move instruction. Also, 'fail' is allowed as an explicit instruction in the BAM, to invoke backtracking; it can also appear as the destination of a jump, which does the same thing.

The unification instructions are used mainly for implementing unification.[4] Two notable exceptions to this are the move instruction, which is also used in implementing cut (as previously mentioned), and the equal instruction, which is also used in implementing deterministic clause selection. The instructions here bear little resemblance to the WAM unification, get, set, and put instructions. Rather than a number of special case instructions used for performing unification, the BAM

---

3. See Table 39 in Appendix A.
4. See Table 40 in Appendix A.

instructions are, for the most part, simpler and more conventional. They consist of instructions like move and push, found on many actual processors, and more Prolog-specific instructions, such as deref, trail, equal, and unify_atomic. By decomposing the unification algorithm into smaller operations, the BAM unification instructions allow significant portions to be optimized away. For example, the WAM always assumes the arguments to be unified must be dereferenced. The BAM makes this a separate operation, which can be removed when not needed. By making trail a separate instruction, trailing can be reduced also. The equal, move, push, and unify_atomic instructions allow simple code to be generated for some of the simple cases of unification, such as unifying atomic values. The unify instruction supports general unification, with mode information provided, when available, to allow some optimizations to be applied. Details about these instructions and their use in implementing unification can be found in [84].

The arithmetic instructions are used mainly to implement the is/2 built-in, and have no counterpart in the WAM.[5] Support for is/2 and other built-in predicates is glossed over in most descriptions of the WAM.

2.2.3 Other Execution Models

Other execution models have been suggested which vary significantly from the original WAM. We briefly describe a few of these below.

Vienna Abstract Machine (VAM)

Since one of the basic operations of Prolog is the pattern matching, or unification, of goals and their arguments with clause heads, the Vienna Abstract Machine (VAM[6]) has two program counters: one for the goal and one for the clause head [47]. Execution in this model proceeds by attempting to match the goal arguments with the clause heads. This approach seems interesting for parallel processing, as it could be extended to have a program counter for each clause and allow pattern matching with all clauses simultaneously. Unfortunately, the instructions still seem rather coarse grain and the

5. See Table 41 in Appendix A.
6. Krall and Berger's VAM is not related to the Vienna Definition Language and VAM of Lucas et al.

instruction set seems to be lacking the instructions needed for a full implementation of Prolog (e.g., is/2 and other built-ins).

Parma's Intermediate Code

In the Parma Prolog compiler, Taylor chose to follow the terminology of most compiler writers, and described an intermediate code, between the Prolog source and the target assembly code [80]. Unlike the abstract machines, which tend to be independent of the target, Taylor's was designed for efficient translation to the MIPS processor. Also, there is little to link it back to the Prolog source. It consists of very simple operations: loading from and storing to memory, performing an operation (e.g., addition, multiplication) on two operands, conditionally and unconditionally branching, and calling a procedure. This allows the MIPS target assembly code to be generated quite easily, by translating each of these intermediate instructions into one or two MIPS instructions, but makes some peephole optimizations in the back-end of the compiler difficult to perform, since the link back to the Prolog semantics is hard to follow. For example, the Aquarius compiler performs a peephole optimization in which it eliminates choice instructions if all code following a choice instruction is guaranteed to succeed.

An intermediate code at this low level may allow optimizations not available even with the BAM instruction set. It would seem, therefore, to be a good choice for exploring efficient execution of Prolog. Unfortunately, Parma is not a full implementation (for example, it can't compile itself) and was not available for use.

Parallel Execution Models

Most abstract models address sequential execution of Prolog. There are a few oriented towards parallel execution.

Hermenegildo defined a model for And-parallel execution of Prolog, named the Restricted And Parallel WAM (RAP-WAM) [33]. This model built on the WAM, the most efficient abstract model for Prolog at the time the RAP-WAM was defined. The RAP-WAM added one memory region, the *goal stack*, for storing *goal frames*. These describe goals which can be executed in parallel and are waiting to execute. In addition to storing environments and choicepoints, the stack contained *parcall* (parallel call) frames. These describe points in execution where goals were added to the goal stack (for parallel execution).

Crammond described a model for the execution of committed choice non-deterministic (CCND) logic languages (e.g., Parlog, Concurrent Prolog and Guarded Horn Clauses) [19, 20]. Although there are similarities to the WAM (they both have a heap), there are also striking differences, as one would expect, since the source languages execute very differently. The WAM stack, used to control sequential execution, was replaced with an and-or process tree, describing alternating levels of goal (or) and clause (and) processes. The instruction set concentrated on manipulating the processes in this process tree. The myriad of WAM instructions for implementing unification are replaced by three very general term unification instructions.

2.3 Prolog Compilation

This section describes Prolog compilation, identifying similarities with compilation for other languages, as well as differences. It gives examples from the Parma and Aquarius Prolog compilers. More details are provided about the Aquarius compiler, however, since it was used in this research.

Figure 6 shows the basic processing steps, or phases, found in most compilers. Although the details of what goes on in these steps may be slightly different for Prolog than for other languages, the overall structure remains about the same.

2.3.1 Source Preparation

During the first phase, the source code is prepared for compilation. For most compilers, this consists of lexical analysis and syntactic analysis, reading the input source code and parsing it into a form suitable for further processing by the compiler.

While this is done in a Prolog compiler, a compiler written in Prolog has an advantage in that Prolog has a built-in term reader, which *automatically performs these functions*, returning an already parsed clause to the compiler.

Once the clauses of a Prolog program have been read, a Prolog compiler will typically perform a number of “normalizing” transformations on this source, to simplify its structure, while retaining its semantics. In the Parma compiler, this results in a

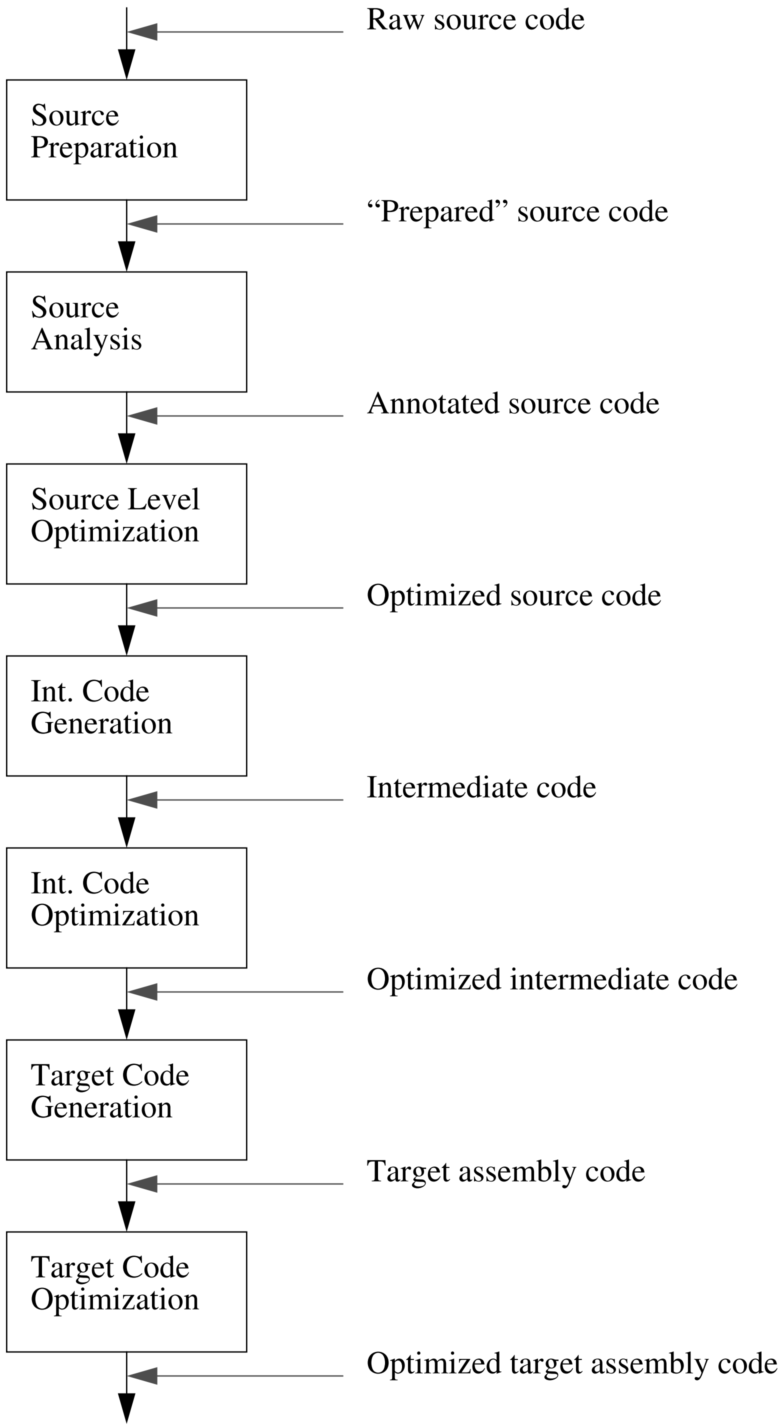


Figure 6: Structure of a Compiler

"normalised" Prolog. In the Aquarius compiler, the result is called "Kernel Prolog". Following are examples of the types of transformations which may be applied:

- Combine all clauses together into a single predicate.
- "Unravel" clause heads into distinct variables, with all head unification becoming explicit in the clause body.
- "Unravel" goals into unifications which prepare the arguments, followed by a call, whose arguments are variables, representing the arguments.
- Flatten nested control structures, like conditionals and disjunctions, into calls to compiler-generated predicates, turning all predicates into disjunctions of conjunctions of simple goals.
- Flatten complex unifications into a small number of simpler cases.

Figure 7 shows the sample Prolog program from Figure 4 after translation into Kernel Prolog by the Aquarius compiler.

```
nreverse(A,B) :-
        A=[], B=[], true;
        A=[E|L], nreverse(L,L1), append(L1,[E],B), true;
        fail.

append(A,B,C) :-
        A=[], B=C, true;
        A=[E|L1], C=[E|L3], append(L1,B,L3), true;
        fail.
```

Figure 7: Sample Kernel Prolog Program

2.3.2 Source Analysis

Once the source program has been read and converted into a form acceptable to the remainder of the compiler, it may be analyzed. This can be simple, local analysis or a global analysis of the whole program. In most compilers, this is referred to as semantic analysis. During this phase, things like type checking are performed.

For Prolog, the source analysis phase can be very complex. In fact, this is where most of the work of this research is concentrated. Rather than *checking* types, a Prolog compiler may need to *infer* types, modes, and a number of other properties of the program. This is typically done using a dataflow analysis technique known as abstract interpretation. Both the Parma and Aquarius compilers have such a dataflow analyzer.

2.3.3 Source-Level Optimization

The results of source analysis can be used to perform source-to-source transformations, in order to optimize the source code prior to, or to place it in a more appropriate form for, code generation. For Prolog, this consists of things like reordering goals in a clause and selecting goals to use for deterministic clause selection. Both Parma and Aquarius perform these types of source transformations. More advanced transformations have been suggested, such as code reordering [74], loop unrolling and combining [22], and partial evaluation [30], but these haven't been incorporated into a full compiler, yet.

Figure 8 illustrates the form of the Prolog program from Figure 4 after source-level optimization in the Aquarius compiler. The clause selection has now become explicit, through the introduction of the '$equal'/2 built-in, which tests for atomic equality, and the '$name_arity'/3 built-in, which performs a non-binding functor test.

2.3.4 Intermediate Code Generation

The next phase is, in some sense, the true compiler. The source code is translated to an intermediate form. This can be an intermediate language, such as three-address code or instructions belonging to some abstract machine. According to Aho et al., the intermediate form "should be easy to produce, and easy to translate into the target program [4]."

The intermediate form in the Aquarius compiler is the BAM assembly language, whereas for the Parma compiler it is a simple three-address code. In both cases, the intermediate code generation is a recursive process, compiling predicates, clauses, and finally goals.

Predicate compilation deals with generating clause selection code. If possible, clause selection should be deterministic, that is, creation of choicepoints should be

```
nreverse(A,B) :-
        ( '$equal'(A,[]) ->
                B=[]
        ; '$name_arity'(A,'.',2) ->
                A=[E|L], nreverse(L,L1), append(L1,[E],B)
        ;       fail
        ).
append(A,B,C) :-
        ( '$equal'(A,[]) ->
                B = C
        ; '$name_arity'(A,'.',2) ->
                A=[E|L1], C=[E|L3], append(L1,B,L3)
        ;       fail
        ).
```

Figure 8: Optimized Kernel Prolog Predicate

avoided. Much of this depends on the results of the global analysis. Both unifications and simple tests can be used to select clauses. Sometimes there are several possible tests that can be performed to select between clauses and a decision tree must be created. The building of this tree is somewhat of an art. Both Parma and Aquarius use heuristics to choose which tests will be used and in what order. These seem to operate well in practice.

Control constructs such as cut and if-then-else are dealt with at the level of the predicate compiler. These can be handled directly by the predicate compiler, or as is done in the Aquarius compiler, turned into simpler code as part of source preparation.

Clause compilation deals with multiple goals. It is responsible for determining when variables are allocated, and when they must be saved across calls to goals. Register allocation can either be performed at this stage, as is done in Aquarius, or deferred to a later phase, as is done in Parma.

The goal compilation process tends to have three cases: compiling calls, unifications, and special cases (e.g., is/2).

The call compiler needs to construct arguments, load argument registers, and generate the call. Argument construction and loading may be very similar to unification. In fact, in the Parma compiler, the arguments to goals are turned into variables, during source transformation, with values assigned through explicit unifications before the call.

The unification compiler tends to be case driven, based on the information available from the analysis and the flexibility allowed by the intermediate code. The BAM provides much more flexibility than the WAM. Although the intermediate code used in Parma is at a lower level than the BAM, it is not clear if this helps generate more efficient code since the BAM instruction set was designed to support the various cases of unification detectable through analysis.

There are a number of goals that demand special handling if efficient code is to result. Many of the simple built-in predicates fall into this category, such as arithmetic comparison (A<B) or evaluation (A is B+5) and type testing (var(X), integer(X)). Aquarius does a good job for most built-ins, and a reasonable job for arithmetic built-ins. Aquarius maintains all arithmetic values as tagged. It could benefit from remembering when values are untagged to perform operations, retagging only after arithmetic evaluation is complete. The Parma compiler gave more consideration to optimizing arithmetic and tagging/untagging operations. This is probably because Parma's target, the MIPS, didn't have the tag support that the VLSI-BAM did, making reduction of untagging and retagging more important for Parma than for Aquarius.

2.3.5 Intermediate Code Optimization

Once the intermediate code has been generated, numerous optimizations can be applied to it. Aho et al. [4] describe a number of generally applicable optimizations, many of which are provided in Parma. This can consist of such optimizations as dead code elimination, instruction peephole optimization, and code migration. Most of these depend only on the semantics of the intermediate code.

In addition to these optimizations, the Aquarius compiler has one optimization, called *determinism optimization*, which is more dependent on the semantics of Prolog or of a section of intermediate code, taken as a whole. Although source transformations are performed to make determinism[7] explicit, some cases are difficult to detect. An example of this is the unification of a variable with a term. Aquarius handles cases which are easily recognizable at the source level as being deterministic. Some other cases can be and are handled at the intermediate code level. In determinism optimization, if a choice instruction is followed by code known to never fail and then by a cut instruction, the choice instruction is removed (since the code is deterministic). This increases determinism for cases where deterministic code is generated, but detecting that this is going to occur, at the source level, is difficult (e.g., it requires too much information about how the code generator operates).

2.3.6 Target Code Generation

Once the intermediate code has been optimized, it can be translated into target assembly code. This can be a very simple task or somewhat involved, depending on how removed from the target the intermediate code is. For Parma, the translation is very simple, since each intermediate instruction becomes one or two MIPS instructions.

For Aquarius, the translation is a little more involved, but is still not difficult. Aquarius has a separate back-end, capable of targeting the BAM intermediate code to a number of different processors, including the MIPS.

2.3.7 Target Code Optimization

Once the target code has been generated, another optimization phase can be performed. The optimizations applied here are similar to those applied at the intermediate level. The differences and similarities depend on how close the intermediate and target code languages are and whether there is any added value in repeating optimizations applied earlier.

---

7. Code is deterministic when at most one choice of many will ever succeed. If the compiler can detect that some code (predicate) is deterministic, it can generate code that selects a choice based on simple tests, as opposed to creating a choicepoint for backtracking.

In Parma, for example, register allocation is performed very late in the compilation process. Therefore, some optimizations based on register use can't be performed until this last phase, whereas these happen at the intermediate level in Aquarius. This may increase the compilation time (since the steps before register allocation will have to deal with less optimized code), but can result in better register allocation and ultimately better code.

The target code optimizer in Aquarius is, for the most part, fairly simplistic, assuming that the majority of the optimizations have been applied at the intermediate level. One major optimization in Aquarius, which is currently only applied to the SPARC processor, is instruction reordering to fill delay slots and to reduce memory stalls. This can't really be done before we have the target assembly code.

2.4 Compile-time Optimization

The previous section touched briefly on the different steps in the compilation process where optimizations can be applied and what some of these optimizations may be. This section addresses this in more depth. Although we address compile-time optimization, in general, emphasis is given to optimizations based on dataflow analysis information.

2.4.1 When to Optimize

As we saw previously, the overall structure of a Prolog compiler isn't much different from other compilers. Therefore the points in the compilation process at which we can apply optimizations remain the same. We will consider three categories corresponding to the three main languages we work with: source level, abstract (or intermediate) level and assembly level optimizations.

At each level, optimizations can be applied either during the generation of code at one level, or as a transformation of the code, once generated. Performing optimizations during the generation of code tends to make the code generation algorithm more complex because there are more cases to be dealt with. Applying optimizations as transformations on the code is simpler to describe and implement, but can be more time consuming since it may be undoing or redoing some of the operations performed during code generation.

The compiler-writer has some flexibility in choosing the compilation phase in which to apply various optimizations. There is no "best" place in general. In fact, all optimizations could be performed in the final phase of the compiler if sufficient information is passed down, but this is hardly efficient. Therefore, we give some advice and examples in the following sections to provide guidance on these decisions.

2.4.1.1 Source-Level Optimizations

Source-level optimization occurs shortly after dataflow analysis. These optimizations simplify the source code. Following are examples of and reasons for source-level optimizations:

- Elimination of tests known always to succeed and of code known always to fail. This reduces the amount of code that must be translated into abstract code.
- Determinism detection. If one clause in a set can be selected deterministically, Prolog's general choicepoint mechanism can be replaced by much simpler tests. Hickey and Mudambi identify two types of such determinism: head determinism, in which the clause head unification can be used to select a clause, and primitive determinism, in which primitive tests (e.g., X>Y) appearing in the clause bodies can be used for clause selection [31].
- Source level transformations [28]. Examples of these are:
    1) program specialization [30], where specialized versions of predicates are generated to deal with different uses within a given program;
    2) fold/unfold transformations [74], which promote goals up or down in the program hierarchy or rearrange goals in an effort to reduce the amount of searching required by a program (the ultimate result would be making a program deterministic); and
    3) loop optimizations [22], e.g. loop fusion and moving loop-invariant code out of loops.

2.4.1.2 Abstract-level Optimizations

During and following the generation of the abstract machine code, there are a number of optimizations that can be applied to either eliminate abstract code or to allow simpler code to be substituted. Debray describes techniques for applying optimizations to abstract code, explaining how to deal with Prolog specific features, like backtracking [23].

Following are examples of and reasons for abstract-level optimizations:

- Unification simplification. Much can be done, as we will see later, to simplify unification, based on dataflow information about the variables in the unification. Although this can be done as a source-level transformation [8], it seems better to do this during abstract code generation. Unification optimizations can then be done easily, during the generation of abstract code. Attempting to capture all of the different cases through source transformations may result in either a loss of optimization opportunities or a wide range of transformations.
- Simplification and elimination of basic operations. Operations, such as dereferencing and trailing, can be simplified or eliminated, given dataflow information. This can be done either during the generation of the abstract code or as a transformation of the abstract code.
- Classical instruction optimizations (if applicable at the abstract level) [4]. Examples include dead and duplicate code elimination, jump and label elimination, strength reduction on operands, and peephole optimization.
- Prolog-specific optimizations. Examples include Van Roy's last call peephole optimization and determinism optimization [84]. Other examples, such as delaying environment allocation and removing redundant bounds checks can be found in [23].

2.4.1.3 Low-level Optimizations

There are a number of low-level optimizations that can be performed during and after translation of the abstract code into target code [4] (assuming, of course, that they haven't been applied previously). Many of these are classical optimizations, like peephole optimizations and dead and duplicate code removal, which are fairly independent of where the target code came from. For intermediate languages that look like assembly languages (as is the case for both Parma and Aquarius), these optimizations may be very similar to those performed at the abstract code level. In fact, it usually is only worth applying these at one level. Unless there are good reasons to wait, it is better to apply them at the abstract level, where the amount of code tends to be smaller. Following are examples of and reasons for low-level optimizations, in addition to those mentioned previously:

- Simple peephole optimizations, to deal with artifacts of the translation from abstract code to target code. In Aquarius, for example, operands in memory are loaded into temporary registers to operate upon them. Rather than reloading such a value if it is used in the next instruction, it is reused from the temporary register. A better scheme is to perform a more global assignment of variables and values to registers, as is done in Parma.
- Removing register movement which might be introduced by a simplistic register allocation algorithm.
- Combining multiple stack increments (for example, when building a term on the heap).
- Removing tagging and untagging operations, by remembering what values have been untagged [23].

2.4.2 What to Optimize

We will now look at some specific optimizations. We will describe the optimizations, identify what type of information is needed in order to apply the optimization, and when the optimization should be applied.

2.4.2.1 Unification

The type of optimization most often described for Prolog is simplification of unification [8, 31, 40, 54, 80, 84]. Perhaps this is because unification is so basic to the operation of logic programming, consisting not only of explicit unification goals in clause bodies, but also implicitly occurring as part of clause selection, in head unification. In fact, after making head unification explicit, unification comprises 53% of the body goals in our benchmark suite.[8]

In this section, we will describe optimizations within a single unification goal, leaving broader optimizations (e.g., clause selection) for later. We will start by

8. The benchmark suite is described in Chapter 6.

examining the general unification algorithm, given in Figure 10, and discuss ways to simplify this algorithm.[9] In the general case, most, if not all, of this algorithm will be implemented by a run-time support subroutine. By implementing part of it in-line, we allow more opportunities for optimization, at the expense of code size. Aquarius implements general unification by providing a subroutine for unifying two dereferenced, bound values. The remainder of the algorithm is implemented in-line as shown in Figure 9.

```
aquarius_unify( V1, V2 ):
    if (tag(V1)==tvar && (tag(V2)!=tvar || V1>V2))
            { trail(V1); mem(V1) = V2; }
    else if (tag(V2) == tvar)
            { trail(V2); mem(V2) = V1; }
    else unify_nonvars( V1, V2 );
```

Figure 9: General Unification in Aquarius

Most of the optimizations we will look at are based on knowing the modes [61] of the variables involved. We will also examine some optimizations which require more implementation-dependent information, such as reducing or eliminating trailing and dereferencing.

To begin with, we will assume that a source-level transformation has been applied to ensure that all unifications are of the form, X=Y, where X is a variable and Y is either a variable or a term. This is a syntactic transformation, requiring no dataflow information. Basically, this reduces the number of cases that we must consider. The only impact on our ability to apply optimizations or gather dataflow information is the breaking apart of unification of two terms. In this case, we either knew at compile-time that the unification would fail (because the functor or arity didn't match), or we broke the unification into the unification of the corresponding arguments. Separating this

9. A C-like syntax is used herein when describing algorithms.

```
unify( V1, V2 ):
    V1 = deref(V1); V2 = deref(V2);
    if (tag(V1)==tvar && (tag(V2)!=tvar || V1>V2))
            { trail(V1); mem(V1) = V2; }
    else if (tag(V2) == tvar)
            { trail(V2); mem(V2) = V1; }
    else if (V1 == V2)
            return;
    else switch (tag(V1)) {
            case tint: case tflt: case tatm:
                    if (V1 != V2) fail;
                    break;
            case tlst: if (tag(V2) != tlst) fail;
                    unify( mem(V1), mem(V2) );
                    unify( mem(V1+1), mem(V2+1) );
                    break;
            case tstr: if (tag(V2) != tstr) fail;
                    if (mem(V1) != mem(V2)) fail;
                    for (i=1; i<arity(mem(V1)); i++)
                            unify( mem(V1+i), mem(V2+i) );
                    break;
            }
```

Figure 10: Unification Algorithm for a WAM-like Memory Model

unification can cause the dataflow analysis to be less precise or require a more complex dataflow analysis. In the benchmark programs, this type of unification never occurs. We will address the dataflow analysis aspects of this transformation later.

Now, we will consider the case of unifying a variable with a (non-variable) term. The unification algorithm simplifies into one of three cases, depending on the term's type, as shown in Figure 11. Further simplifications can be applied if the mode of the variable is known. If the variable is known to be unbound, the unification consists of constructing the term on the heap and making the variable point to the newly created term (or a simple assignment if the term is atomic). If the variable is known to be bound, the unification becomes a functor comparison, followed by unification of the arguments, with further optimizations applied if the functor of the variable is known or the modes of the variable's arguments are known.

```
case 1: /* The term, Y, is atomic */
    if (tag(X) == var) { trail(X); mem(X) = Y; }
    else if (X != Y) fail;
case 2: /* The term, Y, is a list */
    if (tag(X) == var) {
            trail(X); mem(X) = r(h);
            construct list on heap; }
    else if (tag(X) != tlst) fail;
    else recursively unify mem(X) with head(Y) and
            mem(X+1) with tail(Y);
case 3: /* The term, Y is a structure */
    if (tag(X) == var) {
            trail(X); mem(X) = r(h);
            construct the structure on the heap;}
    else if (tag(X) != tstr || mem(X) != functor(Y)) fail;
    else recursively unify arguments of X and Y;
```

Figure 11: Unifying with a (nonvariable) term

In the case where the mode of the variable is not known, code can be generated to test the mode, followed by code to handle the bound and unbound cases. This can expand the code, however. This is the approach taken by the Aquarius compiler.

Instead, the term could be constructed on the heap and the general unification algorithm invoked. This approach could also be used for the nonvar case, to reduce the code size.

Now, we will examine the case of unifying two variables. Basically, there are four cases in the unification, based on the modes of the variables: unifying two unbound variables, unifying an unbound variable with a bound variable, unifying a bound variable with an unbound variable, and unifying two unbound variables. If we know something about the modes of the variables at compile-time, we can eliminate some of these cases. Of course, this is only useful if we can break the unification algorithm into pieces. The WAM did not allow this [88]; the BAM does (as described previously).

When binding an unbound variable, the variable is normally trailed, so that it may be restored to an unbound state if backtracking occurs. This may be eliminated if it is known that the variable was created after the current choicepoint was created.

When one of the variables is known to be atomic (or when unifying with an atomic term), the unification simplifies to the algorithm shown in case 1 of Figure 11. If the other variable is known to be unbound, this becomes a simple assignment. If it is known to be bound, this becomes a comparison. Furthermore, this unification does not require backtracking on failure when unifying with a bound value (comparison) and may, therefore, be used in clause selection optimization (see below). Unification of two ground variables may also be used in this manner, but this requires a special unification routine or a simplified choicepoint, since failure of general unification usually causes backtracking to occur.

Citrin describes a technique to perform multiple unifications in parallel, given information about the independence of variables [12]. This is useful for parallelizing either head unifications or explicit unification, when unifying two terms. When Citrin proposed this technique, he envisioned a machine with multiple unification units, each able to perform an independent general unification. Recent investigations into superpipelined and super-scalar architectures may benefit from parallel unification techniques for specialized unification (i.e., those we can expand in-line) by interleaving the code for two or more unifications. Type and aliasing analysis can provide the information needed for these optimizations.

Marriott and Sondergaard [56] discuss ways to safely remove the occur-check [70] from unification. To do this, we need variable independence information. However, most Prolog implementations do not include the occur-check in unification, anyway. So, we mention this optimization, but will not explore it in detail.

2.4.2.2 Clause Selection

Nominally, each clause is one choice in a set of choices for a predicate, with a choicepoint created at predicate entry and each clause tried through backtracking. Quite often, however, this very general clause selection mechanism can be replaced with a specialized, deterministic mechanism [31, 40, 54, 84].

Hickey and Mudambi describe two types of determinism: head determinism and primitive determinism [31]. Clauses are considered to be head deterministic if the corresponding head arguments to some call argument known to be bound are not themselves unifiable. The predicates in the sample program given in Figure 4 are head deterministic if we assume the first argument of each is bound when the predicates are called. Clauses are considered to be primitive deterministic if some test(s) of the arguments are mutually exclusive across clauses. Figure 12 provides an example, where an arithmetic comparison is required to select a clause. We will assume head unraveling has been done, so that all head unifications actually appear in the body, and therefore qualify as primitive tests. Van Roy calls these collections of tests, *test-sets*. Table 2 provides a list of possible test-sets.

```
min(A, B, A) :- A =< B.
min(A, B, B) :- A > B.
```

Figure 12: Example of Primitive Determinism

To determine when clauses are deterministic, therefore, it helps to know the modes of arguments (specifically, we are looking for bound arguments). Also, we need to know when body goals can be reordered (to move primitive tests closer to the front of the clause). To do this, we also need mode information and variable independence information (two goals can be reordered if their variables are independent and at most

Table 2: Test-sets used for clause selection

Test-set	Description
equal	Select based on whether two variables match or not (X==Y). This requires one of the variables to be simple (atomic or unbound).
hash(atomic)	Select based on the atomic value of a variable. As a special case, select based on whether or not a variable matches a specific atomic value.
hash(structure)	Select based on the functor of a variable bound to a structure. As a special case, select based on whether or not a variable has a given functor.
tag	Select based on the tag (type) of a variable. There are a number of special cases, testing for a single tag value or pair of tag values (e.g., integer(X)).
comparison	Select based on an ordered comparison of a variable with either a variable or a constant. This can use either arithmetic or standard ordering.
list	Select based on whether a list is empty or non-empty.

one has, or makes use of, side effects). Both the Aquarius and the Parma compilers are fairly conservative in their reordering. They restrict the reordering to unification goals and simple built-in tests.

If dataflow analysis was unable to provide sufficiently "rich" modes in order to do clause selection, Van Roy performs "type enrichment" in order to give the compiler something to work with [84]. He collects the set of arguments which can be used for clause selection and determines if any are known to be bound. If no such arguments are found, he selects one argument which is also not known to be unbound (i.e., analysis was unable to determine the mode of the argument, or the predicate is known to be called with the argument in different degrees of instantiation). In this case, Van Roy's compiler transforms the predicate into a mode test, followed by a conditional call to one of two variants of the original predicate: one in which the argument is bound and one in which it is unbound. Figure 13 shows the result of this transformation for the first argument of `append/3` from Figure 4.

```
% The original append/3 predicate has been replaced by code which tests the mode
% of the first argument and calls an appropriate specialization.
    append(A,B,C) :-
            ( var(A) -> '$v_append'(A,B,C) ; '$nv_append'(A,B,C) ).

% Because of the mode test in append/3, the first argument of '$v_append'/3 is
% guaranteed to be unbound.  A choicepoint will be created and used to sequence
% through the clauses.
    '$v_append'([],L,L).
    '$v_append'([E|L1],L2,[E|L3]) :- append(L1,L2,L3).

% Because of the mode test in append/3, the first argument of '$nv_append'/3 is
% guaranteed to be bound.  Although this code looks almost identical to the
% previous, the different mode information will cause different code to be
% generated.  Because the first argument is known to be bound, it can be used for
% clause selection.
    '$nv_append'([],L,L).
    '$nv_append'([E|L1],L2,[E|L3]) :- append(L1,L2,L3).
```

Figure 13: Result of Type Enrichment Transformation

Hickey and Mudambi perform a related source-level transformation, specializing the predicates based on call modes, to improve determinism [31]. They determine which arguments of each predicate are ground, remembering each call separately. They then generate a specialized version of each predicate for each different calling condition. This improves head determinism. An example of this transformation is shown in Figure 14.

Normally, failure during unification causes backtracking to the latest choicepoint. Occasionally, this causes useless work to be performed, since the unification failure was due to bindings earlier in the program. Semi-intelligent backtracking attempts to remedy this by backtracking to a point where useful work may be performed [11, 59].

```
main :- append(X,Y,[1,1,1,1,1]), append(X,[0|Y],L), write(L), fail.
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).
```

after specialization, this becomes:

```
main :- append_ddc(X,Y,[1,1,1,1,1]), append_ccd(X,[0|Y],L), write_c(L), fail.

% This version is non-deterministic, and requires a choicepoint.
append_ddc([],L,L).
append_ddc([H|T],L,[H|R]) :- append_ddc(T,L,R).

% This version is deterministic, and can perform clause selection on the
% tag of the first argument.
append_ccd([],L,L).
append_ccd([H|T],L,[H|R]) :- append_ccd(T,L,R).
```

Figure 14: Example of Ground Parameter Specialization

2.4.2.3 Basic Operations

There are a number of basic operations used in the implementation of Prolog which can be simplified or removed. These optimizations occur at either the abstract or assembly code level. Because of this, they are somewhat implementation-dependent. We address them by examining the BAM instruction set (described in Appendix A), with comments on general applicability.

Run-time Garbage Collection

One of the nice features of the Prolog language is that much garbage collection occurs automatically, relieving the programmer from performing this odious task and ensuring it's done correctly (this is also true of LISP). In order to perform garbage collection, the Prolog run-time system needs some way to determine what data is still "active", that is, what data can still be accessed. It does this by defining a "root set" of places from which it must begin searching for active references. Typically, the root set will be some subset of the registers.

For the WAM, one could consider searching from all registers, but it would be better to know if any of the argument registers are not needed. For the BAM, it is necessary to know exactly which argument registers to include, since some of them may contain untagged values. Garbage collection can change the values of tagged pointers if it compacts memory; therefore, it is important that it not try to access untagged values (which might look like tagged pointers). In addition, there are typically run-time support routines which can be written more efficiently if it is known that garbage collection will not occur during execution (e.g., term copying in Aquarius).

BAM solves this by only performing garbage collection at points in the code identified with `entry` instructions. This instruction also indicates the number of argument registers that must be included in the root set. Nominally, this is placed at the entry to each predicate, but instruction-level optimizations of the BAM code can move it around. If the rate at which information is added to the various memory sections is known, `entry` instructions can be eliminated (as long as the garbage collection check occurs frequently enough). A simple way to approximate this at the Prolog level is to limit the number of operations (e.g., unifications, built-ins, etc.) performed between predicate entries.

Local Stack Allocation

Both the WAM and the BAM typically store both environments and choicepoints on the same stack, called the local stack. One common way to implement this is to have two “top of stack” pointers: one for the current environment and one for the current choicepoint. All environments are linked together in a backward chained, linked list, as are the choicepoints. Items are popped off this stack by following the linked list backwards. Therefore, to find the real top of the stack when adding an item, we must find the maximum of the environment and choicepoint pointers.

If we know what the top item on the stack is, we can eliminate this pointer comparison. This could be approximated through a global analysis of the intermediate code or the source code. Analysis at the source code level will tend to be very imprecise, however, since it will be based on an approximation of when permanent variables (an environment) are needed and when a choicepoint (for non-deterministic clause selection) must be created. It can also be done as a peephole optimization on the intermediate or target code. This lacks the larger view given by global analysis.

Meier describes a technique for reusing environments and choicepoints on the local stack, for tail recursive predicates [60]. This is an optimization best applied on the abstract code since it requires knowledge of when choicepoints and environments are being created and destroyed and how large environments are.

Backtracking Simplification

The failure routine, which is executed when a given search path (choice) fails, is responsible for backtracking to the last choice point and continuing with the next choice. To do this, the machine must be restored to the state it was in when the choice point was created. On most implementations, this means detrailing any assignments that have been made to unbound variables (or at least, those made to variables created before this choice point), and restoring some of the registers.

For the WAM, all argument registers are saved and restored, thereby making the failure routine independent of the point from which it is invoked. For the BAM, backtracking is broken into choice-independent and choice-dependent operations (see Appendix A). The choice-independent operations consist of detrailing and restoring certain special purpose registers, such as the environment and choicepoint pointers. The choice-dependent operations consist of restoring those argument registers needed in the choice (and only those registers). This is determined after register assignment, by collecting the set of registers used within each choice. Hickey and Mudambi also suggested this as a modification to the WAM [31].

The choice-independent failure processing can be optimized for a given choice if we know about changes to the program state. For example, detrailing is not needed if no bindings have been created since the last choice. In the extreme case, where no changes have been made to the program state, backtracking becomes a simple jump to the next choice (actually, this is what clause selection optimization is all about). The problem with this is that it requires restoring different state information for each failure condition depending on what state information has been modified. This can either be done in-line, or by having multiple failure routines. In either case, this expands the code.

Dereferencing Simplification

Most Prolog implementations handle variable aliasing by making one variable point to another. Therefore, to obtain the real value of a variable, it is necessary to follow this (potentially lengthy) chain of pointers; this is referred to as dereferencing. On the WAM, all instructions whose operands might contain pointer chains begin by dereferencing the arguments. On the BAM, this is done by a separate instruction, `deref`, thereby allowing this operation to be optimized [54, 78, 84].

Touati and Despain showed that for the WAM, most accesses do not require dereferencing and when dereferencing was required, the reference chain was never longer than two [82]. Table 3 shows similar results on the BAM for our benchmarks, compiled using the Aquarius compiler. 78% of all dereferences can be resolved with no extra memory reads. Virtually all remaining dereferences require only a single memory read. Table 3 also shows that the time spent dereferencing is around 12% on average.

If the exact length of the reference chain is known, we can replace the `deref` instruction with a fixed number of indirect references. Even when the length is known only to within a range, we can directly follow those references we know exist and then conditionally follow the remaining references. For example, if we know the reference chain for a given variable is between 1 and 2 references long, we can perform an indirect reference, followed by a conditional indirect reference. While faster than the general dereference operation, this expands the code size.

Other optimizations can be applied if we know the mode of the variable. Normally, the dereference operation ends when it encounters a bound value (a value whose tag is not ‘tvar’) or an unbound value (a value whose tag is ‘tvar’ and value matches the address from which it was read). If the mode of the variable is known, this can be reduced to a single termination test.

Trailing Removal

When backtracking, we must unbind any variable bound since the creation of the choice point. In order to do this, both the WAM and the BAM record bindings in a trail

Table 3: Dereferencing in Benchmarks

benchmark	# derefs	depth 0	depth 1		depth 2		% time
		% nv	% var	% nv	% var	% nv	
deriv	2	0.0	100.0	0.0	0.0	0.0	0.4
nreverse	0	0.0	0.0	0.0	0.0	0.0	0.0
qsort	0	0.0	0.0	0.0	0.0	0.0	0.0
serialise	620	96.9	1.6	1.5	0.0	0.0	9.2
mu	2503	79.4	20.6	0.0	0.0	0.0	24.4
pri2	0	0.0	0.0	0.0	0.0	0.0	0.0
queens_8	0	0.0	0.0	0.0	0.0	0.0	0.0
fast_mu	859	81.0	13.7	5.2	0.0	0.0	6.5
query	0	0.0	0.0	0.0	0.0	0.0	0.0
press1	1172	91.8	7.3	0.6	0.0	0.3	6.3
tak	0	0.0	0.0	0.0	0.0	0.0	0.0
sendmore	21150	95.9	4.1	0.0	0.0	0.0	3.3
poly_10	3129	0.0	100.0	0.0	0.0	0.0	2.2
zebra	50915	99.8	0.2	0.0	0.0	0.0	3.7
prover	1084	82.3	7.7	10.0	0.0	0.0	11.5
meta_qsort	6487	69.1	25.9	5.0	0.0	0.0	14.5
nand	25054	98.7	1.2	0.2	0.0	0.0	13.1
chat_parser	181443	75.0	24.0	0.8	0.2	0.0	12.7
browse	1711774	77.5	11.8	10.7	0.0	0.0	20.1
unify	1293	68.3	31.6	0.1	0.0	0.0	11.2
flatten	433	47.1	51.3	1.6	0.0	0.0	6.9
crypt	4650	70.8	29.2	0.0	0.0	0.0	21.1
simple_analyzer	12968	61.7	37.5	0.8	0.0	0.0	6.5
reducer	50364	59.6	32.3	8.2	0.0	0.0	15.0
boyer	548117	82.7	17.3	0.0	0.0	0.0	6.0
Total:	2624017	78.6	14.1	7.2	0.0	0.0	12.1

stack. This stack stores the addresses of variables which have been bound. In order to keep the size of the trail stack from getting too large, both the WAM and the BAM only record bindings for variables older than the choice point; variables created after the choice point are thrown out by restoring the heap pointer to the value it had when the choice point was created

For the WAM, the trailing operation is an integral part of unification. For the BAM, it has been separated out when unifying with an atomic value. Therefore, the trailing operation can be eliminated at compile-time if it is known that the variable being bound is newer than the current choicepoint [31, 78]. In order to do this, we need to know when choicepoints and variables are created. Taylor determines this at the source level, by approximating which predicates will create choicepoints. It may be better to wait until intermediate code has been generated, and it is known exactly which predicates generate choicepoints. The age of variables still must be approximated, however. Van Roy did not implement such an optimization directly. Instead, his uninitialized variable optimization deals with the most common cases.

Tag Manipulation

Because Prolog is a dynamically typed language, all implementations require some way to determine the type of a data value at run-time. This is typically done by having each word consist of a data tag and a data value. Table 4 describes the tags used in the BAM.

Table 4: Data Tags in the BAM

Tag	Description
tvar	This identifies a reference to another value. The data value is a pointer to the actual value. When the pointer is self-referential, the value is unbound (variable).
tlst	This identifies a list. The data value is a pointer to two tagged words in the heap, containing the head and tail of the list, respectively.
tstr	This identifies a structure. The data value is a pointer to the structure, in the heap. The first word described the name and arity of the structure (see tatm). The following words contain the tagged values of the structure arguments.

Table 4: Data Tags in the BAM

Tag	Description
tflt	This identifies a floating point value. The data value contains the floating point number.
tatm	This identifies a Prolog atom. The data value is a unique integer value identifying the atom. In most implementations, it is an index into the atom name table. This tag is also used for the functor of a structure. The data value identifies both the name and arity of the structure.
tint	This identifies an integer value. The data value contains the integer.

Machines with a tagged architecture, like the VLSI-BAM [34], are specifically designed to deal efficiently with these tags. On most general purpose architectures, however, a significant amount of time can be spent splitting words into tags and data, operating on the data, and putting the result back together. If the tags are known beforehand, this can be simplified [23]. One extreme example is the evaluation of an arithmetic expression [54, 80]. If we don't know whether the data is integer or floating point, we must first test this, extract the numeric data from the word, operate on it, and recombine it with the original tag. If it is known, for example, that the data is integer, all handling of floating point can be eliminated. Further, if the expression contains addition or subtraction of a constant, the expression can quite often be evaluated without ever removing the integer tags.

Similar optimizations can be applied to pointer tags. Since most machines support indirect addressing with a constant offset, this offset can be used to remove the pointer tag. Furthermore, the value of a pointer tag is almost always known from context, since we typically won't access the data until we know its type (probably through clause selection).

To perform these optimizations, we need to know the tags of the variables or, to put it another way, we need to know the variables' types.

2.4.2.4 Built-in Predicates

Some benchmark programs spend a significant portion of their execution time within the built-in predicates. Therefore, it is important that the built-ins be efficient. If they are written in Prolog (as most of them are for Aquarius), all of the techniques described in this dissertation can be applied to optimize them.

Some of the built-ins are expanded to in-line code. These can be eliminated or greatly simplified if enough is known about the arguments [8, 84]. For example, var(X), nonvar(X), and ground(X) may be eliminated if the mode of the argument is known. Some calls to functor/3 can be implemented as simple tag tests (and eliminated if the type of the first argument is known).

For out-of-line built-ins, a collection of specialized versions can be developed for each built-in, which are called depending on information known about the arguments. Van Roy called this *modal entry* selection. The Aquarius system uses this technique. For example, the `functor/3` built-in has 20 different versions, described in Table 5. This technique provides a 3% performance improvement for all the benchmark programs and 8.4% for those that use built-ins with specializations, as shown in Table 6.[10]

2.4.2.5 Compile-time Garbage Collection

One of the advantages Prolog has over many other languages is automatic allocation and freeing of dynamic data structures. This relieves the programmer of the burden of explicitly allocating and deallocating memory and ensures that this is done correctly (e.g., that memory isn't freed before its last use). Dynamic memory allocation occurs when an unbound variable is unified with a term (either explicitly or through head unification). Memory deallocation is more complex, however. These dynamically allocated terms must be kept in the heap until they are no longer accessible, at which time their storage can be reclaimed. The Aquarius system implements this with a simple stop and collect garbage collection algorithm [64].

If it is known at compile-time when data is no longer needed, we might be able to reuse some of the memory at run-time, thereby reducing the frequency with which we would need to perform run-time garbage collection. This detection and reuse of memory

10. These multiple versions share a lot of code. In fact, the more general versions tend to perform simple tests (e.g., mode tests) and then call the appropriate specialized versions.

Table 5: Specialized Versions of functor(A,B,C) Built-in Predicate

Name	Specialization
functor	No specialization.
'$ functor *1'	A is unbound and independent. Used to construct a functor.
'$ functor 1'	A is bound (has a functor). Used to get/compare functor and arity.
'$ functor *2'	B is unbound and independent. Used to get functor.
'$ functor *3'	C is unbound and independent. Used to get arity.
'$ functor 2'	B is atomic. Used to compare functor of A or construct new term, given the functor.
'$ functor 3'	C is an integer. Used to compare arity of A or construct new term, given the arity.
'$ functor *1 2'	A is unbound, independent. B is atomic. Used to construct new term, given the functor.
'$ functor 1 *2'	A is bound. B is unbound. Used to get the functor of A.
'$ functor 1 2'	A is bound. B is atomic. Used to test the functor of A.
'$ functor *1 3'	A is unbound, independent. C is an integer. Used to construct a new term, given the arity.
'$ functor 1 *3'	A is bound. C is unbound. Used to get the arity of A.
'$ functor 1 3'	A is bound. C is an integer. Used to test the arity of A.
'$ functor *2 *3'	B and C are unbound and independent. Used to get the functor and arity of A (assuming it is bound).
'$ functor 2 3'	B is atomic. C is an integer. Used to test functor/arity of A or construct a new term, given both the functor and arity.
'$ functor *1 2 3'	A is unbound, independent. B is atomic. C is an integer. Used to construct a new term, given both the functor and arity.
'$ functor 1 *2 *3'	A is bound. B and C are unbound, independent. Used to get the functor and arity of A.
'$ functor 1 *2 3'	A is bound. B is unbound, independent. C is an integer. Used to get the functor of A and test its arity.
'$ functor 1 2 *3'	A is bound. B is atomic. C is unbound, independent. Used to get the arity of A and test its functor.
'$ functor 1 2 3'	A is bound. B is atomic. C is an integer. Used to test the functor and arity of A.

Table 6: Performance Effect of Built-in Specialization

Benchmark	Specialized built-ins used by benchmark	Instructions executed		Speedup
		Original	w/ spec.	
deriv	none	3622	3622	0.0%
nreverse	none	7024	7024	0.0%
qsort	none	8515	8515	0.0%
serialise	none	21482	21482	0.0%
mu	none	39274	39274	0.0%
pri2	none	29424	29424	0.0%
queens_8	none	64129	64129	0.0%
fast_mu	length	51954	51909	0.1%
query	none	77707	77707	0.0%
press1	functor, arg	69461	60307	15.2%
tak	none	1415344	1415344	0.0%
sendmore	none	2052216	2052216	0.0%
poly_10	none	999838	999838	0.0%
zebra	none	4150368	4150368	0.0%
prover	none	37652	37652	0.0%
meta_qsort	none	196659	196659	0.0%
nand	recorded, erase, recorda	578947	578476	0.1%
chat_parser	none	5000476	5000476	0.0%
browse	length, functor, arg	36022549	36012047	0.1%
unify	functor, arg, compare	57568	49164	17.1%
flatten	=.., copy, name	29627	27260	8.7%
crypt	none	92044	92044	0.0%
simple_analyzer	functor, arg, compare, sort, keysort	894080	780335	14.6%
reducer	compare, functor, arg	1705085	1626219	4.9%
boyer	functor, arg	39615086	34005215	16.5%
Geometric mean (overall):				3.0%
Geometric mean (benchmarks using specialized built-ins):				8.4%

at compile-time is referred to as compile-time garbage collection [8, 9, 54, 59, 66]. In addition to reducing run-time garbage collection, this can reduce memory references occurring in the body of the program [35].

The sample program in Figure 4 can benefit from compile-time garbage collection. The top-level predicate (nreverse/2) reverses a list by appending the first element to the end of the reversal of the remainder of the list. The append/3 predicate performs the list concatenation by copying the first list up to its end, which it changes into a pointer to the second list. For a list of length N, this results in copying N-1 elements, then N-2, and so forth, for a total of N(N-1)/2 list element copies. Since it is known that the first input to append/3 is not needed after the call completes, the concatenation can be performed without copying. Instead, append/3 can simply traverse to the end of the first list and change the end of the list to a pointer to the second list. The BAM code to implement append/3 naively and using compile-time garbage collection is given in Figure 15. To perform this optimization, we need to know the types of variables involved (i.e., the size of the structure being allocated or deallocated) and when data becomes "dead" (i.e., when it is no longer needed). The latter requires information about variable independence (if two variables are dependent, their memory cannot be reused until we are done with both variables).

2.5 Summary

This chapter provided an overview of Prolog and related topics. It gave a brief introduction to the language and then described a number of models for Prolog execution. It described the general form of a Prolog compiler, comparing and contrasting this to compilers for other languages. Finally, it showed numerous opportunities for compile-time optimization of Prolog.

In the following chapters, we will see how to perform global analysis of Prolog programs at compile-time, in order to obtain the information needed to apply these optimizations.

Naive implementation:

```
procedure( append/3 ).
    entry( append/3, 3 ).
    equal( r(0), tatm^[], l(append/3,1) ).
    move( r(1), [r(2)] ).
    return.
label( l(append/3,1) ).
    test( ne, tlst, r(0), fail ).
    move( [r(0)], r(3) ).
    move( [r(0)+1], r(0) ).
    move( tlst^r(h), [r(2)] ).
    push( r(3), r(h), 1 ).
    move( tvar^r(h), 1 ).
    move( tvar^r(h), r(2) ).
    adda( r(h), 1, r(h) ).
    entry( append/3, 3 ).
    equal( r(0), tatm^[], l(append/3,1) ).
    move( r(1), [r(2)] ).
    return.
```

With compile-time garbage collection:

```
procedure( append/3 ).
    entry( append/3, 3 ).
    equal( r(0), tatm^[], l(append/3,1) ).
    move( r(1), [r(2)] ).
    return.
label( l(append/3,1) ).
    move( r(0), [r(2)] ).
label( l(append/3,2) ).
    equal( [r(0)+1], tatm^[], l(append/3,3) ).
    trail_bda( [r(0)+1] ).
    move( r(1), [r(0)+1] ).
    return.
label( l(append/3,3) ).
    move( [r(0)+1], r(0) ).
    jump( l(append/3,2) ).
```

Figure 15: Example of Compile-time Garbage Collection

Chapter 3: Abstract Interpretation

This chapter provides an overview of abstract interpretation, describes how to apply it to Prolog in general, and then specifically for compile-time analysis and optimization.

