CONF-9006169--1

DE90 010060

Multiple Specialization of Logic Programs with Run-time Tests

Dean Jacobs and Anno Langen
University of Southern California, CS Dept.
Los Angeles, CA 90089-0782 USA

Will Winsborough
Argonne National Laboratory, MCS Div.
Argonne, IL 60439-4844 USA

Abstract

This paper presents a framework for multiple specialization of logic programs that incorporates run-time testing. This framework supports specialization on the basis of a compiler-generated "wish list" of requirements that enable useful optimizations. In addition, entry mode declarations are used to restrict the class of reachable activations. Our goal is to generate code containing tests at outer levels of the call tree that guard high-performance specialized procedures that are likely to be called.

The starting point for this work is the multiple specialization technique of Winsborough, which introduced the notion of a *call-path automaton*. The transition function for a call-path automaton specifies which version of a procedure should be invoked by each call in each version of each procedure. As a first step, we extend such automata to include "test arcs" that select different versions. We then show how to construct an automaton containing a given set of tests at given test sites. Finally, we discuss techniques for abductively deriving a test at a given site that, upon success, ensures some property at a requirement site.

1 Introduction

Most compiler optimizations for logic programs can be performed only if certain "safety requirements" hold. For example, unification can be simplified if certain arguments to a call are free or ground, data structures can be destructively updated if they are no longer referenced, and independent calls can be scheduled for parallel execution. A *specialized* implementation of a program component incorporates optimizations that are safe only for particular activations of that component. Commonly, procedures are specialized according to their possible activations as determined by global flow analysis. In *single* specialization [2, 4, 8, 10, 11, 12], one version of each procedure is created to handle all of its activations. In *multiple* specialization [13], sev-

---

eral versions of each procedure are created, and the appropriate version is selected at each invocation.

An alternative way of introducing specialization is to explicitly guard different versions of program components by run-time tests. Applications of this approach, e.g., DeGroot's formulation of restricted-and parallelism [5], have inserted tests directly at the site of the optimization. This can lead to excessive testing that, at least in some cases, offsets the benefit of the optimization. Nevertheless, this approach has considerable merit since it allows specialization to be directed towards specific optimizations. In contrast, specialization based solely on flow analysis allows an optimization to be performed only if an activation enabling it happens to be generated. An ideal compromise would be to use tests at outer levels of the call tree to enable specialized versions of procedures with repeated inner-level optimizations.

This paper presents a framework for multiple specialization of logic programs that incorporates run-time testing. This framework supports specialization on the basis of a compiler-generated "wish list" of requirements that enable useful optimizations. In addition, entry mode declarations are used to restrict the class of reachable activations. Our goal is to generate code containing tests at outer levels of the call tree that guard high-performance specialized procedures that are likely to be called.

The starting point for this work is the multiple specialization technique of Winsborough [13], which introduced the notion of a *call-path automaton*. In this context, a *version* of a procedure is associated with a class of activations of its clauses. The transition function for a call-path automaton specifies which procedure version should be invoked by each call in each version of each procedure. The compiler uses the call-path automaton, together with a set of entry versions, to determine which versions of procedures should be created and how versions should call each other. This paper extends the notion of a call-path automaton to include "test arcs"; here, an automaton is viewed as a directed graph. Upon traversing a test arc, the automaton moves to either of two different versions, depending on the outcome of the test. Thus, tests provide additional information about procedure activations, thereby setting up new specializations. This research uses the framework of abstract interpretation to parameterize the specialization process with respect to the type of safety requirements used.

