The Costs and Benefits of Abstract Interpretation-driven Prolog Optimization

Thomas W. Getzinger
Hughes Aircraft Company
Fullerton, CA 92634 USA
tgetzinger@msmail2.hac.com

1 Introduction

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. Using this taxonomy and an analysis 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%.

2 Prolog Analysis

Numerous abstract domains have been proposed to capture various run-time properties [2, 10, 15, 17, 22, 29, 32, 39, 42]. We have constructed a taxonomy, shown in Figure 1, 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.

The following sections describe increasingly precise domains for capturing these properties. Details are provided for those domains we implemented and evaluated. Other types of analysis are touched on briefly for completeness. More details can be found in [8, 13].

2.1 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 [19, 26].

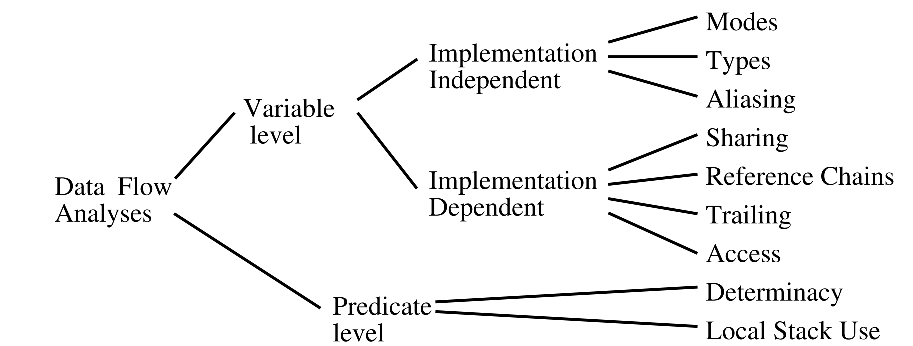

Figure 1. Taxonomy of Data Flow Analyses

Mode Analysis

Mode analysis attempts to determine the degree of instantiation of each variable. This is the most often described form of analysis [1, 3, 10, 23, 28, 30, 32, 36, 37, 39, 42], probably because it is one of the easiest to understand and provides very useful information. 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. Figure 2 presents the abstract domains for mode analysis by showing the lattice in each of the domains for one variable.

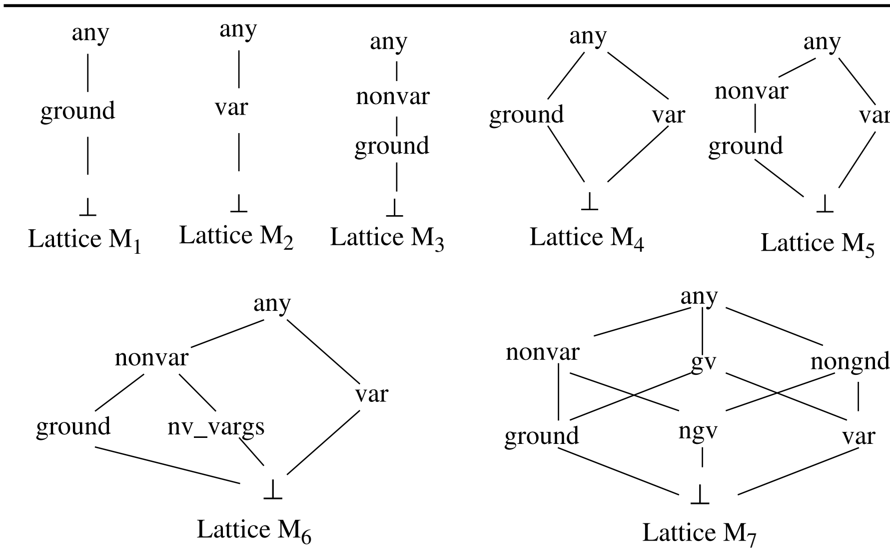

Figure 2. Lattices for Mode Abstract Domains

Figure 3 places the domains, themselves, in a lattice, showing how one domain approximates another [6, 7]; the higher entries on this lattice provide more detailed information. The bottom element is the domain of no information (in essence, the result of performing no analysis).