3.1 A Theoretical Model for Dataflow Analysis

Program analysis has long been recognized as a useful tool for compiler optimizations. Allen and Cocke describe some early applications of data and control flow analysis [1, 2, 13]. Cousot and Cousot provided a unified view of these and other data flow analyses with a technique they called abstract interpretation [17]. Since then, many researchers have applied this technique to the analysis of logic programs [24, 48, 61, 63, 80, 84].

3.1.1 Overview

Abstract interpretation provides a good theoretical model in which a large number of program analyses may be performed. Attempting to analyze a program by simulating its execution for all possible inputs is, in general, unsolvable. Abstract interpretation solves this problem by looking for (over or under) approximate solutions. That is, abstract interpretation will indicate what might happen when the program executes or what is guaranteed not to happen. The closeness of the approximation to reality is referred to as its *precision*.

Abstract interpretation replaces the standard semantics of the program being executed with a "collecting semantics" [18]. This semantics is used to "collect" information about program states reachable during the execution of the program, by replacing concrete data values with abstract descriptions.

If E is the powerset of a set of data objects (the concrete domain) in a program, P, and D is a partially ordered set[1] of descriptions (also called the abstract domain), an abstract interpretation is defined formally by four functions: $E_P : E \rightarrow E$, $D_P : D \rightarrow D$, α: $E \rightarrow D$, and $\gamma : D \rightarrow E$. These functions must meet the conditions given in Table 8.

---

1. The ordering of descriptions reflects approximation. In other words, one description approximates another if it describes a larger set of concrete values.

E_P describes a single step in the execution of the program P. D_P describes abstract execution in the domain of the descriptions. α describes a mapping from the concrete domain (E) to the abstract domain (D).[2] γ describes a mapping back from the abstract domain to the concrete domain. Figure 16 illustrates these functions.

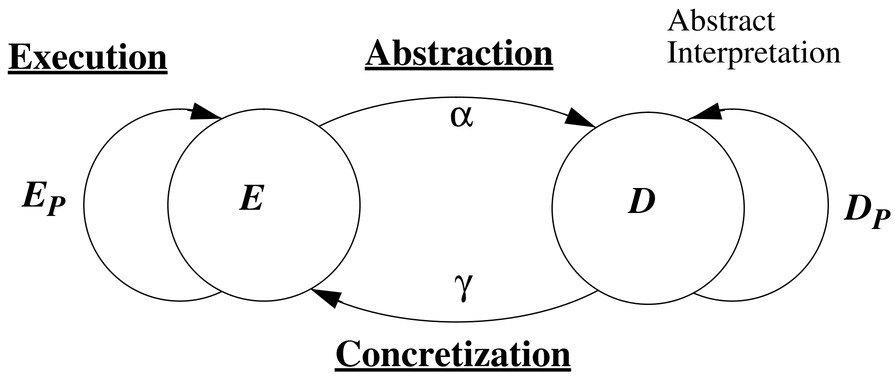


Figure 16: Abstract Interpretation Functions

The conditions given in Table 7 ensure that the interpretation is accurate and complete. Figure 17 illustrates how the last condition ensures that abstract interpretation safely mimics concrete interpretation, that is, the results from abstract interpretation ($\gamma(D_P(\text{d}))$) will include those from concrete interpretation ($E_P(\gamma(\text{d}))$), but may be larger. Table 7 provides some intuition about these conditions.

3.1.2 Example of Abstract Interpretation

The domain of signs gives a simple example of abstract interpretation [18]. The data objects are real numbers. For descriptions, we will use the lattice shown in Figure 18, which gives all possible states of sets of real numbers, based on their signs. A lattice value appearing below another value, connected by a line, indicate two values related by the ordering relation ($\sqsubseteq$). Table 8 gives examples for the six conditions in the domain of signs.

2. α is not required in order to perform abstract interpretation. It is only needed in order to prove the correctness of an abstract interpretation.

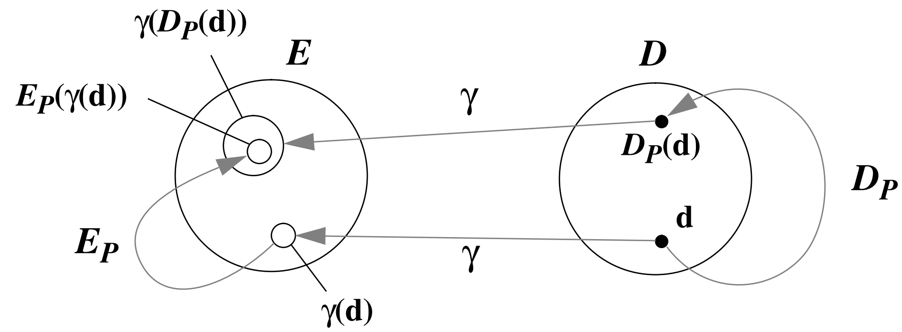


Figure 17: Abstract Interpretation Mimics Concrete Interpretation

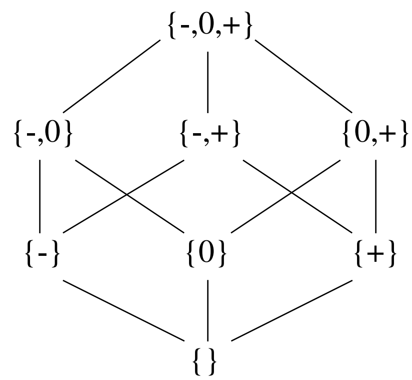


Figure 18: Abstract Domain Lattice for Signs

Table 7: Conditions for Abstract Interpretation

Condition:	Intuition:
1) α and γ are monotonic	Ensures order is preserved in conversion (but might be approximate).
2) $\forall\ d \in D : d = \alpha(\gamma(d))$	Concretization followed by abstraction is exact.
3) $\forall\ e \in E : e \subseteq \gamma(\alpha(e))$	Abstraction followed by concretization is accurate, but approximate.
4) $\forall\ d \in D :$ $E_P(\gamma(d)) \subseteq \gamma(D_P(d))$	Abstract interpretation safely mimics concrete interpretation (but might be approximate).

Table 8: Examples from the Domain of Signs

Function/Condition:	Example:
$E_P : E \rightarrow E$ $D_P : D \rightarrow D$	inc(X) ≡ X + 1 /* inc is an operator in our language */ E_P(inc({0,1,3})) = {1,2,4} D_P (inc({0,+})) = {+}
$\alpha: E \rightarrow D$ $\gamma : D \rightarrow E$	$\alpha(\ \{0,1,3\}\ ) = \{0,+\}$ $\gamma(\ \{0,+\}\ ) = \{x \mid x \geq 0\ \}$
1) α and γ are monotonic	$\{1,3\} \subseteq \{0,1,3\} \rightarrow$ $\alpha(\ \{1,3\}\ ) = \{+\} \sqsubseteq \{0,+\} = \alpha(\ \{0,1,3\}\ )$ $\{+\} \sqsubseteq \{0,+\} \rightarrow$ $\gamma(\ \{+\}\ ) = \{x \mid x > 0\ \} \subseteq \{x \mid x \geq 0\} = \gamma(\ \{0,+\}\ )$
2) $\forall\ d \in D : d = \alpha(\gamma(d))$	$\alpha(\ \gamma(\ \{0,+\}\ )\ ) = \alpha(\ \{x \mid x \geq 0\ \}\ ) = \{0,+\}$
3) $\forall\ e \in E : e \subseteq \gamma(\alpha(e))$	$\gamma(\ \alpha(\ \{1,3\}\ )\ ) = \gamma(\ \{+\}\ ) = \{x \mid x>0\} \supseteq \{1,3\}$
4) $\forall\ d \in D :$ $E_P(\gamma(d)) \subseteq \gamma(D_P(d))$	$E_P(\ inc(\ \gamma(\ \{0,+\}\ )\ )\ ) = E_P(\ inc(\ \{x \mid x \geq 0\}\ )\ ) = \{x \mid x \geq 1\}$ $\cap\vert$ $\gamma(\ D_P\ (\ inc(\ \{0,+\}\ )\ )\ ) = \gamma(\ \{+\}\ ) = \{x \mid x>0\}$

3.2 Abstract Interpretation of Prolog

For Prolog, E_P is typically defined to be the standard top-to-bottom, left-to-right operational semantics of Prolog, based on SLD-resolution [52, 42]. D_P usually represents sets of possible substitutions for the program variables. Large sets of substitutions can be described by logical conditions such as ground(x) (x is instantiated to a ground (fully instantiated) value) or nonvar(x) (x is instantiated to a term, some of whose arguments may be uninstantiated). Figure 19 shows a lattice used when analyzing Prolog programs to determine mode information. It assigns variables to the sets of substitutions described by ground (ground values), nonvar (instantiated values), var (uninstantiated values), or any (all possible values). The set of ground substitutions is a subset of the set of nonvar substitutions. All sets of substitutions are subsets of "any." Bottom ($\perp$) is a special value typically found in all lattices used for analysis. It indicates the empty state or empty set of substitutions, i.e., that no possible values are valid for the variable. During analysis, this indicates portions of code that have not yet been analyzed. When the analysis is complete, it indicates portions of code that are unreachable.

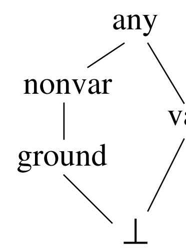


Figure 19: An Example Abstract Domain for Prolog

3.3 Abstract Interpretation Frameworks

Originally, analyzers were built for specific abstract domains [27, 61]. The attraction of abstract interpretation, however, was that it allowed data flow analysis to be parameterized by the abstract domain. This was used in later work to define abstract interpretation frameworks in which a number of analyses could be described [42, 10, 67, 87, 43].

There are three basic pieces that together form an abstract interpretation, as illustrated in Figure 20:

- An abstract domain (for example, the mode domain shown in Figure 19).
- A specific collection of operations defined over the values of the abstract domain.
- An abstract interpreter which implements analysis of a program, but which operates on values in an abstract domain by using a specific collection of operations.

The abstract domain is almost always defined to be a complete lattice[3] (see Appendix C), that is, a set of values, referred to as abstract descriptions, over which a

3. A semi-lattice is sufficient to perform abstract interpretation.

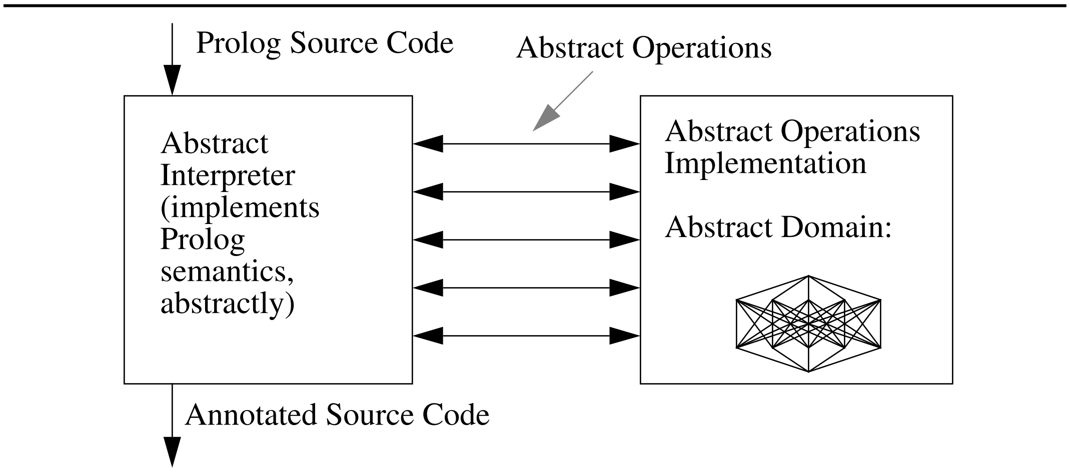


Figure 20: Structure of an Abstract Interpretation

partial ordering ($\sqsubseteq$) and the least upper bound ($\sqcup$) and greatest lower bound ($\sqcap$) of any subset are defined, and which includes top ($\top$) and bottom ($\bot$) elements.

Most abstract interpretation frameworks are defined in terms of the abstract interpreter, the collection of abstract operations that must be provided for an abstract domain, and the properties these operations must exhibit. These frameworks are then shown to be sound (they give correct, if approximate, results) and to terminate (the analysis is guaranteed to finish) if the operations meet the described properties. The following sections describe some of these frameworks, in order to show some of the options available when defining an abstract interpreter.

3.3.1 A Simple Framework

The simplest framework, described by Jacobs [36, 69], requires that only two operations be defined:

$$\overline{init} : Clause \rightarrow Desc$$

$$\overline{pass} : Term \times Desc \times Term \times Desc \rightarrow Desc$$

$\overline{init}$ provides an "initial" description for a clause. This models the concrete operation:

$$init\text{: } Clause \rightarrow \wp(Subst)$$

init returns the set of all possible representations for the identity substitution over the variables in a clause.

$\overline{pass}$ models unification, returning the description obtained from passing information from one term to another. $\overline{pass}$ is defined formally in [69]. $\overline{pass}$ is used both to pass information from a call term into a clause and to return the result after analysis of the clause.

Figure 21 gives the algorithm used in this framework to perform abstract interpretation. The basic idea is to start with the assumption that each clause is called under the most restrictive set of conditions (the bottom element of the abstract domain), analyze from these conditions, propagating changes until we reach the fixpoint (no further changes are propagated). During the analysis, the algorithm records clause input and output conditions in an extension table [29]. When the algorithm terminates, this table covers the conditions that will be encountered at run-time.

As given, the algorithm in Figure 21 is not very efficient. There are a number of things that can be done (and typically are, in an implementation of this framework) to improve the efficiency. First, when choosing unmarked entries, those that lead to the most progress should be selected first (e.g., facts should be chosen over clauses). Second, when unmarking entries, only those affected by a change to the extension table should be unmarked, not the entire table.

Following are some features of this framework:

- It implements top-down (SLD-resolution) semantics, based on substitutions.
- It records, for each clause, multiple call/return conditions for each clause (as opposed to generalizing).
- It doesn't allow an abstract operation implementor to differentiate between clause entry and exit (i.e., $\overline{pass}$ is used in both cases).
- It relaxes the restrictions on the abstract domain, allowing it to be a semi-lattice, instead of a complete lattice.

3.3.2 A Semantics Based Framework

Jones and Sondergaard describe what they refer to as "a semantics-based framework [42]." They begin by defining a "standard" top-down semantics and then factoring it into a "core" semantics and an "interpretation". The "core semantics" corresponds to our abstract interpreter and the "interpretation" to our abstract operations. They use this

proc analyze(A_0 : *Term*, d_0 : *Desc*)
    /*A_0 is the initial term. d_0 is the initial description */
    T := ∅; /* Initially, the extension table is empty */
    executeGoal(d_0, A_0); /* Find out where this atom leads us */
    while T contains unmarked entries /* haven't reached a fixpoint yet */
        /* Find a description, term, and clause # needing analysis */
        choose d_I, A, n where T[d_I, A, n] = (d_O, unmarked);
        C := Clause[Â,n];
        d'_O := executeBody(d_I, C_B); /* Find result of executing that clause */
        T[d_I, Â, n] := (d'_O, marked); /* Add result to extension table */
        if d'_O d_O then /* if the result changed */
            unmark all entries in T; /* Reanalyze everything */

/* Find the result of executing a list of goals */
proc executeBody(d_I : *Desc*, B : *Goal s*) : *Desc*
    d := d_I
    for i = 1 to length(B)
        d := executeGoal(d_I, B_i)
    return d

/* Find the result of executing a single goal */
proc executeGoal(d_I : *Desc*, A : *Term*) : *Desc*
    d_O := ⊥; /* Start with the most restrictive result */
    for n = 1 to NumClauses[Â] /* Analyze for each clause in the called predicate */
        C := Clause[Â, n]; /* Get the next clause */
        d := $\overline{pass}$(C_H, $\overline{init}$(C), A, d_I); /* Find result of entering the clause */
        if fact(C) then /* If there is no body */
            Desc := d; /* Use "d" as the end-of-clause result */
        else if T[d_I, Â, n] is undefined then /* If clause wasn't analyzed yet */
            Desc := ⊥; /* There is no result from this clause (yet) */
            T[d_I, Â, n] := (Desc, unmarked); /* But indicate it needs analyzing */
        else (Desc, Status) := T[d_I, Â, n]; /* Otherwise, use result from table */
        /* Find result of exiting clause and combine with other clause results */
        d_O := d_O ⊔ $\overline{pass}$(A, d_I, C_H, Desc);
    return d_O /* This is the final result from all clauses */

Figure 21: Analysis Algorithm for a Simple Framework

framework to present both the original standard semantics as well as a "collecting" semantics that collects all clause call conditions while executing from some initial query. They then use this framework to present three abstract interpretations: one for collecting groundness information, one for collecting variable aliasing information, and a combination of the two useful in detecting unifications that may need an occur check [70].

Their framework is interesting in that it requires two related domains, as opposed to the single abstract domain required by most others. The two domains, Csub and Asub, abstractly represent substitutions and sets of substitutions, respectively. Jones and Sondergaard then define certain operations (e.g., executing a goal) as starting with a value from Csub (a single substitution) and resulting in a value from Asub (a set of substitutions). This allows us to determine the set of substitutions derivable from the execution of a goal starting from some (single) substitution. This poses a problem, however. The core semantics begins from a single (abstracted) substitution, but after interpreting a single goal, this becomes an (abstracted) set of substitutions. To find the result of analyzing the next goal, it is necessary to interpret the goal starting from all (abstracted) substitutions covered by the current (abstracted) set of substitutions. If Csub = Asub, this is easy, since the two are the same. If Asub = $\wp$(Csub), the interpretation must be done with all subsets of the current (abstracted) set of substitutions, which is feasible if undesirable. Since these are the most common ways in which Csub and Asub are defined, this may not be much of a problem.

The framework requires three operations: *call*, *return*, and *newlog*. The functions *call* and *return* model clause entry and return. The function *newlog* is used to collect clause entry information, in a manner similar to the extension table used in the previous framework.

Although never mentioned, it appears that the framework assumes an extension table approach is being used, as it wouldn't terminate otherwise.

Following are some features of this framework:

- It implements top-down (SLD-resolution) semantics, based on substitutions.
- It can record, for each clause, an abstraction of the set of substitutions with which that clause is called. What (if anything) is actually recorded is up to the abstract operation implementor (as specified in the *newlog* operation).

3.3.3 A Theoretical Framework

Nilsson described a theoretical framework [67] which is very similar to that of Jones and Søndergaard [42]. The main difference is that the proof of correctness is more rigorous. The same groundness example that was used in [42] is used to illustrate this framework.

Following are some features of this framework:

- It implements top-down (SLD-resolution) semantics, based on substitutions.
- It records, for each program point, an abstraction of the set of substitutions that occur during execution at that program point. The program points that are captured are clause entry and exit, and between each goal in a list of goals.
- It requires two abstract operations, *call* and *return*.

3.3.4 A Practical Framework

Bruynooghe described what he referred to as a "Practical framework [9, 10]." This framework differed in three aspects from other frameworks. First, the framework was defined based on the construction of an abstract And/Or tree rather than an extension table of results. This allows different call/return conditions to be recorded based on the caller. Second, the framework is parameterized based on a number of small, easier to understand abstract operations. Last, the framework is described at a level meant for efficient implementation, while being more lax in formal proofs of correctness.

The operations required by this framework are restriction, renaming, initialization, backward unification, upper bound, extension, and abstract interpretation of explicit unification. Bruynooghe claims that this collection of small operations are easier to understand and therefore to correctly implement for a given abstract domain than one or two larger operations. These operations are described, briefly, in the following:

- The restriction operation restricts a description to refer to a subset of variables; this is used to restrict to the variables in a call before calling a predicate or to the head variables after completing a clause.
- The renaming operation changes the variable names within a description; this is used after restriction to rename from the variables in a call to those in the head

of a predicate. The framework assumes that all calls and heads have been unraveled, turning non-variable arguments into explicit unifications.

- The initialization operation extends the result of renaming a description to the head variables of a predicate to include variables in a clause about to be analyzed. This is related to the $\overline{init}$ operation described by Jacobs and Langen [36].
- The backward unification and extension operations, together, are similar to the *return* operation from other frameworks. The backward unification operation computes the result of returning information from the head arguments of a predicate, after it has been analyzed, to the call arguments. The extension operation combines these results with the original description (before the call), to compute the results after the call.
- The upper bound operation is exactly the lub ($\sqcup$) operation on the abstract domain. The framework also assumes that values from the domain can be compared, at least for equality, and possibly for order ($\sqsubseteq$), although these aren't called out explicitly as operations.
- The abstract interpretation of explicit unification operation does exactly what the name implies. This framework assumes that all unification has been broken down into either unification of two variables or unification of a variable with a term, all of whose arguments are variables. This is the first framework which has addressed explicit unification.

Following are some features of this framework:

- It implements top-down (SLD-resolution) semantics, based on substitutions.
- It records, for each program point in an abstract AND/OR execution tree, an abstraction of the set of substitutions at that point. The program points are predicate entry/exit, clause entry/exit, and between each goal in a list of goals.
- It is parameterized by seven primitive operations, which includes an abstraction of explicit unification.

3.3.5 A Generic Interpreter

Le Charlier and Van Hentenryck developed a generic abstract interpretation algorithm very similar to Bruynooghe's [49]. Unlike Bruynooghe's, this algorithm constructs an extension table of results. The required set of operations, however, are almost the same as those described by Janssens and Bruynooghe [9]. The only difference is that the renaming operation has been combined with restriction to form a restriction operation intended to be used specifically when abstractly calling a goal. Other operations have been renamed to make their purpose more explicit. The main goal of this algorithm was to reach a fixpoint (terminate the analysis) with a minimum of effort.

3.4 Summary

In this chapter, we addressed abstract interpretation. We provided an overview of abstract interpretation and described how to apply it to Prolog in general. We then addressed how it can be used specifically for compile-time analysis and optimization of Prolog. We described a number of existing analysis frameworks. In the next chapter we will describe the framework we developed for the analysis and compilation of Prolog.

Chapter 4: An Analysis Framework for Compilation

The frameworks described in the previous chapter show some of the alternatives available in the design of an abstract interpreter. However, none of these frameworks, as given, provide all of the functionality required for an abstract interpreter integrated with a compiler.

In this chapter we describe our analysis framework. This framework was designed to be used as part of a compiler, for both the global analysis performed to obtain information for applying optimizations and the local analysis performed during code generation to maintain information local information about the clauses being compiled. *Using this unified approach we cut the compilation time almost in half.*

4.1 Overview

The following is a list of requirements and goals for our analysis framework:

- As with any other abstract interpreter, the framework must be guaranteed to terminate and provide correct results.
- The framework must provide resulting descriptions for program points which can be used by the compiler for optimization of code. Since most Prolog compilers generate code on a per predicate basis, this means that predicate entry descriptions must be provided. In order to allow dataflow analysis to be repeated locally to a clause during code generation, predicate exit descriptions should also be collected.[1]
- If the compiler is capable of generating code for different versions of a predicate, based on calling conditions, multiple predicate entry/exit descriptions or an abstract And/Or tree must be maintained. Otherwise, it is sufficient to generalize over all call/return conditions.[2]
- The framework must provide for abstract interpretation of all predefined predicates. This includes not only unification, but other built-ins (e.g., is/2,

1. Predicate exit descriptions are used when abstractly interpreting a goal in a list of goals (e.g., within a clause).
2. The Aquarius compiler does not support generation of specialized versions of predicates.

functor/3, etc.) as well. In addition, it should allow user-written predicates to be described.

- The framework should provide the user with the ability to specify starting conditions, i.e., the entry conditions for any predicates which can be called from outside the code being compiled.
- The framework must provide for abstract interpretation of undefined predicates.
- The framework must support abstract interpretation of concrete semantics that operate at a level much lower than that of substitutions. The applicability of compile-time optimizations occasionally depends on low level knowledge of the execution of a program.
- The framework must provide the code generation phase of the compiler with the ability to extract information from abstract descriptions throughout the compilation process. This requires that the code generator either be able to recreate the intermediate analysis results (the results between goals in a clause) or this information be saved during the original analysis.

The design of the analysis framework is somewhat affected by the compiler into which it is integrated, in our case the Aquarius compiler. There are a number of reasons for this:

- Aquarius compiles each predicate independently and does not support specialization of predicate compilation. There is little value, therefore, in saving more than a single pair of entry/exit descriptions per predicate.
- The compiler performs certain source transformations before (e.g., transformation to Kernel Prolog [84] and clause factoring) and after (e.g., determinism extraction) global dataflow analysis. The transformations occurring before the analysis affect the assumptions that can be made by the abstract interpreter about the code it receives. The efficacy of the transformations occurring after the analysis are affected by the results of the analysis. This has a bigger impact on which analyses should be performed, however, than on the design of the abstract interpretation framework.

- The code generator requires specific information in order to apply certain optimizations. To obtain this information, the abstract operations need to be able to abstractly model operations occurring at the level of the generated code (e.g., dereferencing, trailing, and variable initialization deferring).

The analysis framework is written in Prolog, as is the rest of the compiler, making use of Van Roy's Extended Definite Clause Grammer (EDCG) notation to provide a form of global variables, while maintaining an applicative programming style [83]. When describing abstract operations in the following sections, the global information available to that operation is also described.

The first goal in the development of the framework for compilation is to develop a generic abstract interpreter for dataflow analysis. This basic framework is similar to, for example, the generic abstract interpretation algorithm given by Le Charlier and Van Hentenryck [49]. To this, we add the functions needed by the compiler.

4.2 Basic Abstract Domain Operations

The first thing that must be defined is the abstract domain. For the analysis framework, this means defining the order comparison ($\sqsubseteq$) and least upper bound ($\sqcup$) operations

4.2.1 Order Comparison in the Domain

The predicate `less_than/2` implements the order comparison operation for the abstract domain. `less_than( X, Y )` returns successfully if and only X $\sqsubseteq$ Y, that is, if the set of execution states described by the first argument is a subset of the set of execution states described by the second argument. Both arguments are descriptions in the same scope (e.g., over the same set of variables). By definition, the bottom element ($\perp$) is less than all elements of the domain. Figure 22 gives an example definition of this operation for the groundness domain.

4.2.2 Least Upper Bound

The predicate `upper_bound/3` implements the least upper bound operation ($\sqcup$) for two values in the abstract domain. On entry, the first two arguments are bound to elements of the domain, over the same set of variables, and the third argument is unbound. On return, the third argument represents the least upper bound of the first two

```
proc less_than( X : Desc, Y : Desc ) : boolean
    /* Returns true iff X ⊑ Y */
    /* Both X and Y are descriptions of definitely ground variables */
    /* These are represented as sets of variables */
    if X ⊆ Y then
        return true
    else return false
```

Figure 22: Less_than Operation for Groundness

arguments, that is, it is the smallest description (from the abstract domain) which represents the sets of program states represented by the two input descriptions. If either input argument is the bottom element ($\perp$), the result is the value of the other argument. Figure 23 gives an example definition of this operation for the groundness domain.

```
proc upper_bound( X : Desc, Y : Desc ) : Desc
    /* Returns X ⊔ Y */
    /* Both X and Y are descriptions (sets) of definitely ground variables */
    return X ∩ Y
```

Figure 23: Least Upper Bound Operation for Groundness

4.3 Basic Abstract Interpretation Operations

The next collection of operations provide the basis for abstract interpretation of the Prolog code. They define the abstract results of basic steps in the execution of Prolog. As can be seen from the variety apparent in the frameworks described previously, the set of operations specified for this can vary significantly. Bruynooghe, for example, described Prolog execution in terms of six basic operations: restriction, renaming, initialization, backward unification, extension, and abstract interpretation of unification [10]. Bruynooghe claims these operations are simple to understand, and therefore, to define for a given domain. Jacobs and Langen take an opposite approach, defining only two operations: init and pass [36]. Jacobs claims this provides the best opportunity for precision. We have taken a middle ground, attempting to reach a good compromise to the following goals:

- The operations should provide good opportunities for precision (they shouldn't be broken up into too many pieces).
- The operations should be able to capture implementation-dependent information or predicate-level information, not just information based on substitutions.
- The operations should be described in terms of concrete operations, understandable by the implementor.

4.3.1 Abstract Interpretation of Predicate Entry

The predicate `abs_int_entry/4` describes abstract interpretation of predicate entry. This is used to interpret the entry into a predicate from a goal in a clause or from some initial query. From a concrete standpoint, this consists of saving temporary values (variables) in permanent locations (an environment), constructing terms on the heap for arguments, loading these arguments into the argument registers (if they aren't there, already), and either calling the predicate or jumping to it (if last-call optimization applies). Table 9 describes the arguments to this predicate. Figure 24 provides an example definition of this operation for the groundness domain.

<u>proc</u> abs_int_entry(Goal : *Term*, Head : *Term*, Init_Desc : *Desc*) : *Desc*

  Head_Desc := $\emptyset$

  <u>for</u> i = 1 <u>to</u> arity(Goal) <u>do</u>

    <u>if</u> vars($Goal_i$) $\subseteq$ Init_Desc <u>then</u>

      Head_Desc := Head_Desc $\cup$ $Head_i$

  <u>return</u> Head_Desc

Figure 24: Abs_int_entry for Groundness

4.3.2 Abstract Interpretation of Clause Initialization

After entering a predicate, the next execution step involves initialization for each clause in the predicate. From a concrete standpoint, this can consist of performing clause selection either deterministically, through some series of tests, on non-

Table 9: Arguments for abs_int_entry/4

Argument:	Description:
Goal	This is the call goal, as it appeared in the Prolog source. The arguments of the goal may be arbitrary terms (e.g., variables, atoms, structures), unless the goal was unraveled.
Head	This is the head of the predicate about to be entered. The functor and arity of Head will be the same as that of the goal. The arguments will be distinct variables, representing the predicate arguments. This is done by "unraveling" the heads of all clauses and placing head unification into the body of each clause.
Init_Desc	This is the description valid before executing the goal. It will describe the possible execution states when that goal is reached, in terms of the scope in which the goal appears (e.g., it will address the variables in the clause containing the goal). This should never be the bottom element, as, in that case, the abstract interpreter would not continue analysis of the body.
Head_Desc	On return, this will be the description valid after entry into the predicate (but before actual entry into any particular clause). This is typically computed by restricting Init_Desc to only those variables appearing in Goal and propagating the information into the head variables.

deterministically, by creating or updating a choicepoint, and creating an environment. This is handled by the predicate `initialize/3`. This is called for each clause in the predicate. Table 10 describes the arguments to `initialize/3`. For the groundness domain, nothing needs to be done for clause initialization. Typically, however, this operation modifies the description to indicate that the variables appearing in the clause body, but not the head, are unbound and unaliased.

4.3.3 Abstract Interpretation of Clause Termination

Once all the goals in a clause body have been interpreted, the clause is terminated. This is not the same as returning from the predicate (predicate exit), which will be discussed later. Instead, it merely provides a hook to provide any desired processing at the end of each clause. For example, this may consist of restricting the description to only cover those variables appearing in the head or reflect the removal of an environment. The description valid after successful completion of a predicate is

Table 10: Arguments for initialize/3

Argument	Description
Clause	This is the clause being initialized, provided in normal form: Head :- Goal1, Goal2, ..., true.
Head_Desc	This is the description valid after entry into the predicate. This is the description that was returned from `abs_int_entry/4`. It is possible (but unlikely) that this will be the bottom element.
First_Desc	On return, this will be the description valid after initialization of the clause.

computed by taking the least upper bound of the descriptions valid after terminating each clause; this is the description saved for the predicate. In order to keep the saved information small and to simplify the lub calculation, it is worthwhile to remove unnecessary information from the descriptions at the end of the clauses. The arguments to `terminate/3` are described in Table 11. An example of this operation for the groundness domain is provided in Figure 25.

<u>proc</u> terminate(Clause : *Clause*, Last_Desc : *Desc*) : *Desc*

<u>return</u> Last_Desc $\cap$ Clause_H

Figure 25: Terminate/3 for Groundness

Table 11: Arguments for terminate/3

Argument	Description
Clause	This is the clause being terminated, provided in normal form: Head :- Goal1, Goal2, ..., true. This is the same as what was provided to initialize/3.
Last_Desc	This is the description valid after interpretation of the last goal in the clause body.
Tail_Desc	On return, this will be the description valid after terminating the clause.

4.3.4 Abstract Interpretation of Predicate Exit

The final step in execution of a goal is to return from the called predicate to the calling goal. This is provided by the predicate `abs_int_exit/5`. The arguments of this predicate are described in Table 12. This operation doesn't reflect much actual code in an implementation. From an abstract standpoint, however, this operation will typically involve renaming information about the variables in the head of the predicate to those appearing in the arguments of the calling goal and extending this information to other possibly affected variables in the scope in which the goal appears. An example of this operation is provided in Figure 26 for the groundness domain.

<u>proc</u> abs_int_exit(Head : *Term*, Goal : *Term*, Init_Desc : *Desc*,
    Tail_Desc : *Desc*) : *Desc*
  Next_Desc := Init_Desc
  <u>for</u> i = 1 <u>to</u> arity(Goal) <u>do</u>
    <u>if</u> $Head_i \in$ Tail_Desc <u>then</u>
      Next_Desc := Next_Desc $\cup$ vars($Goal_i$)
  <u>return</u> Next_Desc

Figure 26: Abs_int_exit for Groundness

Table 12: Arguments for abs_int_exit/5

Argument	Description
Head	This is the head of the predicate from which we are returning. It is the same term that was passed to abs_int_entry/4.
Goal	This is the calling goal, as it appeared in the Prolog source. It is the same term that was passed to abs_int_entry/4.
Init_Desc	This is the description valid before executing the goal. It describes the possible execution states when that goal was reached, in terms of the scope in which the goal appears (e.g., it will address the variables in the clause body containing the goal). This is the same value that was passed to abs_int_entry/4.

Table 12: Arguments for abs_int_exit/5

Argument	Description
Tail_Desc	This is the description valid after interpretation of the predicate. This is computed as the least upper bound of the descriptions obtained for each clause by calling terminate/3. In other words, the set of execution states reachable at the end of the predicate is the set of states that can be reached by at least one clause.
Next_Desc	On return, this will be the description valid after returning from the predicate to the body.

4.4 Advanced Abstract Interpretation Operations

The operations in the previous section allow the framework to analyze Prolog code provided by the user. To fully analyze a program, however, the framework needs to be able to deal with predicates not defined by the user. Although a few frameworks have addressed analysis of explicit unification, by and large, most have ignored treatment of predicates not appearing in the code being analyzed. In order to provide a framework complete enough for compiling realistic benchmarks, this is an issue that must be addressed. These predicates fall into three categories: built-in predicates, known by the analyzer; user-written predicates, defined elsewhere but described to the analyzer; and completely unknown predicates. The operations needed to deal with these cases are defined in the following sections.

4.4.1 Abstract Interpretation of Predefined Predicates

The Aquarius Prolog system defines a large collection of built-in predicates. In addition to these, the user is allowed to describe predicates defined in source files other than the one being compiled.

Externally-defined user predicates are described by providing Aquarius mode declarations [84]. These declarations describe the expected entry conditions for the predicate and the conditions that will be true when the predicate completes. These mode declarations are used to construct two descriptions: a head description (similar to those returned by abs_int_entry) and a tail description (similar to those given to abs_int_exit). The head description is used to ensure the call to the predicate is what was expected when it was compiled. The tail description is used to propagate the results of executing the predicate back to the caller. Figure 27 shows the algorithm used to analyze externally-defined predicates.

```
proc analyzeExternal(
        Goal : Goal, /* This is the call goal */
        Head : Goal, /* This is a canonical head for the predicate */
        Init_Desc : Desc, /* This is the description before the call */
        Head_Desc : Desc, /* This is the expected entry description */
        Tail_Desc : Desc ) /* This describes the result of the predicate */
        : Desc /* This procedure returns the result of execution the predicate */
    Desc := abs_int_entry( Goal, Head, Init_Desc )
    if not ( Desc ⊑ Head_Desc ) then
        Complain that predicate is not being called as expected
    return abs_int_exit( Head, Goal, Init_Desc, Tail_Desc )
```

Figure 27: Analysis of Externally Defined Predicates

In addition to user-defined predicates, this approach is sufficient for most built-in predicates. There are a few predicates which require special treatment in order to get reasonable results. Unification is a prime example of this. Other examples are special built-ins, introduced during source transformations (e.g., transformation to Kernel Prolog). These built-ins and others requiring special handling are described in Table 13. The predicate `abs_int_builtin/3` is used to analyze built-in predicates. Its arguments are defined in Table 14. If it fails, the head and tail descriptions are looked up in a table derived from both user declarations and built-in declarations, given by an abstract domain implementor, and the approach described in Figure 27 is used to determine the results of executing the built-in.

4.4.2 Abstract Interpretation of Unknown Predicates

Hopefully, all predicates appearing in the program are either defined, and can be analyzed through abstract interpretation, or are described, as shown above. There needs

Table 13: Built-in Predicates Requiring Special Treatment

Built-in:	Purpose/Reason:
'$cut_load'(X) '$cut_shallow'(X) '$cut_deep'(X)	These special built-ins implement the cut/1 built-in. '$cut_load' loads the current choicepoint pointer (r(b)) into a variable (X). The other two built-ins cut the choicepoint stack back to the pointer in the variable, X.
'$add'(A,B,C), '$sub(A,B,C), '$mul'(A,B,C), '$fdiv'(A,B,C), '$idiv'(A,B,C), '$mod'(A,B,C), '$and'(A,B,C), '$or'(A,B,C), '$xor'(A,B,C), '$sll'(A,B,C), '$sra'(A,B,C), '$not'(A,B), '$if2f'(A,B), '$if2i'(A,B)	These special built-ins implement the is/2 built-in. The result is returned in the last argument, which is assumed to be an uninitialized register argument, i.e., the result is returned in a register, not in a memory location passed to the built-in. The type of the result can depend on the types of the inputs.
'$name_arity'(X,N,A), '$atom_nonnil'(X)	These implement specialized functor tests and are generated, after dataflow analysis, as part of determinism extraction. N is always atomic and A is an integer. '$name_arity'/3 is similar to functor/3, except it only tests, never binds. '$atom_nonnil' tests to see if its argument is an atom other than [].
'$uni_args'(X,T), '$equal'(X,Y)	These are specialized versions of unification, generated during determinism extraction. '$uni_args' unifies the variable X with the term T, assuming that X is a bound variable whose tag matches that of the term T. '$equal' compares either two variables, one of which is atomic, or a variable with an atomic value; in either case, this is a single word comparison.
'$deref'(X)	This is used during code generation to explicitly dereference its argument.

to be a mechanism for interpreting predicates when this is not the case.. This is provided by the predicate `abs_int_unknown/3`. The arguments to this predicate are the same as to abs_int_builtin/3.

This operation must make a worst case assumption about the operation of the unknown predicate. That is, given the possible set of initial execution states (as described by the call description), it must compute the element of the abstract domain

Table 14: Arguments for abs_int_builtin/3

Argument	Description
Call	This is the goal used to call the predefined predicate. With the exception of unification, the arguments can be arbitrary terms. For unification, the first argument will always be a variable. If unifications are unraveled, the second argument will be either a variable or a term with variable arguments.
Call_Desc	This is the description valid before executing the goal. It will describe the possible execution states when that goal was reached, in terms of the scope in which the goal appears (e.g., it will address the variables in the clause body containing the goal).
Succ_Desc	On return, this will be the description valid after executing the goal.

which covers the smallest set of execution states which includes all possible execution states resulting from calling an unknown predicate. This may consist of indicating all previously unbound variables are now possibly bound and possibly aliased. Some domains allow more precise treatment of this case by making use of the information known when the predicate is called. For example, if it is known that all the arguments are ground on entry, the predicate cannot further affect the bindings of any variable appearing in the scope of the call. It might further restrict the values (e.g., through tests), but we must assume the worst case.

4.5 Implementing the Framework

The second major class of operations deals with the implementation of the framework and providing hooks to optimize the analysis. Default definitions can be provided for these, thereby simplifying the task of implementing an abstract interpretation.

4.5.1 Abstract Interpretation Initialization and Termination

Typically, there is no special (domain-dependent) processing that must be performed before beginning or after completing the analysis. It is possible that this may be needed, however. For example, Janssens' implementation of type analysis stored type graph nodes in the Prolog database [40, 41]. The implementation required that certain entries be placed in the database initially, and that the database be cleaned up afterwards. To support this programming style, two hooks are provided in the framework: `init_absdom/0`, which is called before the analysis, and `term_absdom/0`, which is called after the analysis.

If this feature is not needed, this predicates should be defined as simple facts.

4.5.2 Displaying Domain Values

When debugging an abstract interpretation, it is useful to see the descriptions during the analysis and after the fixpoint has been reached. Rather than simply writing out the internal representation of the domain elements, the framework provides a hook for printing a description, `write_desc/3`. This predicate should generate an understandable representation of a description on the current output stream as a sequence of one or more lines. Table 15 describes the arguments to this predicate.

Table 15: Arguments for write_desc/3

Table Entry	Description
Desc	This is the description to be written out.
Clause_Vars	The description is usually given relative to the scope of some clause. Clause_Vars is a list of the variables appearing in that clause, starting with the head variables. This can be used to enumerate or name the variables appearing in the description. The variables will always be given in the same order for any particular clause.
Prefix	This is an atom that should be written out at the beginning of each line written out. This is used to ensure these lines are treated as comments, since they are written to the same file as the generated code.

4.5.3 Completely Static Analysis

Since any abstract interpreter must iterate to a fixpoint, it may perform certain operations on the same values a number of times. Typically, these are operations performed relative to a predicate or the clauses in a predicate. Examples of this are obtaining the set of variables in a predicate's head, obtaining the set of variables appearing only in the body of a clause, or obtaining the set of variables appearing multiple times in a clause body.

In order to keep from having to do these operations multiple times, the framework provides hooks to collect this type of information one time and pass it back to the abstract operations which can make use of it.[3] There are three predicates which must be specified to support this: `prepare_head/2`, `prepare_clause/5`, and `prepare_tail/3`, with arguments as described in Tables 16, 17, and 18, respectively. One value (which can be a structure) is gathered for each predicate and one for each clause in each predicate. The algorithm in Figure 28 shows how these values are collected for a predicate.

<u>proc</u> prepare(Pred : *Pred*, Pred_Info : *Term*, Clause_Info : <u>array</u> <u>of</u> *Term*)

  prepare_head(PredH, Pred_Info)

  <u>for</u> i := 1 <u>to</u> NumClauses(Pred) <u>do</u>

    prepare_clause($Pred_H$, $Pred_i$, Clause_Info[i], Pred_Info, Pred_Info')

    Pred_Info := Pred_Info'

  prepare_tail($Pred_H$, Pred_Info, Pred_Info')

  Pred_Info := Pred_Info'

  <u>return</u>

Figure 28: Precomputing Static Information

Table 16: Arguments for prepare_head/2

Argument:	Description:
Head	The canonical head of the predicate. This is the same term passed to abs_int_entry and abs_int_exit.
Pred_Info	On return, this is a term containing information derived from the predicate head.

3. This is very similar to "memoization [29]."

Table 17: Arguments for prepare_clause/5

Argument:	Description:
Head	The canonical head of the predicate. This is the same term passed to abs_int_entry and abs_int_exit.
Body	The body of the clause, as a conjunction of goals, terminated with the atom `true`. This is the same body passed to initialize and terminate.
Clause_Info	On return, this is a term containing information from the clause. This information is available for use in initialize, terminate, abs_int_builtin, and abs_int_unknown.
Pred_Info	This is a term containing information from the predicate head, as well as all clauses prepared so far.
Pred_Info'	On return, this is an updated copy of Pred_Info, including information from this clause.

Table 18: Arguments for prepare_tail/3

Argument:	Description:
Head	The canonical head of the predicate. This is the same term passed to abs_int_entry and abs_int_exit.
Pred_Info	This is a term containing information from the predicate head, as well as all of the clauses.
Pred_Info'	On return, this is an updated copy of Pred_Info. This information is available for use in abs_int_entry, initialize, terminate, and abs_int_exit. This provides a last opportunity to update the predicate information.

4.6 Integration into a Prolog Compiler

The final class of operations provides the interface to the compiler. These operations provide support for user-specified starting points for analysis and allow the code generator to extract information from the analysis results.

4.6.1 Interpreting Mode Formulas

In order to construct the tables described previously for built-in and user-described predicates, it is necessary to convert Aquarius mode formulas [84] into head and tail descriptions. This is done by the predicate `formulas_to_descs/6`, whose arguments are described in Table 19. The information for built-in predicates is provided by an abstract domain implementor, whereas for user-described predicates it comes from mode declarations in the source code of the program being compiled.

This operation is also used to determine starting points for analysis. Aquarius entry declarations, which look very much like mode declarations, are used to determine the initial set of predicates for analysis and the description to use when starting the analysis.

Table 19: Arguments for formulas_to_descs/6

Argument:	Description:
Head	The canonical head of the predicate. This is the same term passed to abs_int_entry and abs_int_exit.
Head_Formula	This is an Aquarius mode formula describing the conditions under which this predicate will be called. The description is in terms of the variables in the head.
Tail_Formula	This is an Aquarius mode formula describing the conditions that will be true when this predicate completes. The description is in terms of the variables in the head.
Bound_Vars	This is the set of head variables which are possibly bound by a call to this predicate. This is extra information available for built-in predicates which can be used or ignored, depending on whether or not is has any meaning in the description.
Head_Desc	This is the description expected to be valid after entering the predicate.
Tail_Desc	This is the description valid after the predicate completes.

4.6.2 Extracting Information From Descriptions

During code generation, it is necessary to obtain information from abstract descriptions which can be used to optimize the generated code. To begin, the code generator has the head and tail descriptions for each predicate, as determined by analysis. When compiling a predicate, it alternates between generating code for clauses and goals, and propagating the descriptions through the code being compiled, in essence, redoing the last analysis pass (the one in which we had determined we had reached a fixpoint). Although this increases the compilation time slightly, it permits the analyzer to save descriptions only at predicate head and tail points, and not all program points. This reanalysis is performed in basically the same manner as it was the first time. The code generator requires a mechanism to extract information from the descriptions that are obtained by this analysis. This mechanism is provided by the predicate `desc_implies/2`, whose arguments are described in Table 20. This predicate succeeds if and only if a specified condition holds for the given description. The set of conditions which may be tested by the compiler is very dependent on the code generation approach. Table 21 lists conditions that are currently tested. As further optimizations are added to the compiler, this list may grow.

Table 20: Arguments for desc_implies/2

Argument:	Description:
Condition	The condition for which to test. Table 21 lists conditions used currently in the compiler.
Desc	An abstract description.

Table 21: Conditions tested by desc_implies/2

Condition(s):	Description:
var(X), nonvar(X), ground(X)	These conditions test the mode of some variable, X. Basically, they test if the mode testing built-in of the same name will always be true for the given description.
integer(X), float(X), number(X), atom(X), atomic(X), simple(X), cons(X), structure(X), rlist(X), '$name_arity'(X,N,A), X\==A	These conditions test the type of some variable, X. Basically, they test if the type testing built-in of the same name will always be true, for the given description. '$name_arity'(X,N,A) tests if the variable X is bound to a term with the functor N/A. X\==A tests to see if X is definitely not bound to the atom A. rlist(X) tests if X is a recursive list, terminated by nil ([]).

Table 21: Conditions tested by desc_implies/2

Condition(s):	Description:
deref(X), rderef(X)	These conditions test for the presence of reference chains in the memory representation of variable X. deref(X) is true if no dereferencing is required to access X. rderef(X) is true if X is dereferenced and the arguments of X, if it is compound, are also recursively dereferenced.
new(X), uninit(X), uninit_reg(X)	These test for very specific and useful conditions for which the Aquarius compiler can generate optimized code. new(X) means the variable X appears only in the body of a clause and hasn't appeared up to this point in the clause; this allows the allocation and initialization of the variable to be deferred until its first occurrence. uninit(X) is true for a variable as long as its initialization can be deferred (but it might be allocated in memory). uninit_reg(X) is true at the end of a predicate for an argument variable if the value can be returned in the argument register (i.e., it isn't bound until the end of each clause).

4.7 The Analysis Algorithm

This section presents the dataflow analysis algorithm, which makes use of the operations described in the previous sections to provide a generic abstract interpreter similar to that given by Jacobs [38] and Pabst [69]. More attention has been given to the use of this algorithm as part of a Prolog compiler, however. We don't claim that this is the most efficient algorithm. The techniques employed by Le Charlier and Van Hentenryck [51] and Tan and Lin [77] could be used to improve the performance of this algorithm. Global information used during the analysis is described in Table 22. The analysis algorithm is given in Figure 29.

```
proc analyze_program( Preds : set of Pred )
    init_external_preds()
    init_internal_preds( Preds, Entry_Clauses )
    init_absdom()
    analyze_closure( Entry_Clauses )
    term_absdom()
    return

proc init_external_preds()
    for each built-in declaration, (Head, Head_Form, Tail_Form, Bound_Vars), do
        formulas_to_descs( Head, Head_Form, Tail_Form, Bound_Vars,
                           Head_Desc, Tail_Desc )
        prepare_head( Head, Pred_Info )
        External[functor(Head)] :=
            (Head, Head_Desc, Tail_Desc, Pred_Info )
    for each user mode declaration, ( Head, Head_Form, Tail_Form ), do
        formulas_to_descs( Head, Head_Form, Tail_Form, ∅,
                           Head_Desc, Tail_Desc )
        prepare_head( Head, Pred_Info )
        External[functor(Head)] :=
            (Head, Head_Desc, Tail_Desc, Pred_Info )
    return
```

Figure 29: Generic Abstract Interpretation Algorithm

```
proc init_internal_preds( Preds : set of Pred, Entry_Clauses : set of ClauseID )
    Entry_Clauses := ∅
    for each Pred ∈ Preds do
        F := functor( Pred_H )
        Head[F] := Pred_H;  Body[F] := Pred_B;  Tail_Desc[F] := ⊥
        prepare_head( Pred_H, Pred_Info )
        for i := 1 to num_clauses( Pred_B ) do
            prepare_clause( Pred_H, Pred_i, Clause_Info[F,i], Pred_Info, Pred_Info )
            for j := 1 to num_goals( Pred_i ) do
                if Pred_{i,j} is a call to a predicate in Preds then
                    Callers[functor(Pred_{i,j})] := Callers[functor(Pred_{i,j})] ∪ {(F,i)}
        prepare_tail( Pred_H, Pred_Info, Pred_Info[F] )
        if PredH has an entry declaration, Entry_Form, then
            formulas_to_descs( Pred_H, Entry_Form, true, ∅, Head_Desc[F], _ )
            Reached[F] := true
            Entry_Clauses := Entry_Clauses ∪ {(F,i) | 1 ≤ i ≤ num_clauses(PredB)}
        else if arity(Pred_H) = 0 then
            formulas_to_descs( Pred_H, true, true, ∅, Head_Desc[F], _ )
            Reached[F] := true
            Entry_Clauses := Entry_Clauses ∪ {(F,i) | 1 ≤ i ≤ num_clauses(PredB)}
        else
            Reached[F] := false
            Head_Desc[F] := ⊥
    return
```

Figure 29: Generic Abstract Interpretation Algorithm (contd.)

```
proc analyze_closure( Clauses : set of ClauseID )
    while Clauses  ∅ do
        Changed_Preds := ∅
        for each F, Cs where Cs = {i | (F,i) ∈ Clauses } do
            Pred_Info := Pred_Info[F]
            analyze_disj( Cs, F )
        Clauses := ∅
        for each F ∈ Changed_Preds do
            for each (P,C) ∈ Callers[F] do
                if Reached[P] then
                    Clauses := Clauses ∪ {(P,C)}
    return

proc analyze_disj( Cs, F )
    Tail_Desc := ⊥
    for each i ∈ Cs do
        Clause_Info := Clause_Info[F,i]
        analyze_conj( Head[F], Body[F]_i, Head_Desc[F], Desc )
        Tail_Desc := upper_bound( Tail_Desc, Desc )
    if not less_than( Tail_Desc, Tail_Desc[F] ) then
        Tail_Desc[F] := upper_bound( Tail_Desc, Tail_Desc[F] )
        Changed_Preds := Changed_Preds ∪ {F}
    return
```

Figure 29: Generic Abstract Interpretation Algorithm (contd.)

```
proc analyze_conj( Head, Body, Head_Desc, Tail_Desc )
    initialize( Head, Body, Head_Desc, Desc )
    for i := 1 to num_goals( Body ) do
        analyze_goal( Body_i, Desc, Desc )
        exit loop if Desc = ⊥
    terminate( Head, Body, Desc, Tail_Desc )
    return

proc analyze_goal( Goal, Init_Desc, Next_Desc )
    G := functor(Goal)
    if G = '='/2 or External[G] is defined then
        Next_Desc := abs_int_builtin( Goal, Init_Desc )
    else if Head[G] is defined (G is an internally defined predicate) then
        Head_Desc := abs_int_entry( Goal, Head[G], Init_Desc )
        if not Reached[G] or not less_than( Head_Desc, Head_Desc[G] ) then
            Head_Desc[G] := upper_bound( Head_Desc, Head_Desc[G] )
            Reached[G] := true
            analyze_disj( {i | 1 ≤ i ≤ num_clauses(G)}, G )
        Next_Desc := abs_int_exit( Head[G], Goal, Init_Desc, Tail_Desc[G] )
    else
        Next_Desc := abs_int_unknown( Goal, Init_Desc )
    return
```

Figure 29: Generic Abstract Interpretation Algorithm (contd.)