The basis of our framework is an algorithm for constructing a call-path automaton containing a given set of tests at given test sites. This leaves open the problem of generating suitable tests and test sites from the compiler wish list. One approach is to insert tests, perhaps suitably strengthened, directly at the requirement site given on the wish list. In certain cases, the results of such tests will be propagated by the automaton across clause boundaries, enabling repeated inner-level optimizations. More generally, one may wish to abductively derive a test at a given site that, upon success, ensures some property at the requirement site. We discuss techniques for deriving such tests. As a first step, using forward abstract execution, we show how to

determine whether the success of a given test leads to the satisfaction of a requirement. Second, using the condensing technique presented in [6, 7], we discuss how to collapse this forward-flow computation into essentially a single unification. This process reduces the general abduction problem to abduction for a single unification. Generally speaking, an efficient solution of even this simpler problem must be domain-dependent. We have worked out efficient domain-dependent solutions for two nontrivial domains for deriving groundness information and plan to report these results elsewhere.

This paper is organized as follows. Section 2 presents a brief overview of abstract interpretation for logic programs and extends the standard framework to deal with run-time testing. Section 3 introduces call-path automata. Section 4 sketches the proof that our specialization technique is sound. Section 5 addresses the issue of selecting suitable tests and test sites and discusses abduction. Section 6 presents concluding remarks.

2 Abstract Interpretation and Testing

In this section, we present a brief overview of abstract interpretation for logic programs and extend the standard framework to deal with run-time testing. Our intention here is to give an intuitive understanding of the concepts used in this paper, thus, we omit certain details. These concepts are formally defined in Section 4, where the soundness of our technique is discussed.

Abstract interpretation [1, 3] is a framework for deriving data-flow information about a program. In this approach, the given language is assigned both a concrete semantics and an abstract semantics. The domain of computation states in the concrete semantics is replaced by a domain of *descriptions* of states in the abstract semantics. Each basic operation on the concrete domain is replaced by a corresponding operation on the abstract domain. Execution of a program according to the abstract semantics produces an approximation to the data-flow information as given by the concrete semantics. This approximation specifies a data-flow analysis.

In this paper, we are concerned with a particular form of concrete semantics, called a *collecting semantics*, that specifies the set of all substitutions that can occur at each point in a program. In this semantics, the basic domain of values $Subst_C$ is taken to be the set of all sets of substitutions ordered by subset, alternately written $\subseteq$ and $\sqsubseteq_C$. We formalize the meaning of programs in terms of several operations, including the following.

- $entry_C(k, a, S)$ is the set of all substitutions that can occur on entry to the k^{th} clause when atom a is called under a substitution in $S \in Subst_C$.
- $propagate_C(b, S)$ is the set of all possible substitutions that can result from executing the clause body b starting with a substitution in $S \in Subst_C$.

In the abstract semantics, $Subst_C$ is replaced by a domain $Subst_A$ of descriptions of substitutions ordered by $\sqsubseteq_A$. The operations $entry_C$ and $propagate_C$ on the concrete domain are replaced by the operations $entry_A$ and $propagate_A$ on the abstract domain. These abstract operations are expected to approximate the concrete operations in the following sense. We assume there is a *concretization function* $\gamma : Subst_A \rightarrow Subst_C$ such that $\gamma(S)$ is the set of all substitutions that are described by S. For each concrete operation f_C and corresponding abstract operation f_A we require that for all $S \in Subst_A$, $f_C(\gamma(S)) \subseteq \gamma(f_A(S))$. γ is required to respect the orderings on the domains, i.e., for all $S_1, S_2 \in Subst_A$, $S_1 \sqsubseteq_A S_2 \Longrightarrow \gamma(S_1) \subseteq \gamma(S_2)$. In all but Section 4 of this paper, we focus on the abstract domain and operations and hence drop the subscript A from abstract operations.