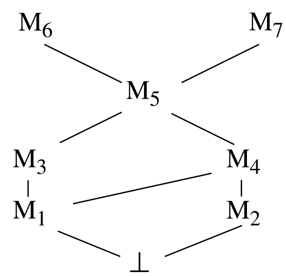

Figure 3. Lattice of Mode Domains

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 [30, 36, 37]. M_1 captures definitely ground variables. M_2 captures definitely unbound (var) variables.

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 [1, 3, 28, 10]. Domain M_5 adds all of these states [42]. 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 (nv_vargs) [32].

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 [23]. 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.

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). In addition, type domains can capture a single type for each variable or a set of possible types (*"or"* types).

The following paragraphs 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 4 shows how these type domains approximate one another.

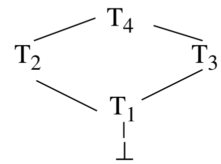

Figure 4. Lattice of Type Domains

Domain T_1 is a flat type domain with no “or” types. This is similar to the mode domains, in that there is a collection of distinct types, ordered in a lattice. Taylor described such a domain [40], shown in Figure 5. 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’.

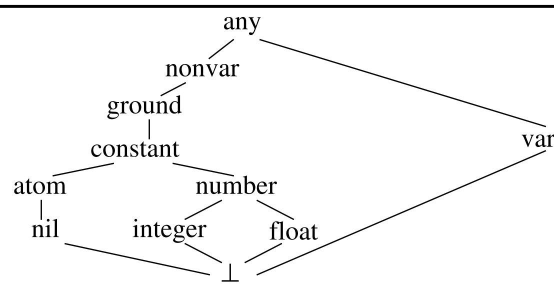

Figure 5. Lattice of Type Descriptions for Domain T_1

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 1. 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 [12].

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 [39, 40]. Tan and Lin later proposed a simpler version which had fewer atomic types and limited recursive lists to nil-terminated lists [38]. We proposed a third version of this domain, which goes the other direction, adding types which improve the detection of the data tags for variables [13]. In order to keep these domains finite, nested type specifications are depth limited. Both Taylor [40] and Tan and Lin [38] limit nested types to a depth of four.

Table 1. 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_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 [2, 17, 18].

Aliasing Analysis

Early mode analysis was found to be unsound since it didn't consider the effects of aliasing [10]. 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. Many aliasing domains have been proposed [3, 4, 5, 17, 19, 22]. 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. When examining these domains, the following substitution will be used as an example:

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

Weak Coupling

It is important to know when two variables are definitely independent, in order to restrict the effects of binding a variable. This is captured by determining which variables are possibly aliased (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.

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. The example substitution is represented in this domain as {U,V,W,X,Y,Z}. Even though X and Y are independent, this information is lost. In fact, under this domain, we must assume that all variables are dependent.

Domain WC_2 was proposed by Chang to detect data dependencies in order to implement semi-intelligent backtracking [4]. 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. The example substitution is represented in this domain as { {T}, {U,V}, {W,X,Y,Z} }. 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.

The most expressive domain for weak coupling, WC_3, was proposed by Jones and Sondergaard [19]. 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. One common representation is a set of pairs of variables (representing the arcs). The example substitution is represented in this domain as { (U,V), (V,U), (W,X), (W,Z), (X,W), (X,Z), (Y,Z) }. It is now possible to detect that U and X are independent, and that X and Y are independent.

Strong Coupling

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*.

Domain SC_1, the simplest strong coupling domain, captures sets of equivalent variables. This was suggested by Bruynooghe et al. in [1]. An element of this domain 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 is 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.

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) [5]. This was proposed as an improvement to Chang's SDDA [4]. 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).

Coupling Domain CC

Langen proposed a domain that included both strong and weak coupling, and can address the weakness of SC_2 [22]. 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, 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. 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).

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.

The first equivalence domain partitions the set of variables into sets of equivalent variables [1]. This is identical to 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.

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 [40]. Janssens provided the equivalence information separately from the types by specifying sets of selectors for equivalent values [17]. 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 [17]. 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.

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* [22]. Jones and Sondergaard called them *reocc* (repeated occurrence) variables [19]. Janssens called them *NUNI* (for Not-UNIque) variables [17].

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], we would have to assume that the variables H and T are coupled, unless it is known that X is linear.