Table 22: Global Information in the Analysis Algorithm

Name/Type	Description
External: array [*Functor*] of structure Head: *Goal* Head_Desc: *Desc* Tail_Desc: *Desc* Pred_Info: *Term*	This array contains information describing each external predicate. The information maintained is the canonical head (Head), the head and tail descriptions (Head_Desc and Tail_Desc) and the static analysis information (Pred_Info).
Head: array [*Functor*] of *Goal*	This array contains the canonical heads of internally defined predicates.
Body: array [*Functor*,*integer*] of Goals	This array contains the clause bodies for internally defined predicates. Body[F] refers to all clauses for a predicate. Body[F,i] refers to the *ith* clause.
Reached: array [*Functor*] of *boolean*	This array indicates if a predicate has been "reached" in the analysis. Initially, all entry points have been reached. During analysis, any predicate encountered during the analysis of a "reached" predicate is considered to have been "reached".
Head_Desc: array [*Functor*] of *Desc*	This array contains the current head (entry) description for each internally defined predicate. Initially, all head descriptions are set to bottom ($\perp$) except for the entry points, which are set based on user-given declarations.
Tail_Desc: array [*Functor*] of *Desc*	This array contains the current tail (exit) description for each internally defined predicate. Initially, all elements are set to bottom ($\perp$).
Pred_Info: array [*Functor*] of *Term*	This array contains the static analysis information (from prepare_head et al.) for each internally defined predicate.
Clause_Info: array [*Functor*,*integer*] of *Term*	This array contains the static analysis information (from prepare_clause) for each clause of each internally defined predicate.
Callers: array [*Functor*] of set of *ClauseID*	This array indicates the set of clauses that call each internally defined predicate. This is used to indicate which clauses need reanalysis when a predicate's exit description changes.

Table 22: Global Information in the Analysis Algorithm

Name/Type	Description
Pred_Info	This is the static analysis information for the predicate currently being analyzed. It is available for use in the abstract operations.
Clause_Info	This is the static analysis information for the clause currently being analyzed. It is available for use in the abstract operation.

Chapter 5: Prolog Analysis

Numerous abstract domains have been proposed to capture various properties [9, 27, 36, 40, 48, 57, 61, 78, 84]. We have constructed a taxonomy, shown in Figure 30, to organize these properties. The properties have been grouped into three broad categories: implementation-independent variable analyses, implementation-dependent variable analyses, and predicate-level analyses. These categories reflect the source of information being collected in a given dataflow analysis.

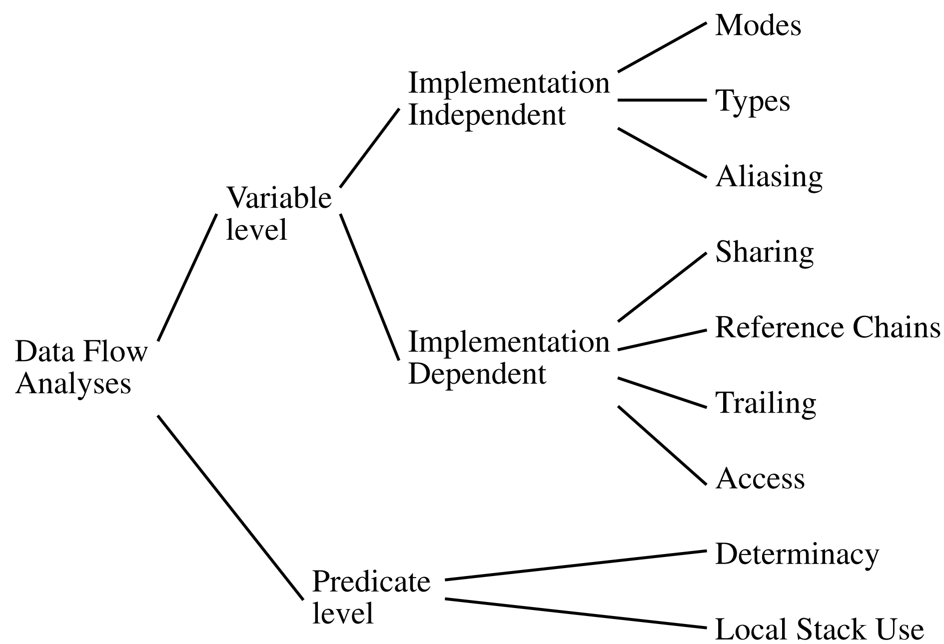


Figure 30: Taxonomy of Data Flow Analyses

The following sections describe increasingly precise domains for capturing these properties. After describing domains for some collection of properties, we give a number of observations, which are backed up by the results in Chapter 6.

5.1 Implementation-Independent Variable Analyses

The dataflow analyses described in this section collect information about the states that variables take on during execution which are independent of a given Prolog implementation. In other words, these analyses capture abstract properties describing the possible substitutions that occur during execution in the standard operational (SLD-resolution) semantics [42, 52].

5.1.1 Mode Analysis

Mode analysis attempts to determine the degree of instantiation of each variable. This is the most often described form of analysis [8, 10, 27, 49, 54, 59, 61, 67, 69, 78, 84], probably because it is one of the easiest to understand and provides very useful information. As we saw previously, many optimizations depend on knowing the modes of variables. Most optimizations depend only on knowing if a variable is bound or unbound. A few depend on knowing that a variable is ground.

In each abstract domain for mode analysis, an abstract description consists of a mapping from variables to modes; the only difference between these domains is the set of modes each one supports. Table 23 defines the various modes used in the abstract domains. At any point in the execution of a program, the state of each variable (the set of values the variable may take on) is described by an entry in this table. Figure 31 presents the abstract domains for mode analysis by showing the lattice in each of the domains for one variable. The lattice shows the set of modes provided in a given domain, and the relationship between modes; a mode appearing above another mode, connected by solid lines, describes a larger set of concrete values

Figure 32 places the domains, themselves, in a lattice, showing how one domain approximates another [17, 16]; the higher entries on this lattice provide more detailed information. The top element is the concrete domain. The bottom element is the domain of no information (in essence, the result of performing no analysis).

Mode Domains M_1 and M_2

Abstract domains M_1 and M_2 are the simplest of mode domains, capturing the set of variables that are definitely a given mode at some program point [59, 67, 69]. M_1 captures definitely ground variables; this is the groundness domain that was used as an example in the previous chapter. M_2 captures definitely unbound variables.

These domains are good examples of domains capturing some boolean property for variables. There are two types of these domains: those that capture variables that *definitely* have a property (such as these), and those that capture variables that *possibly*

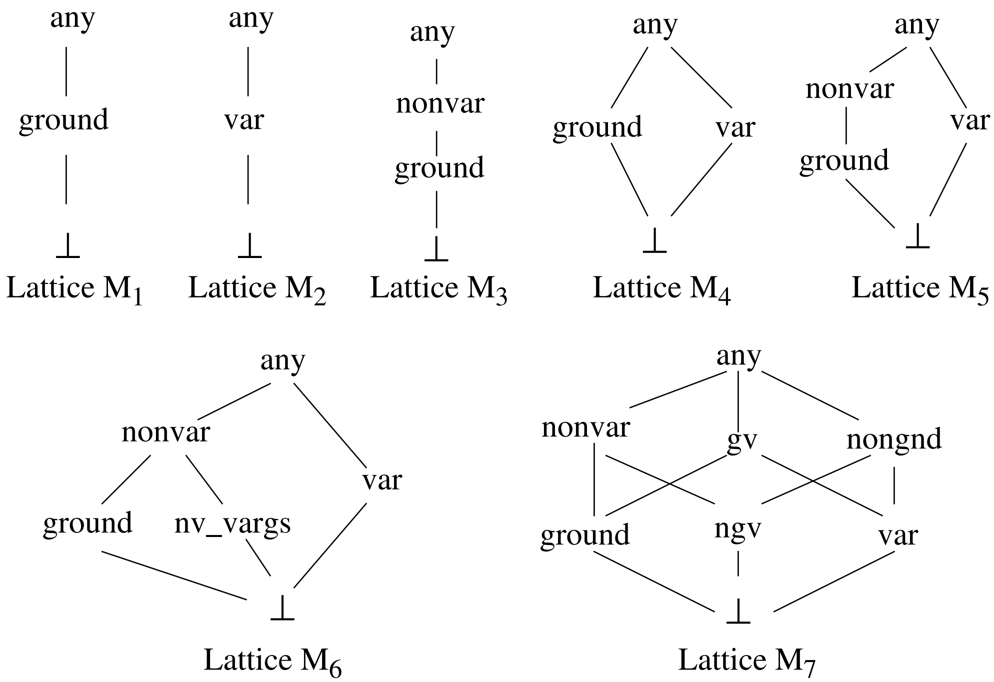


Figure 31: Lattices for Mode Abstract Domains

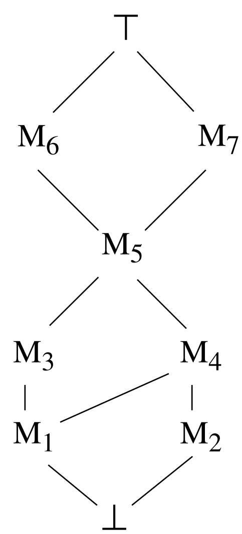


Figure 32: Lattice of Mode Domains

have a property. In either case, these domains can be represented efficiently as a set of

Table 23: Mode Definitions for a variable, X

Mode	Description	Concretization: $\gamma(X)$
any	X could be anything.	Term
ground	X is ground.	$\{ T \mid var(T) = \emptyset \}$
nonvar	X is not unbound.	$\{ T \mid T \notin Var \}$
var	X is unbound.	Var
nv_vargs	X is a compound term with all arguments unbound.	$\{ f(t_1, ..., t_n) \mid n > 0 \wedge t_i \in Var, 0 < i \leq n \}$
gv	X is either ground or unbound.	$\{ T \mid var(T) = \emptyset \vee T \in Var \}$
nongnd	X is not ground.	$\{ T \mid var(T) \neq \emptyset \}$
ngv	X is neither ground nor unbound.	$\{ T \mid var(T) \neq \emptyset \wedge T \notin Var \}$

variables; i.e., the domain is the powerset of the variables and a domain element is a subset of the set of variables. For *definite* domains, the ordering relation is superset. An element indicating that more variables definitely have a property is more restrictive than one in which fewer variables have that property; the empty set is the top element. For *possible* domains, the ordering relation is subset and the set of all variables is the top element.

Mode Domains M_3 through M_6

Descriptions in the next four mode domains consist of mappings from the set of variables to modes, lifted pointwise from the ordering of the modes. The set of modes, however, varies from one domain to the next.

Since it is more important to know whether or not a variable is bound than to know if it is ground, the next domains add this type of information. Domain M_3 extends domain M_1 with the state where a variable is known to be bound (nonvar), but not necessarily ground. Domain M_4 combines the information from domains M_1 and M_2 [8, 10, 54, 27]. Domain M_5 adds all of these states [84]. Domain M_6 extends domain M_5 even further by adding the state where a variable is bound to a compound term, but all of the arguments are unbound [61].

Mode Domain M_7

Descriptions in mode domain M_7 are similar to those in the previous domains, in that they are mappings from variables to modes. Rather than being a collection of modes chosen for some unknown reason, however, a more methodical approach was taken [49]. This domain begins by partitioning all possible values into a number of mutually exclusive, all covering sets: ground, unbound (var), and all others (ngv). The abstract domain is now formed by taking the powerset of the set, {ground, var, ngv}, ordered by subset. The elements of the lattice each represent some subset of this set, with the full set appearing at the top and the empty set appearing at the bottom. A simple technique for implementing these kinds of descriptions is to use individual bits in an integer to represent the elements of the set. Using this technique, the least upper bound can be computed using a bitwise or of two integers. This approach is used in some of the following domains, as well.

Observations

Many compiler optimizations can be applied when it is known if a variable is unbound (var), bound (nonvar), or ground. Mode domain M_5 is the simplest domain containing all of this information. Mode domains M_6 and M_7 add more complexity. This complexity would be warranted if it improved the ability to get useful modes, but for our benchmark suite, these domains gave almost no improvement.

5.1.2 Type Analysis

Type analysis captures the *type* associated with each variable, that is, the set of terms to which a variable may be bound. This goes beyond mode analysis by providing information about the functors and atomic values appearing within the structure of the variable's value.

Type domains can capture type information to one level (*flat* types), some fixed number of levels, or an infinite number of levels (recursive types). Recursive types can capture a fixed set of types (e.g., just lists) or arbitrary types (type expressions or type graphs). Another choice in developing a type domain is whether to capture a single type for each variable or a set of possible types (*or* types).

Early work considered types separately from modes [6, 8]. Janssens and Bruynooghe referred to these as *rigid types* [40]. They went on to introduce *integrated types*, which allow partially instantiated terms to be described with considerable precision. Since mode information is so important to compile-time optimization, we will only consider integrated types.

The following sections describe a number of abstract domains used to approximate type information. In each case, an abstract description consists of a mapping from variables to type descriptions. These type descriptions can be simple names or complex type expressions. Figure 33 shows how these type domains approximate one another, by placing the domains in a lattice, with the concrete domain at the top and the domain of no (type) information at the bottom.

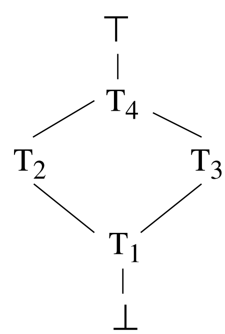


Figure 33: Lattice of Type Domains

Type Domain T_1

The first type domain is a flat type domain with no "or" types. This is similar to the mode domains, in that there are a collection of distinct types, ordered in a lattice. Taylor described such a domain [80]. The lattice for his types is shown in Figure 34. These lattice elements are described in Table 24. The similarity to domain M_5 should be noted.

This domain suffers imprecision from two sources: with the exception of "ground", it cannot express anything about the arguments of a compound value, and it has limited ability to represent the intersection (upper bound) of two sets of values. As an example, the representation for a variable which is either unbound or bound to an integer is 'any', the most general element.

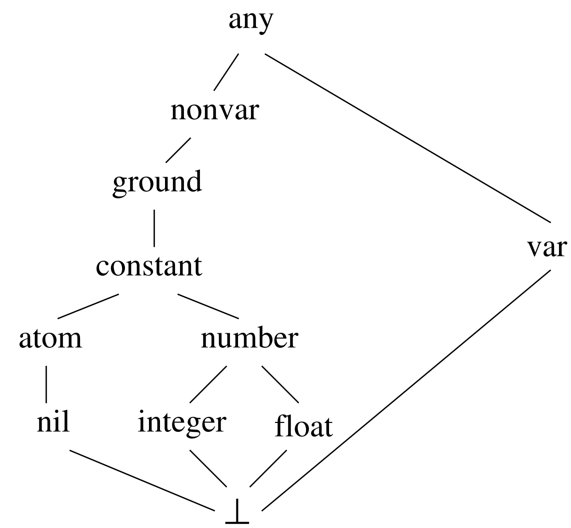


Figure 34: Lattice of Type Descriptions for Domain T_1

Table 24: Definition of Type Descriptions for Domain T_1

Element	Definition
any	The variable could be anything.
nonvar	The variable is bound.
ground	The variable is ground.
var	The variable is unbound.
constant	The variable is bound to an atomic constant (either an atom or a number).
atom	The variable is bound to an atom.
nil	The variable is bound to the nil atom ([]).
number	The variable is bound to a number (either an integer or floating point).
integer	The variable is bound to an integer constant.
float	The variable is bound to a floating point constant.

Type Domain T_2

Type domain T_2 addresses the inadequate handling of type intersections in domain T_1. It is still a flat type domain, however. Domain T_2 is formed by taking the powerset of a set of mutually exclusive type elements, with domain elements ordered by subset; this is similar in form to domain M_6. The concrete values represented by such a description are the union of the values represented by each element of the subset, as defined in Table 25. For example, the subset {posint,zero,negint,var} represents a value which is either bound to an integer or is unbound. This domain is capable of expressing all values from the previous domain, and can express some values even more precisely (as seen in the previous example). In addition, it is able to differentiate between negative and nonnegative integers, between atomic and compound values, and between lists and structures. In other words, it is able to differentiate the data tags that are used in the VLSI-BAM [34].

Table 25: Definition of Type Elements for Domain T_2

Element	Definition (set of Terms)
nil	The nil atom ([]).
nonnil_atom	Any atom, other than the nil atom.
posint	Any positive integer constant.
negint	Any negative integer constant.
float	Any floating point constant.
ng_cons	Any non-empty non-ground list (functor of ./2).
g_cons	Any non-empty ground list (functor of ./2).
ng_struct	Any non-ground compound term, except for functor of ./2.
g_struct	Any ground compound term, except for functor of ./2.
var	Any unbound value.

Type Domain T_3

Type domain T_3 addresses the flatness of domain T_1 by adding recursive lists and nested compound types. Taylor was the first to propose a type domain of this class [78, 80]; we refer to this domain as T_{3a}. Tan and Lin later proposed a simpler version which had fewer atomic types and limited recursive lists to nil-terminated lists [77]; we refer to this domain as T_{3b}. We propose a third version of this domain, which goes the other direction, adding types which improve the detection of the data tags for variables. We refer to this domain as T_{3c}. Table 26 describes the type descriptions in these domains.

The recursive list type provides additional information about the list. The amount of additional information varies across the domains. All three domains specify the type for the list elements (which is recursively an element of this type domain). Domains T_{3a} and T_{3c} specify the type of the list terminator. This is usually either 'nil' or 'var'. Domain T_{3b} only supports nil-terminated lists. Domain T_{3c} additionally allows the fact that a list is non-empty (has at least one element) to be captured.

Structure types specify the functor of the structure and the types of the arguments. Type domain T_{3c} also allows a more abstract representation for structures with the types 'struct' and 'gndstr'. In order to keep these domains finite, nested type specifications are depth limited. Both Taylor [80] and Tan and Lin [77] limit nested types to a depth of four.

Type Domain T_4

Type domain T*4* is the most expressive of the type domains. It captures fully recursive types and type intersections by defining type graphs or type expressions. Janssens and Bruynooghe called these *integrated types* because they include mode information [9, 40, 41].

A type graph is a directed graph with a root node. The root node is the "top" of the type graph. Each node is labelled, to indicate what it represents. The arcs point to the node's successors. The node labels used by Janssens are defined in Table 27. Janssens and Bruynooghe describe a number of restrictions on type graphs that are required in order to make them finite [40].

Table 26: Definition of Type Descriptions in Domains T_{3a} - T_{3c}

Description	Domain	Definition (set of Terms)
any	All	The variable could be anything.
nonvar	All	The variable is bound.
ground	All	The variable is ground.
struct	T_{3c}	The variable is a structure (has a structure tag).
gndstr	T_{3c}	The variable is a ground structure.
list(t_1, t_2)	T_{3a}	The variable is a recursive list, with elements of type t_1 and terminated by a value of type t_2.
list(t_1)	T_{3b}	The variable is a recursive, nil-terminated list, with elements of type t_1.
list(t_1,t_2,c)	T_{3c}	The variable is a recursive list, with elements of type t_1 and terminated by a value of type t_2. If 'c' is true, the list is non-empty (has at least one element).
struct(f(t_1,..,t_n))	All	The variable is a compound term with functor f/n and arguments of type t_1 through t_n, respectively.
var	All	The variable is unbound.
constant	All	The variable is bound to an atomic constant (either an atom or a number).
atom	T_{3a}, T_{3c}	The variable is bound to an atom.
nil	T_{3a}, T_{3c}	The variable is bound to the nil atom ([]).
number	T_{3a}, T_{3c}	The variable is bound to a number (either an integer or floating point).
integer	T_{3a}, T_{3c}	The variable is bound to an integer constant.
float	T_{3a}, T_{3c}	The variable is bound to a floating point constant.

Observations

Flat type domains are most useful when capturing flat information, such as numbers. This can be seen in benchmarks like sendmore and tak. Flat type analysis decreases

Table 27: Definition of Node Labels for Type Graphs

Label	Definition (set of Terms)
any	Any term (including unbound values).
var	An unbound value.
integer	An integer constant.
float	A floating point constant.
OR	The union of the sets of terms defined by the node's successors.
f/n	A structure with functor f and arity n (0 for an atom). The node's successors are ordered and describe the arguments of the structure.

execution time by 20% for tak and 35% for sendmore over mode analysis. Many benchmarks don't benefit from this since they store information in nested data structures (such as a list). This information is lost very quickly without some way to capture it in the type domain.

Generalized "or" types can provide a wealth of information for optimization, but type domain T_4 is currently too expensive to be useful. It appears that a type domain like T_{3c}, which implements recursive lists and limited nested types provides much of the information for optimization with much lower analysis time.

There are two directions worth exploring, starting from domain T_{3c}. First, it might be useful to have some "or" types. A simple indication that a variable is either unbound or has some given type would capture many useful cases. This could be captured by adding a 'var' flag to each type description. Second, the type hierarchy of T_{3c} might be more complex than is strictly needed. Integer and float types are added to simplify arithmetic code when the data type is known. If this information is lost, code must be generated to handle either integer or float data. Many of these cases could be dealt with by adding a flag for each variable, indicating if the variable contains any integers or floats (e.g., the domain could be the powerset of {integer,float}). This can reduce the number of type tests performed as part of arithmetic built-ins.

5.1.3 Aliasing Analysis

Early mode analysis was found to be unsound since it didn't consider the effects of aliasing [27]. Consider the example program from [27], shown in Figure 35. After the call to `q/2`, the variables X and Y are aliased together. The call to `r/1` binds both X and Y to the atom `a`, but without aliasing information, it may seem that Y is not affected. To correct this, it is necessary to assume that instantiations occurring to some variables may further instantiate other variables aliased with the modified variables. Aliasing analysis can restrict the scope of these changes by determining what aliasing exists between variables.

```
p(X,Y) :- q(X,Y), r(X), s(Y).
q(Z,Z).   r(z).   s(_).
```

Figure 35: Example of Aliasing in Mode Analysis

In addition to improving mode and type information [65], aliasing analysis can be used in its own right for automatic parallelization of Prolog code [9] and removal of occur-check from unification (or detection of where occur-checks may be needed, since most Prolog implementation don't perform occur-checks during unification) [42].

Many aliasing domains have been proposed [11, 12, 10, 40, 42, 48]. Most are constructed as the product of simpler domains, such as weak coupling, strong coupling, linearity, and modes. Modes have already been addressed. The remaining sub-domains are described in the following sections. Some terminology related to variable aliasing is defined in Table 28.

When examining these domains, the following substitution will be used as an example:

{ S → a, T → E, U → D, V → f(D,D,D), W → A, X → A, Y → B, Z → f(A,B,C) }

5.1.3.1 Weak Coupling

As the example in Figure 35 showed, it is important to know when two variables are definitely independent, in order to restrict the affects of binding a variable. This is

Table 28: Variable Aliasing Terminology

Term	Definition
independence	Two (or more) program variables are independent if their values share no variables. A variable is independent if it is independent of all other variables.
weak coupling	Two (or more) variables are weakly coupled if their values share one or more variables. This is the opposite of being independent.
strong coupling	Two (or more) variables are strongly coupled if grounding one also grounds the other(s). This is true if and only if the set of variables in the value of one variable is the same as that in the other(s).
covering	A variable, X, covers another variable, Y, if grounding X also grounds Y. This is a unidirectional form of strong coupling.
linearity	A program variable is linear if it has no variable appearing more than once in its value.
equivalence	Two program variables are equivalent if they have the same value. For partially instantiated structures (or unbound variables), this means the variables, themselves, must match.

captured by determining which variables are possibly aliasing (weakly coupled). If it is not the case that two variables are possibly coupled, then they must be independent and therefore, any binding to one variable will not affect the other.

Weak Coupling Domain WC_1

A simple domain for capturing weak coupling is the powerset of the set of variables, ordered by subset. An element of this domain represents the set of possibly coupled variables, i.e., all variables not in this set are definitely independent. To the best of our knowledge, this domain has never been proposed previously.

The example substitution would be represented in this domain as {U,V,W,X,Y,Z}. This is shown graphically in Figure 36. Even though X and Y are independent, this information is lost. In fact, under this domain, it must be assumed that all variables are dependent.

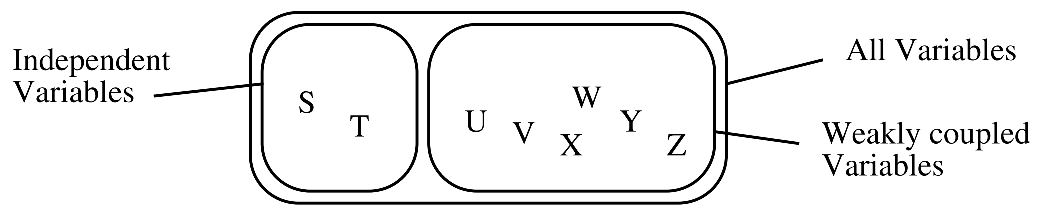


Figure 36: Graphic Depiction of Aliasing in Domain WC_1

Weak Coupling Domain WC_2

Domain WC_2 was proposed by Chang to detect data dependencies in order to implement semi-intelligent backtracking [11]. This domain addresses the deficiency of the previous domain. An element of this domain is a set of sets of variables, partitioning the variables into sets of possibly coupled variables. This means the domain is the powerset of the powerset of the set of variables, with the restriction that no variable can appear in more than one set.

The example substitution would be represented in this domain as { {T}, {U,V}, {W,X,Y,Z} }. This is shown graphically in Figure 37. It is now possible to detect that U and X are independent, but it still isn't possible to detect that X and Y are independent.

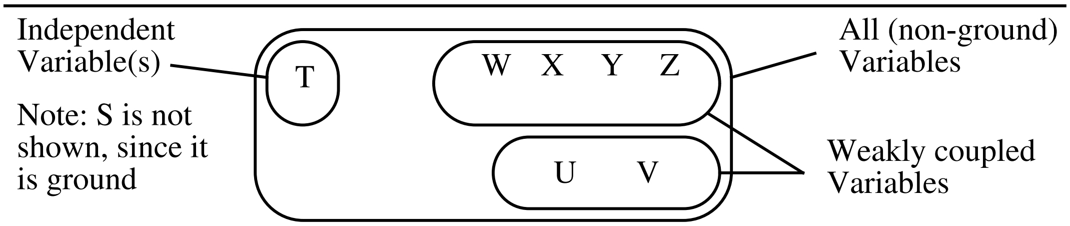


Figure 37: Graphic Depiction of Aliasing in Domain WC_2

A singleton set (a set containing a single variable) indicates an independent variable. When this domain was first proposed, as part of a technique called Static Data Dependency Analysis (SDDA) [11], it also captured groundness; any variables not appearing in the abstract description were considered to be ground. This can still be done, or groundness can be captured using mode domain M_1. Groundness information can be very important in maintaining the precision of aliasing analysis because ground variables are definitely independent.

Weak Coupling Domain WC_3

The most expressive domain for weak coupling, WC_3, was proposed by Jones and Sondergaard [42]. In this domain, weak coupling is captured as an undirected graph in which the arcs represent possible coupling between two variables. There are many ways to represent this graph [3]. One common representation is a set of pairs of variables (representing the arcs). This makes the abstract domain the powerset of the cross product of the set of variables with itself, ordered by subset.

The example substitution would be represented in this domain as { (U,V), (V,U), (W,X), (W,Z), (X,W), (X,Z), (Y,Z) }. This is shown graphically in Figure 38. It is now possible to detect that U and X are independent, and that X and Y are independent.

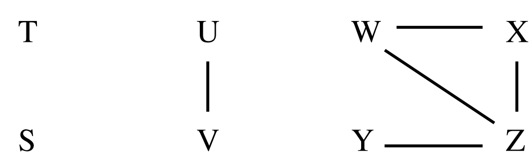


Figure 38: Graphic Depiction of Aliasing in Domain WC_3

5.1.3.2 Strong Coupling

As mentioned previously, knowing when variables become ground helps maintain the precision of aliasing analysis, since ground variables are definitely not aliased. Interestingly enough, aliasing information can also help improve approximation of groundness. For example, if variable Z in the example substitution becomes ground, this also makes variables W, X, and Y ground. Weak coupling doesn't allow this to be detected, however, since it only addresses the *possibility* of aliasing. Instead, we need to know when variables are *definitely* aliased, in fact, when the sets of variables they share are identical, so that the grounding of one grounds the other(s). This is known as *strong coupling*. The following sections describe some domains for detecting this type of coupling.

Strong Coupling Domain SC_1

The simplest strong coupling domain captures sets of equivalent variables. This was suggested by Bruynooghe et al. in [8]. The domain is the powerset of the powerset of the set of variables, ordered by subset. An element is a set of sets of variables, with no variable appearing in multiple sets (if a variable was equivalent to two sets of variables, all variables in both sets would be equivalent).

The example substitution would be represented in this domain as { {W,X} }. It is now possible to detect that grounding of W grounds X, and vice-versa. This domain can't be used to detect the strong coupling of U and V, however.

Since this domain can only be used to detect when variables are exactly equivalent (e.g., they have been unified together), it isn't very useful. This type of equivalence happens very infrequently.

Strong Coupling Domain SC_2

Citrin proposed a domain which had the same form as SC_1, except the descriptions captured sets of strongly coupled variables (i.e., grounding of one variable in the set grounded all other variables in the set) [12]. This was proposed as an improvement to Chang's SDDA [11]. It is interesting to note that in his domain, which was the product of SC_2 and WC_2, each weak coupling set was further divided into multiple strong coupling sets. In other words, the strong coupling sets were a further partitioning of the variables, beyond the partitioning due to the weak coupling.

The example substitution would be represented in domain SC_2 as { {W,X}, {U,V} }. It is now possible to detect that grounding of U grounds V, and vice-versa. This domain can't be used to detect that Z covers W, X, and Y (i.e., grounding of Z grounds W, X, and Y).

5.1.3.3 Coupling Domain CC

Langen proposed a domain that included both strong and weak coupling, and can address the weakness of SC_2 [48]. He called this domain *sharing*, but since we use sharing, later, to refer to the sharing of memory words in an implementation, we call his domain *combined coupling*, or simply, CC.

This domain looks similar to domains WC_2 and SC_2, in that elements are sets of sets of variables (the domain is the powerset of the powerset of the set of variables), except variables can appear in multiple sets. An element of the domain is a set of *sharing groups*. Each sharing group identifies the variables that (may) share a set of variables in their value. We say *may* because the set of variables they share may be the empty set.

The example substitution would be represented in this domain as { {W,X,Z}, {Y,Z}, {Z}, {U,V}, {T} }. The first set corresponds to those variables containing A in their value, the second to those containing B, and so on. Any variable not appearing in a sharing group must be ground. Since a domain element is a set, if there had been two variables appearing in the same set of variables, there would only be a single sharing group to reflect this. Also, this domain element represents a single substitution. If it were abstracting two substitutions, the given example, and one in which all of the variables were ground, the result would be the same; this is because it represents possible sharing groups (and this is where that key word *may* comes from).

Two variables are weakly coupled if and only if they appear together in some coupling group. Therefore, it can be determined that X and Y are independent (making this look as good as WC_3). Two variables are strongly coupled if and only if they appear together in all coupling groups in which one appears. Therefore, it can be determined that U and V are strongly coupled and that W and X are strongly coupled (making this look as good as SC_2). Whenever a variable is bound to a ground term, all coupling groups containing that variable are removed from the description (since any variables in the value of that variable are now ground). This can be used to show that the grounding of Z makes W, X, and Y ground (which couldn't be done with the previous domains).

5.1.3.4 Equivalence

In addition to being used to improve groundness information, aliasing information can improve more general mode and type information. For example, if it is known that two variables, X and Y, are equivalent and the mode (or type) of X changes, the mode of Y must also change. In fact, the new mode for both variables is the mode obtained by performing abstract unification between the mode of Y and the modified mode of X. This extends directly to sets of equivalent variables.

Equivalence Domain E_1

The first equivalence domain partitions the set of variables into sets of equivalent variables [8]. This is identical to strong coupling domain SC_1. It is merely being used here for a different purpose. For the same reasons given previously, this is also not a very useful domain for capturing equivalence.

Equivalence Domain E_2

The second equivalence domain allows equivalence to be described for portions of the values of variables. This is most useful when combined with a non-flat type domain, providing some structure to variable values to which this equivalence relationship can be attached. Taylor included equivalence within the type descriptions, but only for unbound variables and structure arguments [80]. Janssens provided the equivalence information separately from the types by specifying sets of selectors for equivalent values [40]. These selectors allowed a variable to be selected or some part of the structure of a variable (e.g., the head of a list). In addition, these equivalence relationships could be over any values. For type domains with multiple subtypes of the type ground (e.g., type domain T_1), this can be useful to maintain equivalence relationships over even ground values, since unification can then further restrict these ground values (e.g., restricting a set of equivalent ground values to all be integer).

Equivalence domain E_2 is defined similar to the description used by Janssens [40]. A description is a set of equivalence sets, each of which contains a number of pairs of variables and selector lists. A (possibly empty) selector list selects parts of the term to which a variable is bound. A selector is either `head` (to select the head of a list), `tail` (to select the tail) or `arg(n)` to select the nth argument of a structure.

The example substitution would be represented in this domain as { {[U],[V,arg(1)],[V,arg(2)],[V,arg(3)]}, {[W],[X],[Z,arg(1)]}, {[Y],[Z,arg(2)]} }. This captures all aliasing, exactly. In fact, when this domain has been used in conjunction with a coupling domain, typically only coupling not represented by the equivalence relationship is addressed with a coupling description.

5.1.3.5 Linearity

If the variables V and Z from the example substitution are unified, variables X and Y become aliased. If V had a value of f(D,E,F) instead, X and Y would not become aliased. To detect this, it is necessary to know when a variable's value contains multiple occurrences of some variable (as is the case with V). Langen called these variables *non-linear* [48]. Jones and Sondergaard called them *reocc* (repeated occurrence) variables [42]. Janssens called them *NUNI* (for Not-UNIque) variables [40].

Knowledge of linearity is very important during predicate calling/returning. If a call argument is a variable, X, and the head argument is a term, [H|T], it would have to be assumed that the variables H and T are coupled, unless it is known that X is linear. After all, what if X is bound to [A|A]?

Domain L captures this information as the set of definitely linear variables. Therefore, the domain is the powerset of the set of variables, ordered by subset. The example substitution would be represented as {S,T,U,W,X,Y,Z} (all variables are definitely linear except V).

5.1.3.6 Proposed Aliasing Domains

This section reviews a number of proposed aliasing domains. It shows how these domains are constructed, using the simpler domains defined in the previous section. Table 29 describes the domains. Figure 39 shows how these domains approximate one another, by placing the domains in a lattice, with the concrete domain at the top and the domain of no (aliasing) information at the bottom.

5.1.3.7 Observations

Aliasing information is very important in obtaining precise results for mode/type analysis (and vice versa). The types of aliasing analysis worth performing depend highly on how the information will be used. For example, strong coupling is very important for a parallelizing compiler where the objective is to detect possible coupling. For a sequential compiler, it isn't as useful.

For a sequential compiler, there are two main goals:

- *Worst case information propagation due to possible (but not actually occurring) coupling should be reduced. Knowledge of weak coupling helps here.*

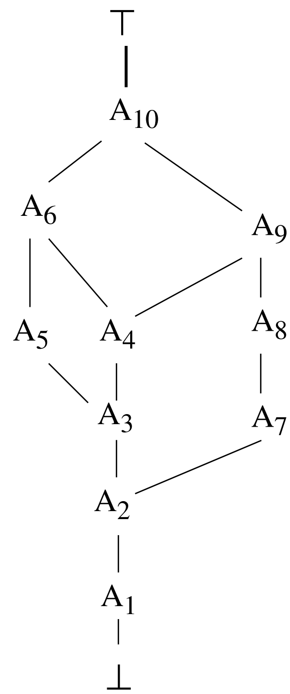


Figure 39: Lattice of Aliasing Domains

- *When performing type analysis, types should be propagated back to strongly coupled variables. Equivalence with selectors helps here.*

Therefore, the best domain for a sequential compiler appears to be domain A_6. Although domain A_9 may provide more precise coupling information, it is too expensive to compute for predicates with a large number of variables (e.g., the zebra benchmark). Besides, A_9 doesn't provide equivalence information.

5.2 Implementation-Dependent Variable Analyses

The previous sections described analyses based on substitutions, a representation that is independent of the Prolog implementation. The following sections describe analyses that approximate properties specific to a given Prolog implementation.

5.2.1 Sharing Analysis

This section describes a number of abstract domains used to approximate sharing information. Sharing is similar to aliasing, expect that it refers to the sharing of memory structures between variables. Therefore, it is possible to have sharing between two

Table 29: Proposed Aliasing Domains

#	Reference	Definition:	Comments:
A_1	None	WC_1	This is a simple domain, provided for completeness.
A_2	Chang [11]	WC_2	When combined with M_1, this is SDDA.
A_3	None	WC_3	This adds non-transitive weak coupling (A and B are coupled, B and C are coupled, but A and C are not).
A_4	Jones & Sondergaard [42]	$WC_3 \times L$	This adds linearity. This is important for predicate entry and exit.
A_5	Bruynooghe et al. [8]	$WC_3 \times E_2$	This drops linearity, but adds equivalence.
A_6	Jannsens [40, 41]	$WC_3 \times E_2 \times L$	This adds both. It was used along with T_4.
A_7	Citrin [12]	$WC_2 \times SC_2$	This improves on SDDA by adding strong coupling classes.
A_8	Langen [36]	CC	This captures both weak and strong coupling in a single, combined representation which is more expressive than both combined.
A_9	Langen [48]	$CC \times L$	This adds linearity in order to keep the information more precise (and computationally feasible). It also includes M_2.
A_{10}	None	$CC \times L \times E_2$	This adds equivalence. It is more precise than all other abstract domains.

ground variables. Clearly, the amount of sharing between variables depends highly on when the implementation copies information as opposed to creating pointers to shared data structures.

Knowing about sharing is important in order to perform compile-time garbage collection [8, 35, 45], that is, to reuse memory structures when they are no longer needed (they become dead).

Because sharing so closely resembles aliasing, the domains will also. In fact, any domain used for aliasing can be used for capturing sharing, with an appropriate change to the definitions of the abstract operations over the domain.

The following sections describe some domains that have been proposed for capturing sharing information.

Sharing Domain S_1

Bruynooghe et al. proposed a sharing domain to be used for compile-time garbage collection [8]. The domain is the product of a possible sharing and definite sharing domain. The possible sharing domain can describe the following:

- X AL Y: variables X and Y may share the same memory structure.
- Comp(T,X) AL Y: Values of type, T, within the structure of variable X may be shared with variable Y.
- Comp(T,X) AL Comp(T,Y): Values of type, T, within the structure of variable X may be shared with values of the same type within the structure of variable Y.

The definite sharing domain can describe the following:

- X PART Y: Variable X is certainly a part (not the whole) of Y.

Sharing Domain S_2

Mulkers et al. provided a more elegant description for a sharing domain [66]. In addition, they describe an *instrumented concrete semantics*, which captures actual sharing occurring in both structure copying and structure sharing implementations. They use this semantics to prove the correctness of their abstract semantics. The form of this domain is similar to SC_2, with a domain element consisting of a set of sharing sets. Each sharing set contains (variable and type) selectors for values that share. It is different from SC_2 in that it describes possible sharing.

5.2.2 Reference Chain Analysis

As described previously, dereferencing values (following pointer chains) is a basic operation in Prolog. We have found that nearly 93% of all dereference operations have no pointer chain to follow on the BAM. This is consistent with Touati's findings on the WAM [82]. Reference chain analysis attempts to determine at compile time, the length of the pointer chain for each variable.

This section describes a number of abstract domains used to approximate this information. These domains are characterized by the categories of reference chains described at the top level of a value and the categories of reference chains described for arguments of compound terms. Figure 40 shows how these domains approximate one another by placing the domains in a lattice, with the concrete domain at the top and the domain of no (reference chain) information at the bottom.

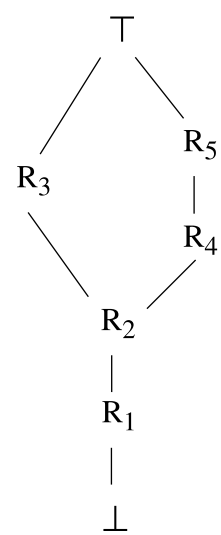


Figure 40: Lattice of Reference Chain Domains

Reference Chain Domain R_1

Domain R_1 is the simplest reference chain domain. This is a boolean domain capturing those variables which are definitely dereferenced (contain no pointer chain to the top-level value).

Reference Chain Domain R_2

Domain R_2 was described by Van Roy [84]. The lattice for each variable consists of the following set of values (listed in order): {any, deref, rderef}. These values have the following meaning:

- any: Nothing is known about the references chains for a variable, either at the top-level or for the arguments.

- deref: There is no reference chain at the top level of the variable, but nothing is known about the arguments if the value is compound.
- rderef: There are no reference chains anywhere in the value of the variable (i.e., it is recursively dereferenced).

Reference Chain Domain R_3

Domain R_3 is a refinement on R_2. We discovered while experimenting with the Aquarius compiler that it could occasionally claim that a variable was recursively dereferenced when in fact it was not. This was due to an error in the abstract operation for predicate exit.

Figure 41 illustrates the problem. After calling p/2, variables X and Y are aliased, but 'rderef'. After the unification in q/1, A (which is bound to Y) is no longer dereferenced. Its use in the next goal (B is A+1) causes A to become dereferenced again. The compiler remembers this and uses the dereferenced copy for all subsequent uses of A in the clause. At the end of the clause (when returning to the main predicate), the analyzer propagated this information back, keeping Y as rderef. It should have assumed that Y was not rderef (especially since it wasn't). There are two ways to solve this:

- The compiler can perform a source transformation whenever it knows variables are used in calls which require dereferenced arguments (e.g., many built-in predicates). This would make q/1 look like: q(A) :- A=5, '$deref'(A,A'), B is A'+1. Now, the second goal causes A' to become a dereferenced copy of A, which can then be used in the remainder of the clause. When returning to main/0, the non-deref mode for A will be propagated back to Y. The problem with this is that it will increase the number of variables appearing in clauses, which in turn increases the compilation and analysis time.
- A new reference chain mode can be added to reflect what is happening. This mode, which we call 'locally_deref', indicates a variable which has been dereferenced by a call to a predicate requiring a dereferenced argument. This is treated as 'deref' for the remainder of the clause, but as 'any' when performing predicate exit.

We chose the second alternative, arriving at a lattice of {any, deref, locally_deref, rderef}. Although this doesn't improve performance over Van Roy's lattice, it does allow the same performance with a sound treatment of reference chains.

Program:

```
main :- p(X,Y), q(Y), r(Y).
p(A,A).
q(A) :- A=5, B is A + 1.
r(5).
```

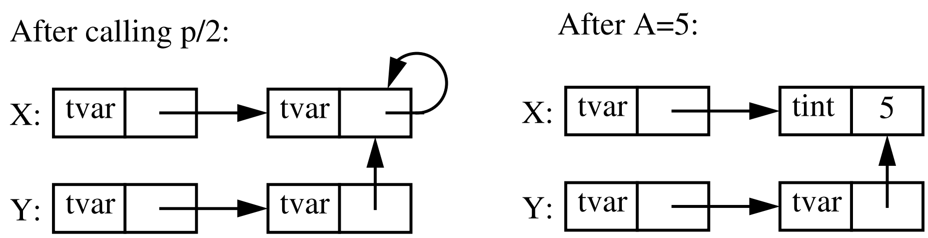


Figure 41: Propagation of Ref Chain Info Through Predicate Exit

Reference Chain Domain R_4

Domain R_4, proposed by Taylor, captures more categories of chains and allows different descriptions for different arguments of compound values [78]. To support this, it needs to be combined with a type domain which allows non-flat types to be captured, for example T_{3c}. He categorized reference chains based on implementation issues, as seen below:

- 0: The reference chain is of length 0 (i.e., no reference chain). The value can be accessed directly.
- 1: The reference chain is of length 1. The value can be accessed by following one link (an indirect memory reference).

- 0-1: The reference chain is of length 0 or 1. The value can be accessed with a single test for a reference chain.
- ?: The reference chain is of an unknown length. The value must be dereferenced using the general dereference algorithm.

For a small set of benchmarks, Taylor found that this would remove 70-90% of the general dereference loops.

In addition to allowing this reference chain description for the top level of a value, he allowed it for some types from domain T_{3a}. A similar description was provided for the arguments of a ground value, for individual arguments of a structure type, for the elements of recursive lists, and for the cdr (tail) link of recursive lists. It is not clear why he didn't also allow this for arguments of nonvar and 'any' types (as he did for ground types). With this addition, this domain becomes more expressive than R_2. Even without types, this domain could be used by specifying a pair of reference chain descriptions per variable, one for the top-level, and one for all arguments.

Reference Chain Domain R_5

Domain R_5, proposed by Mariën et al., provides even more levels by describing a reference chain as having a minimum and maximum length [54]. For example, 1..3 would describe a reference chain that had between 1 and 3 links. By combining this with a type domain, this provides a very expressive reference chain domain.[1]

The least upper bound of two elements in this domain is computed by taking the minimum of the two minimums and the maximum of the two maximums. The minimum is limited by zero, but there is no limit on the maximum length. It is not clear, therefore, how to keep this domain finite. One partial solution, given in [54], is to always reduce the length of a chain by the minimum before calling a predicate. Therefore, the minimum at predicate entry will always be zero. Also, it is unlikely that this adds much performance over R_4, since only a fraction of a percent of reference chains ever exceed a length of one [82].

---

1. Mariën et al. combined R_5 with T_4, but other type domains could be used.

5.2.3 Trailing Analysis

Another basic operation in Prolog is trailing of variables in order to restore the program state during backtracking. Trailing analysis indicates when variables definitely do or do not need to be trailed during operations which may (or do) bind them [78]. Obviously, a variable does not need to be trailed if it is already bound. In addition, a variable does not need to be trailed if it can be shown that the variable was created after the current choicepoint. This requires predicate-level knowledge, which will be addressed later. Without this knowledge, we can only talk about local variables (created after any choicepoint).

Trailing Domain TR_1

Domain TR_1 is a simple domain for capturing trailing information. It was first proposed by Taylor. The domain is the powerset of the set of variables, ordered by subset. Each element identifies the set of variables which can safely be bound without trailing (i.e., they are newer than the latest choicepoint).

Taylor proposed this domain in conjunction with his type domain (T_{3a}) [78]. He allowed any possibly unbound places in the type description to be annotated with a 'trailing not needed' flag.

Trailing Domain TR_2

Trailing domain TR_1 attempts to eliminate unneeded trailing operations. We propose a domain, TR_2, which also attempts to eliminate trail checks when it is known that a variable is definitely older than the latest choicepoint. This domain maps variables onto values from the lattice shown in Figure 42. A variable which is 'newer' than the current choicepoint doesn't need to be trailed. A variable which is 'older' than the current choicepoint must be trailed when bound (and therefore doesn't even need to be compared with the heap address stored in the choicepoint). A variable which is 'any' needs the full trail check and conditional trailing.

5.2.4 Access Analysis

There are a class of analyses that capture when certain type of accesses occur for a variable. Examples include the first access or last access. We refer to these as *access* analyses. They are described in the following sections.

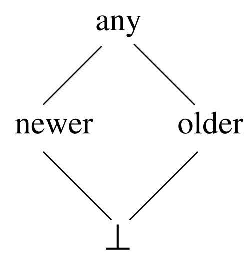


Figure 42: Lattice for Trailing Domain TR_2

Liveness Analysis

To perform compile-time garbage collection, two types of information are required: sharing of data structures (which has been described) and variable liveness. Liveness analysis determines when variables (or structures in variables) are no longer live, that is, when their memory can be reused. This is done by determining the last access to a variable or part of its value. Bruynooghe described a domain for capturing this information in conjunction with sharing domain S_1 [8]. This liveness domain describes what values may still be needed when returning from the current predicate, through the following types of statements:

- LIVE X: variable X may still be needed (live) on return.
- LIVE_Comp(T,X): Values of type, T, within the structure of variable X may still be live on return.

Initialization Analysis

Initialization analysis attempts to find the first ‘unprotected’ access to an unbound variable. Detecting this allows the initialization of the variable to be deferred. There are a number of advantages to this. An uninitialized variable does not need to be trailed when it is bound, since it has no value. In fact, it can be bound with a simple assignment. This reduces the size of the code and the trail. This occurs frequently when variables are passed to predicates which will bind them (i.e., they are output variables). Taylor achieved this by deferring initialization of variables known to be unbound and unaliased [80].