The examples in this paper are based on the abstract domain *Prop* adapted from [9]. Elements of this domain are propositional formulae that describe groundness dependencies between variables. For example, $\mathtt{X} \leftrightarrow \mathtt{Y}$ describes the set of all substitutions where X and Y are in the same state of groundness and all subsequent instantiation will leave them in the same state of groundness. Thus, $\mathtt{X} \leftrightarrow \mathtt{Y}$ describes the substitutions $\{\mathtt{X} \mapsto \mathtt{a}, \mathtt{Y} \mapsto \mathtt{b}\}$ and $\{\mathtt{X} \mapsto \mathtt{Y}\}$, but does not describe the substitutions $\{\mathtt{X} \mapsto \mathtt{a}\}$ or $\{\mathtt{Y} \mapsto \mathtt{f(Z)}, \mathtt{X} \mapsto \mathtt{f(Q)}\}$. Further, *true* describes all substitutions ($\top$) and *false* describes no substitutions ($\bot$). The concretization function is given by $\gamma(P) = \{\sigma | \forall \theta : P \text{ holds under } \sigma \circ \theta\}$ where variable v holds under substitution σ iff σ grounds v and the propositional connectives have their usual interpretation. Thus, for example, $\gamma(\mathtt{X} \wedge \mathtt{Y}) = \{\sigma | \sigma \text{ grounds X and Y}\}$. Abstract operations are defined over this domain, e.g., if the head of clause k is p(X,Y), then $entry_A(k, \mathtt{p(f(Q),R)}, \mathtt{Q} \leftrightarrow \mathtt{R}) = \mathtt{X} \leftrightarrow \mathtt{Y}$. This clause-activation description for clause k tells us that, if either X or Y subsequently becomes ground, both X and Y will then be ground.

We assume there is a concrete domain of tests *Test* that, when executed at run-time, test attributes of call arguments that are expressible in $Subst_A$. For example, for the domain *Prop*, elements of *Test* should determine whether call arguments are ground or contain precisely the same variables. We assume tests are applied to arguments, rather than to individual variables in a clause. This simplifies the presentation and, in any case, is a reasonable approach when one is interested only in procedure-level optimizations. Our framework can be generalized to incorporate clause-level optimizations directly.

We assume there are functions

$$\textit{then-branch}_C : \textit{Test} \times \textit{Atom} \times \textit{Subst}_C \rightarrow \textit{Subst}_C$$
$$\textit{else-branch}_C : \textit{Test} \times \textit{Atom} \times \textit{Subst}_C \rightarrow \textit{Subst}_C$$

that interpret tests on the concrete level. In particular, $\textit{then-branch}_C(t, a, S)$ produces the set of all substitutions $\sigma \in S$ such that $a\sigma$ satisfies t. Similarly, $\textit{else-branch}_C(t, a, S)$ produces the set of all substitutions $\sigma \in S$ such that $a\sigma$ does not satisfy t. The abstract versions of these functions, $\textit{then-branch}_A$

and *else-branch*$_A$, interpret tests on the abstract level. As usual, we require that for all $S \in Subst_A$

$$\begin{array}{c} \textit{then-branch}_C(t, a, \gamma(S)) \subseteq \gamma(\textit{then-branch}_A(t, a, S)) \\ \textit{else-branch}_C(t, a, \gamma(S)) \subseteq \gamma(\textit{else-branch}_A(t, a, S)). \end{array}$$

In addition, we assume the analysis does not lose information when tests are added, i.e., $\textit{then-branch}_A(t, a, S) \sqsubseteq_A S$ and $\textit{else-branch}_A(t, a, S) \sqsubseteq_A S$.

3 Call Path Automata

In this section, we introduce call-path automata, the fundamental structure in our multiple specialization technique. First, we introduce the notion of call-path automata without tests. The transition function for such an automaton specifies which procedure version should be invoked by each call in each version of each procedure. Our formulation of this concept is taken from [13], except that we use *entry* and *propagate*. Second, we introduce the notion of call-path automata with tests. The transition function for such an automaton can select versions of procedures on the basis of the outcome of tests.

Throughout our discussion, we assume a fixed rule base with n clauses where the i^{th} clause has m_i calls in its body.

3.1 Automata without Tests

In our framework, a *version* of a procedure is associated with a class of activations of its clauses. A class of activations of a clause is given by an element of the abstract domain *Subst* describing the state of the computation immediately upon entry to the clause. It is convenient to formalize the notion of a version as an association of an element of *Subst* to every clause in the rule base.

$$\textit{Version} \quad = \quad \{1 \ldots n\} \rightarrow \textit{Subst}$$

All versions that arise in our constructions are everywhere $\perp$ except for the clauses of the procedure being invoked. We say the version is *for* this procedure.