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).

Proposed Aliasing Domains

A number of aliasing domains have been proposed by researchers. We have decomposed these domains into the sub-domains just described. Table 2 describes the domains and gives the decomposition. Figure 6 shows how these domains approximate one another.

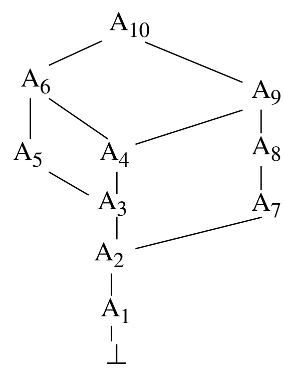

Figure 6. Lattice of Aliasing Domains

Table 2. Proposed Aliasing Domains

#	Definition	Comments
A_1	WC_1	This is a simple domain, provided for completeness.
A_2	WC_2	When combined with M_1, this is SDDA [4].
A_3	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	$WC_3 \times L$	This adds linearity. This is important for predicate entry and exit [19].
A_5	$WC_3 \times E_2$	This drops linearity, but adds equivalence [1].
A_6	$WC_3 \times E_2 \times L$	This adds both. It was used along with T_4 [17, 18].
A_7	$WC_2 \times SC_2$	This improves on SDDA by adding strong coupling classes [5].
A_8	CC	This captures both weak and strong coupling in a single, combined representation which is more expressive than both combined [15].
A_9	$CC \times L$	This adds linearity in order to keep the information more precise (and computationally feasible). It also includes M_2 [22].
A_{10}	$CC \times L \times E_2$	This adds equivalence. It is more precise than all other abstract domains.

2.2 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.

Sharing Analysis

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 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 [1, 14, 21], 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.

Reference Chain Analysis

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 [13]. This is consistent with Touati's findings on the WAM [41]. 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 7 shows how these domains approximate one another.

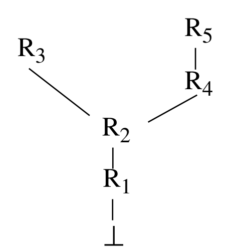

Figure 7. Lattice of Reference Chain Domains

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

Domain R_2 was described by Van Roy [42]. 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 reference 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).

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. The simplest solution was to add a new description, locally_deref, which indicates a predicate argument that has been dereferenced within the predicate.

Domain R_4, proposed by Taylor, captures more categories of chains and allows different descriptions for different arguments of compound values [39]. To support this, it needs to be combined with a type domain which allows non-flat types to be captured, for example T_3. For a small set of benchmarks, Taylor found that this would remove 70-90% of the general dereference loops. The descriptions are:

- 0: The reference chain is of length 0 (i.e., no reference chain).
- 1: The reference chain is of length 1.
- 0-1: The reference chain is of length 0 or 1.
- ?: The reference chain is of an unknown length.

Domain R_5, proposed by Mariën et al., provides even more levels by describing a reference chain as having a minimum and maximum length [28]. 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. It is unlikely, however, that this adds much performance over R_4, since only a fraction of a percent of reference chains ever exceed a length of one [41].

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 [39]. 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).

Access Analyses

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

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 [1]. This is done by determining the last access to a variable or part of its value.

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 [40]. Van Roy took a more aggressive approach, relaxing the aliasing restriction [42]. 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* [42]. 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).

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. 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.

2.3 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. Although abstract interpretation can be used to capture predicate-level information [33], 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. However, abstract interpretation is still a good technique to use for proving the correctness of predicate-level analyses.

Determinacy Analysis

Mellish defines a predicate to be *determinate* if it will never return more than one solution [34]. 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. Taylor used determinacy information to reduce variable trailing [39]. 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 [9]. 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.

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.

Local Stack Analysis

Many implementations of the WAM and the BAM place both environments and choicepoints onto a single stack, called the local stack [27]. If it is known what objects are added to the stack, there are a number of optimizations that can be applied. For example, environments and choicepoints can be reused [31]. 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.

In order to perform precise trailing analysis, it is necessary to know when predicates can generate choicepoints [39]. 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.

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.

These predicate-level analyses 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.

2.4 Combining Analyses

In the previous descriptions, we have (as much as possible) considered each analysis in isolation. Figure 8 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.