Van Roy took a more aggressive approach, relaxing the aliasing restriction [84]. He does assume, however, that the uninitialized variable will be at the end of any reference chain created during unification (if not, the variable would have to be initialized to know when the chain ended). The problem with deferring the initialization of an aliased variable is what happens when a variable aliased with the uninitialized variable is accessed. If an attempt is made to dereference the aliased variable, the uninitialized variable will be read, giving garbage. Therefore, Van Roy uses the following rules to identify variables whose initialization can (will) be deferred:

- The variable must be unbound.
- The variable must be 'rderef' (at the end of the reference chain).
- There must be no reference to the variable (either directly or indirectly). The variable must be initialized before such a reference might occur. This is what we previously referred to as an 'unprotected' access.

Van Roy combined uninitialized variable information with modes. We prefer to separate it and keep it along with information about when a variable is actually allocated in memory (its first occurrence). The domain maps variables onto the values: {init, uninit, unalloc}. This more aggressive deferring of variable initialization reduces the value in attempting to eliminate trailing; most trailing which would be eliminated is caught with 'uninit' variables.

Uninit_Reg Analysis

In addition to deferring initialization of variables, Van Roy attempts to defer their allocation when their first use is as an argument to a call which binds the variable, using a technique he calls *uninitialized register arguments* [84].

In all Prolog implementations we have seen, all arguments are passed in to a called predicate. This means that for arguments which would be considered 'output' arguments in other languages (i.e., arguments which are unbound on entry and bound on return), memory must be allocated in the caller and the address passed in; the called predicate would then use this pointer to bind the return value. Van Roy's initialization deferral makes this a less costly operation than on most implementations, but it still requires passing all values through memory.

Van Roy attempts to model true output arguments by having a predicate return values in argument registers. This is very useful for predicates which return numeric information (e.g., the tak benchmark), or other small constants which fit into a register. In order to do this, an argument must be unallocated at the time of the call and bound either by a simple built-in at the end of the called predicate or passed to a predicate which returns the value in the same argument register (i.e., it is passed as an uninitialized register argument in the same position in the last goal of the predicate). This last condition ensures that last call optimization can be applied in the presence of uninitialized register arguments. Normally the call to the last goal in a clause is turned into a jump. This couldn't be done if an uninitialized register argument was in the wrong position, as it would be returned in the wrong argument register and a move instruction would be needed to correct this.

The domain for capturing this information consists of the powerset of the set of variables, ordered by superset. Propagation through the clause restricts the set of possible arguments until, at the end, the resulting set are those arguments whose values can be returned in argument registers.

Binding Analysis

When a called predicate modifies one or more of its arguments, this change must be propagated to other variables in the caller which are possibly aliased with the arguments. Aliasing analysis attempts to restrict the scope of this change by determining a smaller set of possibly aliased variables. Another approach is to determine arguments which were definitely *not* modified (bound) by the call. For example, if a variable is ground before a call, it can't be modified by the call. In addition, there are a number of built-ins which perform tests on the arguments, and do not bind them (e.g., structure(X)).

The domain for capturing this information is the powerset of the set of variables, ordered by subset. An element of this domain indicates the set of possibly bound variables. The empty set is the bottom element, since it indicates that no variables were bound.

As indicated, this information can be used at predicate exit to restrict the propagation of changes. In fact, we have implemented only a minimalistic version of this which has a preset value for each of the built-ins (indicating which arguments they bind) and makes a worst case assumption for all user predicates (assuming all arguments have been modified). When combined with information about which variables were initially ground and aliasing information, this provides sufficient restriction on the propagation of other dataflow information.

5.2.5 Observations

Probably the most important of these analyses are those allowing initialization and allocation of variables to be deferred. In addition to the direct benefits of reducing memory access, they reduce trailing and dereferencing, since uninitialized variables don't need either of these operations applied to them.

Domain R_3 does a good job at removing dereferencing. R_5 is almost certainly overly complex. Probably a good compromise would be a domain containing pairs of values from domain R_4 for each variable: one value addressing the top of the structure and the other the rest of the structure. Another possible direction would be to consider strongly and weakly dereferenced variables. A strongly dereference variable remains dereferenced even when modified (as long as the value being bound with the variable is dereferenced), whereas a weakly dereferenced variable may not. The difference has to do with variable arguments to a structure (or unbound variables). An unbound argument might point to itself, in which case binding it keeps it dereferenced, or it might point to another cell, in which case binding it makes it have a chain of length 1. Alternatively, this value could have been called non-dereferenced initially, which keeps the domain unchanged, but may provide less precise results.

Sharing and liveness analysis should provide useful information for compile-time garbage collection. It is not clear yet what the best domain will be for this application. Much work remains to be done here.

5.3 Predicate-level Analyses

All of the previous analyses capture information about variables. Predicate level analyses collect information about attributes of predicates. This might be a boolean flag for each predicate, or something more complex. The following sections describe a number of attributes like this.

Although abstract interpretation can be used to capture predicate-level information [62], this is probably not the most efficient technique. Consider, for example, attempting to detect the set of recursive predicates. This can be done by performing a transitive closure on the call graph and then collecting the set of predicates that eventually call themselves. Abstract interpretation is still a good technique to use for proving the correctness of predicate-level analyses.

5.3.1 Determinacy Analysis

Mellish defines a predicate to be *determinate* if it will never return more than one solution [63]. It may succeed or fail, but will never be able to return alternate solutions through backtracking. In terms of an implementation, this means that the predicate does not leave a choicepoint on the stack.

A predicate is determinate if every clause other than the last contains a cut and every goal after the cut calls determinate predicates. Mode information can be used, as well, to determine when clauses are mutually exclusive, which reduces the need for cuts.

As described, this doesn't address what happens for side-effects. Consider, for example, the following code: p(a). p(X) :- write(hello), fail. This will never return more than one solution, but must be backtracked to in order for the side-effects to occur.

Mellish uses determinacy information to generate a different kind of call instruction, but then he had a virtual machine, named POPLOG [63], which is very different from the WAM.

Taylor used determinacy information to reduce variable trailing [78]. Any variable being bound which was created after the current choicepoint was created does not need to be trailed.

Debray and Warren extend the idea of determinacy to the concept of *functional* predicates [26]. A functional predicate is one whose outputs depend, functionally, on the inputs; it may return multiple results, but the results are identical for all values that are used (e.g., an output which varies may never be accessed after returning). This concept is less dependent on the use of cuts and extends easily to parallel logic languages.

5.3.2 Local Stack Analysis

Many implementations of the WAM and the BAM place both environments and choicepoints onto a single stack, called the local stack [53]. If it is known what objects are added to the stack, there are a number of optimizations (already described) that can be applied. For example, environments and choicepoints can be reused [60]. This is also related to the determinacy analysis just described, for determining when a predicate leaves a choicepoint on the stack.

This information cannot be captured effectively at the level of Prolog; there is too much imprecision. It may be necessary to perform abstract interpretation of the intermediate code in order to get a usable approximation of this type of information.

Choicepoint Creation

In order to perform precise trailing analysis, it is necessary to know when predicates can generate choicepoints [78]. Taylor does this at the Prolog level and claims to get good results. Better results could be obtained at the intermediate code level, where the choicepoint instructions actually exist.

The domain for this is boolean, capturing those predicates which definitely do not create a choicepoint. For trailing domain TR_2, it may be worthwhile also to capture those predicates which definitely *do* create a choicepoint.

Local Stack State

In order to apply other local stack optimizations (such as environment or choicepoint reuse or simplifying the creation of choicepoints and environments), it is necessary to know what objects are added to the stack. Keeping track of this requires an infinite domain, however, since one predicate can add some number of objects - a possibly infinite number for a predicate with tail recursion. To keep it finite, it is probably sufficient to approximate the top object on the stack at predicate entry and the objects added to the stack on exit (generalizing after some finite limit). The benefits from this analysis don't appear to warrant the added complexity of analyzing the intermediate code.

5.3.3 Observations

The analyses described above are probably too costly for the value added. They would most likely require a second abstract interpreter, executing at the intermediate code level in order to get usable precision. In addition, the optimizations they allow would only have a small (~1%) impact on performance.

The Aquarius compiler performs determinacy analysis locally, at both the Prolog level and the intermediate level, in order to remove the creation of choicepoints. The results are good. It is doubtful if a more complex analysis scheme could do much better.

5.4 Combining Analyses

The previous descriptions have (as much as possible) considered each analysis in isolation. This section looks at the interactions between analyses.

Figure 43 shows the interdependencies between the different types of analyses. As might be expected, the implementation independent analyses are important to the other types of analysis.

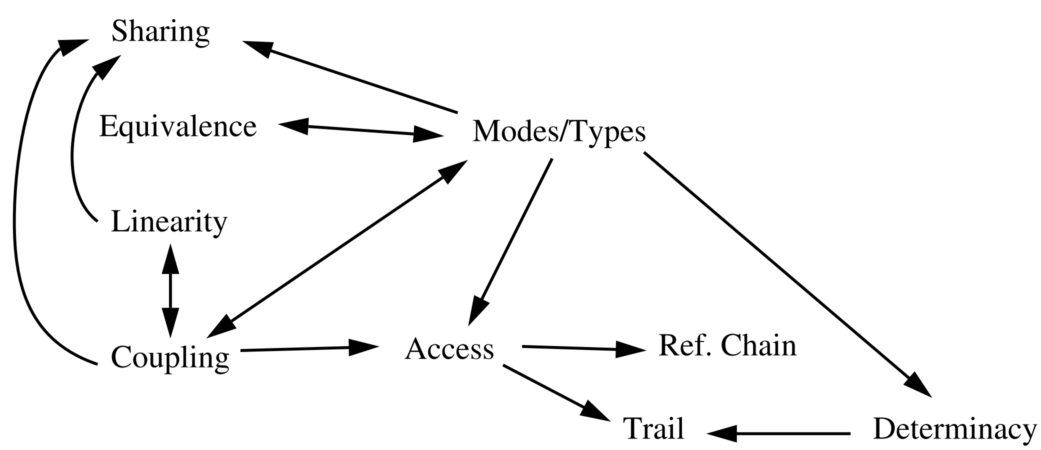


Figure 43: Interdependencies between Types of Analyses

The selection of type (or mode) analysis is very important to the implementation of the other analyses. Many of the other analyses can provide information about multiple levels of a variable's value. But, there is little point in talking about parts of a variables's value beyond that expressed in the type domain.

One goal in this research has been to provide a partitioned description of analyses. This was easily done until non-flat types were introduced. At that point, it became difficult to capture, for example, equivalence without some knowledge of the structure of the type domain. As part of a tool for exploring different analyses and optimizations, separation between domain descriptions was important. Once a "good" collection of domains is selected, however, we expect that the abstract operations for this collection will be reimplemented and optimized, as a whole. Therefore, the difficulty of maintaining partitioned descriptions isn't seen as a major problem.

Chapter 6: Evaluation

This chapter explores the costs and benefits of various dataflow analyses, examining the relationship between dataflow analyses and compile-time optimizations. We begin by examining eight different mode and type domains. After selecting the best of these, we add aliasing analysis. To this we then add implementation-dependent analysis.

Detailed results for experiments can be found in Appendix E. This chapter summarizes and analyzes the results.

6.1 Benchmarks

Benchmark selection is a difficult process. There are a number of pitfalls to avoid. Benchmark programs may not be representative of the "actual" use of a machine or language. Benchmarks may over-emphasize good or bad points in some implementation. Continued use of the same set of benchmarks, although accepted by the community at large, may become the driving function for enhancements, ending up with systems tailored to give good performance for those benchmarks (perhaps at the detriment of other programs).

Nevertheless, some approach must be taken to evaluate system performance. The benchmarks used for the measurements given in the following sections were chosen because performance figures are available for these benchmarks on other Prolog systems. We could have attempted to make this set "more complete", but it is not clear where there might be gaps, or how useful additional information might be. Besides, as the size of the benchmark suite grows, this reduces the number of tests we can perform.

The benchmarks chosen are those used by Van Roy [86] and Taylor [80]. They are described in Table 30. See Appendix D for source code.

6.2 Evaluation Measures

As with benchmark selection, there are no universally agreed upon measures to use when comparing two systems. We have attempted to select a set large enough to address the important aspects, yet not so large as to be overwhelming. The cost of including analyses or optimizations is given in terms of analysis time and compilation time relative to some baseline case. Performance is measured in terms of static code size (for

Table 30: Benchmark Descriptions

Benchmark	Lines	Preds	Clauses	Description
deriv	32	6	15	Symbolic differentiation of four equations. Van Roy reported performance for one equation (times10).
nreverse	10	3	5	Naive reversal of a list of 30 integers. This reflects list manipulation, a common operation in Prolog.
qsort	19	3	6	Quicksort on a list of 50 integers.
serialise	29	7	13	Computes the “serial” number for values in a list. This demonstrates tree manipulation and variable aliasing.
mu	33	8	16	Solves a theorem in Hofstadter’s mu-math system.
pri2	28	5	9	Sieve of Erastosthenes for first 100 integers.
queens_8	30	7	11	Finds all solutions to the eight queens problem (exhaustive search).
fast_mu	57	7	15	A “faster” solution to mu-math, making use of knowledge of the system.
query	68	5	54	Database lookup and integer math to answer a simple query.
press1	277	46	12 2	A simplified version of the PRESS symbolic equation solver.
tak	15	2	3	Highly recursive integer function (takeuchi function).
sendmore	47	4	22	Depth-first search to solve a cryptographic problem.
poly_10	86	11	32	Symbolic polynomial expansion.
zebra	44	5	9	Solution to a simple word problem (where does the zebra live?) through exhaustive search.

Table 30: Benchmark Descriptions

Benchmark	Lines	Preds	Clauses	Description
prover	83	9	32	A simple theorem prover.
meta_qsort	74	7	25	A simple meta-interpreter for Prolog, interpreting qsort.
nand	495	42	137	A logic circuit synthesis program, based on heuristic search.
chat_parser	1139	156	514	The sentence parser for a natural language system.
browse	92	14	29	A "classic" LISP benchmark, exercising pattern matching.
unify	122	12	60	A unification compiler, from the Aquarius compiler.
flatten	159	27	58	Flattens Prolog code into a simpler syntax, from the Aquarius compiler.
crypt	69	9	27	Depth-first search to solve a crypto-arithmetic problem.
simple_analyzer	451	67	143	A simple mode analyzer, from the Aquarius compiler.
reducer	302	41	119	A graph reducer for T-combinators.
boyer	375	25	135	A Boyer-Moore theorem solver (a "classic" LISP benchmark).

the generated code) and execution time relative to some baseline case. The baseline we are using for our measurements is the Aquarius Prolog compiler as currently distributed.

Other measures that could have been chosen were dynamic memory usage during compilation, or dynamic memory usage during execution. While these may be interesting or even important measurements to some researchers, they were not the focus of this research, and so any change in them was incidental.

The benchmarks were executed on a DECstation 3100, running ULTRIX version 3.1. This machine is based on the MIPS R2000 RISC processor, running at 16 MHz, with 16Mb of main memory, 64Kb of cache, and 64 Mb of swap space. Execution times were measured by using the ‘pixie’ and ‘pixstats’ commands to obtain an actual instruction count. By reporting instruction counts, the results won’t vary based on system load or other unknown factors. Also, these results can be compared with results obtained on other system configurations (e.g., Taylor’s results are from a 25 MHz MIPS R3000 machine with different memory resources and a different operating system).

The benchmarks were compiled using the Aquarius Prolog compiler, integrated with our analysis framework, running under Quintus Prolog on an HP 9000/700 workstation. This machine was used rather than the DECstation because it was faster and Quintus Prolog was not available to us on the DECstation. Aquarius Prolog could have been run on the DECstation, but development time using Quintus Prolog is shorter.

6.3 Performance on Aquarius Prolog

In order to provide perspective on the cost and performance of our compiler, it is important to compare it with other Prolog compilers. There are a number of commercial Prolog systems available, for example Quintus, SICStus, and BIM. These systems are not state of the art, and therefore don’t provide a good baseline. Besides which, they weren’t available for comparison. Instead, we use Van Roy’s Aquarius system to provide a baseline. This system was used as the starting point for our work and therefore provides a good point for comparison.

Van Roy’s Aquarius system has been available in a number of different states during its development. The version we used for establishing a baseline is the first version released to the research community, as of April, 1993. This version is available by sending e-mail to `listserv@acal-server.usc.edu`. In addition to a number of bug fixes since Van Roy’s dissertation was written (in 1990), support has been added for floating point and garbage collection, features not available when he performed measurements on the SPARC [86]. Table 31 shows performance figures for the benchmarks compiled by Aquarius Prolog. The first column shows the compilation time, in seconds. The next two columns show the global analysis time, in seconds and as a percentage of the compilation time. The next column shows the static code size of the compiled benchmark. The last column shows the execution time, in terms of the number of instructions executed.

Table 31: Performance of Aquarius Prolog System

Benchmark	Compilation time (Sec)	Analysis time		Static code size (Bytes)	Execution time (insts)
		(Sec)	% C		
deriv	10.7	0.5	4.7	2,540	3,622
nreverse	3.0	0.2	6.7	752	7,024
qsort	5.4	0.5	9.3	1,496	8,515
serialise	9.2	0.4	4.3	3,624	21,482
mu	12.6	0.9	7.1	5,348	39,274
pri2	5.1	0.5	9.8	1,016	29,424
queens_8	8.8	0.6	6.8	2,652	64,129
fast_mu	37.9	2.5	6.6	10,440	51,909
query	9.5	0.5	5.3	3,368	77,707
press1	100.7	8.0	7.9	40,196	60,307
tak	4.3	0.2	4.7	864	1,415,344
sendmore	82.4	0.8	1.0	8,236	2,052,216
poly_10	54.6	2.3	4.2	5,816	999,838
zebra	25.8	0.3	1.2	5,148	4,150,368
prover	27.8	1.3	4.7	8,876	37,652
meta_qsort	24.0	0.9	3.8	9,292	196,659
nand	351.0	35.3	10.1	56,160	578,476
chat_parser	447.5	35.4	7.9	150,524	5,000,476

Table 31: Performance of Aquarius Prolog System

Benchmark	Compilation time (Sec)	Analysis time (Sec)	Analysis time % C	Static code size (Bytes)	Execution time (insts)
browse	23.8	2.0	8.4	8,412	36,012,047
unify	68.4	9.8	14.3	35,116	49,164
flatten	38.6	3.4	8.8	16,388	27,260
crypt	24.7	1.2	4.9	8,436	92,044
simple_analyzer	176.3	11.4	6.5	34,132	780,335
reducer	618.5	6.2	1.0	160,724	1,626,219
boyer	139.5	3.2	2.3	60,360	34,005,215

6.4 Performance Comparison and Analysis

We would like to measure the performance of our compiler on the benchmark suite for all combinations of abstract domains available to us. Unfortunately, there are too many combinations. Therefore, we need to reduce the search space. To do this, we begin by examining the relationship between the different dataflow analyses described in Chapter 5 and the optimizations described in Chapter 2. Figure 44 shows which optimizations depend on which type of dataflow analysis. From this figure, it appears that mode and type analysis is the most important. The next most important is probably aliasing analysis, to improve the modes and types, followed by access analysis (specifically, initialization analysis) and reference chain analysis. We use this information to direct our search.

6.4.1 Minimum Analysis

We begin our search with the bottom domain, that is, the domain of no analysis. Although the benchmarks compile almost twice as fast (52% of the time of Aquarius), the resulting code is almost three times as large and over three times as slow.[1] The

1. See Table 50 in Appendix E.

	Modes	Types	Aliasing	Sharing	Ref. Chains	Access	Trailing	Local Stack	Determinacy
Unification	X	X							
Clause Selection:									
Determinism	X	X							X
Mode Enrichment	X	X							
Semi-Intelligent Backtracking	X	X	X						X
Basic Operations:									
Local Stack Allocation								X	
Dereferencing	X	X			X	X			
Trailing	X	X				X	X	X	
Tag Manipulation	X	X							
Built-ins	X	X							
Compile-time Garbage Collection				X		X			

Figure 44: Information Required for Compiler Optimizations

reason this is so significantly different from Aquarius is that even when not performing global dataflow analysis, Aquarius still performs extensive local analysis during the code generation process. We use the same domain for both global and local analysis.

To compensate (somewhat) for this, the next domain we selected was $AC_1 \times R_1$. This domain keeps track of 'new' variables and dereferenced variables. This removes extraneous operations being generated local to a clause due to the lack of local analysis. It simulates some of the local analysis being performed by Aquarius without adding significant global analysis. New variable analysis is always local to a clause and dereferenced variable analysis usually is. This decreases both the code size and run time to around twice that of Aquarius.[2] Also, it actually decreases the compilation time (to

2. See Table 51 in Appendix E.

45% of Aquarius') since it now it doesn't need to generate as much code. The analysis time increases slightly, to 62% of Aquarius' analysis time. In all subsequent experiments, we include at least these domains.

6.4.2 Mode and Type Analysis

Next, we add either mode or type analysis to this minimal set of analyses. The performance results for mode domains M_1 through M_7 and type domains T_1 through T_3 are shown in Figure 45. This figure shows the performance measures annotating a lattice of mode and type domains. The lattice orders these domains by their degree of precision. The performance figures shown are compilation time, analysis time, static code size and instructions executed, relative to Aquarius.

From this figure, we can see that analysis time increases and execution time decreases as the domains get more precise, as one would expect. As more precise analysis is performed, we would expect to see the code size decreasing, since the analysis eliminates cases at compile time, requiring less code to be generated. This is the case except for when comparing domains M_3 or M_4 with M_5. At this point, the code size increases. The reason for this is that M_5 is the simplest domain for which *mode enrichment* [84] can be performed. This optimization reduces execution time, but does so by increasing code size.

Compilation time is a little more complex since it consists of several parts: source preparation time, analysis time, code generation time, and code optimization time. The source preparation time is independent of the analysis being performed. The analysis time has been shown to increase as more precise analysis is performed. The code generation and optimization times tend to decrease with more precise results (since less code needs to be generated and optimized). For these domains, we see the total compilation time increasing through the mode domains and then reverse direction through the type domains.

The most precise type analysis we examined (T_3) reduces code size by 20% and execution time by 32% over no mode or type analysis. This increases the compilation time by 62% (which is still 27% less than Aquarius'). Mode domain M_2 provides almost no improvement over no analysis. This is as expected, since the domain only captures

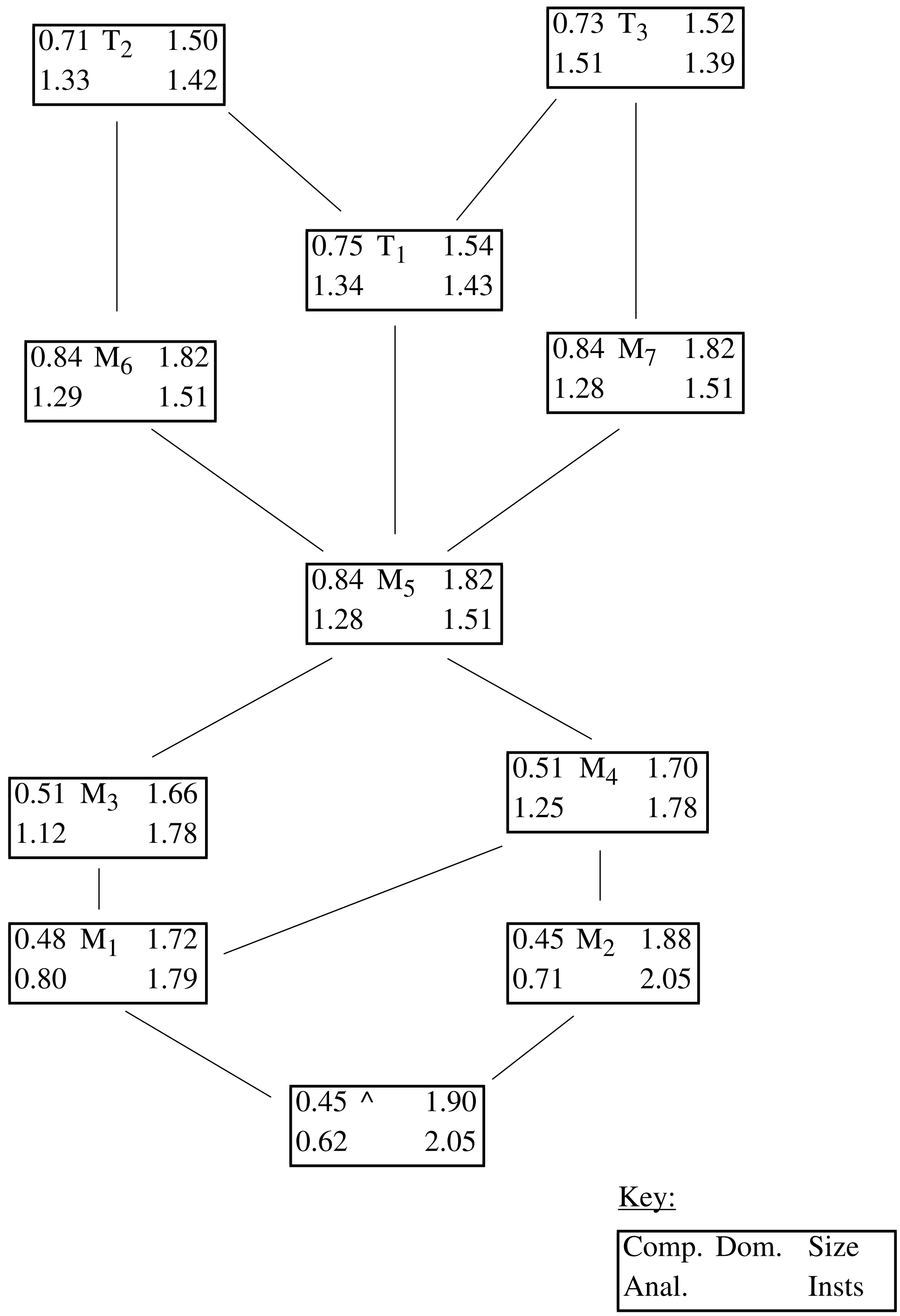


Figure 45: Performance Lattice for Mode/Type Analysis

unbound variables. This is a mode which is quickly lost without some form of aliasing information. Mode domain M_5 provides the best incremental improvement in execution time. This is the first domain containing all of the modes useful for optimization: var, nonvar, and ground. Domains M_6 and M_7, which add refinements on these modes, provide no added advantage. Perhaps the lesson to be learned here is that a domain should contain the information needed for optimizations and little else.

All of the type domains provide a significant decrease in static code size. Before performing type analysis, code was generated to handle both integers and floating point for expressions. With type analysis, we can determine, at compile-time, that many of the operands in arithmetic expressions are integers. As expected, the best type domain is T_3, which supports nested structures and recursive lists. Unfortunately, we were unable to get complete results for domain T_4, which supported fully recursive types and or-types. We will return to this later.

6.4.3 Aliasing Analysis

Next, we add aliasing analysis. It would be nice if we could explore all aliasing domains in combination with all mode and type domains, but this is not feasible. Therefore, we chose to explore the impact that aliasing analysis has in combination with type domain T_3, since this is the most expressive type domain we have fully implemented, and therefore most likely to benefit from aliasing analysis. The performance results for aliasing domains A_1 through A_{10} in combination with $T_3 \times R_1 \times AC_1$ are shown in Figure 46. This figure shows the performance measures annotating a lattice of aliasing domains. The lattice orders these domains by their degree of precision. The performance figures shown are compilation time, analysis time, static code size and instructions executed, relative to Aquarius.

From this figure, we see similar trends to those we saw for modes and types. As the domains get more precise, code size and run time decrease while analysis time increases for the most part. It is interesting to note that the analysis time actually decreases when going from domain A_8 to A_9 and again from A_9 to A_{10}. The reason for this is that, even though the domains are getting more complex, this complexity keeps the values in the portion of the domain which is most costly to compute (sub-domain CC) small.

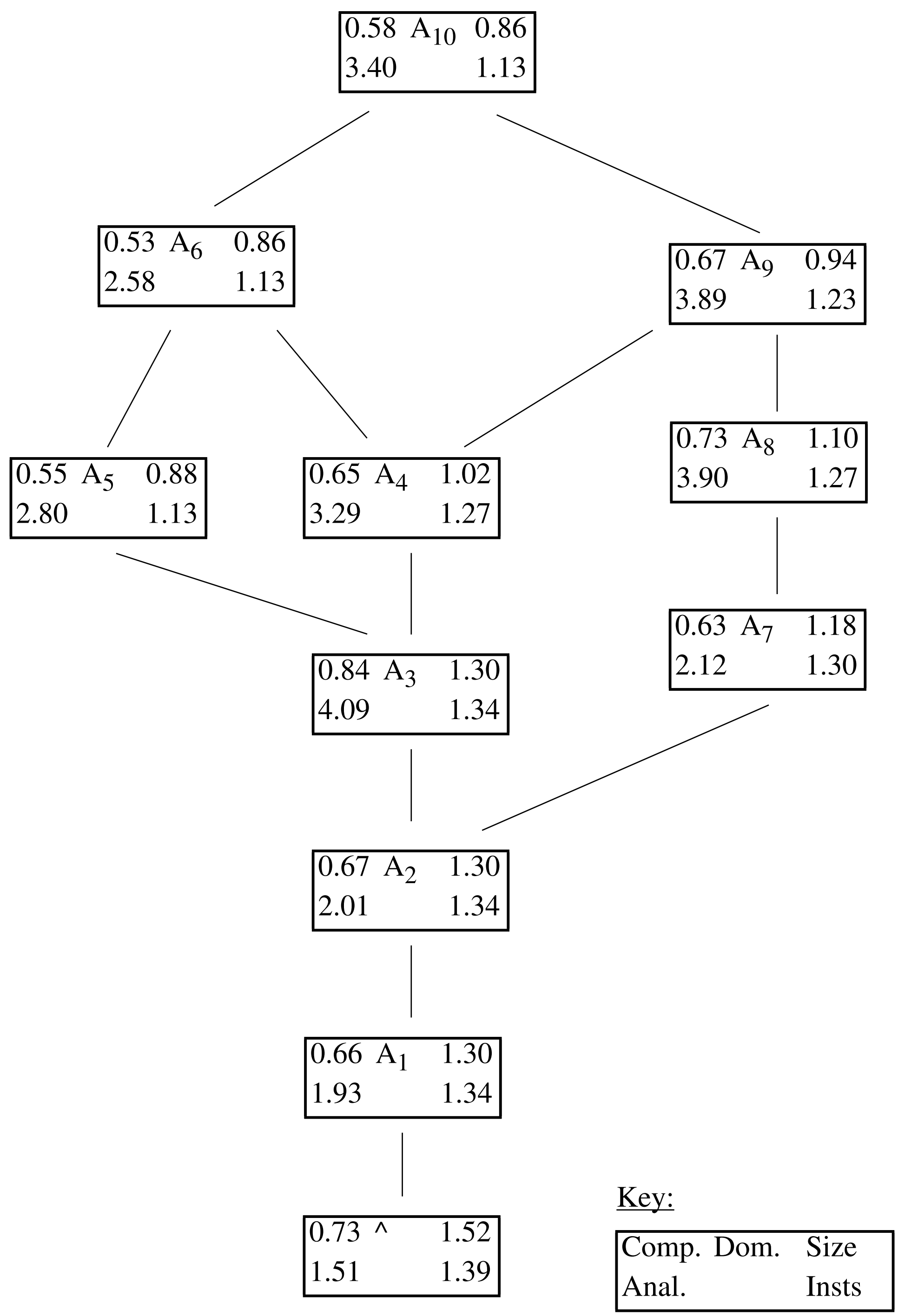


Figure 46: Performance Lattice of Aliasing Analysis

For the most part, compilation time decreases as the aliasing domains get more complex. Again, this is because the added analysis information reduces the amount of code that is generated and that needs to be optimized. Ultimately, this trend will reverse again, as the analysis time dominates the compilation time.

The most precise aliasing analysis (A_{10}) reduces code size by 43% and execution time by 19% over no aliasing analysis. Interestingly enough, this *decreases* the compilation time by 21% (although the analysis time more than doubles). Also, the resulting code is 14% *smaller than Aquarius*. There are some interesting things to note in Figure 46:

- Domain A_2 (sub-domain WC_2) adds little over domain A_1 (sub-domain WC_1) and domain A_3 (sub-domain WC_3) adds nothing over domain A_2. Sub-domain WC_1, which we added simply to provide a very simple aliasing domain, seems almost as useful as the simplest domain proposed previously. Non-transitivity (WC_3) doesn't help by itself.
- Equivalence is an important property. This is apparent from the improvements obtained by going from A_3 to A_5 and from A_9 to A_{10}.
- Domain A_6 provides most of the benefits of aliasing analysis with lower costs than domain A_{10}. For a sequential execution model, strong coupling doesn't seem very important. The added precision of domain A_{10}, which includes combined (weak and strong) coupling and equivalence, may be more important for parallelizing compilers.

6.4.4 Implementation-Dependent Analysis

Next, we add implementation-dependent analysis. This will go beyond the minimal analysis we chose earlier. We will examine this analysis in conjunction with $T_3 \times A_6$, which we determined in the previous selection to be the best implementation independent analysis in our problem domain: sequential execution. The performance results for reference domains R_1 and R_3 and access domains AC_1 through AC_6 are shown in Figure 47. This figure shows the performance measures annotating a lattice of implementation-dependent analysis domains. The lattice orders these domains by their degree of precision. The performance figures shown are compilation time, analysis time, static code size and instructions executed, relative to Aquarius.

Once again, code size and run time decrease while analysis time increases as the domains get more precise. The compilation time is less predictable, but it only varies between 53% and 60%.

The most precise analysis ($AC_6 \times R_3$) reduces code size by 22% and execution time by 31% over minimal implementation-dependent analysis. Analysis time increases by 43% and compilation time by 9%. The resulting code is 36% *smaller* than Aquarius and 28% *faster*.

Bound variable analysis (going from odd AC domains to even ones) adds very little. It only has a 1% impact when going from AC_3 to AC_4. This domain was proposed by Van Roy, who had no true aliasing analysis. The domain implementation is trivial, however. It only has non-bottom elements for the built-in predicates, where it has a table of exit results. If this information were truly propagated through the analysis, it may turn out to be more important, but it is unlikely to be significant if aliasing analysis is being performed.

The impact of reference chain analysis varies significantly, depending on how much initialization analysis is performed. With only new variable analysis (AC_1), the impact of R_3 over R_1 is only 1%. When uninitialized variable analysis is added (AC_3), the impact becomes 6%. With the addition of uninit reg analysis (AC_5), is becomes 8%. The reason for this is that the optimizations performed using the more advanced initialization analysis result in fewer reference chains, which can be detected using advanced reference chain analysis. There are probably more benefits to be gotten using more precise reference chain domains (e.g., R_4).

Initialization analysis is very important. The addition of uninitialized variable analysis (AC_4) decreases execution time by around 20%. Uninit reg analysis (AC_6) removes another 15%. We have been unable to either find or envision initialization analyses more advanced than AC_6. It is possible, however, to improve the precision of AC_6 by improving the abstract operations. For example, the assumption that an uninitialized variable must be at the end of a reference chain can be relaxed if the length

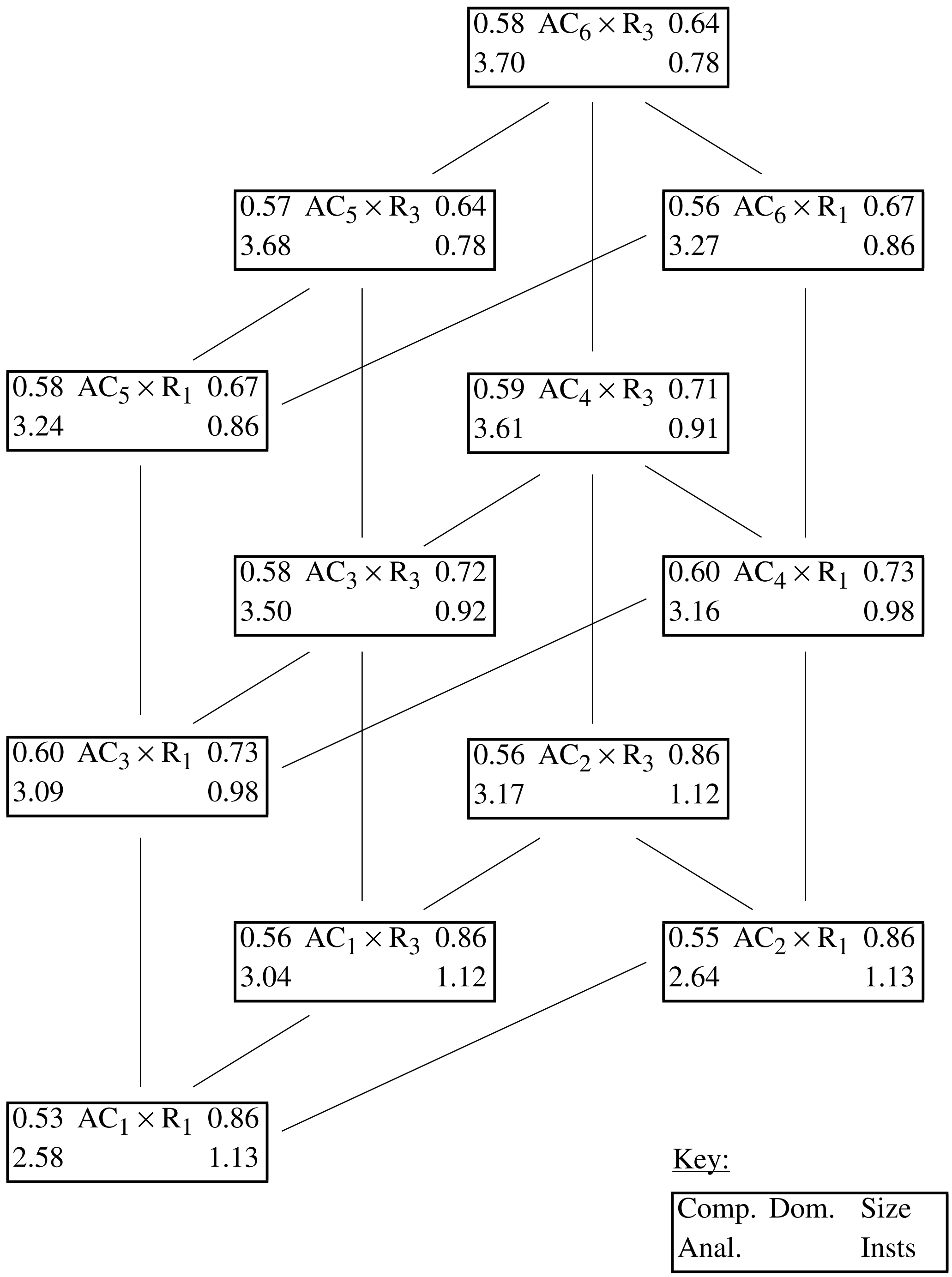


Figure 47: Performance Lattice of Implementation-Dependent Analysis

of the chain is known exactly. Improvements in these operations may improve the precision slightly, but probably only in very specialized cases.

Other types of implementation-dependent analysis (e.g., trailing and local stack analysis) were not implemented in our compiler. There are a number of reasons for this:

- There was insufficient time to implement all of the analyses.
- Many of the remaining analyses required (or at least, suggested) a second analysis pass, operating on the level of intermediate code, in order to provide reasonable results.
- The expected benefits were low (e.g., optimizations based on trailing analysis would improve execution time by at most 1%).

6.4.5 Fully Recursive Types (T_4)

We saved the analysis of fully recursive types until the end because the results are partial. These partial results are given in Table 32. The analysis domain used is $T_4 \times A_5 \times AC_5 \times R_3$. To make the incremental improvement more obvious, this domain is compared with $T_3 \times A_5 \times AC_5 \times R_3$, instead of with the Aquarius domain.

Although we allowed the global analyzer to run for over three CPU hours on an HP 9000/700 workstation, the analysis failed to terminate for almost half of the benchmarks. There are a number of reasons why this might have occurred:

- There may have been errors in my reimplementation of Janssens' type analysis algorithms [41]. In fact, I found several.
- There may have been errors in Janssens' type analysis algorithms or the original implementation. I found one algorithm error which caused non-termination and one implementation error which caused incorrect results.
- The analysis might have terminated if given more time. Although this might be the case for some benchmarks (the zebra benchmark took 3.6 hours for the analysis to complete), it is unlikely that it is the cause for all eleven benchmarks.

For the benchmarks for which we were able to get results, these results are somewhat discouraging. There was a code size reduction in only five of the benchmarks and a reduction in execution time for only three of the benchmarks. One benchmark (fast_mu) got even larger and slower! This is due to the inability of domain T_4 to express the

Table 32: Results of Fully Recursive Type Analysis (Domain T_4)

Benchmark	Compile Time		Analyze Time		Static Size		Run Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	Never finished global analysis.							
nreverse	10.4	4.3	8.2	9.2	488	1.00	5503	1.00
qsort	66.9	16.7	62.7	35.0	780	1.00	6142	1.00
serialise	30.8	5.4	23.5	10.7	1648	0.80	11268	0.98
mu	36.3	4.6	27.8	11.4	4040	0.94	39111	1.00
pri2	8.0	2.0	5.1	3.0	680	1.00	24147	1.00
queens_8	8.7	2.0	5.8	3.1	1188	1.00	37252	1.00
fast_mu	41.2	3.1	30.0	7.4	5908	1.09	40339	1.26
query	11.2	2.4	6.4	6.2	2276	1.00	58757	1.00
press1	Never finished global analysis.							
tak	3.7	1.3	2.4	2.7	228	1.00	842866	1.00
sendmore	6.2	1.1	3.0	1.6	1228	1.00	1055780	1.00
poly_10	194.2	9.3	130.2	23.2	3896	0.76	775802	0.81
zebra	17815.6	659.8	13131.1	1873.2	8192	1.00	2450694	1.00
prover	Never finished global analysis.							
meta_qsort	393.5	17.9	316.7	129.3	4532	0.64	173699	0.89
nand	Never finished global analysis.							
chat_parser	Never finished global analysis.							
browse	Never finished global analysis.							
unify	Never finished global analysis.							
flatten	Never finished global analysis.							
crypt	30.1	2.4	19.8	4.7	4912	0.95	76855	1.00
simple_analyzer	Never finished global analysis.							
reducer	Never finished global analysis.							
boyer	Never finished global analysis.							
Geometric Mean:		5.6		12.6		0.93		0.99

results of tests added to the code during determinism extraction. These are tests like structure(X) and nonvar(X). We added these classes of types to domain T_3, but not to T_4.

The average reduction in code size was 7% and in run time was 1%. This is about the same improvement as we saw when moving from domain T_1 to domain T_3. However, to achieve this required an increase in compilation time by *over five times*. Most of this increase was due to one benchmark: zebra. This benchmark has a large number of variables (78) in the main predicate, all but one of which appear only once. This greatly increases the cost of analysis, due to type propagation from possible variable coupling.

We have a number of suggestions for improving the results of this domain:

- There is little value in having variables in a description before they are encountered in a clause. These variables should be left out until they are encountered.
- Similarly, after some point certain variables no longer contribute anything (e.g., after their last occurrence in the clause). Eliminating these variables from the description will reduce analysis time by making descriptions smaller and less complex.
- Equivalence is captured for points in the type graphs [40]. However, weak coupling and linearity are captured for entire variables. This results in a loss of precision, which can result in expensive computation at analysis time. These properties should be captured for points in the type graph, as well.
- Janssens' type analysis algorithms consist of a number of incremental steps to arrive at the final result. Although this makes the algorithms simpler to understand and prove correct, it makes them more expensive. The amount of time spent in each algorithm should be measured and this information used to select candidates for improvement.
- Type descriptions needed to represent the results of determinism extraction should be added (as we did to domain T_3). This consists of the following node labels, with the obvious meanings: nonvar, structure, ground, ground_structure, and atom.

6.5 Comparison with Parma

In this section, we compare our results with those reported by Taylor [80]. The goal of his Parma compiler, developed about the same time as Van Roy's Aquarius, was to show that high performance execution of Prolog could be obtained on a modern general–purpose architecture (the MIPS R3000). The Parma compiler was not available for measurements. Therefore, the performance measures for Taylor's Parma system come from his dissertation [80]. There are some problems with this:

- The figures may not be exact since Parma's performance figures were given relative to the performance of SICStus Prolog running on a 25 MHz R3230. Parma's performance has to be computed from these relative figures.
- It is not clear what level of support, if any, Parma had for features found in Aquarius, such as floating point numbers and garbage collection.

Table 33 shows the performance measures for Parma. Compilation time was not available. Information was not available for a number of the benchmarks. These results are compared with our results using domain $T_3 \times A_6 \times AC_5 \times R_3$. Our average execution time is very close to that of Parma, although the variance is rather large. Our code size is a dissatisfying 75% larger. There are two reasons for this:

- The Parma compiler was targeted only for the MIPS. It performs a number of optimizing passes over the low-level intermediate code. This code and these optimizations were tailored[3] for efficient MIPS code generation. The Aquarius system, on the other hand, was intended to generate intermediate code which could easily be translated to code for a number of targets.
- The implementation of garbage collection in Aquarius introduces a number of instructions at the beginning of every procedure call. We believe the Parma approach is different and does not have this overhead.

---

3. As opposed to being "Taylored".

Table 33: Performance of Parma Compiler

Benchmark	Analysis time (Sec)	Static code size (Bytes)	Us / Parma	Execution time (insts)	Us / Parma
deriv	2.0	1,630	1.16	1979	1.25
nreverse	0.42	414	1.18	4375	1.26
qsort	0.84	672	1.16	5325	1.15
serialise	1.12	1,240	1.67	5750	2.00
mu	0.80	1,956	2.30	18750	1.90
pri2	0.21	414	1.64	25000	0.97
queens_8	0.35	432	2.75	35350	1.05
query	1.1	948	2.40	78338	0.75
press1	15.0	18,464	1.45	294113	0.19
tak	0.44	192	1.19	757575	1.11
poly_10	3.1	2,504	2.05	1000000	0.95
zebra	1.9	3,178	2.27	3250000	0.38
prover	5.3	5,392	1.45	25000	1.48
nand	20.0	25,000	2.09	391255	1.40
chat_parser	37.0	67,488	1.96	3157900	1.45
browse	1.3	5,052	1.44	16666662	2.09
crypt	1.7	948	5.45	84900	0.90
reducer	9.3	27,352	1.48	1708863	1.09
boyer	15.0	40,352	1.11	76335878	0.43
Geometric mean:			1.75		1.00

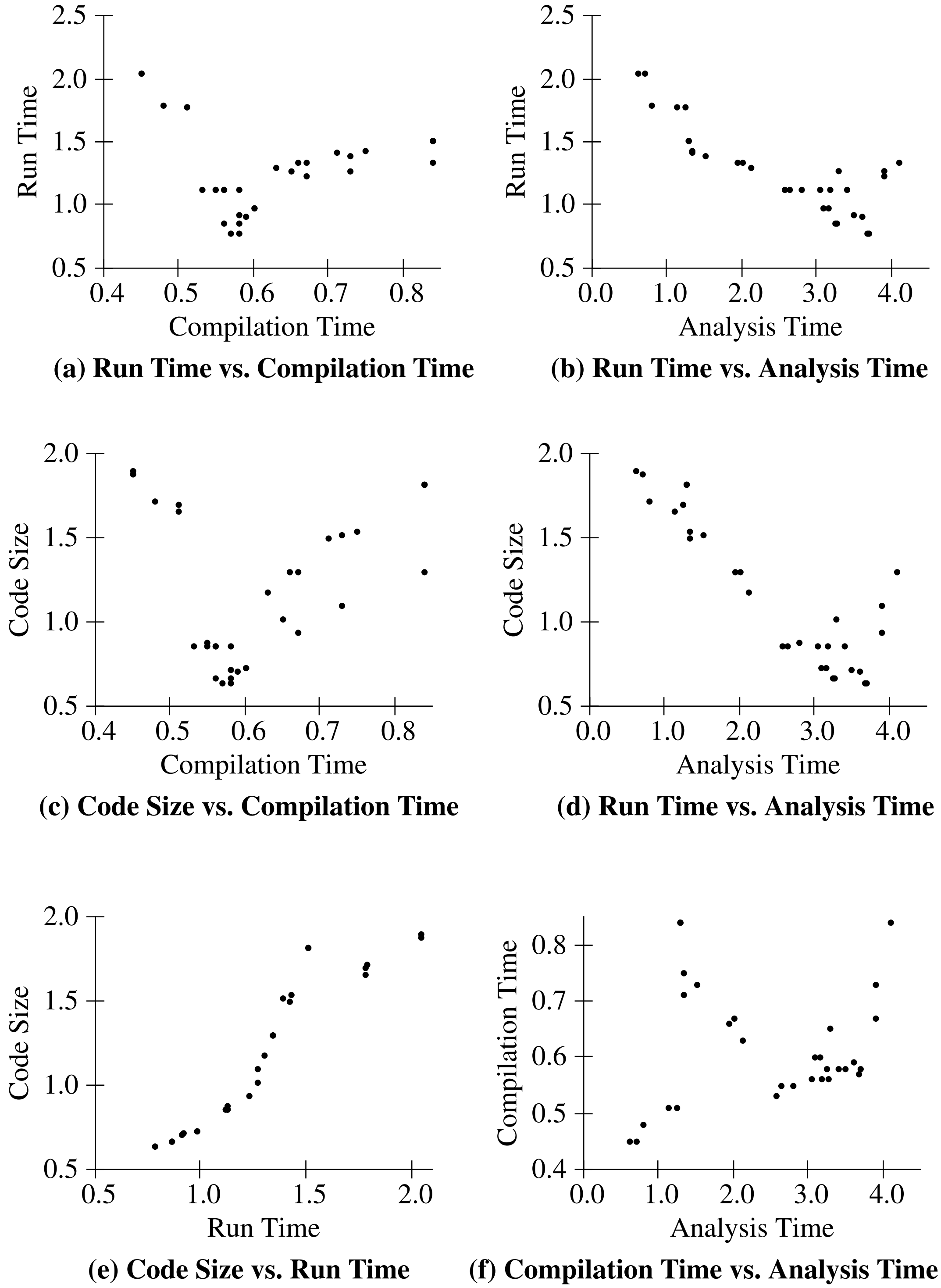
6.6 Summary

In this chapter, we have evaluated the costs and performance gains due to the analysis results obtained from various combinations of abstract domains. These results are summarized in Figure 48.

Figure 48f seems to show compilation time first decreasing as analysis time increases and then increasing. This occurs because compilation time is initially driven by the amount of code which must be generated and optimized. As more analysis is performed, less code is generated (since more cases are eliminated at compile-time), thereby reducing compilation time. Ultimately, however, the major contributor to compilation time becomes analysis time and we see compilation time increasing again.

This trend reversal in compilation time explains why the corellation between our performance measures and compilation time is not very high. Run time and code size do seem to be inversely correlated to analysis time, however. In other words, the more time we spend analyzing, the better our results are (as we would expect).

The highest correlation is between our two performance measures: code size and run time. Although we would expect the programs to both run faster and require less code as analysis removes cases that must be dealt with at run-time, we suspect this trend should eventually reverse, as techniques such as code specialization and in-line expansion are used.



(a) Run Time vs. Compilation Time **(b) Run Time vs. Analysis Time**

(c) Code Size vs. Compilation Time **(d) Run Time vs. Analysis Time**

(e) Code Size vs. Run Time **(f) Compilation Time vs. Analysis Time**

Figure 48: Summary of Performance Evaluation

Chapter 7: Conclusion

This chapter summarizes the research that was performed, the primary results and contributions of that research, and provides directions for future work.

7.1 Contributions

The following sections describe the major contributions of this research.

Test of the Thesis

The primary contribution of any dissertation is the test of the thesis. We set out to show that the right collection of abstract interpretations driving aggressive optimizations could provide a significant improvement in execution performance of Prolog. While more remains to be done, we believe we have demonstrated this thesis.

When compared to Aquarius, we demonstrated a speedup of 28% and code size reduction of 36% while at the same time reducing compilation time by 43%. From a relative standpoint, we demonstrated a performance improvement of 4.4 times and code size reduction of 4.6 times over performing no analysis, while the compilation time only varied by at most a factor of two. This shows that there is a good trade-off between performance (execution time or code size) and cost (compilation time).

Taxonomy of Abstract Analyses

In order to perform more than just an ad-hoc selection of analyses, we began by developing a taxonomy of abstract analyses for Prolog. We populated this taxonomy with numerous abstract domains, at varying levels of complexity and precision. Many of these domains had been suggested, in one form or another, by other researchers. We added some domains to fill out this taxonomy and showed the structure and relationship underlying these domains.

This taxonomy will serve as a tool to help future researchers decide which abstract domains fit their needs or where more work needs to be done in the construction of abstract domains.

Tool for Exploring Compilation and Analysis

In order to evaluate the power of various abstract domains for driving compile-time optimizations, we extended the Aquarius compiler to include an analysis framework. This framework is unique in that it is integrated into all phases of a complete Prolog compiler, allowing the power of abstract interpretation to be explored in terms of its impact on run-time execution for realistic programs.

Refined Abstract Execution Model

We made a number of refinements to the BAM abstract model in the course of this research. The resulting abstract machine is described in Appendix A. The following summarizes the main differences:

- We made numerous changes to simplify the instruction syntax and make instruction operations and operands more regular and orthogonal [32]. This makes the model easier to understand and to translate to target assembly code, and permits peephole optimizations not possible previously.
- We added a mode flag to the deref instruction (like the unify flags). This allows some optimizations for the dereference loop. This has a larger impact on code size than on execution time.
- We made the hash and switch instructions more general. This allows better translation to target code (e.g., multiple switch instructions become one, allowing the tag to be extracted one time).

High-Performance Prolog System

When this work began, the Aquarius Prolog system was only partially functional. Over the last two and a half years I have worked, along with a group spread across the globe, to shake the bugs out of the system. Aquarius Prolog is now able to run on five different platforms and is available to the research community.[1] In addition to being a valuable tool for general Prolog usage, it is hoped that it will stimulate further investigation into high performance Prolog execution.

Fundamental Insights

During any successful research, there are insights gained which are valuable in their own right. Sometimes these are “hindsights”, easily seen once we’re done, although not

---

1. For a copy of the Aquarius Prolog system, send e-mail to `listserv@acal-server.usc.edu`

obvious at the outset. The following list summarizes the insights we gained (or in some instances, regained) throughout this research.

- There is a significant amount of structure to the kinds of dataflow analyses that can be performed on logic programs. Our taxonomy exposes this. Some we were aware of initially. Others, such as the factoring in the aliasing domain into various sub-domains, weren't obvious.
- More precise analysis can actually reduce the overall compilation time. The added analysis can eliminate run-time possibilities, allowing less code to be generated by the compiler. The most striking example of this is the addition of linearity during aliasing analysis (going from A_3 to A_4). The compilation time was reduced by 24%. At the same time, execution time was improved by 7% and code size by 28%.
- The analysis of definite equivalence is an important part of aliasing analysis, especially when combined with type analysis. This property is not useful, by itself, for driving compiler optimizations. It is useful, however, in simplifying other forms of aliasing analysis and in improving the precision of type and mode analysis. The addition of equivalence analysis (going from A_3 to A_5) reduces execution time by 16% and code size by 32%, while reducing compilation time by 35%.
- An abstract domain should contain descriptions which are precise enough to capture the information that is useful for performing optimizations, but usually little else. This can be seen in the mode domains, where domains M_5 and M_6 added more precision over M_4, but this precision bought nothing in terms of added execution performance or reduced code size. The following are reasons for adding more precision than is directly useful for driving optimizations:
    - Added precision is worthwhile if it simplifies the implementation of the abstract operations, making them easier to prove correct.
    - Added precision is worthwhile when it improves the precision of other domains, used for optimizations.

- Implementation-dependent domains are difficult to implement and difficult to prove correct. They must be based on a low-level semantics that expresses the properties present in the abstract domain. The semantics may be based on optimizations that are applied based on analysis results, causing circular reasoning. It is important to think about the analysis independent of the optimizations, and then applying optimizations based on the analysis results.

7.2 Directions for Future Work

Abstract Interpretation

A number of design decisions were made during the development of the analysis framework. Other choices may give interesting results. One such decision is how to deal with multiple call/return descriptions. We chose to generalize and maintain only one call/return pair of descriptions for each predicate. Instead, multiple conditions could be maintained. These could be generalized after the analysis terminates; this might give better results, but probably not much better. A better direction would be to use this information to generate multiple versions of predicates, specialized as to their usage. The problem is selecting useful specializations without an excessive expansion in code size.

Another direction is to rethink the representation of abstract descriptions. When constructing a product of multiple domains, we attempted to keep individual domains as independent as possible. This worked very well until non-flat types were introduced. At this point, almost all other domains could benefit from describing information as detailed as the types that were known for variables. This meant either generalizing this information over a variable (as was done sometimes) or making the domains aware of the structure of the type descriptions (which was also done). A better solution would be to start with types (for which modes are a degenerate case) as the centerpiece and have all other domains provide descriptions in terms of the known points in type descriptions.

As our product domain got larger and larger, analysis time kept increasing (as one would expect), going from 6% to 30% of the compilation time. Not all components of the domain are equally complex, however, and the simpler parts can reach a fixpoint sooner. It should be possible to include the product domain as an integral part of the analysis framework and describe conditions for early termination of parts of the domain. This may provide a significant speedup to the analysis phase. There are also other ways to attack the analysis time issue [50, 77].

Abstract Domains

Some work remains to be done in exploring the taxonomy of abstract domains presented here.

- Fully recursive types (T_4) deserve more investigation to see if they have a substantial payback, as well as to find ways to reduce the analysis time.
- As more interest leans towards parallel models and languages, aliasing information becomes increasingly important. Current domains are either too imprecise or too expensive on programs with many variables. Domain A_{10} adds equivalence to a very precise aliasing domain. We have only partially integrated the two sub-domains. A fuller integration in the expression of the abstract operations should result in a more expressive, less expensive aliasing domain.
- There may be gains in other domains, such as those for reference chains, trailing, and the various predicate-level domain. Except for reference chains, the remaining benefits in these areas are probably small, however. Domain R_4 should provide a worthwhile improvement. Based on the amount of time still spent dereferencing, we estimate an improvement between 5% and 10% is obtainable.

Tool Improvements

We hope that the tool we have developed for exploring abstract interpretation and Prolog compilation is useful to other researchers. There are two areas where it could be improved.

Aquarius was developed as a testbed for Prolog compilation ideas and was therefore designed with a number of very general mechanisms. Because of this, it can be very slow (sometimes taking an hour to compile a single benchmark). This limited the number and size of benchmarks we could use and the number of experiments we could perform. Tan and Lin report a decrease in analysis time by two orders of magnitude [77]. They claim this is due to a technique called *abstract compilation*, in which the program to be analyzed is first compiled (into a WAM-like program) and the compiled program is then used to drive the analyzer.[2]

Providing the abstract operations for an abstract domain can be tedious. Although we have provided some utilities to make this easier, more could be done. For example, the order relationship can be converted automatically into the least upper bound operation for many domains, using techniques like partial evaluation. It would be interesting to see if a concise formal description of the operations could be converted automatically into an implementation.

Prolog Compilation

There are numerous techniques for compiling Prolog into efficient target code. We have explored many of them, but weren't able to examine them all. In addition to investigating the power of individual techniques, the order in which they should be applied is not clear. Optimizing Prolog compilers tend to perform well only when given code written in the style they expect. Perhaps there is some "optimal" ordering of transformations that will make this process less dependent on the programming style.

Compile-time garbage collection should be a very useful optimization. In addition to reducing dynamic memory requirements, it can reduce the time spent reading and writing memory [35]. The tools are now in place to give it the scrutiny it deserves.

Recursive types are another area that deserve further study. Unfortunately, the domain we had made different assumptions than our compiler (e.g., the domain required unraveled calls whereas the compiler generated more efficient code when this wasn't the case). This mismatch introduced inefficiencies. This is related to the previous issue of programming style. The compiler should be able to handle differing assumptions in the abstract domains. Once this has been fixed, the true power of recursive types should be seen.

The abstract execution model also hampered our search for performance. Taylor targeted his compiler for one machine, the MIPS. This is a major reason for his higher

---

2. Most of the improvement comes from the use of C rather than Prolog for implementing the analyzer. We hope someday to see such complex programs running as efficiently in either language.

performance. The BAM model, while a good model for targeting to many machines, is not ideal for any one machine. A lower level model, like that used by Taylor, might not suffer from this. For example, the BAM reloads argument registers immediately upon entering a choice during backtracking, often to move these values to different registers or into an environment. Instead, the values could be accessed directly from the choicepoint when needed, as if this were a read-only environment.

Language Directions

This work is not limited to just sequential execution of Prolog. The ideas should be applicable to other, related languages. Here are some directions in the language area:

- As mentioned previously, the Aquarius compiler can be rather slow. The capability to compile separate modules of code would be helpful here. There are a number of interesting issues in the interaction between separate compilation and abstract interpretation. Dean Jacobs, Thomas Pabst and I explored these somewhat, integrating a module system to the Aquarius compiler [39]. Some of the issues are deciding what a user should specify about the interface (should he provide abstract information about the expected calling/return conditions or should this be derived and maintained automatically).
- There are interesting variants of and extensions to Prolog that may have new abstract domains to be discovered. Constraint logic programming is an example of this, where domains might capture definiteness and freeness of variables [57].
- Another major direction should be towards automatic parallelization of Prolog and investigation into concurrent logic languages. These languages have their own set of modes, for example Inbound, Outbound, Infree and Outfree [75].

7.3 Summary

In this research, we set out to show that a significant performance increase could be obtained in Prolog execution through a good selection of global data flow analyses driving compiler optimizations. To demonstrate this, we augmented an optimizing Prolog compiler with an abstract interpretation framework and integrated the code generator with this framework to allow it to make use of the results. We then developed a taxonomy of abstract analyses and populated this taxonomy with families of abstract domains. Exploring this collection, we demonstrated and measured the value of abstract interpretation as a tool for Prolog compilation.

Although the results were not the revolutionary jump in performance we expected, we did demonstrate an evolutionary increase in performance of 28% and code size reduction of 36% over the original compiler. We showed a wide range of performance and code size figures and the costs associated with achieving these. The analysis time varied from 39% of Aquarius' analysis time to almost four times as much as in Aquarius. The compilation time varied between 45% and 84% of Aquarius', with it decreasing sometimes when the analysis time increased.

References

[1] F. Allen and J. Cocke. A Program Data Flow Analysis Procedure. In *Comm. of the ACM*, Vol 19 #3, pp. 137-147, March 1976.

[2] F. Allen. Control Flow Analysis. In *ACM SIGPLAN Notices 5:7*, pp. 1-19, January 1970.

[3] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley. June 1974.

[4] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley. March 1986.

[5] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press. 1991.

[6] A. Bansal and L. Sterling. An Abstract Interpretation Scheme for Logic Programs based on Type Expressions. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp. 422-429, November 1988.

[7] R. Barbuti and M. Martelli. A tool to check the non-floundering logic programs and goals. In *Lecture Notes in Computer Science, Vol. 348 (Programming Languages Implementation and Logic Programming International Workshop PLILP '99)*, pp. 58-67. May 1988.

[8] M. Bruynooghe, G. Janssens, Alain Callebaut, and B. Demoen. Abstract Interpretation: Towards the Global Optimisation of Prolog Programs. In *1987 IEEE Symposium on Logic Programming*, pp. 192-204, 1987.

[9] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inferencing. In *Logic Programming: Proceedings of the 5th International Conference*, pp. 669-683, August 1988.

[10] M. Bruynooghe. *A Framework for the Abstract Interpretation of Logic Programs*. Report CW 62, Dept. of Computer Science, K.U. Leuven, October 1987.

[11] J. Chang and A. Despain. Semi-Intelligent Backtracking of Prolog Based on A Static Data Dependency Analysis. In *Logic Programming conference*, July 1985.

[12] W. Citrin. *Parallel Unification Scheduling in Prolog*. Ph.D. Thesis, University of California, Berkeley, Report UCB/CSD #88/415, 1988.

[13] J. Cocke. Global Common Subexpression Elimination. In *ACM SIGPLAN Notices 5:7*, pp. 20-24, January 1970.

[14] C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation for Concurrent Logic Languages. In *Proceedings of the North American Conference on Logic Programming '90*, pp. 215-232, October 1990.

[15] M. Codish, D. Dams, and E. Yardeni. Bottom-up Abstract Interpretation of Logic Programs. Technical Report CS90-24. The Weizmann Institute of Science. October 1990.

[16] A. Cortesi, G. File, and W. Winsborough. Comparison of Abstract Interpretations. Internal report 14 - 18.11.1991. Departmento di Matematica. November 1991.

[17] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th ACM POPL*, pp. 238-252, June 1977.

[18] P. Cousot and R. Cousot. *Abstract Interpretation and Application to Logic Programs*. Research Report 92-12, LIENS Laboratoire d'Informatique de l'Ecole Normale Superieure, June 1992.

[19] J. Crammond. An Execution Model for Committed-Choice Non-Deterministic Languages. In *Proceedings of the 1986 Symposium on Logic Programming*, pp. 148-158, September 1986.

[20] J. Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In *Proceedings of the North American Conference on Logic Programming '90*, pp. 642-657, October 1990.

[21] S. Debray. Register Allocation in a Prolog Machine. In *Proceedings of the IEEE 1986 Symposium on Logic Programming*, pp. 267-275, September 1986.

[22] S. Debray. Unfold/Fold Transformations and Loop Optimization of Logic Programs. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 297-307, June 1988.

[23] S. Debray. A Simple Code Improvement Scheme for Prolog. In *Logic Programming: Proceedings of the 6th International Conference*, pp. 17-32, June 1989.

[24] S. Debray. Flow Analysis of Dynamic Logic Programs. In *Journal of Logic Programming*, Vol. 1989:7, pp. 149-176. 1989.

[25] S. Debray. The Mythical Free Lunch (Notes on the Complexity/Precision Trade-off in Dataflow Analysis of Logic Programs) (unpublished). March 1991.

[26] S. Debray and D. S. Warren. Detection and Optimization of Functional Computations in Prolog. In *Proceedings of the Third International Conference on Logic Programming*, pp. 490-504. July 1986.

[27] S. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. In *IEEE 1986 Symposium on Logic Programming*, pp. 78-88, September 1986.

[28] D. De Schreye and M. Bruynooghe. An Application of Abstract Interpretation in Source Level Program Transformation. In *Lecture Notes in Computer Science*, Vol. 348 (Programming Languages Implementation and Logic Programming International Workshop PLILP '88) pp. 35-57. May 1988.

[29] S. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Proceedings of the 4th International Symposium on Logic Programming*, pp. 264-272. 1987.

[30] J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialisation. In *Logic Programming: Proceedings of the 7th International Conference* pp. 732-746. MIT Press. June 1990.

[31] T. Hickey and S. Mudambi. Global Compilation of Prolog. In *Journal of Logic Programming*, Vol. 7, pp. 193-230, 1989.

[32] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers. 1990.

[33] M. Hermenegildo. An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs. In *Proceedings of the Third International Conference on Logic Programming*, pp. 25-39. July 1986.

[34] B. Holmer, et al. Fast Prolog with an Extended General Purpose Architecture. In *The 17th Annual International Symposium on Computer Architecture Conference Proceedings*, pp. 282-291, June 1990.

[35] G. Gudjonsson and W. Winsborough. *Update in Place: Overview of the Siva Project*. Technical Report CS-93-11, Pennsylvania State University, May 1993.

[36] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *Logic Programming: Proceedings of the North American Conference 1989*, pp. 154-165, October 1989.

[37] D. Jacobs. *Constructing and Optimizing Multi-Directional Logic Programs*. (Unpublished). March 1991.

[38] D. Jacobs. *A Framework for the Abstract Interpretation of Logic Programs*. (Unpublished). October 1991.

[39] D. Jacobs, T. Pabst and T. Getzinger. *Modules and the Compile-Time Optimization of Prolog*. (Unpublished). June 1992.

[40] G. Janssens and M. Bruynooghe. *Deriving descriptions of possible values of program variables by means of abstract interpretation.* Report CW 107, Department of Computer Science, K. U. Leuven. March 1990.

[41] G. Janssens and M. Bruynooghe. *Deriving descriptions of possible values of program variables by means of abstract interpretation: definitions and proofs.* Report CW 108, Department of Computer Science, K. U. Leuven. April 1990.

[42] N. Jones and H. Søndergaard. *A Semantics-Based Framework for the Abstract Interpretation of Prolog*. Report No. 86/14, Institute of Datalogy, University of Copenhagen, 1986.

[43] R. Kemp and G. Ringwood. An Algebraic Framework for Abstract Interpretation of Definite Programs. In *Proceedings of the North American Conference on Logic Programming '90*, pp. 516-530, October 1990.

[44] F. Kluzniak. Type Synthesis for Ground Prolog. In *Logic Programming: Proceedings of the 4th International Conference*, pp. 788-816, May 1987.

[45] F. Kluzniak. Compile Time Garbage Collection for Ground Prolog. In *Logic Programming: Proceedings of the 5th International Conference*, pp. 1490-1505, 1988.

[46] R. Kowalski. *Logic for Problem Solving*, Elsevier North-Holland, 1979.

[47] A. Krall and T. Berger. *A Prolog Compiler based on the VAM*. (Unpublished) 1992.

[48] A. Langen. *Advanced Techniques for Approximating Variable Aliasing in Logic Programs*. PhD Thesis. University of Southern California. December 1990.

[49] B. Le Charlier and P. Van Hentenryck. *Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog*. Technical Report No. CS-91-55, Brown University, August 1991.

[50] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. *Efficient and Accurate Algorithms for the Abstract Interpretation of Prolog Programs*. Research Paper No. RP-90/9, University of Namur, Belgium, August 1990.

[51] B. Le Charlier and P. Van Hentenryck. *Reexecution in Abstract Interpretation of Prolog*. Technical Report No. CS-92-12, Brown University. March 1992.

[52] J. Lloyd. *Foundations of Logic Programming*, Springer-Verlag 1987.

[53] A. Marien and B. Demoen. On the Management of Choicepoint and Environment Frames in the WAM. In *Logic Programming: Proceedings of the North American Conference 1989*, pp. 1030-1047, October 1989.

[54] A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The impact of abstract interpretation: an experiment in code generation. In *Logic Programming: Proceedings of the 6th International Conference*, pp. 33-47, June 1989.

[55] K. Marriott and H. Søndergaard. Bottom-up Abstract Interpretation of Logic Programs. In *Logic Programming: Proceedings of the 5th International Conference and Symposium*, pp. 733-748. 1988.

[56] K. Marriott and H. Søndergaard. On Prolog and the Occur Check Problem. In *ACM SIGPLAN Notices 24:5*, pp. 76-82, May 1989.

[57] K. Marriott and H. Søndergaard. Analysis of Constraint Logic Programs. In *Proceedings of the North American Conference on Logic Programming '90*, pp. 531-547, October 1990.

[58] K. Marriott, H. Sondergaard, and P. Dart. A Characterization of Non-Floundering Logic Programs. In *Proceedings of the North American Conference on Logic Programming '90*, pp. 661-680. 1990.

[59] H. Mannila and E. Ukkonen. Flow Analysis of Prolog Programs. In *1987 IEEE Symposium on Logic Programming*, pp. 205-214, 1987.

[60] M. Meier. Recursion vs. Iteration in Prolog. In *Proceedings of the Eighth International Conference on Logic Programming*, pp. 157-169, June 1991.

[61] C. Mellish. *Automatic Generation of Mode Declarations for Prolog Programs (Draft).* DAI Research Paper 163, Dept. of Artificial Intelligence, University of Edinburgh, August 1981.

[62] C. Mellish. Abstract Interpretation of Prolog Programs. In *Proceedings of the Third International Conference on Logic Programming*, pp. 463-474, July 1986.

[63] C. Mellish. Some Global Optimizations for a Prolog Compiler. In *Journal of Logic Programming, Vol 2*, pp. 43-66, April 1985.

[64] F. Morris. On a Comparison of Garbage Collection Techniques. In *Communications of the ACM*, 22(10), page 571, October 1979.

[65] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information Through Abstract Interpretation. In *Proceedings of the North American Conference on Logic Programming '89*, pp. 166-185, August 1989.

[66] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs. In *Logic Programming: Proceedings of the 7th International Conference*, pp. 747-762. MIT Press. June 1990.

[67] U. Nilsson. Towards a Framework for the Abstract Interpretation of Logic Programs. In *Lecture Notes in Computer Science, Vol. 348 (Programming Languages Implementation and Logic Programming International Workshop '88)*, pp. 68-82, May 1988.

[68] R. O'Keefe. Finite Fixed-Point Problems. In *Logic Programming: Proceedings of the 4th International Conference*, pp. 729-743. May 1987.

[69] T. Pabst, Dataflow Analysis and Modular Logic Programs. Diplomarbeit, TU-Berlin, November 1991.

[70] D. Plaisted. The Occur-Check Problem in Prolog. In *Proceedings of the 1984 International Symposium on Logic Programming*, pp. 272-280, February 1984.

[71] C. Ponder, P. McGeer, and A. Ng. Are Applicative Languages Inefficient? In *ACM SIGPLAN Notices 23:6*, pp. 135-139, June 1988.

[72] G. Ringwood. SLD: A Folk Acronym? In *ACM SIGPLAN Notices 24:5*, pp. 71-75, May 1989.

[73] P. Schnupp and L. Bernhard. *Productive Prolog Programming*. Prentice Hall, 1987.

[74] H. Seki and K. Furukawa. Notes on Transformation Techniques For Generate and Test Logic Programs. In *1987 IEEE Symposium on Logic Programming*, pp. 215-223, 1987.

[75] Z. Somogyi. A system of precise modes for logic programs. In *Logic Programming: Proceedings of the 4th International Conference*, pp. 769-787. May 1987..

[76] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proceedings of the 3rd International Conference on Logic Programming*, pp. 84-98. 1986.

[77] J. Tan and I. Lin. *Compiling Dataflow Analysis of Logic Programs.* (Unpublished) 1992.

[78] A. Taylor. Removal of Dereferencing and Trailing in Prolog Compilation. In *Logic Programming: Proceedings of the 6th International Conference*, pp. 48-60, June 1989.

[79] A. Taylor. LIPS on a MIPS: Results from a Prolog Compiler for a RISC. In *Logic Programming: Proceedings of the 7th International Conference,* June 1990.

[80] A. Taylor. *High Performance Prolog Implementation.* PhD Thesis. University of Sydney, June 1991.

[81] A. Thayes. *From Standard Logic to Logic Programming*. John Wiley & Sons Ltd. 1988.

[82] H. Touati and A. Despain. An empirical study of the Warren Abstract Machine. In *Proceedings of the 1987 Symposium on Logic Programming*, pp. 114-124. San Francisco 1987.

[83] P. Van Roy. A Useful Extension to Prolog's Definite Clause Grammar Notation. In *ACM SIGPLAN Notices, Volume 24, No. 11*, pp. 132-134. November 1989.

[84] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?*. Ph.D. Thesis, University of California, Berkeley, Report UCB/CSD #90/600, December 1990.

[85] P. Van Roy, B. Demoen, and Y. D. Willems. Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection, and Determinism. In *Lecture Notes in Computer Science, Vol. 250 (TAPSOFT '87)*, pp. 111-125, March 1987.

[86] P. Van Roy and A. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. In *Computer*, pp. 54-68, January 1992.

[87] A. Wærn. An Implementation Technique for the Abstract Interpretation of Prolog. In *Logic Programming: Proceedings of the 5th International Conference*, pp. 700-710, August 1988.

[88] D. Warren. *An Abstract Prolog Instruction Set*. Technical Note 309. SRI International Artificial Intelligence Center. October 1983.

[89] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis. In *International Conference and Symposium on Logic Programming*, pp. 684-699. August 1988.

[90] W. Winsborough and A. Wærn. Transparent And-Parallelism in the Presence of Shared Free Variables. In *Logic Programming: Proceedings of the 5th International Conference*, pp. 749-764, August 1988.

Appendix A: Semantics of the Berkeley Abstract Machine (BAM)

A.1 Introduction

This appendix describes the semantics of the Berkeley Abstract Machine (BAM). Much of this information is derived from [84], but it has been updated to include refinements we have made since that time.

The BAM is an abstract machine meant for describing compiled Prolog. BAM instructions have a finer grain than previous Prolog execution models, notably the Warren Abstract Machine (WAM) [88]. This provides more opportunities for optimization. BAM is meant to be an intermediate code, which can easily be translated to a target machine's instruction set. This appendix provides some hints about this translation and about trade-offs available when implementing this execution model.

A.1.1 Implementation Choices

There are a number of design and implementation choices that affect this execution model. Some of these affect the semantic descriptions that follow. Others are lower level, implementation issues. We summarize these choices in the following sections. More details are provided in the body of the appendix.

Destructive Assignment

Pure Prolog is a write-once language. For the sake of efficiency, Aquarius Prolog has included some optional non-logical features: backtrackable and stepped destructive assignment. These features are described in detail in the Aquarius Prolog User Manual.

To support these features, two registers (r(sda_queue) and r(sda_queue_next)), one memory section (the SDA queue), and three instructions (trail_bda, quue_sda, and step) must be added. In addition, the trail, choice, and fail instructions must be changed. We have described the BAM as if these features are included, with notes on how they could be excluded if not needed.

Choicepoint Register Saving

When a choicepoint is created, the “active” argument registers must be saved. This is the set of registers that will be used in at least one subsequent choice. For example, if the first choice needs registers r(0) and r(3), the second needs r(0) and r(2), and the third needs r(1) and r(2), the only registers that must be saved are r(0)-r(2) (r(3) doesn’t need to be saved, since it won't ever be restored). There are a number of ways this saving can be done:

- All argument registers can be saved, regardless of whether or not they are needed. This can make the code much smaller, since the saving can be done in a called procedure and the restoring can be done in the failure routine (see the fail instruction), but, since the BAM has over 255 argument registers, this does not seem practical. This is similar to the approach taken in the WAM. For the WAM, choicepoints are only created immediately following predicate entry, and therefore only need to save argument registers up to the arity of the entered predicate.
- All argument registers up to the highest numbered register listed in the choice instruction can be saved. This simplifies the code, somewhat, and can help on architectures with double word loads/stores.
- Only those registers listed can be saved. This approach generates the smallest choicepoints and is usually the fastest. This is the approach described here. In some circumstances, on machines with double word loads/stores, it can actually be slower than the previous approach.

Data Representation

There are a number of issues involved in representing data, such as the size of a word, the size, position, and values of tags, and even which tags are available. More details are provided in a later section, dealing with data representation in detail.

Hash Table Representation

There are several different ways to encode hash tables. Here we describe a few:

- The hash table could be stored and searched as a linear list of key, value pairs. This is the most compact, and also the slowest.
- The hash table could be stored as a more traditional, fixed-length table of hash buckets, with each bucket pointing to a linear list of key, value pairs for that bucket.
- The hash table could be stored as an almost fixed-length table of hash buckets, each containing a key, value pair, with collisions handled by storing in the next

open bucket. This can cause the table to go beyond its nominal length (2N-1 in the worst case), but in practice it will not extend too far. This is the implementation approach we take, but the model does not describe any particular approach.

Similarly, there are a number of different ways to compute hash functions. Again, we don't describe any one approach as part of the semantics of the BAM. However, since this operation should be fast, we recommend the hash function be something simple, like selecting a fixed number of bits at the bottom of the data word (excluding the tag). This can be computed with one shift and one mask operation. If the tag is stored at the bottom of the word and the tag length is variable (see section A.2.1), this may require testing the tag first. Still, this gives a hash index that should do a reasonably good job of bringing the search time close to constant.

Local stack

Memory in the BAM is partitioned into a number of sections (see section A.2.3). Ideally, these memory sections are totally independent memory spaces. In reality, they tend to be allocated contiguously in a single memory space. Therefore, when one is almost full, the others may need to be moved, to make more room. In addition, each memory section has at least one register (the next location, or top pointer) associated with it.

Two of these sections, the environment stack and the choicepoint stack, can be combined fairly easily into a single section, called the local stack. In order to do this, we need to know where the top of the local stack is. This can be done by maintaining a separate top of local stack pointer, or by taking the maximum of the current environment and choicepoint pointers when we need to add information to this stack.

We have described the BAM as having separate environment and choicepoint stacks, but implementations typically combine them and use the second approach given for maintaining the top of stack. The changes needed to implement this approach are described where appropriate throughout the body of this appendix.

Trail check register

When backtracking occurs, all variables bound since the choicepoint was created must be returned to their unbound state (and backtrackable destructive assignments must be backtracked). This is done by recording information in a memory section known as the trail stack, or simply the *trail*. The trail can record all bindings, only bindings of unbound variables, or only bindings of variables created before the latest choicepoint was created.

The first approach can require a rather large trail, and isn't a useful simplification; it is usually known when binding if the old value is unbound (and may need trailing) or not. Therefore, the last two choices are the only reasonable ones. To implement the third choice, a copy of the heap pointer must be saved when the choicepoint is created and variable bindings compared against this saved copy. The heap pointer must be saved in the choicepoint, regardless, so this memory copy can be used, or a register can be used, for efficiency. We have included this register (r(hb)) in our description of the BAM, with notes describing the impacts of alternate choices. This seems to be the best choice, but should be studied to verify this.

Double word alignment

Some architectures (e.g., the SPARC and the VLSI-BAM) have double word load and store operations. To make efficient use of these, the compiler should try to keep data objects aligned in memory.

For the heap, this is done by adding appropriate pad instructions to ensure the heap is aligned when adding structures and lists and when calling another procedure or returning (so we know the alignment state of the stack at predicate entry and across procedure calls). Variables are stored in a single word and can appear as arguments in a compound term. Therefore, since they won't be aligned in general, there is no reason to align variables created alone on the stack (besides, when reading or writing a variable, we're only accessing a single word, anyway). In practice, however, the Aquarius compiler does align stand-alone variables when alignment is requested, so the heap is always kept aligned after adding anything. If alignment isn't important, the pad instructions can be treated as no-ops.

For environments and choicepoints, alignment is done by adding a pad word. In our descriptions of these memory sections, we show where this pad word should go, if it is needed.

Dealing with the trail is easy. If destructive assignment is supported, all bindings added to the trail are two words long, and therefore the trail is always properly aligned. Otherwise, the trail consists of the addresses of bound variables (one word each) and doesn't need to be aligned.

A.2 Data Organization

A.2.1 Data Representation

All data in the BAM is represented in words. The length of the words and the physical representation of data within a word is implementation-dependent, but for a reasonable implementation, the length of a word should be at least 32 bits. There are two basic data formats: tagged and untagged. Untagged values are used for machine integers and memory addresses. The entire word is used for these values. Tagged values consist of a data tag, identifying the type of data, and a data value, with a tag-dependent interpretation. Figure 49 illustrates the various data representations. Table 34 provides additional details about the tags and their interpretation (along with recommended tag values).

There are two logical places for the tag within a word: the top and the bottom. Placing the tag at the top, however, is only recommended if the architecture has support for these tags (e.g., the VLSI-BAM). Without this support, tag testing, removal, and addition becomes more difficult, since it is not always easy to manipulate the most significant bits of a word without affecting the others.

Therefore, for machines without tag support, we recommend placing the tags in the least significant bits. All tags could be made the same length (three bits for six tags), or they can vary.

For example, most architectures allow an offset to be provided on memory accesses. This can be used as a bias, to remove the pointer tags (tlst, tstr, and tvar) without an additional operation to mask off the tag (when the tag is known). Also, since most architectures are byte addressed and the BAM is word addressed, this leaves the bottom

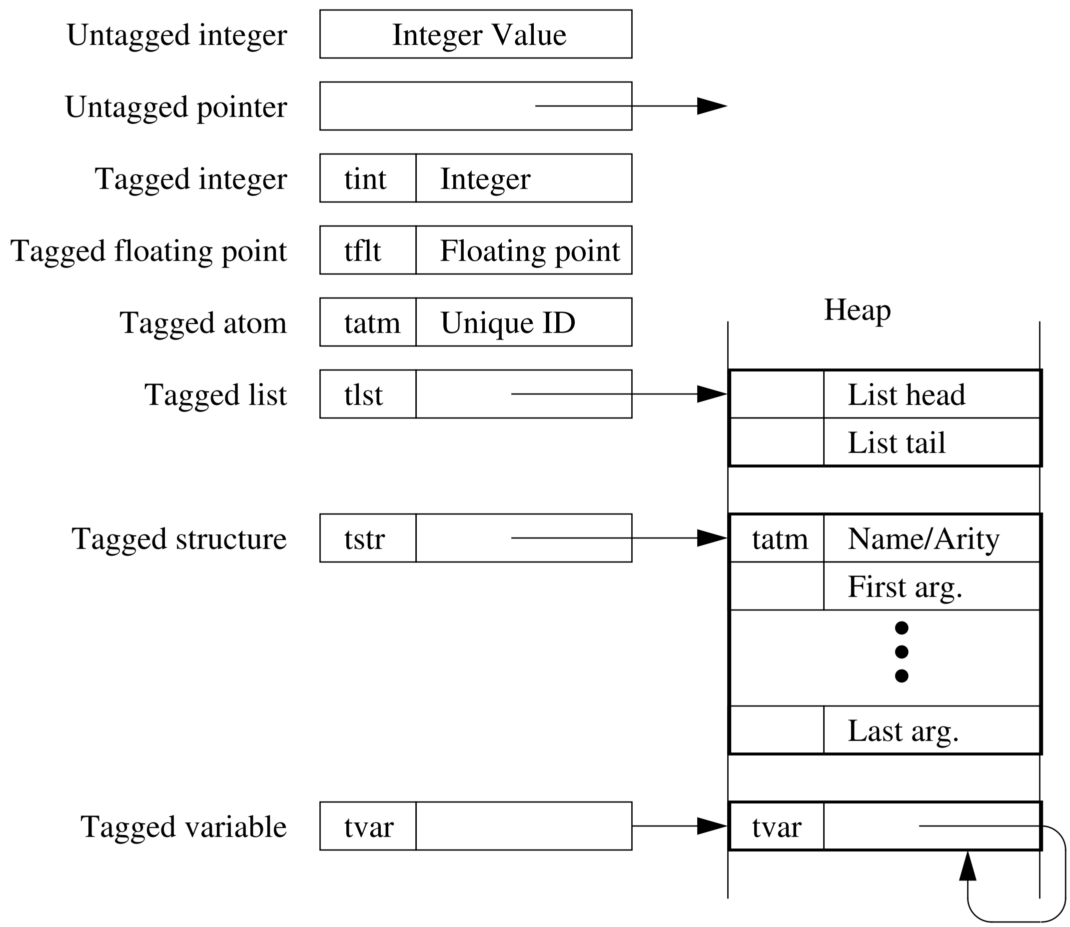


Figure 49: BAM Data Representation

two bits of an address unused. We typically use these for the pointer tags, with tvar having a value of zero, since this is easiest to test for and is probably the most common.

The fourth value for the bottom two bits identifies the atomic tags (tatm, tflt, and tint), with additional bits to differentiate these cases. We could just use two more bits and have an unused bit combination. However, since most floating point values expect the full data word, it seems undesirable to take too many bits away from floating point. Therefore, we typically assign two tags to tflt (actually, it has a unique three bit tag), and leave the last two for tatm and tint.

Table 34: Suggested Tagged Values

Tag	Value	Description
tvar	xx00	This identifies a reference to another value. The data value is a pointer to the actual value. When the pointer is self-referential, the value is unbound (variable).
tlst	xx01	This identifies a list. The data value is a pointer to two tagged words in the heap, containing the head and tail of the list, respectively.
tstr	xx10	This identifies a structure. The data value is a pointer to the structure, in the heap. The first word described the name and arity of the structure (see tatm). The following words contain the tagged values of the structure arguments.
tflt	x011	This identifies a floating point value. The data value contains the floating point number.
tatm	0111	This identifies a Prolog atom. The data value is a unique integer value identifying the atom. In most implementations, it is an index into the atom name table. This tag is also used for the functor of a structure. The data value identifies both the name and arity of the structure.
tint	1111	This identifies an integer value. The data value contains the integer.

There is another possibility for integers that is worth mentioning. We could assign two tags to integers, tpos for nonnegative integers, and tneg for negative integers. By using all zeros for tpos and all ones for tneg, and placing these at the top of the word, a tagged integer would look no different than an untagged integer, and in fact the tag would serve as a “sign bit”. The only difference in dealing with tagged integers is overflow detection. This was the approach taken in the VLSI-BAM, but for machines without tag support, it conflicts with the assignments given based on the previous descriptions. Therefore, we describe the BAM as having a single integer tag.

A.2.2 BAM Registers

Table 35 describes the registers in the BAM. The table provides the name, format (tagged, untagged, or either), and description of each register.

Table 35: BAM Registers

Register	Format	Description
r(e)	Untagged	Top of the environment stack. This points to the current environment.
r(b)	Untagged	Top of the choice point stack. This points to the current choice point.
r(h)	Untagged	Top of the heap (global stack).
r(hb)	Untagged	Top of the heap when the current choice point was created.
r(tr)	Untagged	Top of the trail stack.
r(sda_queue)	Untagged	Bottom of the SDA queue.
r(sda_queue_next)	Untagged	Top of the SDA queue.
r(pc)	Untagged	Program counter.
r(cp)	Untagged	Continuation pointer (return address).
r(tmp_cp)	Untagged	Continuation pointer for simple procedures (see simple_procedure instruction).
r(0) ... r(N)	Either	Argument and temporary registers.
p(0) ... p(N)	Either	Permanent variables (p(i) is equivalent to m(r(e)-i-3))

A.2.3 BAM Memory Sections

Figure 50 illustrates the various memory sections in the BAM. These memory sections are described in the following sections of the document. Two of these, the environment and choicepoint stacks, can actually be (and usually are) combined into a single memory section, the local stack.

The registers appearing in Figure 50 will always point into the indicated memory sections.

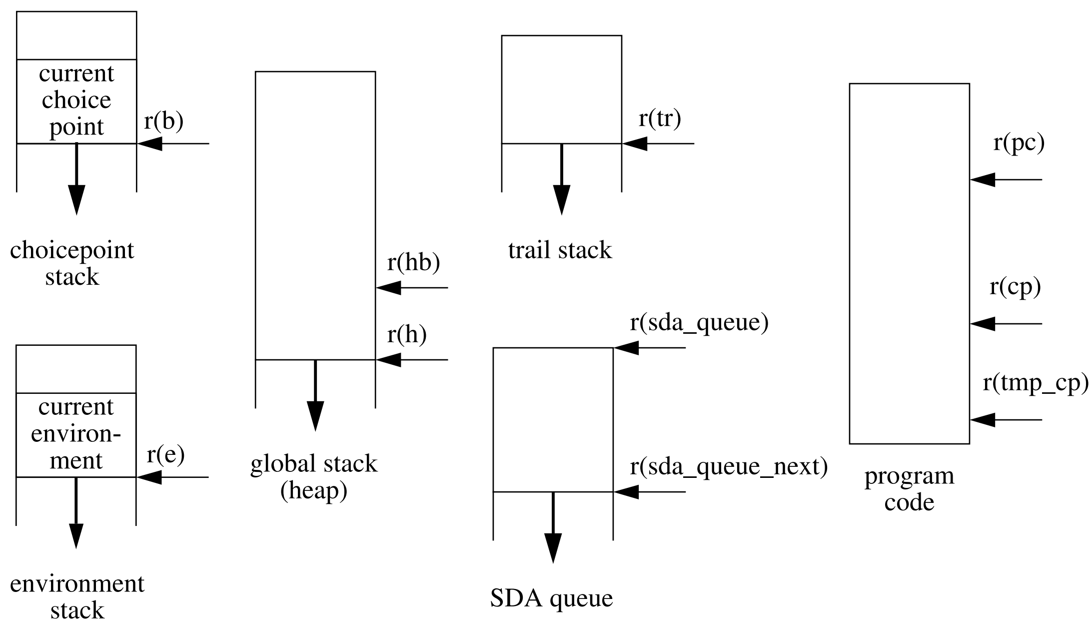


Figure 50: BAM Memory Sections

Program Code Section

The program code section contains the BAM instructions for the program predicates, as well as the run-time library and memory manager. No particular representation for the instructions is provided. They may be encoded in this memory region, but will most likely be translated into actual machine instructions.

Environment Stack

The environment stack contains environments. An environment is created when a predicate needs to call other predicates. It is used to save the return address for the current predicate and to save variables across predicate calls. These variables are called "permanent" variables, and are referred to as "p(i)". An environment is created by an allocate instruction and destroyed by a deallocate instruction. Table 36 describes the structure and contents of an environment. The current (top) environment is pointed to by r(e). An extra word may exist between environments in order to keep them double word aligned on implementations for which this is more efficient.

Table 36: Description of an Environment

Address	Contents	Description
r(e)-2-N through r(e)-3	p(N-1) through p(0)	These are the permanent variables, used to save values across calls to subordinate predicates. Once an environment is created, these may be treated as if they were registers. If a pad word is desired and needed (N is odd) in order to keep the environment stack double word aligned, it is placed at the beginning of the environment (r(e)-3-N) so we don't need to know the size of an environment to access the permanent variables in it.
r(e)-2	r(e)	This is the address of the environment active when this environment was created (i.e., the previous environment).
r(e)-1	r(cp)	This is the return address for the current predicate. It is saved so it isn't lost when subordinate predicates are called (see the call instruction).

Choicepoint Stack

The choicepoint stack contains choicepoints. A choicepoint is used to record an alternate choice (clause) to be invoked via backtracking if the current choice fails. It indicates the address for the next choice and records all state information that must be restored during backtracking. Table 37 describes the structure and contents of a choicepoint. The current (top) choicepoint is pointed to by r(b). An extra word may exist between or within choicepoints in order to keep them double word aligned on implementations for which this is more efficient.

Global Stack (Heap)

The global stack, also referred to as the heap, is used to store Prolog terms. Tagged pointers point to values in the heap. The representation of various terms in the heap was shown previously in Figure 49.

Trail Stack

The trail stack is used to record bindings that have occurred during program execution. This information is used during backtracking to “unbind” variables that have

Table 37: Description of a Choicepoint

Address	Contents	Description
r(b)-X-N through r(b)-X-1	arg(0) through arg(N-1)	Saved arguments. This is used to save argument and temporary registers (r(i)). The registers that are saved do not need to be sequential. See the choice instruction for details.
r(b)-8 through r(b)-X	Implementation-dependent	Implementation-dependent information may be saved in the choicepoint. The current implementation uses this area for a pad word, if one is desired to ensure double word alignment. Therefore, the offset (X) is implementation-dependent.
r(b)-7	r(sda_queue_next)	The end of the SDA queue when the choicepoint was created. This is used to remove entries added after the choicepoint, when backtracking occurs.
r(b)-6	r(h)	The top of the heap when the choicepoint was created. This is used to remove everything on the heap, created after the choicepoint, when backtracking occurs.
r(b)-5	r(tr)	The top of the trail when the choicepoint was created. This is used to unbind variables when backtracking occurs.
r(b)-4	r(e)	The top of the environment stack when the choicepoint was created. This is used to remove unneeded environments from the environment stack when backtracking occurs and to restore the state to the saved environment.
r(b)-3	r(cp)	The value of the continuation pointer (return address) register when the choicepoint was created. This is used to restore the processor state when backtracking occurs.
r(b)-2	r(b)	A pointer to the previous choicepoint. This is used when the last choice is encountered in a choicepoint in order to allow backtracking to previous choicepoints.
r(b)-1	Retry	The address of the BAM code for the next choice associated with this choicepoint. This is referred to as the “retry” address.

been bound since the choicepoint to which we're backtracking was created. To return these variables to an “unbound” state, all that needs to be done is to set them to self-referencing pointers, with a tag of ‘tvar’. Therefore, the only things needed on the trail stack are the addresses of bound variables, not their values.

For implementations which provide backtrackable destructive assignment, however, both the address and the value must be saved on the trail stack. In addition, either there needs to be a way to differentiate between variable recordings and destructive assignments recordings (e.g., the tag of the address saved on the trail) or the address must be stored twice for variables (as both the address and the value). We will typically take the latter choice.

SDA Queue

The SDA queue is used to store requests for stepped destructive assignment. The Aquarius Prolog User Manual describes this in detail. Briefly, this is a feature provided in Aquarius Prolog to allow the programmer to queue up a number of requests for destructive assignment (see queue_sda instruction), and then to commit these changes all at one time (see step instruction). This is implemented as a queue, instead of a stack, to ensure that requests for assignments are superceded by later requests for the same location.

The SDA queue holds pairs. The first entry is the address to be modified, with a tvar tag. The second entry is the tagged value to be place in that location. Because garbage collection looks at the SDA queue to determine live data in memory, all values on the SDA queue must be tagged.

The SDA queue end pointer is saved during choicepoint creation, to restore the state of the SDA queue if backtracking occurs. When the queued requests are committed (by the step instruction), the destructive assignments are trailed as backtrackable destructive assignments, to make SDA fully backtrackable.

A.3 Instruction Set

This section describes the BAM instruction set. First, a quick summary of the instructions is given, followed by detailed descriptions of each instruction. The last part provides a detailed description of the instruction operand types.

A.3.1 Instruction Summary

The BAM instructions fall roughly into four categories: procedural control flow, conditional control flow, unification, and arithmetic instructions. Tables 38 through 41 provide a summary of these instructions.

Table 38: Procedural Control Flow Instructions

Instruction	Description
procedure(P).	Procedure entry point definition.
entry(N).	Entry point for garbage collection.
allocate(N).	Allocate an environment.
deallocate(N).	Deallocate top environment.
call(P).	Call a procedure.
return.	Return from current procedure.
label(L).	Local label definition.
jump(L).	Jump to local label or procedure.
simple_procedure(P).	Simple procedure entry point definition.
simple_call(P).	Call a simple procedure.
simple_return.	Return from a simple procedure.
jump_ind(X).	Jump indirectly to the given address.

Table 39: Conditional Control Flow Instructions

Instruction	Instruction
hash(V, L).	Jump to label based on hash table lookup.
switch(V, TL, OL).	Multi-way branch based on tag of value.
jump(T, C, A, B, L).	Conditional branch based on arithmetic comparison.
choice(I/N, Rs, L).	Choice point management.

Table 39: Conditional Control Flow Instructions

Instruction	Instruction
cut(V).	Choice point removal.
fail.	Failure.

Table 40: Unification Instructions

Instruction	Description
deref(S, D, F).	Dereference value.
unify(V1, V2, F1, F2, fail).	General unification.
trail(V).	Trail variable.
unify_atomic_wt(V, A, L).	Unification with atomic value (with trailing).
equal(S1, S2, L).	Conditional branch based on read-mode unification failure.
move(S, D).	Write-mode unification (copy).
push(S, R, N).	Push onto arbitrary stack.
adda(S, O, D).	Address addition.
pad(N).	Address alignment.
trail_bda(S).	Trail a value for backtrackable destructive assignment.
queue_sda(D, S).	Queue a value for stepped destructive assignment.
step.	Perform assignments queued by queue_sda.
data_label(L).	Data label definition.
word(W).	Data word definition.

A.3.2 Instruction Set Details

The following sections provide detailed information about each of the instructions in the BAM instruction set. Each instruction description provides information on one or

Table 41: Arithmetic Instructions

Instruction	Description
add(T, A, B, D).	Addition.
sub(T, A, B, D).	Subtraction.
mul(T, A, B, D).	Multiplication.
div(T, A, B, D).	Division.
mod(T, A, B, D).	Remainder.
and(T, A, B, D).	Bitwise and.
or(T, A, B, D).	Bitwise or.
xor(T, A, B, D).	Bitwise exclusive or.
sll(T, A, B, D).	Logical shift left.
sra(T, A, B, D).	Arithmetic shift right.
not(T, A, D).	One's complement.
i2f(A, D).	Tagged integer to tagged floating point conversion.
f2i(A, D).	Tagged floating point to tagged integer conversion.
ord(M, A, D).	Tagged value to untagged integer conversion.
val(tint, A, D).	Untagged integer to tagged integer conversion.
val(tatm, N, A, D).	Untagged integers to tagged functor conversion.

more related instructions. This description contains four parts: a textual description, assembler syntax and operand description, an operational description, and an optional note section, providing additional information about the instruction(s) such as implementation choices and decisions. The operational description is provided in a C-like syntax. Within this description, a number of functions are used which are defined in Table 42.

Table 42: Functions used in the operational descriptions

Function	Definition
m(X)	This refers to the contents of the memory location whose address is given by X. This can be used to both read the value and modify it (depending on whether it's the source or destination of an assignment).
length(X)	This function returns the length of the list, X, as an integer.
tag(X)	This function returns the tag of the value, X. The result is undefined if X is not a tagged value.
integer(X)	This function returns a untagged integer, given the tagged integer, X. The result is undefined if X is not a tagged integer.
atom(X)	This function returns an untagged integer representing the unique ID assigned to the atom in the tagged atom or functor X.
arity(X)	This function returns an untagged integer representing the arity of the functor, X, or zero if X is a tagged atom.
Tag^Value	This function constructs a tagged word with a tag of Tag and value of Value.
tagptr(T^S) tagptr(T^(S+O)) tagptr(T^(S-O))	This function constructs a tagged pointer with a tag of T and an address as given by the effective address S, possibly offset by the number of words specified by the integer O.

A.3.2.1 Procedural Control Flow Instructions

This section describes the procedural control flow instructions. These instructions provide unconditional flow of control inside and between predicates.

Procedure(P)

This instruction defines the entry point to procedure P.

Assembler syntax:	Operands:
procedure(P).	proc_label(P)

Operation: N/A

Notes:

- If P is of the form N/A, it is assumed this is a Prolog predicate of arity A, with arguments in registers r(0) through r(A-1). This information may be important to a debugger. If P has any other form, no assumptions are made about the arguments.

Entry(P, N)

This instruction defines an acceptable point in the program where memory overflow checking and garbage collection can occur. The name comes from the fact that this is nominally the entry point to a predicate.