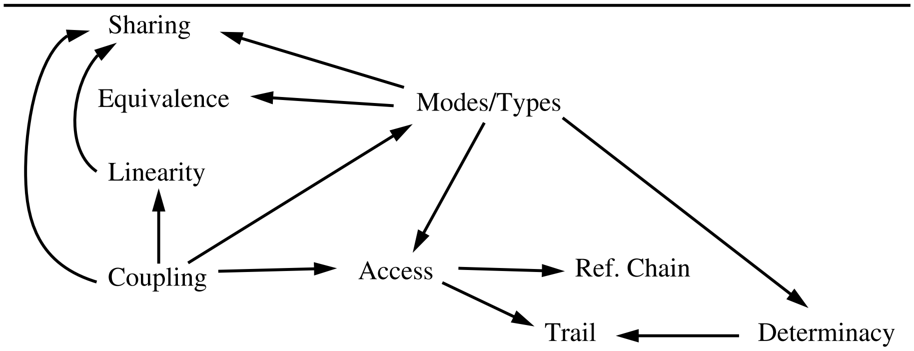

Figure 8. Interdependencies between Types of Analyses

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 variable'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.

3 Evaluation

This section 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.

3.1 An Analysis Framework for Compilation

A number of analysis frameworks have been proposed [2, 3, 15, 19, 20, 23, 36, 37, 44]. However, none of these frameworks, as given, provide all of the functionality required for an abstract interpreter integrated with a compiler. In [13] we describe the analysis framework we developed to perform this research. 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 local information about the clauses being compiled. *Using this unified approach we cut the compilation time almost in half.*

The framework uses 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.

3.2 Benchmarks

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. Specifically, they were used by Van Roy [43] and Taylor [40]. They are described in Table 3.

3.3 Evaluation Measures

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 the generated code) and execution time relative to some baseline case. All measurements are relative to those obtained using the Aquarius Prolog compiler as currently distributed.

The benchmarks were compiled using the Aquarius Prolog compiler, integrated with our analysis framework, running under Quintus Prolog on an HP 9000/700 workstation. 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

Table 3. Benchmark Descriptions

Benchmark	Description
deriv	Symbolic differentiation of four equations.
nreverse	Naive reversal of a list of 30 integers.
qsort	Quicksort on a list of 50 integers.
serialise	Computes the "serial" number for values in a list.
mu	Solves a theorem in Hofstadter's mu-math system.
pri2	Sieve of Erastosthenes for first 100 integers.
queens_8	Finds all solutions to the eight queens problem
fast_mu	A "faster" solution to mu-math.
query	Database lookup and integer math to answer a simple query.
press1	A simplified version of the PRESS symbolic equation solver.
tak	Highly recursive integer function (takeuchi function).
sendmore	Depth-first search to solve a cryptographic problem.
poly_10	Symbolic polynomial expansion.
zebra	Solution to a simple word problem (where does the zebra live?).
prover	A simple theorem prover.
meta_qsort	A simple meta-interpreter for Prolog, interpreting qsort.
nand	A logic circuit synthesis program, based on heuristic search.
chat_parser	The sentence parser for a natural language system.
browse	A "classic" LISP benchmark, exercising pattern matching.
unify	A unification compiler, from the Aquarius compiler.
flatten	Flattens Prolog code into a simpler syntax.
crypt	Depth-first search to solve a crypto-arithmetic problem.
simple_analyzer	A simple mode analyzer, from the Aquarius compiler.
reducer	A graph reducer for T-combinators.
boyer	A Boyer-Moore theorem solver (a "classic" LISP benchmark).

16 MHz, with 16 Mb of main memory, 64 Kb 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, thereby eliminating unknown factors, such as system load.

3.4 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 Section 2 and various Prolog optimizations, as shown in Figure 3. 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.

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. The

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

Figure 9. 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, whereas 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. Also, it actually decreases the compilation time (to 45% of Aquarius') since it now 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.