Assembler Syntax:	Operands:
entry(P, N).	proc_label(P), nat(N)

Operation: N/A

Notes:

- Originally, this operation was part of the procedure/1 instruction, but the compiler, in its peephole optimization phase, can unroll jumps and calls to procedures. For a highly-recursive predicate, this can result in a loop containing no memory overflow checking (because last-call optimization may end up jumping to a label internal to the predicate). Therefore, this operation was separated from the procedure instruction and is unrolled along with the procedure code. This ensures that all loops will contain at least one entry instruction.
- Granularity analysis could be used to reduce the number of places where this instruction is needed. If you can guarantee that an entry instruction was executed not too long ago on all paths leading to a given entry instruction, the entry instruction can be eliminated.
- N is the number of argument registers (r(0) through r(N-1)) which are currently active, i.e., that may contain tagged Prolog terms and therefore must be considered during garbage collection. Normally, N is the arity of the predicate

being entered. It can, however, be less if, for example, the last few arguments are uninit_reg (i.e., have no value on entry). The values in the active registers are used when computing the active portions of the Prolog memory sections (see the Aquarius Implementation Notes for a description of garbage collection). They may be changed if the memory is compacted or shifted. Therefore, they must either be valid tagged values or garbage (as in the case of uninit_reg); they cannot contain untagged values because these may be interpreted as tagged values and modified.

Allocate(N)

This instruction creates an environment, i.e. a new set of permanent variables, of size N on the environment stack.

Assembler syntax:	Operands:
allocate(N).	nat(N)

Operation: Temp = r(e);

r(e) = r(e) + N + 2;

m(r(e)-2) = Temp;

m(r(e)-1) = r(cp);

Notes:

- In addition to creating space for the permanent registers, this instruction saves r(cp). The only reason we need permanent registers is because we are going to call a predicate which might destroy the temporary registers. Therefore, the only time we need to save r(cp), which will be destroyed by the call, is here (see the call instruction). If a subordinate predicate is going to be called, but no permanent registers are needed, we can allocate an environment of size zero.
- The old r(e) is saved so the deallocate instruction can restore access to the previous environment.
- For implementations which use a single local stack for both choice points and environments, the second step in the operation should be:

  r(e) = max(r(e), r(b)) + N + 2;

- For implementations with double word access (e.g., the SPARC), it is more efficient to keep the environment double word aligned. In this case, the increment to r(e) should be rounded up. The processor register assignment should be made to allow either the allocate instruction, the deallocate instruction, or both to use double word access for storing and restoring r(e) and r(cp) in the environment (see deallocate).

Deallocate(N)

This instruction removes the top-most environment, which is of size N, from the environment stack.

Assembler syntax:	Operands:
deallocate(N).	nat(N).

Operation: r(cp) = m(r(e) - 1);
r(e) = m(r(e) - 2);

Notes:

- For implementations which have separate environment and choice point stack, the old r(e) doesn't need to be saved in the environment. Instead, this instruction can use N to decrement r(e) back to the previous environment.

Call(P)

This instruction calls the procedure P.

Assembler syntax:	Operands:
call(P).	label(P)

Operation: r(cp) = r(pc);
r(pc) = P;

Notes:

- This instruction changes the value of r(cp) without saving it. To handle multiple levels of nesting, it is important to save this register. This is done by the allocate/deallocate instructions, which the compiler always puts around call instructions.

Return

This instruction returns from a procedure call.

Assembler syntax:	Operands:
return.	N/A

Operation: r(pc) = r(cp);

Label(L)

This instruction defines label L

Assembler syntax:	Operands:
label(L).	label(L)

Operation: N/A

Jump(L)

This instruction performs an unconditional jump to L.

Assembler Syntax:	Operands:
jump(L).	label(L)

Operation: r(pc) = L;

Jump_ind(X)

This instruction performs an unconditional jump to the address found in the argument, given by X (i.e., if X is a register, the register contains the address; if X refers to a memory location; the memory location contains the address).

Assembler Syntax:	Operands:
jump_ind(X).	aea(X)

Operation: r(pc) = R;

Simple_procedure(P)

This instruction defines the entry point to simple procedure P. A simple procedure is one which doesn't call any other procedures.

Assembler syntax:	Operands:
simple_procedure(P).	proc_label(P)

Operation: N/A

Notes:

- Simple procedures are currently not generated by the compiler. They are meant for implementing low-level routines in the run-time system. For example, support for the unify and hash instructions may be provided by simple procedures.

Simple_call(P)

This instruction calls the simple procedure P.

Assembler syntax:	Operands:
simple_call(P).	label(P)

Operation: r(tmp_cp) = r(pc);

r(pc) = P;

Notes:

- This instruction changes the value of r(tmp_cp) without saving it. Since a simple procedure doesn't call any other procedures, this is not a problem. Furthermore, since a simple procedure can only called from a regular procedure, it is not necessary to save r(cp) when making a call to a simple procedure.

Simple_return

This instruction returns from a simple procedure call.

Assembler syntax:	Operands:
simple_return.	N/A

Operation: r(pc) = r(tmp_cp);

A.3.2.2 Conditional Control Flow Instructions

This section describes the conditional control flow instructions. These instructions are used mainly for clause selection and backtracking.

Hash(V, T)

This instruction looks up the value V in a hash table, T. The hash table is given as a list of pairs of values and labels. The label indicates where to jump when the value matches. If no match is found, execution continues immediately after the hash instruction.

Assembler syntax:	Operands:
hash(V, T).	ea(V), hash_list(T)

```
Operation: if ( tag(V) == tstr )
                Temp = m(V);
           else Temp = V;
           for( J=0; J<length(T); J++ )
           {
                if ( Temp = T[J].value )
                     r(pc) = T[J].label;
           }
```

Notes:

- There are numerous ways this instruction can be implemented. For example, the hash table can be stored as a linear list and searched directly. If the range of values is small, it could be implemented as a direct table lookup. The intended approach, however, is to use a hash table, indexed on the lower bits of the value. Collisions can be handled by storing in the next open hash slot. This requires storing a known terminating value after the last used hash slot (e.g., untagged zero). Therefore, the actual table could be longer than N (in the worst case, 2N-1)

- This instruction is usually implemented using a simple call to a run-time support routine.

Switch(V, TL, L)

This instruction performs a multi-way branch, based on the tag of V. T is a list of tag, label pairs. It branches to the label in this list matching the tag of V. If the tag of V does not appear in this list, it branches to L.

Assembler syntax:	Operands:
switch(V, TL, L).	ea(V), switch_list(TL), label(L)

```
Operation: r(pc) = L;
           for (J=0; J<length(TL); J++)
           {
               if ( tag(V) == TL[J].tag )
                   r(pc) = TL[J].label;
           }
```

Jump(T, C, A, B, L)

This instruction performs a conditional branch, based on a numeric comparison. It compares the values of A and B and branches to L if the comparison is true. The type of comparison is given by C. The data type (e.g., tagged integer) is given by T.

Assembler syntax:	Operands:
jump(T, C, A, B, L).	data_type(T), cond(C), ea(A), ea(B), label(L))

```
Operation: if ( ( C == 'eq' ) && A == B ) r(pc) = L;
           else if ( ( C == 'ne' ) && A != B ) r(pc) = L;
           else if ( ( C == 'lts' || C == 'ltu' ) && A < B ) r(pc) = L;
           else if ( ( C == 'gts' || C == 'gtu' ) && A > B ) r(pc) = L;
           else if ( ( C == 'les' || C == 'leu' ) && A <= B ) r(pc) = L;
           else if ( ( C == 'ges' || C == 'geu' ) && A >= B ) r(pc) = L;
```

Notes:

- The data types of the operands, A and B, must be consistent with the data type specification, T. Possible data types are tagged integers, tagged floating point numbers, and untagged values. Untagged values can either be machine integers or untagged pointers. In either case, the comparison uses the full machine word. The results of this instruction are undefined if the operand data types are not consistent with T.
- Values of C ending with ‘s’ refer to signed comparisons. Values ending with ‘u’ refer to unsigned comparisons. For other values (eq, ne), this doesn’t matter.

Cut(V)

This instruction implements the cut operation, i.e., it removes the latest choicepoint from the choicepoint stack. It assumes that V contains the address of the previous choicepoint (the compiler copies r(b) into this variable at the beginning of a procedure containing a cut).

Assembler syntax:	Operands:
cut(V).	aea(V)

Operation: r(b) = V;

r(hb) = m(r(b)-6);

Notes:

- For implementations which do not include r(hb), the second step (restoring r(hb)) is not done.

Choice(I/N, Rs, L)

This instruction performs choicepoint management for the I-th clause out of N. For the first choice (I=1), this creates the choicepoint, saving the registers given by Rs in it and setting the retry address to L. For intermediate choices (1<I<N), this restores the registers from the choicepoint and updates the retry address. For the last choice (I=N), the registers are restored and the choicepoint is removed.

Assembler syntax:	Operands:
choice(I/N, Rs, L).	pos(I), pos(N), regs(Rs), label(L)

```
Operation: if ( I == 1 ) { Temp = r(b);
                r(b) = r(b) + room for the choicepoint;
                for ( J=0; J<length(Rs); J++ )
                    m(r(b)-X-length(Rs)+J) = r(Rs[J]);
                m(r(b)-7) = r(sda_queue_next);m(r(b)-6) = r(h);
                m(r(b)-5) = r(tr);m(r(b)-4) = r(e);
                m(r(b)-3) = r(cp);m(r(b)-2) = Temp;
                m(r(b)-1) = L; r(hb) = r(h); }
            else { for ( J=0; J<length(Rs); J++ )
                    if ( Rs[J] != “no” ) r(Rs[J]) = m(r(b)-X-length(Rs)+J);
                if ( I < N )
                    m(r(b)-1) = L;
                else { r(b) = m(r(b)-2);
                    r(hb) = m(r(b)-6);
                }
            }
```

Notes:

- For implementations which use a single local stack for both choicepoints and environments, the second line in the operation should be:

r(b) = max(r(e), r(b)) + room for the choicepoint;

- The last choice (I=N) is implemented by removing the choicepoint when we reach this instruction; i.e., we load r(b) with the saved r(b) in the choicepoint. If failure occurs, we will continue from the next choice for the previous choicepoint. Therefore, the label is not used, but must be given as ‘fail’. The first instruction following labels for all other choices is the next choice instruction.
- For implementations which do not include r(hb), restoring r(hb) on the last choice (I=N) is simply ignored.
- The register numbers listed in Rs do not have to be the same for all choices, but there must be the same number of registers. This allows values to be moved to

different registers between choices, although the compiler does not make use of this.

- A value of ‘no’ in Rs indicates the value saved in the choicepoint is not required for a given choice, and should be ignored. This cannot be given for the first choice. Therefore, a choice only needs to restore those registers it uses, but the first choice must save all registers needed by any subsequent choice.
- For implementations with double word access (e.g., the SPARC), it is more efficient to keep the choicepoint double word aligned. The processor register assignment should allow pairs of registers to be saved/restored with double word accesses.

Fail

This instruction untrails all variable bindings and jumps to the retry address in the current choicepoint.

Assembler syntax:	Operands:
fail.	N/A

Operation: Temp = m(r(b)-5);

```
while ( Temp != r(tr) ) {
    m(m(r(tr)-2)) = m(r(tr)-1);
    r(tr) -= 2; }
r(sda_queue_next) = m(r(b)-7);
r(e) = m(r(b)-4);
r(cp) = m(r(b)-3);
r(h) = r(hb);
r(pc) = m(r(b)-1);
```

Notes:

- This describes the failure routine. This is the same operation that occurs whenever ‘fail’ is used as a label in an instruction. This may be implemented with in-line code, a jump to a failure routine, or a synchronous trap to a failure handler. Current implementations jump to a failure routine.

- This instruction performs all state restoration from the trail and choicepoint which is common to all choices. The remainder (e.g., restoring argument registers) is performed by the choice instruction.
- For implementations which do not save r(h) in r(hb) when creating a choicepoint (see trail), r(h) is restored from the choicepoint.
- An implementation may save other values in the choicepoint which must be restored on failure.
- If destructive assignment is not supported, the trail will contain only the addresses of bound variables, tagged with tvar. In this case, the value on the trail is uses as both the address and the contents to restore the variables to unbound.

A.3.2.3 Unification Instructions

The unification instructions are used mainly to implement both read-mode and write-mode unification.

Deref(S, D, F)

This instruction dereferences its first operand and stores the result in the second operand. The first argument is unchanged. This is the only instruction which dereferences its argument; all other instructions assume their arguments are dereferenced.

Assembler syntax:	Operands:
deref(S, D, F).	ea(S), aea(D), nv_flag(F)

Operation: Temp = S;

```
while ( tag(Temp) == tvar && m(Temp) != Temp )
    Temp = m(Temp);
D = Temp;
```

Notes:

- If D is a target processor register, it can be used for Temp. Otherwise, a temporary register should be used.
- F is added as an optimization; it indicates the mode of S, if known. This information allows the loop termination test to be simplified.

Equal(S1, S2, L)

This instruction compares the first two operands to ensure they are equal. If they aren't, it branches to L. This is a full word comparison; both the tags and the values are compared.

Assembler syntax:	Operands:
equal(S1, S2, L).	ea(S1), ea(S2), label(L)

```
Operation:  if( S1 != S2 )
                r(pc) = L;
```

Unify(V1, V2, F1, F2, fail)

This instruction performs a general unification between V1 and V2, branching to the failure routine (fail) if the unification fails.

Assembler syntax:	Operands:
unify(V1, V2, F1, F2, fail).	aea(V1), aea(V2), nv_flag(F1), nv_flag(F2)

```
Operation: if ( tag(V1) == tvar && ( tag(V2) != tvar || V1 > V2 ) )
                { trail(V1); m(V1) = V2; }
            else if ( tag(V2) == tvar )
                { trail(V2); m(V2) = V1; }
            else { switch ( tag(V1) )
                {
                    case tint: case tflt: case tatm:
                        if ( V1 != V2 ) fail;
```

```
                else break;
            case tlst: if ( tag(V2) != tlst ) fail;
                Temp1 = deref ( m(V1) );
                Temp2 = deref ( m(V2) );
                unify ( Temp1, Temp2, any, any, fail );
                Temp1 = deref ( m(V1+1) );
                Temp2 = deref ( m(V2+1) );
                unify ( Temp1, Temp2, any, any, fail );
                break;
            case tstr: if ( tag(V2) != tstr ) fail;
                if ( V1 != V2 ) fail; /* functor/arity don't match */
                for ( i=1; i<=arity(V1); i++ )
                {
                    Temp1 = deref ( m(V1+i) );
                    Temp2 = deref ( m(V2+i) );
                    unify( Temp1, Temp2, any, any, fail );
                    }
            }
        }
```

Notes:

- When unifying two variables, the younger always points to the older. This makes backtracking more efficient since all variables created after a choicepoint can then be ignored during backtracking; we merely restore the old heap pointer, freeing up the younger variables. This would cause problems if the older variable pointed to the younger (which would then be beyond the heap pointer).
- F1 and F2 are added as an optimization; they indicate the modes of the V1 and V2, if known. This information allows a better translation to be done by eliminating a number of cases in the above operation.

Trail(V)

This instruction adds V to the trail stack if V was created before the latest choicepoint. This is used to record the variables to unbind when backtracking.

Assembler syntax:	Operands:
trail(V).	aea(V)

Operation: if(V < r(hb))

```
{
    m(r(tr)++) = V; /* This is the address of the trailed variable */
    m(r(tr)++) = V; /* This is the contents of the trailed variable */
}
```

Notes:

- This instruction assumes the operand is a dereferenced, unbound value, i.e., the tag of V must be tvar. The results are unpredictable if this is not the case.
- For an implementation which does not support destructive assignment, this instruction only needs to save V, not its address. This is because, for an unbound variable, the value *is* the address.
- r(hb) points to the top of the heap when the current choicepoint was created. An implementation may choose to ignore the test and always add variables to the trail. This makes trailing quicker, but requires a larger trail stack. Another alternative is to compare against the saved r(h) in the current choicepoint. This requires fewer registers, but the trail check may take more time.

Unify_atomic_wt(V, A, L)

This instruction unifies the variable V with the atomic term A and branches to L if the unification fails. If the variable is unbound, it is trailed.

Assembler syntax:	Operands:
unify_atomic(V, A, L).	aea(V), ea(A), label(L)

```
Operation: if ( tag(V) == tvar )
           {
               m(V) = A;
               if( V < r(hb) )
               {
                   m(r(tr)++) = V; /* This is the address of the trailed variable */
                   m(r(tr)++) = V; /* This is the contents of the trailed variable */
               }
           }
           else if ( V != A )
               r(pc) = L;
```

Notes:

- This is a special case of the general unification that significantly reduces code size.
- This instruction is used when unifying a variable whose mode is not known with an atomic value. If the mode were known, the compiler would generate either an equal instruction (for nonvar) or a move instruction (for var).

Move(S, D)

This instruction moves S to D. Depending on the addressing modes of S, this instruction can copy a tagged value, or create a tagged pointer.

Assembler syntax:	Operands:
move(S, D).	ea(S), aea(D) tagptr(S), aea(D)

```
Operation: if( the first operand is an effective address or an immediate value )
               D = S;
           else if( the first operand is a tagged pointer construction )
               D = tagptr(S);
```

Push(S, R, N)

This instruction pushes S onto the stack with stack pointer R, then increments R by N words.

Assembler syntax:	Operands:
push(S, R, N).	ea(S), reg(R), pos(N) tagptr(S), reg(R), pos(N)

Operation: if(the first operand is an effective address or an immediate value)

m(R) = S;

else if(the first operand is a tagged pointer construction)

m(R) = tagptr(S);

R = R + N;

Notes:

- If the target processor supports a post-incrementing address mode, this can be used to implement a push with increment of 1.
- N is a word offset. For machines which are not word addressed, this must be appropriately scaled.

Adda(S, O, D)

This instruction adds the offset O to the tagged pointer in S and stores the result in D.

Assembler syntax:	Operands:
adda(S, O, D).	aea(S), ea(O), aea(D)

Operation: D = S + O;

Notes:

- O is an untagged integer, specifying a word offset. For machines which are not word addressed, this must be appropriately scaled.
- The results are unpredictable if S is not a tagged pointer or if O is not an untagged integer.

Pad(N)

This instruction adds N words to the heap pointer, r(h). This is used to ensure the correct alignment of compound terms.

Assembler syntax:	Operands:
pad(N).	pos(N)

Operation: r(h) = r(h) + N;

Notes:

- N is a word offset. For machines which are not word addressed, this must be appropriately scaled.
- On machines which don't care about the alignment of terms on the heap, this instruction can be treated as a no-op.
- The space reserved by this instruction will never be accessed.

Trail_bda(V)

This instruction pushes the address and value of V onto the trail stack if V was created before the current choicepoint. This is used to record the variables modified using destructive assignment, to restore them to their original value when backtracking (see backtrackable destructive assignment in the Aquarius Prolog User Manual).

Assembler syntax:	Operands:
trail_bda(V).	aea(V)

Operation:
```
if( V < r(hb) )
{
    m(r(tr)++) = tagptr(tvar^V);
    m(r(tr)++) = m(V);
}
```

Notes:

- For an implementation which does not support backtrackable destructive assignment, this instruction is illegal.

- The operational description shows one way this operation could be implemented. There must be a way to handle BDA trailings and variable trailings on the trail stack. One way is to have the trail instruction save the tagged variable pointer as the second word (now, detrailing always consists of storing the second word in the address given in the first word); this can make the trail large. Another way is to use a special tag to indicate BDA trailings. In this case, this instruction must add that tag to the address of V; this is what was done on the VLSI-BAM. This makes detrailing take a little longer. Another approach would be to have separate variable and BDA trail stacks. This complicates memory management and requires another stack pointer.
- r(hb) points to the top of the stack when the current choicepoint was created. An implementation may choose to ignore the test and always add values to the trail. This makes trailing quicker, but requires a larger trail stack. Another alternative is to compare against the saved r(h) in the current choicepoint. This requires fewer registers, but the trail check may take more time.

Queue_sda(D, S)

This instruction pushes a request for stepped destructive assignment onto the SDA queue (see stepped destructive assignment in the Aquarius Prolog User Manual). D is the address that will be modified. S is the value that will be placed there.

Assembler syntax:	Operands:
queue_sda(D, S).	aea(D), ea(S)

Operation: m(r(sda_queue_next)++) = tagptr(tvar^D);

m(r(sda_queue_next)++) = S;

Notes:

- For an implementation which does not support stepped destructive assignment, this instruction is illegal.

- The register r(sda_queue_next) is only needed if stepped destructive assignment is supported. If it is, this register points to the sda_queue, a new memory section. This register must be saved in the choicepoint and restored during backtracking.
- Assignments queued by this instruction take effect when the next step instruction is executed.

Step

This instruction performs the stepped destructive assignment requests made by the queue_sda instruction (see stepped destructive assignment in the Aquarius Prolog User Manual).

Assembler syntax:	Operands:
step.	N/A

Operation:
```
Temp = r(sda_queue);
while ( Temp < r(sda_queue_next) )
{
    A = m(Temp)++;
    V = m(Temp)++;
    if ( A < r(hb) )
    {
        m(r(tr)++) = A;
        m(r(tr)++) = m(A);
    }
    m(A) = V;
}
r(sda_queue_next) = r(sda_queue);
```

Notes:

- For an implementation which does not support stepped destructive assignment, this instruction is illegal.

- The bindings made by this instruction are added to the trail in the same manner as BDA trailings, so that stepped destructive assignment is fully backtrackable. Therefore, backtrackable destructive assignment must be supported in order to support stepped destructive assignment.
- The bindings are made in the order in which they were queued to ensure that multiple bindings to the same address results in the last value being bound to the address.
- The register r(sda_queue) is only needed if stepped destructive assignment is supported. It points to the start of the SDA queue.

Data_label(L)

This instruction defines data label L

Assembler syntax:	Operands:
data_label(L).	label(L)

Operation: N/A

Notes:

- A data label is used to identify the beginning of a constant term, created using word pseudo-instructions (see below).
- For implementations which prefer double-word aligned data (e.g., the SPARC), this data label should be double-word aligned. In other words, the assembler location counter should be aligned before the label is defined (perhaps leaving an unused location before the label).

Word(W)

This instruction defines the contents of a data word to be W. This should follow a data_label definition or another word pseudo-instruction.

Assembler syntax:	Operands:
word(W). word(T^L).	imm(W). pointer_tag(T), label(L).

Operation: N/A

Notes:

- This pseudo-instruction defines the contents of a memory word. This can either be a tagged or untagged constant (integer, float, or atom) or a tagged pointer to another data label. This is used for constructing constant terms at compile time.

A.3.2.4 Arithmetic Instructions

This section describes the arithmetic instructions. This includes both tagged and untagged arithmetic, as well as conversion instructions. These instructions are used primarily in implementing the 'is/2' built-in, as well as supporting the needs of the run-time system.

A.3.2.4.1 Typed Arithmetic Instructions

These instructions perform unary and binary arithmetic operations on values of a given data type.

Assembler syntax:	Operands:
add(T, A, B, D). sub(T, A, B, D). mul(T, A, B, D). div(T, A, B, D). mod(T, A, B, D). and(T, A, B, D). or(T, A, B, D). xor(T, A, B, D). sll(T, A, B, D). sra(T, A, B, D).	data_type(T), ea(A), ea(B), aea(D)
not(T, A, D).	data_type(T), ea(A), aea(D)

Operation: switch (instruction) {

case add: D = A + B; break;

case sub: D = A - B; break;

case mul: D = A * B; break;

case div: D = A / B; break;

```
        case mod:  D = A % B; break;
        case and:  D = A & B; break;
        case or:  D = A | B; break;
        case xor:  D = A ! B; break;
        case sll:  D = A << B; break; /* logical shift left A by B places */
        case sra:  D = A >> B; break; /* arithmetic shift right A by B places */
        case not:  D = ! A; break;
    }
```

Notes:

- T specifies the data type. This can be tagged integers, tagged floating point numbers, or untagged integers. If the values of the operands (A and B) do not match the specified data type, the results are undefined.
- The data type cannot be tagged floating point for logical operations (and, or, xor, sll, sra, and not).
- The results of overflow, underflow, or division by zero are undefined. If the shift count on sra or sll instructions is larger than the target integer word size, the results are undefined.
- It's not clear what negative shift counts mean.

A.3.2.4.2 Conversion Instructions

These instructions convert between integer and floating point and between tagged and untagged values.

Assembler syntax:	Operands:
i2f(A, D). f2i(A, D).	ea(A), aea(D)
ord(M, A, D).	mode(M), ea(A), aea(D)
val(tint, A, D).	ea(A), aea(D)
val(tatm, N, A, D).	ea(N), ea(A), aea(D)

```
Operation: switch ( instruction ) {
        case i2f: /* tagged integer to floating point conversion */
        case f2i: /* tagged floating point to integer conversion */
            D = A; break;
        case ord: /* tagged value to untagged integer conversion */
            switch(M) {
                case integer: D = integer(A); break;
                case atom: D = atom(A); break;
                case arity: D = arity(A); break;
            }
        case val(tint): /* untagged to tagged integer conversion */
            D = tint^A; break;
        case val(tatm): /* untagged integer(s) to tagged functor conversion */
            D = tatm^(N/A); break;
    }
```

Notes:

- The results are undefined if the operands are not of the appropriate type.
- The val(tatm,...) instruction is used to create a tagged functor or atom from untagged integers. The integer N is the machine integer for an atom. The integer A is the machine integer for the arity.

A.3.3 Instruction Operands

This section describes the types of operands used for various instructions. The operand types, their syntax, and their descriptions are given in Table 43.

Table 43: BAM Instruction Operands

Operand and Syntax:	Description:
proc_label(P) :- atom(P) proc_label(N/A) :- atom(N), nat(A).	The operand specifies a label for a procedure or a simple procedure. The second form is for Prolog predicates; A is the predicate's arity.
label(P) :- proc_label(P). label(l(P,N)) :- proc_label(P), pos(N). label(i(P,N)) :- proc_label(P), pos(N). label(s(P,N)) :- proc_label(P), pos(N). label(n(P,N)) :- proc_label(P), pos(N).	The operand specifies a label. It must be a non-ground Prolog term. The label 'fail' has a special meaning; see the fail instruction. The last four forms are used for labels local to a procedure.
ea(I) :- imm(I). ea(A) :- aea(A).	The operand specifies a value via an effective address. It can be an immediate value or a value obtained through an alterable effective address.
imm(I) :- integer(I). imm(tint^I) :- integer(I). imm(tflt^F) :- float(F). imm(tatm^A) :- atom(A). imm(tatm^(N/A)) :- atom(N), nat(A).	The operand specifies an immediate value. It can be untagged or tagged. The last form is used to specify a structure functor with name N and arity A.
aea(R) :- reg(R). aea([R]) :- reg(R). aea([R+N]) :- reg(R), integer(N). aea([R-N]) :- reg(R), integer(N).	The operand specifies an alterable effective address. It is the contents of a register or a memory location, addressed indirectly through a register with an optional offset.
reg(r(N)) :- nat(N). reg(p(N)) :- nat(N). reg(r(X)) :- atom(X).	The operand is the contents of a register. It is r(X) or p(N), where N is a natural number and X is either a natural number or a register name (e.g., b). An implementation may provide a number of named special purpose registers. These may be implemented as physical registers or known memory locations.

Table 43: BAM Instruction Operands

<table>
<tr><th>Operand and Syntax:</th><th>Description:</th></tr>
<tr><td>regs([]).
regs([no|Rs]) :- regs(Rs).
regs([R|Rs]) :- nat(R), regs(Rs).</td><td>The operand is a list of register specifiers (for the choice instruction). Each entry in the list is either a natural number (referring to r(i)) or ‘no’, indicating no register.</td></tr>
<tr><td>data_type(ti).
data_type(tf).
data_type(nt).</td><td>The operand specifies the data type for operands in an arithmetic or comparison instruction to be either tagged integers (ti), tagged floating point numbers (tf), or untagged integers or pointers (nt).</td></tr>
<tr><td>hash_list([]).
hash_list([V-L|R]) :- imm(V), label(L),
        hash_list(R).</td><td>The operand specifies a hash table, by providing a list of value/label pairs.</td></tr>
<tr><td>switch_list([]).
switch_list([T-L|R]) :- tag(T), label(L),
        hash_list(R).</td><td>The operand specifies a tag/label pair list for the switch instruction.</td></tr>
<tr><td>tagptr(T^S) :- pointer_tag(T), aea(S).
tagptr(T^(S+O)) :- pointer_tag(T),
        aea(S), int(O).
tagptr(T^(S-O)) :- pointer_tag(T),
        aea(S), int(O).
tagptr(T^L) :- pointer_tag(T), label(L).</td><td>The operand is a tagged pointer. The tag is specified explicitly. The address is computed from the effective address, S, possibly offset (positively or negatively) by O words or is specified to be data label L.</td></tr>
<tr><td>pointer_tag(T) :- T=tstr; T=tlst; T=tvar.</td><td>The operand is a pointer tag.</td></tr>
<tr><td>cond(C) :- C=eq; C=ne.
cond(C) :- C=lts; C=gts; C=les; C=ges.
cond(C) :- C=ltu; C=geu; C=gtu; C=leu.</td><td>The operand specifies an arithmetic comparison condition. Values ending with ‘s’ are signed, ending with ‘u’ are unsigned, and others don’t matter.</td></tr>
<tr><td>nv_flag(var).
nv_flag(nonvar).
nv_flag(any).</td><td>The operand specifies the mode of a variable, if known. It is ‘var’ for unbound, ‘nonvar’ for bound, or ‘any’ for unknown.</td></tr>
</table>

Table 43: BAM Instruction Operands

Operand and Syntax:	Description:
mode(integer). mode(atom). mode(arity).	The operand is the mode for an ord instruction. It is either ‘integer’ to extract the integer from a tagged value, ‘atom’ to extract the ID of the atom, or ‘arity’ to extract the arity of a functor.
int(N) :- integer(N).	The operand is an integer.
pos(N) :- integer(N), N > 0.	The operand is a positive integer.
nat(N) :- integer(N), N >= 0.	The operand is a natural integer.

Appendix B: Aquarius Compiler Abstract Domain Description

B.1 Introduction

In this appendix, we reconstruct the abstract domain used in the Aquarius Prolog compiler [84]. We define the abstract operations needed to perform abstract interpretation over this domain according to the framework we defined in Chapter 4. This domain is interesting in that it captures operational aspects of execution not representable in terms of substitutions. Also, it provides a complex example of the use of our analysis framework. We show that the abstract operations are monotonic, which is important in showing that the analysis reaches a fixpoint (terminates).

We begin by defining the form and semantics of the abstract domain. Next, we define some terminology used in the remainder of the appendix. Then, we define each of the abstract operations required by our framework and a number of utility functions used by multiple operations.

B.2 Abstract Domain

The domain of descriptions used in the Aquarius compiler is given by:

$$Description = D_1 \times D_2 \times D_3 \times D_4$$

$$D_1 = Var \rightarrow (I \times R)$$

$$I = < \{ \bot_I, ground, nonvar, any, uninit, new \}, \sqsubseteq_I >$$

$$R = < \{ rderef, locally\_deref, any \}, \sqsubseteq_R >$$

$$D_2 = < \wp (\{ (d_1, d_2, d_3, d_4) \mid d_1 \in Var, d_2 \subseteq Var, d_3 \subseteq Var, d_4 \in B \}) , \sqsubseteq_2 >$$

$$B = \{ true, false \}$$

$$D_3 = < \wp (Var), \subseteq >$$

$$D_4 = < \wp (Var), \supseteq >$$

where:

$$\bot_I \sqsubseteq_I ground \sqsubseteq_I nonvar \sqsubseteq_I any,\ \bot_I \sqsubseteq_I new \sqsubseteq_I uninit \sqsubseteq_I any,$$

$$rderef \sqsubseteq_R locally\_deref \sqsubseteq_R any,\ \bot_R = rderef$$

$$X \sqsubseteq_2 Y \text{ iff } \forall\ (y_1,y_2,y_3,y_4) \in Y, \exists\ (y_1,y_2,y_3,x_4) \in X \ni x_4 \sqsubseteq_B y_4$$

$$true \sqsubseteq_B false, \text{ and } \bot_B = true.$$

This domain consists of the product of four simpler domains. The first domain approximates state information about the variables in a clause. The next two domains provide information used to improve the precision of these states. The last domain describes another state for variables that could have been expressed in the same manner as the first, but it was felt that a variable set representation was more appropriate since it is a boolean state applying only to head variables.

The first domain describes two types of information for each variable in the clause being analyzed. This information is represented as a function, instead of a set, since it provides a mapping from variables to states; for variables not in the clause, the function returns $(\perp_I, \perp_R)$. The first state describes the degree of instantiation of the variable. This combines modes (Domain M_5) with initialization information (Domain AC_3). The second state describes information about reference (pointer) chains that might need to be traversed to get at the value of the variable, or the value of arguments if the variable is a compound term; this is domain R_3. The ordering relations for these states appear in Figure 52. The meanings of the states are given in Table 44. If the instantiation state is *uninit* or *new*, the reference mode must be *rderef*; although the ordering relation and the least upper bound (lub) operation are defined over all values in the cross product, the abstract operations ensure that this restriction is always met.

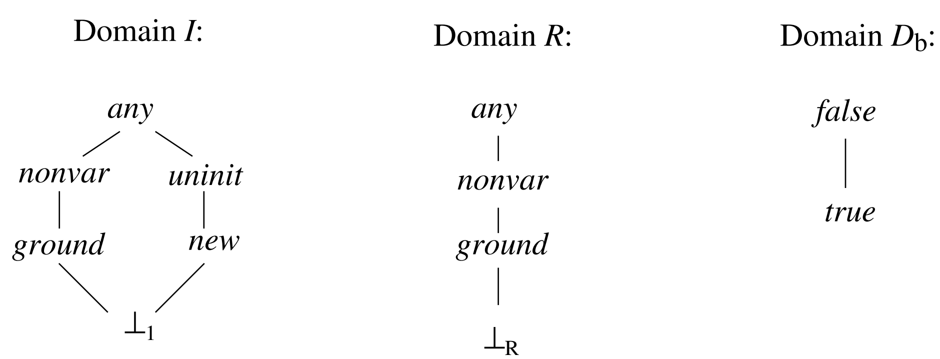


Figure 51: Lattice Diagrams for Domains *I*, *R*, and D_b

Table 44: Definitions of State Values

Value:	Description:
any	The variable is initialized, but may be bound or unbound.
nonvar	The variable is bound, but may contain some unbound arguments.
ground	The variable is completely bound.
uninit	The variable is uninitialized, but possibly allocated.
new	The variable is uninitialized and unallocated (because it hasn't been encountered in the clause, yet).
rderef	There is no pointer chain to the top level of the value (it is dereferenced), and if it's a structure, all arguments are *rderef* (recursively dereferenced).
locally_deref	The variable contains no pointer chains in the current scope, but might in the calling scope. This occurs when a built-in dereferences a simple value, making it *rderef* locally, but still retaining the possibility of a pointer chain in the caller.
any	The variable may contain pointer chains anywhere in its structure.

The second domain provides a form of aliasing information, similar to the equivalence domains described in Chapter 4. It describes the explicit unifications that have occurred so far in a clause. This is used to improve the variable states at the end of the clause. For example, if all variables on one side of a unification are ground at the end of the clause, so are the variables on the other side. An element of the domain consists of a set, with one 4-tuple for each unification that occurred during the clause. The unifications are always of a variable with either a variable or a term.[1] Therefore, the first element of the 4-tuple for a unification identifies the variable and the second element is the set of variables in the term (or simply the variable) unified with the first variable. The third element is the set of variables in the term which were *new* at the time of the unification. The fourth element is a flag that indicates if the unification can be used to improve the reference chain state of the first variable in the unification. Each

1. Unification of terms is broken into unification of the arguments.

element in the set describing unifications provides further restrictions of the set of possible values in the concrete domain. Therefore, the top element is the empty set (no restrictions). The ordering relation ($\sqsubseteq_2$) ensures that for each element in the larger description (the right-hand side), there is a more restrictive element in the smaller description (the left-hand side).

An element of the third domain identifies the set of variables which might have been bound by some operation. This is used during predicate exit to determine the range of changes to variables in the scope of the caller. If all variables possibly bound by a predicate call were either ground, uninit, or new before the call, no other variables are affected by the call. Otherwise, a worst case assumption must be made because of possible aliasing. Currently, this domain element is set to the set of head variables during clause termination (this is a worst case assumption). For built-in predicates, more specific information is provided. Since this description talks about 'possibly' bound variables, the top element is the set of all variables and the ordering relation ($\sqsubseteq_3$) is subset.

The last domain describes the set of head variables which can be returned to the caller in registers, instead of being stored in a memory location provided by the caller. These variables are called *uninit_reg* arguments [84]. For arguments which are new or uninitialized at the time of the call, this is a more efficient argument passing mechanism. The worst case (top element) is when no variables can be treated as *uninit_reg* arguments (empty set). Therefore, the ordering relation ($\sqsubseteq_4$) is superset. In order to use this mechanism for a given predicate argument, a number of criteria must be met:

- The argument must be uninitialized at the time of the call.
- The argument must not appear prior to the last *non-survive* goal in each clause.
- The argument must not appear more than once in any given clause.
- The last goal in each clause must refer to a known predicate.
- If the last goal in a clause is not a survive goal, the argument must appear in the same position in this goal as in the head.

The Aquarius domain is similar to $M_5 \times AC_4 \times E_2 \times R_3$ from Chapter 4.

B.3 Terminology

Table 45 defines some terminology and ancillary functions used in the remainder of this appendix.

Table 45: Definitions of Terminology and Ancillary Functions

Terminology/Function:	Definition:
X_i	the *i*th element of an n-tuple or the *i*th goal in a list of goals.
arg(i,T)	the *i*th argument of the term T.
arity(T)	the arity of the term T.
dups(T)	the set of all variables appearing multiple times in the term T.
fast(H)	true if the predicate whose head is H is a *fast* predicate. This is true iff all goals in the predicate are survive goals.
state(V,D)	the state of variable V in the description D. This is equivalent to D_1(V).
nonvar(T)	true if the term T is bound (it is non-variable).
survive(G)	true if the goal, G, is a survive goal. This is the case iff it is a call to simple built-in predicate which does not modify any argument registers (i.e., they 'survive' across the goal).
update_state(V,M,D)	a new description, everywhere the same as D, except the state of V has been changed to M.
var(T)	true if the term T is unbound (it is variable).
vars(T)	the set of all variables in the term T.
prop(N, F)	the set of variables for which the property, N, is true in the given Aquarius mode formula, F.

B.4 Abstract Interpretation Operations

The following sections define the operations over this abstract domain that are needed for abstract interpretation of the program. The set of operations that are defined are those required by the framework we defined in Chapter 4.

Order Comparison

The function, less_than(X, Y), implements the order relation for the abstract domain. It is defined in Figure 52.

---

function less_than(X : *Description*, Y : *Description*) : *Boolean* =
    if X = $\bot$
        then true
    else if Y = $\bot$
        then false
    else if $\forall$ V $\in$ Vars, state(V,X) $\sqsubseteq_S$ state(V,Y) $\wedge X_2 \sqsubseteq_2 Y_2 \wedge X_3 \subseteq Y_3 \wedge$
            $X_4 \supseteq Y_4$
        then true
    else false

Figure 52: Aquarius Domain Order Comparison

Least Upper Bound

The function, upper_bound(X, Y), returns the least upper bound of two domain values. This function is defined in Figure 53.

Abstract Interpretation of Predicate Entry

The function, abs_int_entry (A, H, ID), returns the description valid after entry to a predicate. H is the most general head of the called predicate. A is the call atom. ID is the description valid before the call. The global variable, Last, is true if and only if A is the last goal in the current clause. This is important when last-call optimizations cause this goal to be treated differently. This function can be seen to be monotonic since the functions it calls are monotonic. The function is defined in Figure 54.

```
function upper_bound( X : Description, Y : Description ) : Description =
    if X = ⊥
        then Y
    else if Y = ⊥
        then X
    else ( f : Var → S, X2 ∩ Y2, X3 ∪ Y3, X3 ∩ Y3 )
where:
    f(V) = state(V,X) ⊔S state(V,Y)
```

Figure 53: Least Upper Bound for Aquarius Domain

```
function abs_int_entry( A : Goal, H : Goal, ID : Description ) : Description =
    if ID = ⊥
        then ⊥
        else ( HM, ∅, UR ∩ uninit_regs ( H ), ∅ )
where:
    HM = pass ( in, H, A, dups(A), f: Var → S, ID1 )
    UR = if Last and fast(H)
            then { V | ∃ i ∋ arg(i,H)=V ∧ UR(i) }
            else vars(H)
    UR(i) = ( ID1(arg(i,A)) ⊑S (new,rderef) ∨ arg(i,A)∈ ID3 )
    f(V) = if V ∈ vars(Head)
            then (new,rderef)
            else ⊥S
```

Figure 54: Predicate Entry for Aquarius Domain

Abstract Interpretation of Clause Initialization

The function, initialize (C, HD), returns the description valid after initialization of a clause. C is the clause with a canonical head, C_H, and a body C_B which is a conjunction of goals. HD is the description valid after entry to the predicate. This function can be seen to be monotonic. The function is defined in Figure 55.

$\underline{\text{function}}$ initialize(C : *Clause*, HD : *Description*) : *Description* =

$\underline{\text{if}}$ HD = $\perp$

$\underline{\text{then}}$ $\perp$

$\underline{\text{else}}$ (f: $Var \rightarrow S$, $\emptyset$, HD_3, $\emptyset$)

where:

f(V) =

$\underline{\text{if}}$ $V \in vars(C_B) \wedge V \notin vars(C_H)$

$\underline{\text{then}}$ (*new*,*rderef*)

$\underline{\text{else}}$ $HD_1(V)$

Figure 55: Clause Initialization for Aquarius Domain

Abstract Interpretation of Clause Termination

The function, terminate (C, LD), returns the description valid after termination of a clause. C is the clause with a canonical head, C_H, and a body C_B which is a conjunction of goals. LD is the description valid after completion of the last goal in the clause. This function can be seen to be monotonic since the functions it calls are monotonic. The function is defined in Figure 56.

Abstract Interpretation of Predicate Exit

The function, abs_int_exit (H, A, ID, TD), returns the description valid after exiting from a predicate. H is the most general head of the called predicate. A is the call atom. ID is the description valid before the call. TD is the description valid after termination of the predicate; this is computed by taking the least upper bound of the results from terminating each clause in the predicate. The global variable, Last, is true if and only if A is the last goal in the current clause. This is important when last-call optimizations cause this goal to be treated differently. This function can be seen to be monotonic since the functions it calls are monotonic. The function is defined in Figure 57.

Abstract Interpretation of Built-in Predicates

The function, abs_int_builtin (C, CD) returns the description valid after calling a "special" built-in predicate (e.g., unification). For the Aquarius domain, unification is

$\underline{\text{function}}$ terminate(C : *Clause*, LD : *Description*) : *Description* =

$\underline{\text{if}}$ LD = $\perp$

$\underline{\text{then}}$ $\perp$

$\underline{\text{else}}$ (f: $Var \rightarrow M$, $\emptyset$, D'_3, vars(C_H))

where:

D' = back_prop_unify (LD)

f(V) = $\underline{\text{if}}$ V $\notin$ vars(C_H)

$\underline{\text{then}}$ $\perp_S$

$\underline{\text{else}}$ $\underline{\text{if}}$ V $\in$ vars(C_H) $\wedge$ $D'_1(V) \sqsubseteq_S$ (*uninit*,*rderef*)

$\underline{\text{then}}$ (*any*,*rderef*)

$\underline{\text{else}}$ $\underline{\text{if}}$ V $\in$ vars(C_H) $\wedge$ $D'_1(V)_2$ = *locally_deref*

$\underline{\text{then}}$ ($D'_1(V)_1$,*any*)

$\underline{\text{else}}$ $D'_1(V)$

Figure 56: Clause Termination for Aquarius Domain

the only "special" built-in. The remaining built-ins are handled through table-lookup as described in Chapter 3. This function can be seen to be monotonic since the functions it calls are monotonic. The function is defined in Figure 58.

Abstract Interpretation of Unknown Predicates

The function, abs_int_unknown (A, CD) returns the description valid after a call to an unknown predicate. A is the call atom. CD is the description valid before the call. The global variable, Last, is true iff this is the last goal in the current clause. This is important when last-call optimizations cause this goal to be treated differently. This function can be seen to be monotonic. The function is defined in Figure 59.

Abstract Interpretation Initialization and Termination

This domain requires no "special" initialization or termination. Therefore, the predicates, init_absdom and term_absdom, are defined as simple facts, always succeeding.

$\underline{\text{function}}$ abs_int_exit(H : *Goal*, A : *Goal*, ID : *Description*, TD : *Description*)

: *Description* =

$\underline{\text{if}}$ TD = $\perp$

$\underline{\text{then}}$ $\perp$

$\underline{\text{else}}$ (NM', ID_2, UR, $\emptyset$)

where:

NM = *pass* (*out*, A, H, $\emptyset$, ID_1, TD_1)

NM' = $\underline{\text{if}}$ $\exists$ (i, V) $\ni$ V $\in$ vars(arg(i,A)) $\wedge$ arg(i,H) $\in$ TD_3 $\wedge$

ID_1(V) $\notin$ { (*ground*,*rderef*), (*new*,*rderef*), (*uninit*,*rderef*), ($\perp$,*rderef*) }

$\underline{\text{then}}$ *worst_case_for_aliasing* (vars(A), NM)

$\underline{\text{else}}$ NM

UR = $\underline{\text{if}}$ Last

$\underline{\text{then}}$ ID_3 $\cap$ { V | var(V) $\wedge$ $\exists$ i $\ni$ arg(i,A)=V $\wedge$ arg(i,H) $\in$ TD_3 }

$\underline{\text{else}}$ ID_3

Figure 57: Predicate Exit for Aquarius Domain

Displaying Domain Values

The predicate, write_desc(D, CV, P), provides a readable interpretation of a description. D is the description to be written. CV is a list of variables appearing in the clause to which the description applies. P is a prefix, to written at the beginning of each line. This predicate is defined in Figure 60.

Completely Static Analysis

There are a number of precomputed values used in the Aquarius domain for predicates and clauses. For predicates, we precompute the set of predicate arguments which may be turned into *uninit_reg* arguments. This eliminates some possibilities, based on the syntax of the clauses, alone. We also determine which predicates are considered to be *fast*. For clauses, we precompute the set of variables appearing only in the clause (not in the head) and the set of variables appearing multiple times in a clause.

```
proc abs_int_builtin( A = B : Goal, CD : Description ) : Description
    if CD = ⊥ ∨ A and B are not unifiable
        then return ⊥
    ID' := save_unify ( B, A, CD, save_unify ( A, B, CD, ID ) )
    AS := term_state ( A, ID' )
    BS := term_state ( B, ID' )
    if var(B) then
        ID' := unify_var_with_var ( unify, A, ID', B, ID', CD )
        ID' := unify_var_with_var ( unify, B, ID', A, ID, CD )
        if AS1 ⊑I uninit ∨ BS1 ⊑I uninit ∨ ( AS1 ⊑I ground ∧ BS1 ⊑I ground )
            then return ID'
            else return worst_case_for_aliasing ( {A,B}, ID' )
    else /* nonvar(B) */
        ID' := unify_var_with_term ( A, ID', B, ID' )
        ID' := unify_term_with_var ( unify, B, ID', A, ID, CD )
        if AS1 ⊑I uninit ∨ ( AS1 ⊑I ground ∧ BS1 ⊑I ground )
            then return ID'
            else return worst_case_for_aliasing ( {A} ∪ vars(B), ID' )
```

Figure 58: Abstract Unification for Aquarius Domain

This is done by three functions, prepare_head, prepare_clause, and prepare_tail, as described in Chapter 4. These functions are defined in Figure 61.

Interpreting Mode Formulas

The function, formulas_to_descs(H, HF, TF, BV, HD, TD), converts Aquarius mode formulas into an entry and exit description for a predicate. H is the canonical head for the predicate. HF is the formula of properties known to be true when this predicate is called. TF is the formula of properties known to be true when this predicate completes. BV is the set of predicate arguments possibly bound by a call to this predicate.[2] On return, HD is the description which must be valid after entering this function abs_int_unknown(A : *Goal*, CD : *Description*) : *Description* =

if CD = $\perp$

then $\perp$

else (f: *Var* $\rightarrow$ *S*, CD_2, CD_3, CD_4)

where:

f(V) = if $CD_1(V)_1 \in \{uninit, new\} \wedge V \in$ vars(A)

then (*any*,*any*)

else if $CD_1(V)_1 \in \{uninit, new\} \wedge V \notin$ vars(A)

then $CD_1(V)$

else if $CD_1(V)_1 \sqsubseteq_I ground$

then $CD_1(V)$

else $(CD_1(V)_1, any)$

Figure 59: Handling Unknown Predicates for Aquarius Domain

proc write_desc(D : *Description*, CV : list of *Var*, P : *string*) :

if D = $\perp$

then write P, "BOTTOM"

else

$\forall$ V $\in$ CV, write P, V, ": ", state(V,D)

$\forall$ u $\in$ D_2, write P, u

write P, "Uninit Reg vars = ", D_3

write P, "Bound vars = ", D_4

Figure 60: Writing Aquarius Domain Descriptions

predicate and TD is the description which is valid prior to exiting it. This function is defined in Figure 62.

2. For user-defined predicates, this is the set of all arguments. For built-ins, more exact information is provided.

proc prepare_head(H : *Goal*, PI : *Term*) :

PI := (vars(H), true)

proc prepare_clause(H : *Goal*, B : *Goals*, CI : *Term*, PI : *Term*, PI' : *Term*) :

CI := (vars(B) - vars(H), dups(B))

PI' := (uninit_regs(H, PI_1, B), $PI_2 \wedge \forall$ i, $1 \le i \le$ length(B), survive(B_i))

proc prepare_tail(H : *Goal*, PI : *Term*, PI' : *Term*) :

PI' := PI

Figure 61: Completely Static Analysis for Aquarius Domain

Extracting Information from Descriptions

The function, desc_implies(C, D), tests to see if condition C holds for domain D. Possible conditions are listed in Table 15. This function is defined in Figure 63.

function desc_implies(C : *Term*, D : *Description*) : *Boolean* =

switch (C) {

ground(X) : return $D_1(X)_1 \sqsubseteq_I$ *ground*

nonvar(X) : return $D_1(X)_1 \sqsubseteq_I$ *nonvar*

var(X) : return $D_1(X)_1 \sqsubseteq_I$ *uninit*

uninit(X) : return $D_1(X)_1 \sqsubseteq_I$ *uninit*

new(X) : return $D_1(X)_1 \sqsubseteq_I$ *new*

uninit_reg(X) : return $X \in D_3$

rderef(X) : return $D_1(X)_2 \sqsubseteq_R$ *rderef*

deref(X) : return $D_1(X)_2 \sqsubseteq_R$ *rderef*

otherwise: return false

}

Figure 63: Extracting Information from Aquarius Domain Descriptions

```
proc formulas_to_descs( H : Goal, HF : Formula, TF : Formula, BV : Var,
                        HD : Description, TD : Description ) :
    if HF = "fail"
        then HD := ⊥
        else HG := prop( ground, HF )
            HN := prop( nonvar, HF )
            HU := prop( uninit, HF )
            HR := prop( rderef, HF )
            HD := ( sets_to_states( H, HG, HN, HU, HR, ∅ ), ∅, ∅, ∅ )
    if TF = "fail"
        then TD := ⊥
        else TG := prop( ground, TF )
            TN := prop( nonvar, TF )
            TR := prop( rderef, TF )
            TU := prop( uninit, TF )
            TL := prop( locally_deref, TF )
            TS := sets_to_states( H, TG, TN, TU, TR, TL )
            TD := ( TS, ∅, prop( uninit_reg, HF ), BV )
```

Figure 62: Converting Mode Formulas to Aquarius Domain Descriptions

B.5 Utility Functions

The following sections define some utility functions used by the previous functions. When these functions operate on descriptions, they assume the descriptions are not ⊥. This case is handled by the calling functions.

Uninit_regs

The function, uninit_regs (H, UR, B) returns the set of argument variables that can possibly be passed to a predicate via the *uninit_reg* mechanism; variables that are not in this set definitely cannot be. H is the canonical head of the predicate. UR is the set of argument registers that are currently still allowed to be uninit_regs. B is the body of a clause of the predicate. This function is called from prepare_clause, for each clause, each time further restricting the possibilities. The function is defined in Figure 64.

<u>function</u> uninit_reg(H : *Goal*, UR : *Var*, B : *Goals*) : *Var* =

    <u>if</u> n = 0 /* This clause is a fact */

        <u>then</u> UR

    <u>else</u> <u>if</u> survive(B_n)

        <u>then</u> UR’

    <u>else</u> <u>if</u> B_n calls an unknown predicate

        <u>then</u> ∅

    <u>else</u> { V | V ∈ UR’ ∧ ∃ j ∋ j   arity(H)∧ j   arity(B_n) ∧ V = arg(j,B_n) = arg(j,H) }

where:

    n = length(B)

    i = smallest integer ∋ i > 0 ∧ ∀ j > i, survive(B_j)

    UR’ = UR - vars (B_1, ... , B_{i-1}) - dups (B_i, ..., B_n)

Figure 64: Determining which arguments can be uninit_reg

Sets_to_states

The function, sets_to_states(H, G, N, U, R, L), converts a number of sets of variables, indicating which arguments to a predicate have certain properties, into an element of the first component of an Aquarius description, mapping the arguments of a predicate into their states. H is the canonical head of the predicate. G is the set of arguments known to be ground. N is the set of arguments known to be bound; this is a superset of G. U is the set of arguments known to be uninitialized. R is the set of arguments known to be recursively dereferenced. L is the set of arguments known to be locally dereferenced; this is used on exit from built-ins to indicate those arguments which are made locally dereferenced by calling the built-in. This function is defined in Figure 65.

function sets_to_states(H : *Goal*, G, N, U, R, L) : (*Var* $\rightarrow$ *S*) = f

where:

f(V) = if V $\notin$ vars(H)

then $\perp_S$

else if V $\in$ U

then (*uninit*,*rderef*)

else (S_I,S_R)

S_I = if V $\in$ G

then *ground*

else if V $\in$ N

then *nonvar*

else *any*

S_R = if V $\in$ R

then *rderef*

else if V $\in$ L

then *locally_deref*

else *any*

Figure 65: Converting property sets to states

Save_unify

The function, save_unify (A, B, CD, ID) updates the description ID to save information about the explicit unification of A with B. This information is used to improve the state information at the end of the clause. This function can be seen to be monotonic. The function is defined in Figure 66.

Back_prop_unify

The function, back_prop_unify (D) updates the description D by back-propagating mode information through explicit unifications that have occurred so far in the clause. This function can be seen to be monotonic. The function is defined in Figure 67.

function save_unify(A : *Term*, B : *Term*, CD : set of *Var*, ID : *Description*) :
Description =

if nonvar(A)
then ID
else (ID_1, ID_2', ID_3, ID_4)

where:

$ID_2' = ID_2 \cup (A, B, BN, DF)$

$BN = \{ V \mid V \in vars(B) \wedge state(V,ID) = (\mathit{new},\mathit{rderef}) \}$

$DF = (state(A,ID)_1 \sqsubseteq_I \mathit{uninit} \wedge A \notin CD \wedge \forall V \in vars(B), state(V,ID) \in \{ (\mathit{nonvar},\mathit{rderef}), (\mathit{ground},\mathit{rderef}), (\mathit{new},\mathit{rderef}), (\mathit{nonvar},\mathit{tmpdrf}), (\mathit{ground},\mathit{tmpdrf}) \}) \})$

Figure 66: Saving information about explicit unifications

Worst_case_for_aliasing

The function, worst_case_for_aliasing (UV, D), updates the description D based on some unification involving the variables in UV to reflect the worst case aliasing among these variables. This function can be seen to be monotonic. The function is defined in Figure 68.

function worst_case_for_aliasing (UV: set of *Var*, X : *Description*) : *Description* =
(f: *Var* → *S*, D_2, D_3, D_4)

where:

$f(V) =$ if $V \notin UV \wedge \mathit{nonvar} \sqsubseteq_I X_1(V)_1$
then ($X_1(V)_1$, *any*)
else $X_1(V)$

Figure 68: Dealing with Worst-case Aliasing

Pass

The function, pass (Dir, A, B, DIV, AD, BD), updates the description AD to reflect the unification of A with B. Dir is the direction of the unification. This is *in* for proc back_prop_unify(D : *Description*) : *Description*

  loop until no more changes in D occur:

    $\forall$ (A,B,BN,DF) $\in$ D_2:

      AS := state(A,D)

      BS := term_state (B, D)

      AS'_1 := uni_var_mode (AS_1, BS_1)

      if DF $\wedge$ $BS_2 \sqsubseteq_R$ *locally_deref* $\wedge$ $\forall$ V $\in$ BN, $D_1(V)_2 \sqsubseteq_R$ *rderef*

        then AS'_2 := *rderef*

        else AS'_2 :=AS_2

      if AS'   AS

        then D := update_state (A, AS', D)

      if $AS_1 \sqsubseteq_I$ *ground* $\wedge$ $BS_1 \not\sqsubseteq_I$ *ground*

        then $\forall$ V $\in$ vars(B), state(V,D) $\not\sqsubseteq_I$ *ground*

          D := update_state (V, (*ground*, $\text{state}(V,D)_2$), D)

  return D

Figure 67: Back-Propagating Unification Information

predicate entry and *out* for predicate exit. AD is the description valid in the scope of A. BD is the description valid in the scope of B. DIV is a set of variables in the scope of B that must definitely be initialized (i.e., variables in A must be initialized as a result of the unification if variables in B are members of DIV). Since this function is used to pass into or out of the most general head of a predicate, it never needs to handle the unification of two terms. This function can be seen to be monotonic since the functions it calls are monotonic. The function is defined in Figure 69.

Unify_var_with_var

The function, unify_var_with_var (Dir, A, X, B, Y, DIV), updates the description AD to reflect the unification of the variable A with the variable B. Dir is the direction of the unification. This is *in* for predicate entry, *out* for predicate exit, and *unify* for explicit unification. ad is the description valid in the scope of A. BD is the description

```
proc pass ( Dir : Atom, A : Goal, B : Goal, DIV : set of Var, AD : Description,
            BD : Description ) : Description
    AD' := AD
    ∀ i, 1   i   arity(A):
        Ai := arg(i,A)
        Bi := arg(i,B)
        if nonvar(Ai)
            then AD' := unify_term_with_var ( Dir, Ai, AD', Bi, BD, DIV )
        else if nonvar(Bi)
            then AD' := unify_var_with_term ( Ai, AD', Bi, BD )
        else AD' := unify_var_with_var ( Dir, Ai, AD', Bi, BD, DIV )
    return AD'
```

Figure 69: Passing information into and out of a Predicate

valid in the scope of B. DIV is a set of variables in the scope of B that must definitely be initialized (i.e., A must be initialized as a result of the unification if B is a member of DIV). This function can be shown to be monotonic, exhaustively, by examining the results based on all values of Dir, the states of A and B, and whether or not B is in DIV. The function is defined in Figure 70.

Unify_var_with_term

The function, unify_var_with_term (A, AD, B, BD) updates the description AD to reflect the unification of the variable A with the term B. AD is the description valid in the scope of A. BD is the description valid in the scope of B. This function can be seen to be monotonic. The function is defined in Figure 71.

Unify_term_with_var

The function, unify_term_with_var (A, AD, B, BD, DIV) updates the description AD to reflect the unification of the term A with the variable B. AD is the description valid in the scope of A. BD is the description valid in the scope of B. DIV is a set of variables in the scope of B that must definitely be initialized (i.e., A must be initialized function unify_var_with_var (Dir, A, AD, B, BD, DIV) =

update_state (A, (AS'_1, AS'_2), AD)

where:

AS = state(A,AD)

BS = state(B,BD)

AS'_1 = if $BS_1 \not\sqsubseteq_I$ *uninit* $\vee$ B $\in$ DIV $\vee$ Dir = *out* $\vee$ $AM_1 \not\sqsubseteq_I$ *new*

then uni_var_mode (AS_1, BS_1)

else *uninit*

AS'_2 =

if AS $\sqsubseteq_S$ (*ground*,*rderef*) $\vee$

Dir = *out* $\wedge$ AS_2 = *rderef* $\wedge$ BS_2 = *rderef* $\vee$

AS $\sqsubseteq_S$ (*uninit*,*rderef*) $\wedge$ $BS_2 \sqsubseteq_R$ *locally_deref* $\vee$

Dir = *unify* $\wedge$ $BS_1 \sqsubseteq_I$ *uninit* $\wedge$ AS_2 = *rderef*

then *rderef*

else if Dir = *unify* $\wedge$ AS_2 = *locally_deref* $\wedge$ BS_1 *uninit* $\vee$

Dir = *out* $\wedge$ AS $\not\sqsubseteq_S$ (*ground*,*rderef*) $\wedge$ BS_2 = *locally_deref* $\vee$

Dir = *out* $\wedge$ AS_2 = *locally_deref* $\wedge$ BS_2 = *rderef* $\vee$

Dir = *out* $\wedge$ $AS_1 \sqsubseteq_I$ *ground* $\wedge$ AS_2 = *locally_deref* $\wedge$ BS_2 = *any*

then *locally_deref*

else *any*

Figure 70: Abstract Unification of Two Variables

as a result of the unification if B is a member of DIV). This function can be seen to be monotonic. The function is defined in Figure 72.

Term_state

The function, term_state (X, D), returns the state of a term or variable X based on the states of the variables in the term. D is a description containing the states for the variables in X. This function can be seen to be monotonic by inspection of tm_1 and the fact that the least upper bound (lub) operation is monotonic. The function is defined in Figure 73.

function unify_var_with_term (A, AD, B, BD) : *Description* =

update_mode (A, (AS'_1, AS'_2), AD)

where:

AS = state(A,AD)

BS = term_state (B, BD)

AS'_1 := uni_var_mode (AS_1, BS_1)

AS'_2 = if $AS \sqsubseteq_S (\textit{ground},\textit{rderef}) \vee (AS_1 \sqsubseteq_I \textit{uninit} \wedge BS_2 \sqsubseteq_R \textit{locally\_deref})$

then *rderef*

else *any*

Figure 71: Abstract Unification of a Variable with a Term

function unify_term_with_var (A, AD, B, BD, DIV) : *Description* =

(f: $\textit{Var} \rightarrow S$, AD_2, AD_3, AD_4)

where:

$f(V) = \text{if } V \notin \text{vars}(A)$

then $AD_1(V)$

else ($N_1(V)$, $N_2(V)$)

BS = state(B,BD)

$DF1 = (BS_2 = \textit{rderef} \wedge (Dir = \textit{out} \vee (Dir = \textit{unify} \wedge BS_1 \sqsubseteq_I \textit{uninit})))$

$DF2 = (BS = (\textit{ground},\textit{rderef}) \wedge Dir = \textit{unify})$

$N_1(V) = \text{if } BS_1 \sqsubseteq_I \textit{uninit} \wedge B \notin DIV \wedge VS_1 \sqsubseteq_I \textit{new} \wedge V \notin \text{dups}(A)$

then *uninit*

else uni_term_mode (VS_1, BS_1)

$N_2(V)$ = if $VS \sqsubseteq_S (\textit{ground},\textit{rderef}) \vee DF1 \wedge VS_2 = \textit{rderef} \vee$

$DF2 \wedge VS_1 \sqsubseteq_I \textit{uninit} \wedge V \notin \text{dups}(A)$

then *rderef*

else *any*

Figure 72: Abstract Unification of a Term with a Variable

function term_state (X : *Term*, D : *Description*) : *S* =

if var(X)

then state(X,D)

else if nonvar(X) $\wedge$ vars(X) = $\varnothing$

then (*ground*,*rderef*)

else (tm_1,tm_2)

where:

tm_1 = if $\forall$ V $\in$ vars(X), state(V,D)$_1$ $\sqsubseteq_I$ *ground*

then *ground*

else *nonvar*

tm_2 = $\sqcup_S$ ({ state(V,D)$_2$ | V $\in$ vars(X) })

Figure 73: Computing the state of a term

Uni_var_mode

The function, uni_var_mode(X, Y) returns the instantiation mode of a variable after unification with another value (term or variable). X is the original mode of the variable. Y is the mode of the other value. This function can be seen to be monotonic by inspection. The function is defined by Table 46.

Table 46: Variable Unification Mode

X: \ Y:	bottom	ground	nonvar	any	uninit	new
bottom	bottom	bottom	bottom	bottom	bottom	bottom
ground	bottom	ground	ground	ground	ground	ground
nonvar	bottom	ground	nonvar	nonvar	nonvar	nonvar
any	bottom	ground	nonvar	any	any	any
uninit	bottom	ground	nonvar	any	any	any
new	bottom	ground	nonvar	any	any	any

Uni_term_mode

The function, uni_term_mode (X, Y) returns the instantiation mode of a variable, which is part of a term, after unifying the term with another value (term or variable). X is the original mode of the variable. Y is the mode of the other value. This function can be seen to be monotonic by inspection. The function is defined by Table 47.

Table 47: Term Unification Mode

X: \ Y:	bottom	ground	nonvar	any	uninit	new
bottom	bottom	bottom	bottom	bottom	bottom	bottom
ground	bottom	ground	ground	ground	ground	ground
nonvar	bottom	ground	nonvar	nonvar	nonvar	nonvar
any	bottom	ground	any	any	any	any
uninit	bottom	ground	any	any	any	any
new	bottom	ground	any	any	any	any

B.6 Conclusions

The abstract domain used in the Aquarius compiler is non-trivial. We have formally defined this domain and the operations over this domain needed to perform abstract interpretation. We have shown these operations to be monotonic, which is crucial is showing that the abstract interpretation will terminate (will reach a fixpoint). This formalization has been useful in uncovering numerous errors in the original operations. Fixing these errors tended to make the results of the analysis less precise. In order to compensate for this, we have made both the abstract domain and the operations more complex. The result is a set of operations which are sound and provide results that are approximately as precise as those from the original analysis.

Appendix C: Review of Lattice Theory

This appendix provides a review of lattice theory, defining the terminology used in this dissertation.

Let S be a set.

- $R \subseteq S \times S$ is a *relation* on S. $(x,y) \in R$ is written in infix notation as xRy.

A relation R on a set S is a *partial order* if

- $\forall\ x \in S$, xRx (the relation is reflexive),
- $\forall\ x,y \in S$, $xRy \wedge yRx \rightarrow x=y$ (the relation is antisymmetric), and
- $\forall\ x,y,z \in S$, $xRy \wedge yRz \rightarrow xRz$ (the relation is transitive).

A *partially ordered set* or *poset* is a pair consisting of:

- a set S and
- a partial order R on that set.

Let S be a set with a partial order $\sqsubseteq$.

- $a \in S$ is an *upper bound* of a subset X of S iff $\forall\ x \in X$, $x \sqsubseteq a$.
- $b \in S$ is a *lower bound* of a subset X of S iff $\forall\ x \in X$, $b \sqsubseteq x$.
- $a \in S$ is the *least upper bound (lub)* of a subset X of S iff a is an upper bound of $X \wedge \forall\ a' \in S$, a' is an upper bound of $X \rightarrow a \sqsubseteq a'$. If the lub of X exists, it is unique and written as $\sqcup X$.
- $b \in S$ is the *greatest lower bound (glb)* of a subset X of S iff b is a lower bound of $X \wedge \forall\ b' \in S$, b' is a lower bound of $X \rightarrow b' \sqsubseteq b$. If the glb exists, it is unique and written as $\sqcap X$.

A poset L is a *complete lattice* iff $\forall\ X \subseteq L$, $\sqcup X$ and $\sqcap X$ exist. In this case:

- $\sqcup L$, the top element, is written as $\top$.
- $\sqcap L$, the bottom element, is written as $\bot$.

Let L be a complete lattice and $T : L \rightarrow L$ be a mapping.

- T is *monotonic* iff $\forall x,y \in L$, $x \sqsubseteq y \rightarrow T(x) \sqsubseteq T(y)$
- $X \subseteq L$ is *directed* iff every finite subset of X has an upper bound in X.
- T is *continuous* iff $\forall X \subseteq L$, X is directed $\rightarrow T(\sqcup X) = \sqcup T(X)$.

Appendix D: Source Code

This appendix describes the source code for the analysis framework, compiler, and benchmark programs. It indicates where this code can be obtained and provides a summary of the contents of the source files.

D.1 Obtaining the Source Code

The source code used to obtain the results reported in this dissertation is available from the USC Advanced Computer Architecture Laboratory (ACAL) list-server. To find out what is available, send an e-mail message, with the word "`help`" as the message body, to `listserv@acal-server.usc.edu`.

There are two archives that will be of interest:

- aquarius-sources: This archive contains the source for the Aquarius compiler. This is a stand-alone implementation which runs on a number of platforms. Although we replaced the compiler, we use the back-end and run-time library for the DECstation. To obtain this archive, you must fill out and return a license agreement available through the list-server.
- compilation-benchmarks: This archive contains the benchmark programs used for measuring performance of the analyzer and the analysis results. The contents of this archive are described below.
- compiler-with-analysis-framework: This archive contains the enhanced Aquarius compiler into which the analysis framework has been integrated. It also includes a number of different abstract domains and operations, the source code for the benchmark programs, and a number of Unix scripts used to run the benchmark programs. The contents of this archive are described below.

D.2 Source Code Archive

Table 48 describes the structure and contents of the ACAL list-server archive containing the source code for the enhanced compiler and benchmarks.

Table 48: Source Code Archive Contents

File	Contents/Purpose
bench/	This directory holds the benchmark source code.
boyer.pl, browse.pl, crypt.pl, deriv.pl, fast_mu.pl, flatten.pl, meta_qsort.pl, mu.pl, nand.pl, nreverse.pl, poly_10.pl, press1.pl, pri2.pl, prover.pl, qsort.pl, queens_8.pl, query.pl, reducer.pl, sendmore.pl, serialise.pl, simple_analyzer.pl, tak.pl, unify.pl, zebra.pl	These files contain the source code for the benchmark programs. They are very similar to those used by Van Roy [86] and Taylor [80]. The main difference is the elimination of input and output (this is done in C in Aquarius and not considered part of the problem we are attacking).
sources/	This directory contains source files from the Aquarius compiler which were not affected by the analyzer.
compile.compiler.QUINTUS2, compile.compiler.QUINTUS3	These are scripts for building the compiler to run under Quintus Prolog.
config.h	This file defines a number of parameters configuring the compiler. We configured it the same as the distributed version of Aquarius Prolog.
copyright.pl	This is the copyright statement for the compiler.
op_defs.pl	This defines operators used in the compiler.
registers.h	This tells the compiler about the target processor's registers.
version.pl	This identifies the Aquarius Prolog version.
build/	This is the directory containing the compiler and analyzer source code. This is also the directory in which the compiler is built.
AQUARIUS_comp_head.pl, QUINTUS2_comp_head.pl, QUINTUS3_comp_head.pl	These are prefix files, used to specify differences in the compiler when compiling it under different Prolog systems (e.g., Quintus or Aquarius).
Makefile	This is the makefile, which builds the compiler.
compile.cpp	This is the top-level control for the compiler. It reads the source code, invokes the various compilation phases, and writes the results.

Table 48: Source Code Archive Contents

File	Contents/Purpose
preamble.cpp	This combines clauses into predicates, by collecting clauses with the same functor in the head. It also performs part of the translation to Kernel Prolog.
expression.cpp	This is used from preamble.cpp to translate arithmetic expressions into calls to Aquarius' low-level arithmetic built-ins (e.g., '$add'(A,B,C)).
transform_cut.cpp	This is called after preamble.cpp to translate cuts and if-then-else into calls to Aquarius' low-level cut built-ins (e.g., '$cut_load'(X)).
factor.cpp	This is called after transform_cut.cpp to combine clauses whose heads have non-trivial most specific generalizers, thereby factoring out common head unifications.
standard.cpp	This is called after factor.cpp to continue the conversion to Kernel Prolog. This file is responsible for unraveling the clause heads.
flatten.cpp	This is called after standard.cpp to complete the conversion to Kernel Prolog. This file ensures that all predicates are 'flat', that is, that they consist of disjunctions of conjunctions of simple goals.
inline.cpp	This is called after flatten.cpp to perform in-line expansion of 'short' predicates.
analyze.cpp	This is called after inline.cpp. It is interface between the compiler and the dataflow analyzer.
fail.cpp	This is the dataflow analysis framework. It performs abstract interpretation of the Prolog code until a fixpoint is reached, saving the analysis results. It it also used during code generation to redo the analysis locally to a clause.
predinfo.cpp	This file contains descriptions of the built-in predicates, used during analysis.

Table 48: Source Code Archive Contents

File	Contents/Purpose
aiutil.cpp	This file contains a number of utility predicates useful when implementing abstract operations for the analysis framework.
absdom_aquarius.cpp	This defines the abstract operations for the Aquarius abstract domains, defined in Appendix B.
absdom_product.cpp, absdom_access.cpp, absdom_aliasing.cpp, absdom_mode.cpp, absdom_type.cpp, absdom_ref.cpp, absdom_rtype.cpp	This defines the abstract operations for the product domain. absdom_product.cpp is the top-level file. The remaining files each provide one component of the product. absdom_rtype.cpp provides fully recursive types and some aliasing.
segment.cpp	This is called from the code generator (in compile.cpp) after the analysis completes. It uses the analysis results to segment clauses into those goals useful for determinism extraction and the remaining goals.
selection.cpp	This is called after segment.cpp to convert the predicates into deterministic code (when possible).
testset.cpp	This is called from selection.cpp to collect test-sets for clauses (tests usable for deterministic clause selection).
proc_code.cpp	This is the procedure compiler. It generates the deterministic or non-deterministic clause selection code, based on the results of selection.cpp.
clause_code.cpp	This is the clause compiler. It generates the code to call each of the goals in a clause.
unify.cpp	This is the unification compiler, called from clause_code.cpp to deal with explicit unifications.
conditions.cpp	This file performs code generation for simple built-ins, such as arithmetic evaluation and comparison and cut.

Table 48: Source Code Archive Contents

File	Contents/Purpose
regalloc.cpp	This is the register allocator, used to allocate temporary and permanent registers used within a clause.
peephole.cpp	This is the BAM instruction-level optimizer, called once all code has been generated for a procedure.
synonym.cpp	This is called from peephole.cpp to perform strength reduction on the instruction operands.
tables.cpp	This contains a number of miscellaneous tables needed by the compiler.
utility.cpp	This contains a number of utility predicates used throughout the compiler.
mutex.cpp	This provides some utilities for manipulating Aquarius mode formulas.
multiple.cpp	This provided an ‘optimization’ which wasn’t included as part of the Aquarius system.

Appendix E: Performance Measurement Results

This appendix documents the performance results from various experiments with different abstract domains. The benchmarks and techniques used to measure performance have been described in Chapter 6.

E.1 Performance on Aquarius Prolog

Table 49 shows performance figures for the benchmarks compiled by Aquarius Prolog. The first column shows the compilation time, in seconds. The next two columns show the global analysis time, in seconds and as a percentage of the compilation time. The next column shows the static code size of the compiled benchmark. The last column shows the execution time, in terms of the number of instructions executed.

E.2 Performance Comparison and Analysis

Tables 50 through 84 show the performance of the benchmarks when compiled using our compiler with various combinations of abstract domains. The first column identifies the benchmark. The remaining columns provide a number of cost and performance results. Each is given as an absolute figure and then relative to Aquarius. The cost figures are compilation time and analysis time, in seconds. The performance figures are static code size, in bytes, and execution time, in number of instructions.

Table 49: Performance using Aquarius Prolog System

Benchmark	Compilation time (Sec)	Analysis time (Sec)	Analysis time % Ct	Static code size (Bytes)	Execution time (insts)
deriv	10.7	0.5	4.7	2,540	3,622
nreverse	3.0	0.2	6.7	752	7,024
qsort	5.4	0.5	9.3	1,496	8,515
serialise	9.2	0.4	4.3	3,624	21,482
mu	12.6	0.9	7.1	5,348	39,274
pri2	5.1	0.5	9.8	1,016	29,424
queens_8	8.8	0.6	6.8	2,652	64,129
fast_mu	37.9	2.5	6.6	10,440	51,909
query	9.5	0.5	5.3	3,368	77,707
press1	100.7	8.0	7.9	40,196	60,307
tak	4.3	0.2	4.7	864	1,415,344
sendmore	82.4	0.8	1.0	8,236	2,052,216
poly_10	54.6	2.3	4.2	5,816	999,838
zebra	25.8	0.3	1.2	5,148	4,150,368
prover	27.8	1.3	4.7	8,876	37,652
meta_qsort	24.0	0.9	3.8	9,292	196,659
nand	351.0	35.3	10.1	56,160	578,476
chat_parser	447.5	35.4	7.9	150,524	5,000,476
browse	23.8	2.0	8.4	8,412	36,012,047
unify	68.4	9.8	14.3	35,116	49,164
flatten	38.6	3.4	8.8	16,388	27,260
crypt	24.7	1.2	4.9	8,436	92,044
simple_analyzer	176.3	11.4	6.5	34,132	780,335
reducer	618.5	6.2	1.0	160,724	1,626,219
boyer	139.5	3.2	2.3	60,360	34,005,215

Table 50: Performance with No Analysis (Domain = ⊥)

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	8.6	0.86	0.3	0.72	12864	5.06	25049	6.92
nreverse	2.0	0.91	0.2	2.50	2912	3.87	63982	9.11
qsort	3.4	0.72	0.3	0.65	4980	3.33	57164	6.71
serialise	5.8	0.68	0.3	0.70	13796	3.81	54605	2.54
mu	5.2	0.44	0.3	0.39	13400	2.51	111642	2.84
pri2	3.7	0.82	0.3	0.68	6144	6.05	123140	4.19
queens_8	4.9	0.60	0.3	0.54	10212	3.85	358567	5.59
fast_mu	12.7	0.33	0.3	0.13	25132	2.41	99837	1.92
query	5.9	0.68	0.3	0.63	13468	4.00	655142	8.43
press1	49.5	0.50	1.7	0.21	113880	2.83	145411	2.41
tak	2.1	0.58	0.2	1.79	3596	4.16	7084251	5.01
sendmore	32.3	0.39	0.3	0.39	22508	2.73	5003629	2.44
poly_10	17.3	0.32	0.5	0.20	38284	6.58	6223318	6.22
zebra	14.6	0.62	0.3	1.00	14968	2.91	5373698	1.29
prover	19.9	0.75	0.4	0.33	30684	3.46	112364	2.98
meta_qsort	10.5	0.45	0.4	0.45	18352	1.98	643950	3.27
nand	252.7	0.71	3.6	0.10	175764	3.13	2569453	4.44
chat_parser	106.6	0.24	6.3	0.18	231308	1.54	9930133	1.99
browse	11.3	0.49	0.5	0.23	28416	3.38	109077666	3.03
unify	45.9	0.67	1.5	0.15	74432	2.12	115217	2.34
flatten	20.8	0.55	1.3	0.38	46168	2.82	49678	1.82
crypt	12.0	0.49	0.4	0.32	23564	2.79	427801	4.65
simple_analyzer	36.4	0.21	2.0	0.18	95040	2.78	1522206	1.95
reducer	107.9	0.18	1.5	0.25	87820	0.55	3751736	2.31
boyer	152.0	1.11	1.5	0.46	168624	2.79	139167302	4.09
Geometric Mean:		0.52		0.39		2.97		3.44

Table 51: Performance with Domain $AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	7.0	0.70	0.5	1.23	7344	2.89	15670	4.33
nreverse	1.9	0.86	0.4	3.90	1712	2.28	25921	3.69
qsort	2.9	0.62	0.4	1.05	3532	2.36	31145	3.66
serialise	4.6	0.54	0.5	1.30	7280	2.01	28127	1.31
mu	4.8	0.40	0.5	0.59	8520	1.59	56401	1.44
pri2	3.6	0.80	0.4	1.05	4060	4.00	77482	2.63
queens_8	4.8	0.59	0.5	0.83	6984	2.63	186815	2.91
fast_mu	13.9	0.36	0.5	0.22	17464	1.67	66415	1.28
query	4.7	0.54	0.5	0.96	9356	2.78	505104	6.50
press1	43.1	0.43	2.3	0.29	74028	1.84	95775	1.59
tak	2.1	0.58	0.4	2.79	1908	2.21	3800657	2.69
sendmore	21.4	0.26	0.5	0.61	13620	1.65	3145769	1.53
poly_10	12.2	0.23	0.8	0.35	21552	3.71	3278830	3.28
zebra	18.4	0.78	0.5	1.93	7976	1.55	4202162	1.01
prover	16.5	0.62	0.7	0.59	20488	2.31	63530	1.69
meta_qsort	9.4	0.40	0.6	0.71	11660	1.25	419052	2.13
nand	144.8	0.41	5.5	0.16	133000	2.37	1240549	2.14
chat_parser	83.7	0.19	8.6	0.24	171708	1.14	7312984	1.46
browse	10.1	0.44	0.8	0.42	15508	1.84	48226553	1.34
unify	40.8	0.60	2.1	0.22	53200	1.51	80576	1.64
flatten	18.8	0.50	1.7	0.51	29872	1.82	35524	1.30
crypt	9.3	0.38	0.7	0.59	14064	1.67	205441	2.23
simple_analyzer	31.2	0.18	3.5	0.31	61976	1.82	1101453	1.41
reducer	91.6	0.15	2.0	0.32	62060	0.39	2258287	1.39
boyer	124.1	0.91	1.9	0.58	120452	2.00	99080599	2.91
Geometric Mean:		0.45		0.62		1.90		2.05

Table 52: Performance with Domain $M_1 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	7.1	0.71	0.6	1.40	6116	2.41	6331	1.75
nreverse	1.9	0.86	0.4	4.40	1356	1.80	25276	3.60
qsort	3.3	0.70	0.6	1.48	3244	2.17	31145	3.66
serialise	4.5	0.53	0.6	1.62	6468	1.78	27512	1.28
mu	5.0	0.42	0.8	0.90	8500	1.59	56386	1.44
pri2	3.8	0.84	0.5	1.27	3804	3.74	76408	2.60
queens_8	5.0	0.62	0.6	1.06	6200	2.34	183605	2.86
fast_mu	14.1	0.36	0.8	0.32	14240	1.36	60305	1.16
query	4.8	0.55	0.5	1.02	8756	2.60	166062	2.14
press1	46.4	0.47	4.2	0.53	70816	1.76	92961	1.54
tak	2.1	0.58	0.4	3.14	1476	1.71	3180468	2.25
sendmore	20.8	0.25	0.5	0.70	11732	1.42	2906829	1.42
poly_10	13.0	0.24	1.8	0.79	20388	3.51	3240554	3.24
zebra	18.1	0.77	0.6	2.04	7976	1.55	4202162	1.01
prover	16.6	0.62	0.9	0.74	20488	2.31	63530	1.69
meta_qsort	9.5	0.41	0.7	0.86	11620	1.25	419052	2.13
nand	166.4	0.47	7.2	0.20	120996	2.15	1001166	1.73
chat_parser	88.0	0.20	10.8	0.31	170768	1.13	7299293	1.46
browse	10.9	0.47	1.6	0.78	14288	1.70	47630462	1.32
unify	41.5	0.61	2.8	0.29	47676	1.36	73135	1.49
flatten	19.1	0.51	2.1	0.62	27088	1.65	34202	1.25
crypt	11.4	0.47	0.8	0.68	10768	1.28	149399	1.62
simple_analyzer	38.9	0.22	4.2	0.37	56984	1.67	1031948	1.32
reducer	148.8	0.24	2.3	0.37	59824	0.37	2165028	1.33
boyer	125.7	0.92	2.0	0.64	120200	1.99	94304699	2.77
Geometric Mean:		0.48		0.80		1.72		1.79

Table 53: Performance with Domain $M_2 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.9	0.69	0.6	1.35	7344	2.89	15670	4.33
nreverse	1.9	0.86	0.4	4.30	1712	2.28	25921	3.69
qsort	2.9	0.62	0.5	1.12	3532	2.36	31145	3.66
serialise	4.7	0.55	0.5	1.41	7280	2.01	28127	1.31
mu	4.8	0.40	0.5	0.64	8520	1.59	56401	1.44
pri2	3.7	0.82	0.5	1.12	4060	4.00	77482	2.63
queens_8	5.0	0.62	0.5	0.94	6984	2.63	186815	2.91
fast_mu	14.1	0.36	0.7	0.27	17464	1.67	66415	1.28
query	4.5	0.52	0.5	0.96	8956	2.66	505097	6.50
press1	44.5	0.45	3.3	0.42	73692	1.83	95769	1.59
tak	2.2	0.61	0.4	3.00	1908	2.21	3800657	2.69
sendmore	21.4	0.26	0.5	0.70	13620	1.65	3145769	1.53
poly_10	12.0	0.22	0.9	0.38	21552	3.71	3278830	3.28
zebra	17.5	0.74	0.6	2.07	7248	1.41	4202156	1.01
prover	16.4	0.62	0.8	0.70	20288	2.29	63504	1.69
meta_qsort	9.4	0.40	0.6	0.75	11660	1.25	419052	2.13
nand	145.0	0.41	6.0	0.17	129856	2.31	1239748	2.14
chat_parser	85.5	0.19	10.1	0.29	169104	1.12	7312984	1.46
browse	10.3	0.44	1.4	0.71	15508	1.84	48226553	1.34
unify	39.5	0.58	2.5	0.26	52900	1.51	80630	1.64
flatten	16.9	0.45	1.8	0.54	29808	1.82	35524	1.30
crypt	9.3	0.38	0.8	0.68	14064	1.67	205441	2.23
simple_analyzer	32.1	0.18	4.0	0.35	61976	1.82	1101453	1.41
reducer	80.0	0.13	2.2	0.36	62316	0.39	2252062	1.38
boyer	123.5	0.90	2.0	0.63	119820	1.99	99080593	2.91
Geometric Mean:		0.45		0.71		1.88		2.05

Table 54: Performance with Domain $M_3 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	7.4	0.74	0.8	1.77	6116	2.41	6331	1.75
nreverse	2.0	0.91	0.6	5.50	1356	1.80	25276	3.60
qsort	3.5	0.74	0.7	1.80	2548	1.70	27888	3.28
serialise	4.8	0.56	0.8	2.05	6020	1.66	26937	1.25
mu	5.3	0.45	0.9	1.14	8500	1.59	56386	1.44
pri2	3.9	0.87	0.6	1.56	3412	3.36	75590	2.57
queens_8	5.2	0.64	0.7	1.31	5968	2.25	183599	2.86
fast_mu	14.7	0.38	1.5	0.63	14104	1.35	64589	1.24
query	5.0	0.57	0.6	1.24	8756	2.60	166062	2.14
press1	48.9	0.49	5.9	0.75	67664	1.68	92488	1.53
tak	2.2	0.61	0.5	3.86	1476	1.71	3180468	2.25
sendmore	21.7	0.26	0.7	0.84	11732	1.42	2906829	1.42
poly_10	13.9	0.26	2.4	1.04	20388	3.51	3240554	3.24
zebra	17.7	0.75	1.1	3.96	5956	1.16	4109254	0.99
prover	23.3	0.88	1.9	1.54	19780	2.23	61150	1.62
meta_qsort	9.9	0.42	0.9	1.12	10804	1.16	389649	1.98
nand	193.7	0.54	11.3	0.32	121416	2.16	986255	1.70
chat_parser	96.0	0.21	15.0	0.43	169124	1.12	7287389	1.46
browse	11.7	0.50	1.8	0.92	14188	1.69	48070562	1.33
unify	40.8	0.60	4.8	0.49	46768	1.33	72385	1.47
flatten	19.9	0.53	2.4	0.71	26800	1.64	34061	1.25
crypt	11.9	0.49	1.6	1.45	10768	1.28	149399	1.62
simple_analyzer	42.7	0.24	6.3	0.55	56148	1.65	1030643	1.32
reducer	154.9	0.25	3.5	0.57	61864	0.38	2163805	1.33
boyer	126.6	0.92	2.5	0.80	120200	1.99	94304699	2.77
Geometric Mean:		0.51		1.12		1.66		1.78

Table 55: Performance with Domain $M_4 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	7.4	0.74	0.8	1.95	6116	2.41	6331	1.75
nreverse	2.1	0.95	0.6	5.60	1328	1.77	25270	3.60
qsort	3.5	0.74	0.7	1.82	3244	2.17	31145	3.66
serialise	4.9	0.58	0.8	2.19	6468	1.78	27512	1.28
mu	5.5	0.46	1.6	1.93	8500	1.59	56386	1.44
pri2	3.9	0.87	0.6	1.56	3804	3.74	76408	2.60
queens_8	5.3	0.65	0.7	1.33	6200	2.34	183605	2.86
fast_mu	14.8	0.38	1.7	0.68	14240	1.36	60305	1.16
query	4.7	0.54	0.6	1.24	8364	2.48	166055	2.14
press1	49.3	0.50	6.3	0.79	70480	1.75	92955	1.54
tak	2.3	0.64	0.6	3.93	1332	1.54	2798812	1.98
sendmore	21.3	0.26	0.7	0.91	11352	1.38	2822641	1.38
poly_10	14.4	0.27	2.5	1.10	20232	3.48	3240520	3.24
zebra	19.0	0.81	1.7	6.19	7248	1.41	4202156	1.01
prover	17.8	0.67	1.9	1.55	20288	2.29	63504	1.69
meta_qsort	10.0	0.43	1.0	1.24	11620	1.25	419052	2.13
nand	178.1	0.50	14.8	0.42	119296	2.12	1000371	1.73
chat_parser	96.8	0.22	16.4	0.46	168164	1.12	7299293	1.46
browse	12.1	0.52	2.0	0.98	14160	1.68	47596462	1.32
unify	43.3	0.63	5.7	0.58	47304	1.35	73187	1.49
flatten	18.9	0.50	2.8	0.84	26960	1.65	34202	1.25
crypt	12.1	0.50	1.8	1.60	10768	1.28	149399	1.62
simple_analyzer	44.3	0.25	6.8	0.61	56836	1.67	1031948	1.32
reducer	139.4	0.23	3.9	0.63	60080	0.37	2158803	1.33
boyer	125.6	0.92	2.8	0.88	119568	1.98	94304693	2.77
Geometric Mean:		0.51		1.25		1.70		1.78

Table 56: Performance with Domain $M_5 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	7.5	0.75	0.8	1.88	6116	2.41	6331	1.75
nreverse	2.3	1.05	0.5	5.40	1160	1.54	18533	2.64
qsort	9.2	1.96	0.7	1.85	3496	2.34	21755	2.55
serialise	7.5	0.88	0.8	2.19	5412	1.49	25417	1.18
mu	9.1	0.76	1.6	1.98	10612	1.98	48609	1.24
pri2	6.1	1.36	0.7	1.61	4212	4.15	58289	1.98
queens_8	6.2	0.77	0.7	1.35	6384	2.41	146971	2.29
fast_mu	19.5	0.50	1.7	0.68	14640	1.40	57037	1.10
query	6.0	0.69	0.7	1.33	10644	3.16	164263	2.11
press1	115.8	1.16	6.6	0.83	78240	1.95	83658	1.39
tak	2.3	0.64	0.6	4.00	1332	1.54	2798812	1.98
sendmore	21.2	0.26	0.7	0.94	11352	1.38	2822641	1.38
poly_10	40.4	0.75	2.6	1.15	24464	4.21	2676213	2.68
zebra	17.7	0.75	1.8	6.52	5228	1.02	4109248	0.99
prover	47.9	1.80	2.0	1.69	21300	2.40	48934	1.30
meta_qsort	41.1	1.76	1.0	1.23	13844	1.49	254983	1.30
nand	268.9	0.75	14.3	0.40	121604	2.17	849467	1.47
chat_parser	267.1	0.60	18.3	0.52	244488	1.62	5769165	1.15
browse	15.0	0.65	2.0	0.99	13740	1.63	41122736	1.14
unify	40.7	0.60	5.9	0.61	47228	1.34	71607	1.46
flatten	37.8	1.01	2.7	0.79	30992	1.89	32127	1.18
crypt	12.8	0.53	1.8	1.57	10896	1.29	132810	1.44
simple_analyzer	56.9	0.32	7.0	0.62	60860	1.78	981683	1.26
reducer	708.9	1.15	3.9	0.63	69664	0.43	1955444	1.20
boyer	383.3	2.80	3.5	1.08	173776	2.88	43815133	1.29
Geometric Mean:		0.84		1.28		1.82		1.51

Table 57: Performance with Domain $M_6 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	7.5	0.75	0.8	1.93	6116	2.41	6331	1.75
nreverse	2.3	1.05	0.6	5.50	1160	1.54	18533	2.64
qsort	9.1	1.94	0.7	1.82	3496	2.34	21755	2.55
serialise	7.6	0.89	0.8	2.27	5412	1.49	25417	1.18
mu	9.0	0.76	1.6	1.95	10612	1.98	48609	1.24
pri2	6.1	1.36	0.7	1.66	4212	4.15	58289	1.98
queens_8	6.2	0.77	0.7	1.37	6384	2.41	146971	2.29
fast_mu	19.5	0.50	1.7	0.70	14640	1.40	57037	1.10
query	5.9	0.68	0.6	1.29	10644	3.16	164263	2.11
press1	115.2	1.16	6.5	0.83	78240	1.95	83658	1.39
tak	2.3	0.64	0.6	4.00	1332	1.54	2798812	1.98
sendmore	21.2	0.26	0.7	0.91	11352	1.38	2822641	1.38
poly_10	40.7	0.76	2.6	1.17	24464	4.21	2676213	2.68
zebra	17.7	0.75	1.8	6.59	5228	1.02	4109248	0.99
prover	47.7	1.79	2.0	1.68	21300	2.40	48934	1.30
meta_qsort	40.9	1.76	1.0	1.23	13844	1.49	254983	1.30
nand	269.4	0.75	14.3	0.40	121604	2.17	849467	1.47
chat_parser	265.2	0.59	18.3	0.52	244488	1.62	5769165	1.15
browse	15.1	0.65	2.0	0.98	13740	1.63	41122736	1.14
unify	41.0	0.60	5.8	0.60	47228	1.34	71607	1.46
flatten	38.1	1.02	2.6	0.79	30992	1.89	32127	1.18
crypt	12.9	0.53	1.8	1.58	10896	1.29	132810	1.44
simple_analyzer	57.2	0.32	7.0	0.62	60860	1.78	981683	1.26
reducer	707.5	1.15	4.1	0.66	69664	0.43	1955444	1.20
boyer	384.6	2.81	3.5	1.10	173776	2.88	43815133	1.29
Geometric Mean:		0.84		1.29		1.82		1.51

Table 58: Performance with Domain $M_7 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	7.5	0.75	0.8	1.93	6116	2.41	6331	1.75
nreverse	2.3	1.05	0.5	5.40	1160	1.54	18533	2.64
qsort	9.1	1.94	0.8	1.88	3496	2.34	21755	2.55
serialise	7.5	0.88	0.8	2.30	5412	1.49	25417	1.18
mu	9.2	0.77	1.7	2.01	10612	1.98	48609	1.24
pri2	6.2	1.38	0.7	1.63	4212	4.15	58289	1.98
queens_8	6.3	0.78	0.7	1.30	6384	2.41	146971	2.29
fast_mu	19.6	0.50	1.7	0.69	14640	1.40	57037	1.10
query	6.1	0.70	0.7	1.35	10644	3.16	164263	2.11
press1	116.0	1.17	6.6	0.83	78240	1.95	83658	1.39
tak	2.3	0.64	0.6	3.93	1332	1.54	2798812	1.98
sendmore	21.3	0.26	0.7	0.94	11352	1.38	2822641	1.38
poly_10	40.5	0.76	2.6	1.15	24464	4.21	2676213	2.68
zebra	17.9	0.76	1.8	6.63	5228	1.02	4109248	0.99
prover	47.7	1.79	2.0	1.69	21300	2.40	48934	1.30
meta_qsort	41.0	1.76	1.0	1.24	13844	1.49	254983	1.30
nand	269.0	0.75	14.1	0.40	121604	2.17	849467	1.47
chat_parser	264.2	0.59	18.2	0.52	244488	1.62	5769165	1.15
browse	15.1	0.65	2.0	0.99	13740	1.63	41122736	1.14
unify	40.9	0.60	5.8	0.60	47228	1.34	71607	1.46
flatten	37.9	1.01	2.7	0.79	30992	1.89	32127	1.18
crypt	13.0	0.53	1.8	1.62	10896	1.29	132810	1.44
simple_analyzer	56.8	0.32	7.2	0.63	60860	1.78	981683	1.26
reducer	705.9	1.15	3.9	0.63	69664	0.43	1955444	1.20
boyer	381.2	2.78	2.9	0.92	173776	2.88	43815133	1.29
Geometric Mean:		0.84		1.28		1.82		1.51

Table 59: Performance with Domain $T_1 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	9.6	0.96	0.9	2.02	6884	2.71	6007	1.66
nreverse	2.4	1.09	0.6	5.80	1160	1.54	18533	2.64
qsort	9.3	1.98	0.8	1.93	3436	2.30	21755	2.55
serialise	7.7	0.91	0.8	2.27	5272	1.45	25370	1.18
mu	8.4	0.71	1.7	2.05	10252	1.92	48574	1.24
pri2	4.8	1.07	0.7	1.76	3016	2.97	54672	1.86
queens_8	5.6	0.69	0.7	1.35	5552	2.09	144208	2.25
fast_mu	11.8	0.30	1.8	0.73	10932	1.05	47875	0.92
query	5.2	0.60	0.7	1.41	9272	2.75	149761	1.93
press1	118.5	1.19	6.6	0.83	77152	1.92	80480	1.33
tak	1.9	0.53	0.6	4.14	464	0.54	2226334	1.57
sendmore	3.9	0.05	0.7	0.92	2220	0.27	1833967	0.89
poly_10	41.0	0.76	2.7	1.20	24180	4.16	2657090	2.66
zebra	18.6	0.79	2.1	7.89	5228	1.02	4109248	0.99
prover	49.2	1.85	2.1	1.74	21300	2.40	48934	1.30
meta_qsort	41.8	1.79	0.9	1.08	13844	1.49	254983	1.30
nand	262.2	0.73	15.3	0.43	114764	2.04	822608	1.42
chat_parser	276.0	0.62	19.3	0.55	244792	1.63	5768753	1.15
browse	14.7	0.63	2.1	1.03	13520	1.61	40678856	1.13
unify	37.5	0.55	6.0	0.61	42396	1.21	66221	1.35
flatten	38.5	1.03	2.7	0.80	29912	1.83	31294	1.15
crypt	11.5	0.47	1.9	1.72	8892	1.05	125502	1.36
simple_analyzer	57.2	0.32	7.8	0.69	56680	1.66	967442	1.24
reducer	709.8	1.15	4.2	0.68	68264	0.42	1920361	1.18
boyer	304.5	2.22	3.5	1.11	154132	2.55	42279338	1.24
Geometric Mean:		0.75		1.34		1.54		1.43

Table 60: Performance with Domain $T_2 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	7.2	0.72	0.8	1.91	5860	2.31	6042	1.67
nreverse	2.3	1.05	0.6	5.70	1160	1.54	18533	2.64
qsort	9.0	1.91	0.7	1.75	3388	2.26	21605	2.54
serialise	7.6	0.89	0.8	2.27	5260	1.45	25343	1.18
mu	8.4	0.71	1.7	2.01	10252	1.92	48132	1.23
pri2	4.7	1.04	0.7	1.61	2980	2.93	54597	1.86
queens_8	5.6	0.69	0.8	1.44	5552	2.09	144208	2.25
fast_mu	11.9	0.31	1.8	0.72	10900	1.04	47335	0.91
query	5.1	0.59	0.7	1.33	9272	2.75	149761	1.93
press1	115.9	1.16	6.5	0.83	76936	1.91	80267	1.33
tak	1.9	0.53	0.7	4.86	464	0.54	2226334	1.57
sendmore	3.7	0.05	0.7	0.92	2220	0.27	1833967	0.89
poly_10	40.8	0.76	2.6	1.17	24168	4.16	2657075	2.66
zebra	18.6	0.79	2.1	7.78	5184	1.01	4104624	0.99
prover	42.0	1.58	2.3	1.91	22936	2.58	47103	1.25
meta_qsort	41.7	1.79	0.9	1.07	13800	1.49	249271	1.27
nand	155.4	0.43	13.6	0.39	80672	1.44	814139	1.41
chat_parser	275.2	0.61	18.8	0.53	247148	1.64	5723446	1.14
browse	14.9	0.64	2.2	1.10	13520	1.61	40678856	1.13
unify	37.0	0.54	6.2	0.64	40780	1.16	63816	1.30
flatten	38.6	1.03	2.9	0.86	29864	1.82	31496	1.16
crypt	11.3	0.47	1.8	1.57	8892	1.05	125502	1.36
simple_analyzer	55.9	0.32	8.1	0.72	55952	1.64	967442	1.24
reducer	693.4	1.13	4.0	0.64	70588	0.44	1913533	1.18
boyer	302.5	2.21	3.5	1.11	153952	2.55	42255206	1.24
Geometric Mean:		0.71		1.33		1.50		1.42

Table 61: Performance with Domain $T_3 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	7.7	0.77	1.6	3.63	5860	2.31	6042	1.67
nreverse	2.4	1.09	0.6	6.20	1148	1.53	18443	2.63
qsort	9.3	1.98	0.8	2.05	3400	2.27	21755	2.55
serialise	7.8	0.92	0.9	2.43	5248	1.45	25295	1.18
mu	8.5	0.71	1.7	2.07	10396	1.94	48064	1.22
pri2	4.8	1.07	0.7	1.73	2992	2.94	54672	1.86
queens_8	5.3	0.65	0.7	1.35	5416	2.04	137625	2.15
fast_mu	12.0	0.31	1.9	0.76	10872	1.04	47367	0.91
query	5.2	0.60	0.7	1.47	9272	2.75	149761	1.93
press1	113.7	1.14	6.9	0.87	74156	1.84	79952	1.33
tak	1.9	0.53	0.6	4.29	464	0.54	2226334	1.57
sendmore	3.9	0.05	0.8	1.00	2220	0.27	1833967	0.89
poly_10	41.7	0.78	2.9	1.30	24168	4.16	2657075	2.66
zebra	22.0	0.94	2.4	8.85	10568	2.05	2874300	0.69
prover	49.5	1.86	2.6	2.12	23032	2.59	47176	1.25
meta_qsort	41.6	1.79	1.4	1.70	13800	1.49	249271	1.27
nand	245.1	0.69	20.0	0.57	110656	1.97	808074	1.40
chat_parser	279.2	0.62	21.1	0.60	246620	1.64	5721818	1.14
browse	14.8	0.64	2.2	1.10	13016	1.55	40670756	1.13
unify	38.2	0.56	6.5	0.67	41488	1.18	66241	1.35
flatten	26.6	0.71	3.8	1.13	21664	1.32	31496	1.16
crypt	8.9	0.37	2.0	1.78	6504	0.77	113195	1.23
simple_analyzer	58.9	0.33	9.5	0.84	56584	1.66	967442	1.24
reducer	700.8	1.14	4.2	0.69	70620	0.44	1916185	1.18
boyer	302.4	2.21	3.7	1.17	153828	2.55	42237323	1.24
Geometric Mean:		0.73		1.51		1.52		1.39