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 10. 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

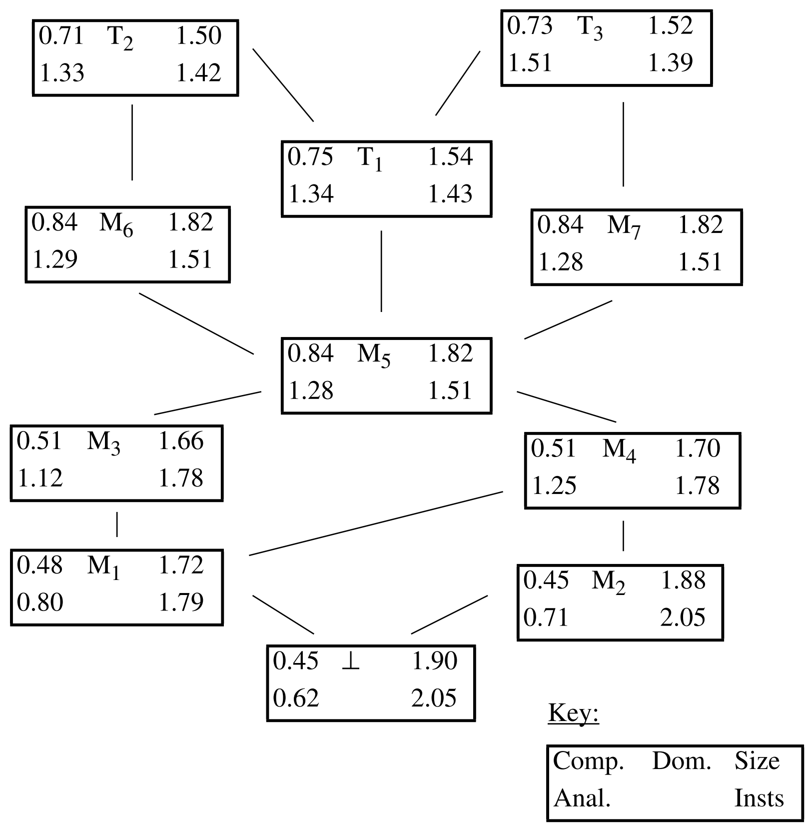

Figure 10. Performance Lattice for Mode/Type Analysis

size increases. The reason for this is that M_5 is the simplest domain for which *mode enrichment* [42] can be performed. This optimization reduces execution time, but does so by increasing code size.

Understanding 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'). Domain M_2 provides almost no improvement over no analysis. We would expect this, since the domain only captures unbound variables. This is a mode which is quickly lost without some form of aliasing information. 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 numbers for expressions. With type analysis, we can determine, at compile-time, that many of the operands in arithmetic expressions are integers. 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 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. 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. Although we allowed the analyzer to run for over three CPU hours, the analysis failed to terminate for almost half of the benchmarks. For the benchmarks we were able to get results, these results are somewhat discouraging. There was a code size reduction in only five of the benchmarks (for an average of 7% reduction) and an execution time reduction in only three (for an average of 1%). One benchmark (fast_mu) got even larger and slower! This is due to the inability of domain T_4 to express the results of tests added to the code during determinism extraction (e.g., struct(X)). The increase in compilation time was *over five times*. Most of this 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 increased the cost of analysis, due to type propagation from possible variable coupling.

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 11.

From this figure, we see similar trends to those we saw for modes and types. For the most part, as the domains get more precise, code size and run time decrease while analysis time increases. The analysis time actually decreases, however, 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.

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*. Note the following in Figure 11:

- 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-

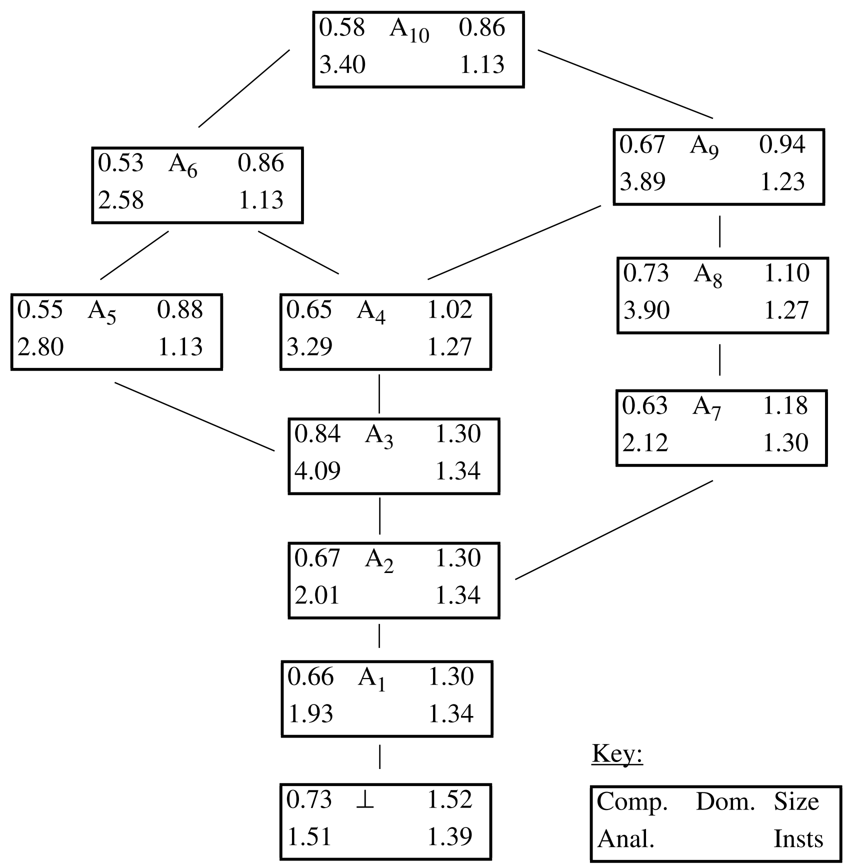

Figure 11. Performance Lattice of Aliasing Analysis

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.

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 section 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 12.