Table 62: Performance with Domain $T_3 \times A_1 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	8.1	0.81	1.9	4.37	5856	2.31	5744	1.59
nreverse	2.9	1.32	0.7	7.30	1264	1.68	18439	2.63
qsort	9.8	2.09	1.6	3.95	3332	2.23	20990	2.47
serialise	8.2	0.96	1.5	4.14	5248	1.45	25295	1.18
mu	8.8	0.74	1.9	2.33	10464	1.96	48020	1.22
pri2	4.8	1.07	0.9	2.15	2784	2.74	53889	1.83
queens_8	5.6	0.69	0.9	1.70	4956	1.87	133581	2.08
fast_mu	12.5	0.32	2.0	0.83	10864	1.04	45564	0.88
query	5.3	0.61	0.8	1.71	6272	1.86	136787	1.76
press1	97.7	0.98	9.3	1.18	61532	1.53	76336	1.27
tak	2.0	0.56	0.7	4.93	464	0.54	2226334	1.57
sendmore	3.9	0.05	0.9	1.22	2180	0.26	1827545	0.89
poly_10	42.9	0.80	4.2	1.88	25028	4.30	2657075	2.66
zebra	22.4	0.95	2.7	9.89	9032	1.75	2727860	0.66
prover	23.4	0.88	3.5	2.93	10048	1.13	43506	1.16
meta_qsort	20.2	0.87	1.6	1.95	8268	0.89	215622	1.10
nand	212.3	0.59	25.7	0.73	95368	1.70	746445	1.29
chat_parser	257.3	0.57	29.2	0.83	197484	1.31	5420199	1.08
browse	15.6	0.67	2.7	1.36	12208	1.45	38509792	1.07
unify	39.6	0.58	8.3	0.85	35404	1.01	63107	1.28
flatten	27.9	0.74	4.5	1.35	19872	1.21	31142	1.14
crypt	9.4	0.39	2.3	2.04	6504	0.77	113195	1.23
simple_analyzer	61.9	0.35	13.1	1.16	50196	1.47	940006	1.20
reducer	698.7	1.14	5.8	0.94	61768	0.38	1876810	1.15
boyer	93.8	0.68	4.3	1.34	62468	1.03	41371627	1.22
Geometric Mean:		0.66		1.93		1.30		1.34

Table 63: Performance with Domain $T_3 \times A_2 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	8.0	0.80	1.9	4.30	5856	2.31	5744	1.59
nreverse	3.0	1.36	0.8	7.90	1264	1.68	18439	2.63
qsort	9.8	2.09	1.6	4.10	3332	2.23	20990	2.47
serialise	8.1	0.95	1.5	4.14	5248	1.45	25295	1.18
mu	9.3	0.78	2.0	2.36	10464	1.96	48020	1.22
pri2	4.9	1.09	0.9	2.20	2784	2.74	53889	1.83
queens_8	5.7	0.70	0.9	1.76	4956	1.87	133581	2.08
fast_mu	12.6	0.32	2.2	0.89	10864	1.04	45564	0.88
query	5.2	0.60	0.8	1.73	6272	1.86	136787	1.76
press1	99.6	1.00	10.6	1.34	60236	1.50	76206	1.26
tak	2.1	0.58	0.7	4.93	464	0.54	2226334	1.57
sendmore	4.0	0.05	0.9	1.21	2180	0.26	1827545	0.89
poly_10	43.2	0.81	4.5	1.97	25028	4.30	2657075	2.66
zebra	22.3	0.95	2.7	9.89	9032	1.75	2727860	0.66
prover	23.7	0.89	3.8	3.13	10048	1.13	43506	1.16
meta_qsort	20.4	0.88	1.6	1.88	8268	0.89	215622	1.10
nand	213.7	0.60	26.3	0.75	95368	1.70	746445	1.29
chat_parser	265.8	0.59	35.7	1.01	196244	1.30	5410101	1.08
browse	15.6	0.67	2.8	1.39	12208	1.45	38509792	1.07
unify	40.3	0.59	8.7	0.90	35404	1.01	63107	1.28
flatten	28.4	0.76	5.0	1.50	19872	1.21	31142	1.14
crypt	9.3	0.38	2.3	2.05	6504	0.77	113195	1.23
simple_analyzer	64.5	0.37	13.9	1.23	49800	1.46	940006	1.20
reducer	704.5	1.15	6.5	1.04	61768	0.38	1876810	1.15
boyer	91.7	0.67	4.4	1.37	60024	0.99	40668167	1.20
Geometric Mean:		0.67		2.01		1.30		1.34

Table 64: Performance with Domain $T_3 \times A_1 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	8.1	0.81	1.9	4.37	5856	2.31	5744	1.59
nreverse	2.9	1.32	0.7	7.30	1264	1.68	18439	2.63
qsort	9.8	2.09	1.6	3.95	3332	2.23	20990	2.47
serialise	8.2	0.96	1.5	4.14	5248	1.45	25295	1.18
mu	8.8	0.74	1.9	2.33	10464	1.96	48020	1.22
pri2	4.8	1.07	0.9	2.15	2784	2.74	53889	1.83
queens_8	5.6	0.69	0.9	1.70	4956	1.87	133581	2.08
fast_mu	12.5	0.32	2.0	0.83	10864	1.04	45564	0.88
query	5.3	0.61	0.8	1.71	6272	1.86	136787	1.76
press1	97.7	0.98	9.3	1.18	61532	1.53	76336	1.27
tak	2.0	0.56	0.7	4.93	464	0.54	2226334	1.57
sendmore	3.9	0.05	0.9	1.22	2180	0.26	1827545	0.89
poly_10	42.9	0.80	4.2	1.88	25028	4.30	2657075	2.66
zebra	22.4	0.95	2.7	9.89	9032	1.75	2727860	0.66
prover	23.4	0.88	3.5	2.93	10048	1.13	43506	1.16
meta_qsort	20.2	0.87	1.6	1.95	8268	0.89	215622	1.10
nand	212.3	0.59	25.7	0.73	95368	1.70	746445	1.29
chat_parser	257.3	0.57	29.2	0.83	197484	1.31	5420199	1.08
browse	15.6	0.67	2.7	1.36	12208	1.45	38509792	1.07
unify	39.6	0.58	8.3	0.85	35404	1.01	63107	1.28
flatten	27.9	0.74	4.5	1.35	19872	1.21	31142	1.14
crypt	9.4	0.39	2.3	2.04	6504	0.77	113195	1.23
simple_analyzer	61.9	0.35	13.1	1.16	50196	1.47	940006	1.20
reducer	698.7	1.14	5.8	0.94	61768	0.38	1876810	1.15
boyer	93.8	0.68	4.3	1.34	62468	1.03	41371627	1.22
Geometric Mean:		0.66		1.93		1.30		1.34

Table 65: Performance with Domain $T_3 \times A_2 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	8.0	0.80	1.9	4.30	5856	2.31	5744	1.59
nreverse	3.0	1.36	0.8	7.90	1264	1.68	18439	2.63
qsort	9.8	2.09	1.6	4.10	3332	2.23	20990	2.47
serialise	8.1	0.95	1.5	4.14	5248	1.45	25295	1.18
mu	9.3	0.78	2.0	2.36	10464	1.96	48020	1.22
pri2	4.9	1.09	0.9	2.20	2784	2.74	53889	1.83
queens_8	5.7	0.70	0.9	1.76	4956	1.87	133581	2.08
fast_mu	12.6	0.32	2.2	0.89	10864	1.04	45564	0.88
query	5.2	0.60	0.8	1.73	6272	1.86	136787	1.76
press1	99.6	1.00	10.6	1.34	60236	1.50	76206	1.26
tak	2.1	0.58	0.7	4.93	464	0.54	2226334	1.57
sendmore	4.0	0.05	0.9	1.21	2180	0.26	1827545	0.89
poly_10	43.2	0.81	4.5	1.97	25028	4.30	2657075	2.66
zebra	22.3	0.95	2.7	9.89	9032	1.75	2727860	0.66
prover	23.7	0.89	3.8	3.13	10048	1.13	43506	1.16
meta_qsort	20.4	0.88	1.6	1.88	8268	0.89	215622	1.10
nand	213.7	0.60	26.3	0.75	95368	1.70	746445	1.29
chat_parser	265.8	0.59	35.7	1.01	196244	1.30	5410101	1.08
browse	15.6	0.67	2.8	1.39	12208	1.45	38509792	1.07
unify	40.3	0.59	8.7	0.90	35404	1.01	63107	1.28
flatten	28.4	0.76	5.0	1.50	19872	1.21	31142	1.14
crypt	9.3	0.38	2.3	2.05	6504	0.77	113195	1.23
simple_analyzer	64.5	0.37	13.9	1.23	49800	1.46	940006	1.20
reducer	704.5	1.15	6.5	1.04	61768	0.38	1876810	1.15
boyer	91.7	0.67	4.4	1.37	60024	0.99	40668167	1.20
Geometric Mean:		0.67		2.01		1.30		1.34

Table 66: Performance with Domain $T_3 \times A_3 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	8.9	0.89	2.2	5.16	5856	2.31	5744	1.59
nreverse	3.1	1.41	0.9	8.60	1264	1.68	18439	2.63
qsort	12.4	2.64	2.9	7.33	3332	2.23	20990	2.47
serialise	10.9	1.28	2.9	7.70	5248	1.45	25295	1.18
mu	11.3	0.95	3.3	4.00	10464	1.96	48020	1.22
pri2	5.8	1.29	1.6	3.85	2784	2.74	53889	1.83
queens_8	6.6	0.81	1.8	3.28	4956	1.87	133581	2.08
fast_mu	15.2	0.39	3.6	1.49	10864	1.04	45564	0.88
query	5.4	0.62	0.9	1.84	6272	1.86	136787	1.76
press1	119.6	1.20	23.5	2.98	60236	1.50	76206	1.26
tak	2.2	0.61	0.8	5.43	464	0.54	2226334	1.57
sendmore	4.1	0.05	1.4	1.83	2180	0.26	1827545	0.89
poly_10	55.0	1.03	10.7	4.74	25028	4.30	2657075	2.66
zebra	41.1	1.75	17.6	65.15	9032	1.75	2727860	0.66
prover	33.9	1.27	9.0	7.42	10048	1.13	43506	1.16
meta_qsort	22.7	0.97	2.2	2.68	8268	0.89	215622	1.10
nand	331.2	0.93	100.6	2.85	95368	1.70	746445	1.29
chat_parser	445.6	1.00	152.6	4.32	196204	1.30	5406213	1.08
browse	18.9	0.81	5.1	2.56	12208	1.45	38509792	1.07
unify	64.2	0.94	26.9	2.76	35404	1.01	63107	1.28
flatten	41.1	1.10	12.5	3.73	19872	1.21	31142	1.14
crypt	10.3	0.42	3.1	2.77	6504	0.77	113195	1.23
simple_analyzer	104.4	0.59	41.6	3.67	49800	1.46	940006	1.20
reducer	736.4	1.20	20.3	3.28	61768	0.38	1876810	1.15
boyer	101.6	0.74	9.0	2.84	60024	0.99	40668167	1.20
Geometric Mean:		0.84		4.09		1.30		1.34

Table 67: Performance with Domain $T_3 \times A_4 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.2	0.62	1.9	4.51	2668	1.05	4211	1.16
nreverse	2.9	1.32	0.8	8.20	844	1.12	14719	2.10
qsort	4.9	1.04	1.6	4.03	1804	1.21	17195	2.02
serialise	7.7	0.91	2.1	5.70	3868	1.07	23689	1.10
mu	9.5	0.80	2.7	3.20	9120	1.71	47411	1.21
pri2	4.2	0.93	1.0	2.46	1836	1.81	50869	1.73
queens_8	5.3	0.65	1.5	2.83	3856	1.45	129768	2.02
fast_mu	13.3	0.34	3.4	1.40	8360	0.80	42476	0.82
query	4.2	0.48	0.8	1.71	4456	1.32	133383	1.72
press1	78.2	0.79	19.7	2.50	47196	1.17	74326	1.23
tak	2.5	0.69	0.8	5.50	464	0.54	2226334	1.57
sendmore	4.0	0.05	1.4	1.84	2180	0.26	1827545	0.89
poly_10	39.6	0.74	10.1	4.46	17788	3.06	2216994	2.22
zebra	30.6	1.30	9.7	35.96	9032	1.75	2727860	0.66
prover	30.6	1.15	7.1	5.90	8836	1.00	42754	1.14
meta_qsort	22.5	0.97	2.4	2.82	8268	0.89	215622	1.10
nand	227.0	0.64	75.4	2.14	73380	1.31	724777	1.25
chat_parser	303.0	0.68	91.3	2.59	155808	1.04	5210561	1.04
browse	16.8	0.72	4.1	2.05	9848	1.17	38467910	1.07
unify	52.0	0.76	20.2	2.07	33480	0.95	62370	1.27
flatten	26.0	0.69	9.7	2.89	14196	0.87	30571	1.12
crypt	10.0	0.41	2.5	2.24	6224	0.74	107625	1.17
simple_analyzer	87.5	0.50	32.3	2.86	43412	1.27	929914	1.19
reducer	284.2	0.46	15.2	2.46	44332	0.28	1802208	1.11
boyer	95.3	0.70	8.1	2.54	57492	0.95	40668167	1.20
Geometric Mean:		0.65		3.29		1.02		1.27

Table 68: Performance with Domain $T_3 \times A_5 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.3	0.63	2.0	4.70	2668	1.05	4211	1.16
nreverse	2.1	0.95	0.7	7.30	648	0.86	11149	1.59
qsort	3.5	0.74	1.0	2.55	1008	0.67	14195	1.67
serialise	5.0	0.59	1.9	5.14	2560	0.71	13895	0.65
mu	6.3	0.53	1.8	2.16	5020	0.94	38098	0.97
pri2	3.5	0.78	1.4	3.46	1176	1.16	45535	1.55
queens_8	4.0	0.49	1.5	2.85	2116	0.80	97479	1.52
fast_mu	12.1	0.31	2.7	1.12	8152	0.78	41450	0.80
query	5.4	0.62	0.9	1.88	6272	1.86	136787	1.76
press1	59.6	0.60	21.0	2.66	34860	0.87	71817	1.19
tak	2.5	0.69	0.8	5.36	464	0.54	2226334	1.57
sendmore	4.1	0.05	1.4	1.84	2180	0.26	1827545	0.89
poly_10	19.8	0.37	4.0	1.78	7780	1.34	1655193	1.66
zebra	18.6	0.79	7.2	26.78	8052	1.56	1374912	0.33
prover	26.9	1.01	5.5	4.52	8836	1.00	42754	1.14
meta_qsort	20.9	0.90	1.8	2.19	8228	0.89	215469	1.10
nand	206.1	0.58	71.5	2.03	77196	1.37	677249	1.17
chat_parser	265.1	0.59	65.3	1.85	150048	1.00	4968574	0.99
browse	16.6	0.72	4.2	2.12	9764	1.16	38467305	1.07
unify	49.8	0.73	17.9	1.84	31948	0.91	61664	1.25
flatten	22.5	0.60	7.8	2.32	13856	0.85	30261	1.11
crypt	9.6	0.40	2.4	2.17	6240	0.74	107572	1.17
simple_analyzer	74.5	0.42	23.4	2.07	40656	1.19	927443	1.19
reducer	284.1	0.46	11.5	1.86	50020	0.31	1830708	1.13
boyer	91.7	0.67	6.8	2.15	56748	0.94	40664513	1.20
Geometric Mean:		0.55		2.80		0.88		1.13

Table 69: Performance with Domain $T_3 \times A_6 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.3	0.63	2.0	4.70	2668	1.05	4211	1.16
nreverse	2.2	1.00	0.8	7.50	648	0.86	11149	1.59
qsort	3.5	0.74	1.0	2.55	1008	0.67	14195	1.67
serialise	4.8	0.56	1.8	4.95	2560	0.71	13895	0.65
mu	6.3	0.53	1.8	2.17	5020	0.94	38098	0.97
pri2	3.5	0.78	1.0	2.41	1176	1.16	45535	1.55
queens_8	4.0	0.49	1.6	2.91	2116	0.80	97479	1.52
fast_mu	11.7	0.30	2.6	1.07	8152	0.78	41450	0.80
query	4.2	0.48	0.9	1.80	4456	1.32	133383	1.72
press1	56.7	0.57	18.6	2.35	34860	0.87	71817	1.19
tak	2.5	0.69	0.8	5.57	464	0.54	2226334	1.57
sendmore	4.1	0.05	1.4	1.88	2180	0.26	1827545	0.89
poly_10	19.9	0.37	4.1	1.81	7780	1.34	1655193	1.66
zebra	14.3	0.61	4.0	14.93	8052	1.56	1374912	0.33
prover	25.0	0.94	4.7	3.84	8836	1.00	42754	1.14
meta_qsort	21.1	0.91	2.0	2.40	8228	0.89	215469	1.10
nand	188.3	0.53	67.0	1.90	68932	1.23	677249	1.17
chat_parser	242.4	0.54	51.3	1.45	148620	0.99	4968574	0.99
browse	15.8	0.68	4.0	2.01	9764	1.16	38467305	1.07
unify	47.2	0.69	15.8	1.62	31948	0.91	61664	1.25
flatten	21.1	0.56	7.0	2.07	13856	0.85	30261	1.11
crypt	9.5	0.39	2.4	2.17	6240	0.74	107572	1.17
simple_analyzer	71.4	0.40	21.9	1.93	40656	1.19	927443	1.19
reducer	280.0	0.46	9.9	1.60	50020	0.31	1830708	1.13
boyer	90.4	0.66	6.2	1.96	56748	0.94	40664513	1.20
Geometric Mean:		0.53		2.58		0.86		1.13

Table 70: Performance with Domain $T_3 \times A_7 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	8.1	0.81	1.9	4.42	5856	2.31	5744	1.59
nreverse	2.2	1.00	0.7	7.40	660	0.88	12454	1.77
qsort	3.6	0.77	1.5	3.67	1416	0.95	16370	1.92
serialise	8.4	0.99	1.6	4.46	5248	1.45	25295	1.18
mu	9.3	0.78	2.0	2.43	10464	1.96	48020	1.22
pri2	3.4	0.76	0.9	2.15	1228	1.21	49714	1.69
queens_8	5.8	0.72	1.4	2.57	4956	1.87	133581	2.08
fast_mu	12.9	0.33	2.3	0.95	10864	1.04	45564	0.88
query	5.3	0.61	0.8	1.67	6272	1.86	136787	1.76
press1	87.4	0.88	11.7	1.48	57388	1.43	76098	1.26
tak	2.1	0.58	0.7	5.07	464	0.54	2226334	1.57
sendmore	4.0	0.05	1.0	1.25	2180	0.26	1827545	0.89
poly_10	43.6	0.81	4.6	2.04	25028	4.30	2657075	2.66
zebra	22.3	0.95	2.7	10.04	9032	1.75	2727860	0.66
prover	24.1	0.91	4.0	3.29	10048	1.13	43506	1.16
meta_qsort	20.4	0.88	1.6	1.94	8268	0.89	215622	1.10
nand	217.5	0.61	28.3	0.80	95368	1.70	746445	1.29
chat_parser	283.5	0.63	43.2	1.22	196244	1.30	5410101	1.08
browse	15.8	0.68	2.9	1.46	12208	1.45	38509792	1.07
unify	41.3	0.60	9.5	0.97	35404	1.01	63107	1.28
flatten	28.8	0.77	5.3	1.57	19872	1.21	31142	1.14
crypt	9.6	0.40	2.4	2.15	6504	0.77	113195	1.23
simple_analyzer	64.7	0.37	15.3	1.35	49800	1.46	940006	1.20
reducer	702.0	1.14	6.8	1.09	61768	0.38	1876810	1.15
boyer	92.3	0.67	4.6	1.46	60024	0.99	40668167	1.20
Geometric Mean:		0.63		2.12		1.18		1.30

Table 71: Performance with Domain $T_3 \times A_8 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	8.4	0.84	2.1	4.88	5856	2.31	5744	1.59
nreverse	2.2	1.00	0.8	7.80	660	0.88	12454	1.77
qsort	3.8	0.81	1.5	3.85	1416	0.95	16370	1.92
serialise	13.3	1.56	4.5	12.03	5248	1.45	25295	1.18
mu	8.5	0.71	2.4	2.88	6688	1.25	40393	1.03
pri2	3.5	0.78	1.4	3.46	1228	1.21	49714	1.69
queens_8	5.2	0.64	1.6	2.94	3852	1.45	125225	1.95
fast_mu	17.2	0.44	4.4	1.80	10524	1.01	44714	0.86
query	5.6	0.64	1.5	3.00	6272	1.86	136787	1.76
press1	105.9	1.06	27.5	3.48	54352	1.35	75904	1.26
tak	2.5	0.69	0.8	5.50	464	0.54	2226334	1.57
sendmore	4.2	0.05	1.5	1.97	2180	0.26	1827545	0.89
poly_10	27.5	0.51	4.0	1.77	17376	2.99	2022883	2.02
zebra	39.3	1.67	14.1	52.37	9032	1.75	2727860	0.66
prover	22.8	0.86	4.1	3.37	9096	1.02	43455	1.15
meta_qsort	22.0	0.94	2.4	2.80	8228	0.89	215469	1.10
nand	282.0	0.79	107.3	3.04	79168	1.41	745917	1.29
chat_parser	452.0	1.01	185.3	5.25	186940	1.24	5380171	1.08
browse	18.9	0.81	4.9	2.45	12092	1.44	38509292	1.07
unify	74.5	1.09	36.7	3.77	35480	1.01	62875	1.28
flatten	32.7	0.87	11.4	3.40	16916	1.03	30897	1.13
crypt	10.0	0.41	2.7	2.39	6352	0.75	113110	1.23
simple_analyzer	97.5	0.55	41.8	3.69	45768	1.34	937324	1.20
reducer	740.3	1.20	28.6	4.62	59928	0.37	1863722	1.15
boyer	101.4	0.74	10.0	3.14	59472	0.99	40668167	1.20
Geometric Mean:		0.73		3.90		1.10		1.27

Table 72: Performance with Domain $T_3 \times A_9 \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.5	0.65	2.1	4.88	2668	1.05	4211	1.16
nreverse	2.3	1.05	0.9	9.00	660	0.88	12454	1.77
qsort	4.1	0.87	1.8	4.45	1416	0.95	16370	1.92
serialise	8.4	0.99	2.4	6.46	3868	1.07	23689	1.10
mu	7.8	0.66	2.5	2.99	5200	0.97	39847	1.01
pri2	3.7	0.82	1.5	3.73	1228	1.21	49714	1.69
queens_8	5.1	0.63	1.6	2.96	3324	1.25	125079	1.95
fast_mu	14.9	0.38	4.6	1.90	8152	0.78	41450	0.80
query	4.4	0.51	0.9	1.88	4456	1.32	133383	1.72
press1	117.7	1.18	32.5	4.11	48060	1.20	75661	1.25
tak	2.6	0.72	0.8	5.93	464	0.54	2226334	1.57
sendmore	4.4	0.05	1.6	2.09	2180	0.26	1827545	0.89
poly_10	20.4	0.38	4.3	1.92	7780	1.34	1655193	1.66
zebra	38.8	1.65	13.9	51.52	9032	1.75	2727860	0.66
prover	23.3	0.88	4.6	3.82	7884	0.89	42703	1.13
meta_qsort	24.9	1.07	3.0	3.51	8268	0.89	215622	1.10
nand	175.3	0.49	46.8	1.33	69604	1.24	719368	1.24
chat_parser	363.1	0.81	113.4	3.21	175016	1.16	5209268	1.04
browse	19.5	0.84	5.5	2.77	10976	1.30	38469535	1.07
unify	75.4	1.10	38.6	3.96	35480	1.01	62875	1.28
flatten	40.5	1.08	15.5	4.62	16396	1.00	30974	1.14
crypt	10.4	0.43	2.9	2.62	6072	0.72	107540	1.17
simple_analyzer	100.8	0.57	45.0	3.98	46412	1.36	935364	1.20
reducer	314.9	0.51	37.3	6.03	43860	0.27	1801556	1.11
boyer	105.0	0.77	13.2	4.15	57492	0.95	40668167	1.20
Geometric Mean:		0.67		3.89		0.94		1.23

Table 73: Performance with Domain $T_3 \times A_{10} \times AC_1 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.9	0.69	2.4	5.53	2668	1.05	4211	1.16
nreverse	2.3	1.05	0.8	8.10	648	0.86	11149	1.59
qsort	3.7	0.79	1.7	4.22	1008	0.67	14195	1.67
serialise	5.4	0.64	2.2	5.92	2560	0.71	13895	0.65
mu	6.8	0.57	2.2	2.64	5020	0.94	38098	0.97
pri2	3.8	0.84	1.6	3.95	1176	1.16	45535	1.55
queens_8	4.4	0.54	1.8	3.30	2116	0.80	97479	1.52
fast_mu	14.1	0.36	4.2	1.72	8152	0.78	41450	0.80
query	4.4	0.51	0.9	1.92	4456	1.32	133383	1.72
press1	63.8	0.64	23.8	3.02	34860	0.87	71817	1.19
tak	2.9	0.81	1.0	7.36	464	0.54	2226334	1.57
sendmore	4.4	0.05	1.6	2.14	2180	0.26	1827545	0.89
poly_10	23.5	0.44	6.5	2.88	7780	1.34	1655193	1.66
zebra	21.5	0.91	9.6	35.70	8052	1.56	1374912	0.33
prover	27.2	1.02	6.4	5.28	8836	1.00	42754	1.14
meta_qsort	21.9	0.94	2.3	2.76	8228	0.89	215469	1.10
nand	160.2	0.45	75.9	2.15	56528	1.01	612174	1.06
chat_parser	287.4	0.64	80.1	2.27	148620	0.99	4968574	0.99
browse	17.6	0.76	5.2	2.59	9764	1.16	38467305	1.07
unify	54.5	0.80	22.4	2.30	32360	0.92	61644	1.25
flatten	23.4	0.62	8.5	2.52	13856	0.85	30261	1.11
crypt	10.9	0.45	3.0	2.71	6204	0.74	107548	1.17
simple_analyzer	75.3	0.43	25.6	2.26	40636	1.19	927443	1.19
reducer	287.6	0.47	13.1	2.12	49740	0.31	1830708	1.13
boyer	94.3	0.69	7.3	2.29	56748	0.94	40664513	1.20
Geometric Mean:		0.58		3.40		0.86		1.13

Table 74: Performance with Domain $T_3 \times A_6 \times AC_2 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.6	0.66	2.1	4.86	2668	1.05	4211	1.16
nreverse	2.2	1.00	0.8	7.70	648	0.86	11149	1.59
qsort	3.6	0.77	1.5	3.73	1008	0.67	14195	1.67
serialise	5.3	0.62	1.3	3.59	2560	0.71	13895	0.65
mu	6.5	0.55	1.9	2.24	5020	0.94	38098	0.97
pri2	3.7	0.82	1.0	2.51	1176	1.16	45535	1.55
queens_8	4.2	0.52	1.6	2.93	2116	0.80	97479	1.52
fast_mu	12.3	0.32	2.7	1.10	8152	0.78	41450	0.80
query	4.3	0.49	0.9	1.76	4456	1.32	133383	1.72
press1	60.1	0.60	18.8	2.38	34860	0.87	71817	1.19
tak	2.6	0.72	0.8	5.57	464	0.54	2226334	1.57
sendmore	4.9	0.06	1.5	1.96	2180	0.26	1827545	0.89
poly_10	20.6	0.38	4.1	1.81	7780	1.34	1655193	1.66
zebra	14.8	0.63	4.1	15.15	8052	1.56	1374912	0.33
prover	25.8	0.97	5.3	4.36	8488	0.96	42703	1.13
meta_qsort	21.6	0.93	2.0	2.43	8228	0.89	215469	1.10
nand	195.0	0.55	68.5	1.94	68932	1.23	677249	1.17
chat_parser	257.1	0.57	54.0	1.53	148620	0.99	4968574	0.99
browse	17.0	0.73	4.1	2.04	9764	1.16	38467305	1.07
unify	49.6	0.73	15.9	1.63	31948	0.91	61664	1.25
flatten	22.9	0.61	7.2	2.15	13856	0.85	30261	1.11
crypt	10.6	0.44	2.4	2.17	6240	0.74	107572	1.17
simple_analyzer	77.2	0.44	22.2	1.96	40656	1.19	927443	1.19
reducer	284.4	0.46	9.8	1.58	50020	0.31	1830708	1.13
boyer	93.7	0.68	6.3	1.99	56748	0.94	40664513	1.20
Geometric Mean:		0.55		2.64		0.86		1.13

Table 75: Performance with Domain $T_3 \times A_6 \times AC_3 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.9	0.69	2.2	5.21	2004	0.79	3083	0.85
nreverse	2.3	1.05	0.8	7.90	536	0.71	6991	1.00
qsort	3.7	0.79	1.6	3.90	840	0.56	10705	1.26
serialise	5.4	0.64	1.9	5.24	2160	0.60	11978	0.56
mu	6.8	0.57	2.0	2.36	4580	0.86	35817	0.91
pri2	3.8	0.84	1.6	3.78	932	0.92	38363	1.30
queens_8	4.4	0.54	1.7	3.09	1720	0.65	84395	1.32
fast_mu	13.7	0.35	3.6	1.48	6664	0.64	36706	0.71
query	4.5	0.52	0.9	1.84	3076	0.91	97237	1.25
press1	64.0	0.64	21.2	2.68	28852	0.72	59390	0.98
tak	2.8	0.78	0.9	6.14	420	0.49	1987801	1.40
sendmore	5.4	0.07	1.6	2.12	1832	0.22	1568389	0.76
poly_10	22.1	0.41	4.4	1.95	6340	1.09	1384663	1.38
zebra	17.9	0.76	5.9	21.81	7872	1.53	1374873	0.33
prover	28.6	1.08	5.8	4.83	8196	0.92	39389	1.05
meta_qsort	22.4	0.96	2.2	2.67	7284	0.78	207210	1.05
nand	213.4	0.60	78.4	2.22	57304	1.02	587126	1.01
chat_parser	283.8	0.63	63.1	1.79	131420	0.87	4582339	0.92
browse	18.5	0.80	4.7	2.33	8292	0.99	36103215	1.00
unify	59.6	0.87	22.1	2.27	30604	0.87	55438	1.13
flatten	26.9	0.72	9.7	2.89	12308	0.75	28290	1.04
crypt	11.6	0.48	2.7	2.40	6148	0.73	104548	1.14
simple_analyzer	89.1	0.51	27.1	2.39	35484	1.04	801169	1.03
reducer	291.7	0.47	11.5	1.85	41276	0.26	1638161	1.01
boyer	88.6	0.65	7.4	2.34	45360	0.75	35130956	1.03
Geometric Mean:		0.60		3.09		0.73		0.98

Table 76: Performance with Domain $T_3 \times A_6 \times AC_4 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.9	0.69	2.2	5.19	2004	0.79	3083	0.85
nreverse	2.6	1.18	0.8	8.00	536	0.71	6991	1.00
qsort	3.7	0.79	1.6	3.90	840	0.56	10705	1.26
serialise	5.4	0.64	1.9	5.27	2160	0.60	11978	0.56
mu	6.8	0.57	1.9	2.35	4580	0.86	35817	0.91
pri2	3.9	0.87	1.6	3.88	932	0.92	38363	1.30
queens_8	4.4	0.54	1.7	3.09	1720	0.65	84395	1.32
fast_mu	13.7	0.35	3.6	1.50	6664	0.64	36706	0.71
query	4.5	0.52	1.4	2.84	3076	0.91	97237	1.25
press1	64.0	0.64	20.6	2.61	28852	0.72	59390	0.98
tak	2.8	0.78	0.8	5.93	420	0.49	1987801	1.40
sendmore	5.5	0.07	1.7	2.16	1832	0.22	1568389	0.76
poly_10	22.0	0.41	4.4	1.96	6340	1.09	1384663	1.38
zebra	17.4	0.74	5.9	21.70	7872	1.53	1374873	0.33
prover	28.1	1.06	5.7	4.74	7864	0.89	39338	1.04
meta_qsort	22.4	0.96	2.2	2.65	7284	0.78	207210	1.05
nand	214.5	0.60	81.5	2.31	57304	1.02	587126	1.01
chat_parser	282.0	0.63	64.7	1.83	131420	0.87	4582339	0.92
browse	18.6	0.80	4.7	2.36	8292	0.99	36103215	1.00
unify	59.1	0.87	22.2	2.28	30604	0.87	55438	1.13
flatten	26.8	0.71	9.7	2.89	12308	0.75	28290	1.04
crypt	11.5	0.47	2.7	2.40	6148	0.73	104548	1.14
simple_analyzer	91.5	0.52	29.3	2.59	35484	1.04	801169	1.03
reducer	290.2	0.47	10.6	1.71	41216	0.26	1636439	1.01
boyer	88.5	0.65	7.5	2.36	45360	0.75	35130956	1.03
Geometric Mean:		0.60		3.16		0.73		0.98

Table 77: Performance with Domain $T_3 \times A_6 \times AC_5 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.8	0.68	2.3	5.44	1944	0.77	2815	0.78
nreverse	2.7	1.23	0.8	8.20	536	0.71	6991	1.00
qsort	3.7	0.79	1.6	3.95	860	0.57	9120	1.07
serialise	5.3	0.62	2.0	5.43	2100	0.58	11593	0.54
mu	6.9	0.58	2.1	2.49	4520	0.85	35768	0.91
pri2	3.8	0.84	1.6	3.85	768	0.76	30133	1.02
queens_8	4.3	0.53	1.9	3.46	1396	0.53	52418	0.82
fast_mu	13.4	0.34	3.7	1.52	5812	0.56	33317	0.64
query	4.2	0.48	1.4	2.80	2356	0.70	70607	0.91
press1	63.7	0.64	21.8	2.76	26920	0.67	55654	0.92
tak	2.7	0.75	0.9	6.36	256	0.30	1304030	0.92
sendmore	5.1	0.06	1.6	2.04	1432	0.17	1188070	0.58
poly_10	21.4	0.40	4.6	2.04	5788	1.00	1219133	1.22
zebra	17.5	0.74	5.9	21.89	7832	1.52	1372964	0.33
prover	27.9	1.05	5.9	4.87	7820	0.88	37011	0.98
meta_qsort	22.0	0.94	2.3	2.73	7036	0.76	194125	0.99
nand	208.8	0.58	82.8	2.35	54040	0.96	553468	0.96
chat_parser	281.8	0.63	66.9	1.89	132624	0.88	4602677	0.92
browse	16.8	0.72	4.9	2.43	7316	0.87	34604951	0.96
unify	57.2	0.84	23.6	2.42	28536	0.81	44677	0.91
flatten	25.7	0.69	9.9	2.94	11652	0.71	25812	0.95
crypt	11.0	0.45	2.7	2.43	5164	0.61	76743	0.83
simple_analyzer	84.7	0.48	27.9	2.46	33440	0.98	767450	0.98
reducer	283.1	0.46	11.8	1.90	41152	0.26	1593120	0.98
boyer	86.2	0.63	7.6	2.38	44956	0.74	32709511	0.96
Geometric Mean:		0.58		3.24		0.67		0.86

Table 78: Performance with Domain $T_3 \times A_6 \times AC_6 \times R_1$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.9	0.69	2.3	5.30	1944	0.77	2815	0.78
nreverse	2.3	1.05	0.8	8.20	536	0.71	6991	1.00
qsort	3.7	0.79	1.6	4.00	860	0.57	9120	1.07
serialise	5.4	0.64	2.0	5.46	2100	0.58	11593	0.54
mu	7.0	0.59	2.1	2.52	4520	0.85	35768	0.91
pri2	3.7	0.82	1.6	3.93	768	0.76	30133	1.02
queens_8	4.3	0.53	1.8	3.30	1396	0.53	52418	0.82
fast_mu	13.5	0.35	3.7	1.52	5812	0.56	33317	0.64
query	4.4	0.51	1.4	2.88	2356	0.70	70607	0.91
press1	63.6	0.64	21.9	2.77	26920	0.67	55654	0.92
tak	2.7	0.75	0.9	6.43	256	0.30	1304030	0.92
sendmore	5.2	0.06	1.6	2.12	1432	0.17	1188070	0.58
poly_10	21.5	0.40	4.6	2.04	5788	1.00	1219133	1.22
zebra	17.5	0.74	5.9	21.93	7832	1.52	1372964	0.33
prover	27.6	1.04	5.9	4.86	7488	0.84	36960	0.98
meta_qsort	22.1	0.95	2.3	2.75	7036	0.76	194125	0.99
nand	209.1	0.59	82.3	2.33	54040	0.96	553468	0.96
chat_parser	280.4	0.63	67.3	1.91	132624	0.88	4602677	0.92
browse	16.8	0.72	5.4	2.71	7316	0.87	34604951	0.96
unify	57.6	0.84	23.9	2.45	28536	0.81	44677	0.91
flatten	25.8	0.69	9.9	2.94	11652	0.71	25812	0.95
crypt	11.1	0.46	2.7	2.43	5164	0.61	76743	0.83
simple_analyzer	84.3	0.48	27.5	2.43	33440	0.98	767450	0.98
reducer	282.4	0.46	11.8	1.91	41092	0.26	1591398	0.98
boyer	85.3	0.62	7.7	2.42	44956	0.74	32709511	0.96
Geometric Mean:		0.56		3.27		0.67		0.86

Table 79: Performance with Domain $T_3 \times A_6 \times AC_1 \times R_3$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.6	0.66	2.2	5.16	2620	1.03	3874	1.07
nreverse	2.7	1.23	0.8	8.40	632	0.84	11056	1.57
qsort	3.8	0.81	1.7	4.33	1008	0.67	14195	1.67
serialise	5.0	0.59	1.9	5.24	2528	0.70	13817	0.64
mu	7.1	0.60	2.0	2.46	5020	0.94	38098	0.97
pri2	3.7	0.82	1.5	3.66	1176	1.16	45535	1.55
queens_8	4.2	0.52	1.7	3.11	2116	0.80	97479	1.52
fast_mu	13.0	0.33	3.6	1.48	8152	0.78	41450	0.80
query	4.4	0.51	0.9	1.84	4456	1.32	133383	1.72
press1	59.5	0.60	20.5	2.60	34860	0.87	71817	1.19
tak	2.7	0.75	0.9	6.43	464	0.54	2226334	1.57
sendmore	4.3	0.05	1.6	2.03	2180	0.26	1827545	0.89
poly_10	22.0	0.41	5.8	2.58	7780	1.34	1655193	1.66
zebra	13.0	0.55	4.7	17.26	7420	1.44	1244607	0.30
prover	26.8	1.01	6.0	4.93	8836	1.00	42754	1.14
meta_qsort	21.5	0.92	2.2	2.62	8228	0.89	215469	1.10
nand	201.0	0.56	78.5	2.23	68964	1.23	677414	1.17
chat_parser	264.6	0.59	63.4	1.79	148620	0.99	4968574	0.99
browse	17.0	0.73	4.4	2.21	9764	1.16	38632005	1.07
unify	52.2	0.76	19.7	2.03	31936	0.91	61658	1.25
flatten	22.7	0.61	7.7	2.28	13808	0.84	30233	1.11
crypt	10.2	0.42	2.6	2.34	6240	0.74	107572	1.17
simple_analyzer	76.8	0.44	25.9	2.29	40656	1.19	927443	1.19
reducer	285.2	0.46	10.8	1.75	50436	0.31	1828970	1.12
boyer	92.2	0.67	6.7	2.10	56748	0.94	40664513	1.20
Geometric Mean:		0.56		3.04		0.86		1.12

Table 80: Performance with Domain $T_3 \times A_6 \times AC_2 \times R_3$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.7	0.67	2.3	5.30	2620	1.03	3874	1.07
nreverse	2.2	1.00	0.8	8.10	632	0.84	11056	1.57
qsort	3.8	0.81	1.7	4.33	1008	0.67	14195	1.67
serialise	5.0	0.59	1.9	5.24	2528	0.70	13817	0.64
mu	7.0	0.59	2.0	2.47	5020	0.94	38098	0.97
pri2	3.7	0.82	1.6	3.85	1176	1.16	45535	1.55
queens_8	4.2	0.52	1.7	3.13	2116	0.80	97479	1.52
fast_mu	12.9	0.33	3.6	1.49	8152	0.78	41450	0.80
query	4.3	0.49	1.4	2.84	4456	1.32	133383	1.72
press1	60.8	0.61	22.1	2.80	34860	0.87	71817	1.19
tak	2.7	0.75	1.3	9.36	464	0.54	2226334	1.57
sendmore	4.3	0.05	1.5	1.99	2180	0.26	1827545	0.89
poly_10	22.0	0.41	5.9	2.60	7780	1.34	1655193	1.66
zebra	13.1	0.56	4.7	17.33	7420	1.44	1244607	0.30
prover	26.3	0.99	5.9	4.87	8488	0.96	42703	1.13
meta_qsort	21.5	0.92	2.2	2.57	8228	0.89	215469	1.10
nand	203.4	0.57	80.5	2.28	68964	1.23	677414	1.17
chat_parser	268.2	0.60	63.8	1.81	148620	0.99	4968574	0.99
browse	17.1	0.74	4.5	2.25	9764	1.16	38632005	1.07
unify	52.2	0.76	19.6	2.01	31936	0.91	61658	1.25
flatten	22.7	0.61	7.7	2.29	13808	0.84	30233	1.11
crypt	10.2	0.42	2.6	2.32	6240	0.74	107572	1.17
simple_analyzer	77.0	0.44	26.0	2.30	40656	1.19	927443	1.19
reducer	284.8	0.46	11.5	1.86	50436	0.31	1828970	1.12
boyer	91.9	0.67	6.7	2.12	56748	0.94	40664513	1.20
Geometric Mean:		0.56		3.17		0.86		1.12

Table 81: Performance with Domain $T_3 \times A_6 \times AC_3 \times R_3$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.9	0.69	2.4	5.53	1956	0.77	2746	0.76
nreverse	2.2	1.00	0.8	8.40	488	0.65	5503	0.78
qsort	3.7	0.79	1.7	4.15	800	0.53	7627	0.90
serialise	5.3	0.62	2.1	5.70	2128	0.59	11900	0.55
mu	7.6	0.64	2.3	2.78	4560	0.85	35737	0.91
pri2	3.8	0.84	1.7	4.22	820	0.81	33193	1.13
queens_8	4.2	0.52	1.7	3.22	1512	0.57	69229	1.08
fast_mu	13.7	0.35	3.9	1.61	6376	0.61	35282	0.68
query	4.3	0.49	1.4	2.96	3032	0.90	85662	1.10
press1	65.0	0.65	24.2	3.06	28740	0.71	58811	0.98
tak	2.7	0.75	0.8	5.86	396	0.46	1510735	1.07
sendmore	4.5	0.05	1.9	2.44	1920	0.23	1711023	0.83
poly_10	21.6	0.40	4.9	2.15	5640	0.97	1129014	1.13
zebra	15.6	0.66	6.9	25.41	7240	1.41	1244568	0.30
prover	29.4	1.11	7.5	6.16	8196	0.92	39389	1.05
meta_qsort	22.3	0.96	2.5	3.01	7284	0.78	207210	1.05
nand	217.3	0.61	94.0	2.66	55452	0.99	582154	1.01
chat_parser	300.0	0.67	91.8	2.60	131096	0.87	4573113	0.91
browse	17.3	0.75	4.9	2.44	8252	0.98	36262515	1.01
unify	61.6	0.90	26.1	2.68	30604	0.87	55438	1.13
flatten	27.0	0.72	11.5	3.41	12260	0.75	28262	1.04
crypt	11.9	0.49	4.4	3.90	6148	0.73	104548	1.14
simple_analyzer	69.9	0.40	15.6	1.38	49800	1.46	940006	1.20
reducer	288.4	0.47	12.4	2.00	40968	0.25	1629094	1.00
boyer	88.6	0.65	8.4	2.64	45360	0.75	35130956	1.03
Geometric Mean:		0.58		3.50		0.72		0.92

Table 82: Performance with Domain $T_3 \times A_6 \times AC_4 \times R_3$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.9	0.69	2.5	5.72	1956	0.77	2746	0.76
nreverse	2.7	1.23	0.8	8.50	488	0.65	5503	0.78
qsort	3.8	0.81	1.7	4.17	800	0.53	7627	0.90
serialise	5.3	0.62	2.1	5.73	2128	0.59	11900	0.55
mu	7.6	0.64	2.3	2.82	4560	0.85	35737	0.91
pri2	3.8	0.84	1.7	4.05	820	0.81	33193	1.13
queens_8	4.2	0.52	1.7	3.22	1512	0.57	69229	1.08
fast_mu	13.8	0.35	4.0	1.63	6376	0.61	35282	0.68
query	4.4	0.51	1.4	2.94	3032	0.90	85662	1.10
press1	64.9	0.65	24.5	3.10	28740	0.71	58811	0.98
tak	2.3	0.64	0.8	5.93	396	0.46	1510735	1.07
sendmore	4.5	0.05	1.8	2.39	1920	0.23	1711023	0.83
poly_10	21.6	0.40	4.8	2.14	5640	0.97	1129014	1.13
zebra	15.6	0.66	6.9	25.52	7240	1.41	1244568	0.30
prover	28.9	1.09	7.5	6.17	7864	0.89	39338	1.04
meta_qsort	22.6	0.97	2.4	2.82	7284	0.78	207210	1.05
nand	224.1	0.63	95.0	2.69	55452	0.99	582154	1.01
chat_parser	304.0	0.68	93.9	2.66	131096	0.87	4573113	0.91
browse	17.7	0.76	5.5	2.73	8252	0.98	36262515	1.01
unify	62.1	0.91	26.6	2.73	30604	0.87	55438	1.13
flatten	27.0	0.72	11.6	3.44	12260	0.75	28262	1.04
crypt	11.7	0.48	4.2	3.73	6148	0.73	104548	1.14
simple_analyzer	88.5	0.50	31.0	2.74	35292	1.03	801169	1.03
reducer	285.7	0.46	12.4	2.00	40908	0.25	1627372	1.00
boyer	87.2	0.64	8.5	2.67	45360	0.75	35130956	1.03
Geometric Mean:		0.59		3.61		0.71		0.91

Table 83: Performance with Domain $T_3 \times A_6 \times AC_5 \times R_3$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.9	0.69	2.5	5.74	1896	0.75	2478	0.68
nreverse	2.2	1.00	0.9	8.70	488	0.65	5503	0.78
qsort	3.6	0.77	1.6	4.10	780	0.52	6142	0.72
serialise	5.2	0.61	2.2	6.03	2068	0.57	11515	0.54
mu	7.6	0.64	2.3	2.82	4500	0.84	35688	0.91
pri2	3.6	0.80	1.7	4.07	680	0.67	24147	0.82
queens_8	4.1	0.51	1.8	3.33	1188	0.45	37252	0.58
fast_mu	13.4	0.34	4.1	1.67	5412	0.52	31946	0.62
query	4.1	0.47	1.5	2.98	2276	0.68	58757	0.76
press1	63.4	0.64	24.8	3.13	26808	0.67	55067	0.91
tak	2.5	0.69	0.8	5.79	228	0.26	842866	0.60
sendmore	4.3	0.05	1.9	2.42	1228	0.15	1052504	0.51
poly_10	20.9	0.39	5.6	2.48	5124	0.88	954097	0.95
zebra	15.6	0.66	6.9	25.63	7200	1.40	1242659	0.30
prover	28.5	1.07	7.6	6.26	7820	0.88	37011	0.98
meta_qsort	22.0	0.94	2.5	2.92	7036	0.76	194125	0.99
nand	215.1	0.60	96.9	2.75	52196	0.93	548516	0.95
chat_parser	306.0	0.68	94.8	2.69	132300	0.88	4592467	0.92
browse	16.1	0.69	5.5	2.77	7300	0.87	34762251	0.97
unify	58.8	0.86	27.3	2.80	28536	0.81	44677	0.91
flatten	26.1	0.70	11.7	3.49	11604	0.71	25777	0.95
crypt	11.1	0.46	4.2	3.73	5164	0.61	76743	0.83
simple_analyzer	86.4	0.49	31.9	2.82	33248	0.97	767334	0.98
reducer	279.7	0.46	12.5	2.02	40584	0.25	1584685	0.97
boyer	84.6	0.62	8.6	2.71	44956	0.74	32709511	0.96
Geometric Mean:		0.57		3.68		0.64		0.78

Table 84: Performance with Domain $T_3 \times A_6 \times AC_6 \times R_3$

Benchmark	Compile		Analyze		Static Size		Exec. Time	
	Sec	Rel	Sec	Rel	Bytes	Rel	Insts	Rel
deriv	6.9	0.69	2.5	5.77	1896	0.75	2478	0.68
nreverse	2.2	1.00	0.8	8.40	488	0.65	5503	0.78
qsort	3.7	0.79	1.7	4.22	780	0.52	6142	0.72
serialise	5.3	0.62	2.2	5.86	2068	0.57	11515	0.54
mu	7.5	0.63	2.3	2.82	4500	0.84	35688	0.91
pri2	3.7	0.82	1.7	4.12	680	0.67	24147	0.82
queens_8	4.0	0.49	1.8	3.35	1188	0.45	37252	0.58
fast_mu	13.5	0.35	4.0	1.66	5412	0.52	31946	0.62
query	4.2	0.48	1.5	2.98	2276	0.68	58757	0.76
press1	65.3	0.66	24.8	3.14	26808	0.67	55067	0.91
tak	2.6	0.72	0.9	6.14	228	0.26	842866	0.60
sendmore	4.3	0.05	1.9	2.47	1228	0.15	1052504	0.51
poly_10	21.4	0.40	5.7	2.50	5124	0.88	954097	0.95
zebra	15.6	0.66	6.9	25.56	7200	1.40	1242659	0.30
prover	28.1	1.06	7.6	6.29	7488	0.84	36960	0.98
meta_qsort	22.1	0.95	2.5	3.00	7036	0.76	194125	0.99
nand	217.3	0.61	98.5	2.79	52196	0.93	548516	0.95
chat_parser	313.1	0.70	95.5	2.70	132300	0.88	4592467	0.92
browse	16.0	0.69	5.5	2.77	7300	0.87	34762251	0.97
unify	58.9	0.86	27.4	2.81	28536	0.81	44677	0.91
flatten	26.1	0.70	11.8	3.52	11604	0.71	25777	0.95
crypt	11.0	0.45	4.2	3.72	5164	0.61	76743	0.83
simple_analyzer	86.0	0.49	32.1	2.84	33248	0.97	767334	0.98
reducer	278.9	0.45	12.6	2.03	40524	0.25	1582963	0.97
boyer	84.8	0.62	8.7	2.72	44956	0.74	32709511	0.96
Geometric Mean:		0.58		3.70		0.64		0.78