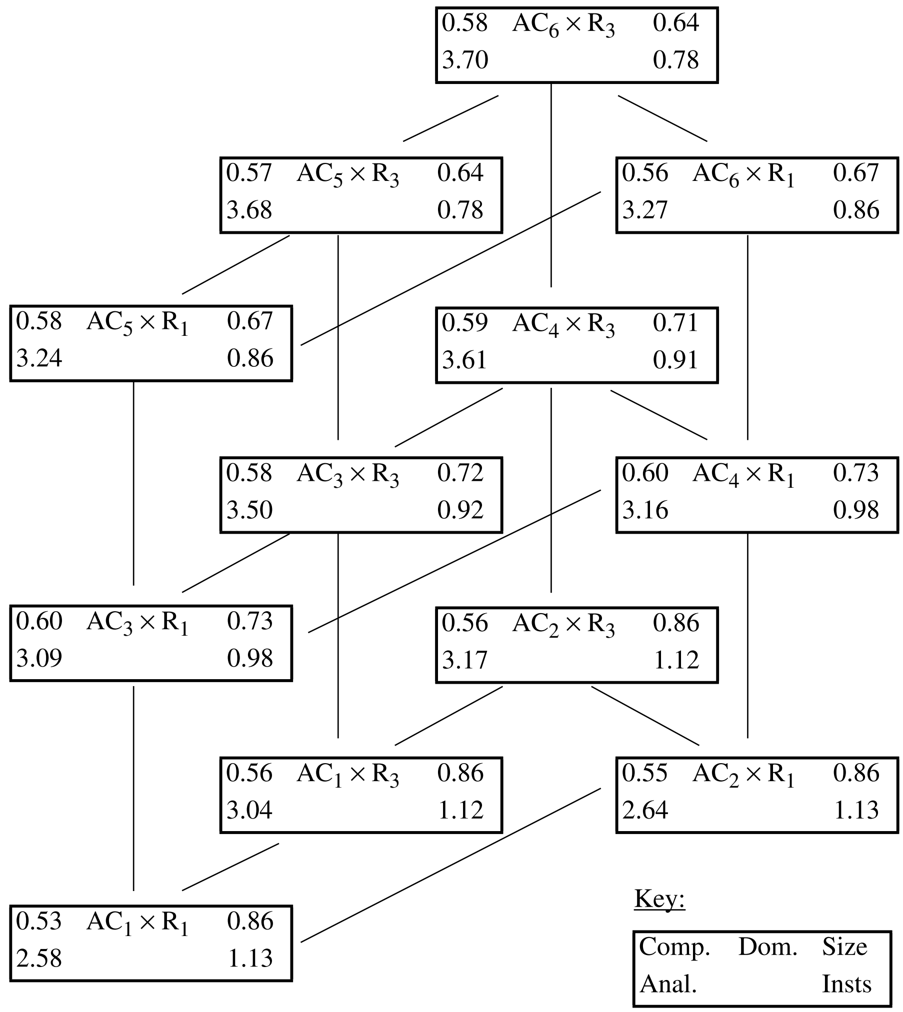

Figure 12. Performance Lattice of Implementation-Dependent Analysis

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), it 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 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%).

3.5 Summary

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 13. All of the data points from Figures 10 through 12 are included.

Figure 13?? 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 (see Figs. 13a and 13??). 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 to trade off code size for run time.

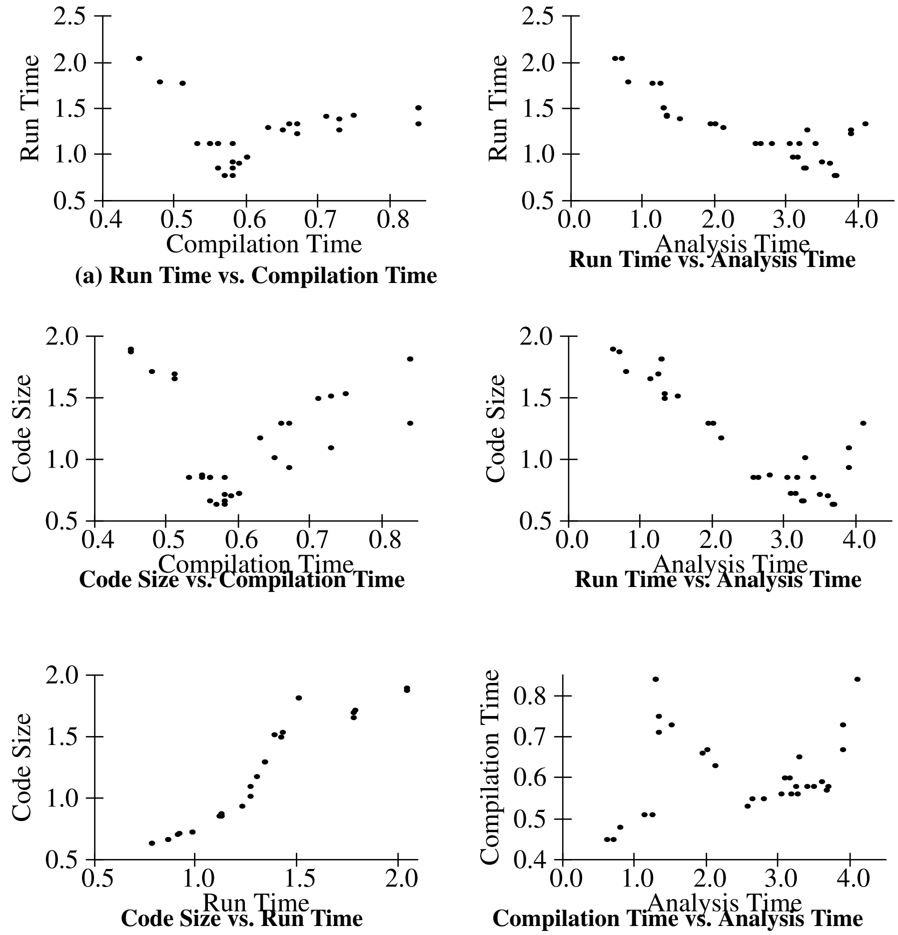

Figure 13. Summary of Performance Evaluation

4 Conclusion

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. Hopefully, 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.

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.

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).

5 Acknowledgements

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

6 References

[1] 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.

[2] 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.

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] T. Getzinger. *Abstract Interpretation for the Compile-time Optimization of Logic Programs*. Ph.D. Thesis, University of Southern California, December 1993.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

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

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

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

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

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

[35] 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.

[36] 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.

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

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

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

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

[41] 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.

[42] 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.

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

[44] 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.