A call in the program is indicated by a pair $\langle i, j \rangle$ where i is a clause number and j is the number of a call within that clause. A call path is given by a sequence of such pairs indicating clauses selected and calls invoked. While scanning a call path, the automaton moves from version to version according to the transition function. It is convenient to combine the current version v with the current call $\langle i, j \rangle$ into a single triple $\langle v, i, j \rangle$ called a *Site*.

$$\textit{Site} = \{ \langle v, i, j \rangle \mid v \in \textit{Version} \wedge 1 \leq i \leq n \wedge 1 \leq j \leq m_i \}.$$

The transition function Δ for the call-path automaton is constructed as follows.

$$\Delta : \mathit{Site} \rightarrow \mathit{Version}$$
$$\Delta\,\langle v, i, j\rangle\, k = \mathit{entry}(k, a_j, \mathit{propagate}(a_1 \ldots a_{j-1}, v\, i)),$$

where $a_1 \ldots a_{m_i}$ is the body of the i^{th} clause and $j \leq m_i$.

The transition function tells the compiler how specialized procedure versions should call one another. This leaves open the problem of selecting suitable entry versions. One way for the compiler to handle an entry call is to invoke the most-general version of the associated procedure. A more satisfactory solution is to derive entry versions from entry mode declarations; this allows specialized versions to be invoked at entry. Conventionally, such mode declarations are viewed as being "import" specifications that tell the compiler what procedure activations it must be prepared to handle. By contrast, in multiple specialization, modes are viewed as being "export" specifications of entry versions that the compiler will make available. The compiler has the option of providing a different specialized implementation for each entry mode declaration. This assumes some mechanism, such as a suitable linker, that selects an appropriate entry version for each entry call. The compiler needs to construct specialized implementations only for those versions that are reachable from the set of entry versions, as formalized below.

$$\begin{array}{rcl} \mathit{EntryVersions} & \subseteq & \mathcal{P}(\mathit{Version}) \\ \mathit{ReachableVersions} & : & \mathit{EntryVersions} \rightarrow \mathcal{P}(\mathit{Version}) \end{array}$$

$\mathit{ReachableVersions}(V)$ is the smallest set satisfying

1. $V \subseteq \mathit{ReachableVersions}(V)$, and
2. $\Delta\,\langle v, i, j\rangle \in \mathit{ReachableVersions}(V)$ if $v \in \mathit{ReachableVersions}(V)$

Example 1: Consider the automaton for the program `all_p`.

```
1. all_p([], _).
2. all_p([H|T], X) :- p(H, X), all_p(T, X).

3. p(A, B) :- q(B, Q), r(A), s(Q, A).
```

If the only entry mode declared has `all_p` called with unknown arguments, i.e., *true* in *Prop*, the reachable versions are $v_1 = \langle \mathit{true}, \mathit{true}, \mathit{false}\rangle$, $v_2 = \langle \mathit{false}, \mathit{false}, \mathit{true}\rangle$, and the trivial version $v_3 = \langle \mathit{false}, \mathit{false}, \mathit{false}\rangle$, which is meaningless and reachable as a consequence of the *false* ($\perp$) elements in versions. So, for example, v_1 says that Clauses 1 and 2 can be activated with any substitution and Clause 3 is not activated. We have $\Delta\,\langle v_1, 2, 1\rangle = v_2$ and $\Delta\,\langle v_1, 2, 2\rangle = v_1$.

Example 2: If in addition to the entry modes assumed in Example 1, we have an entry mode declaring both arguments to `all_p` to be ground, we

obtain two additional reachable versions, $v_4 = \langle true, H \wedge T \wedge X, false \rangle$ and $v_5 = \langle false, false, A \wedge B \rangle$, with $\Delta \langle v_4, 2, 1 \rangle = v_5$ and $\Delta \langle v_4, 2, 2 \rangle = v_4$. Note that v_4 calls v_4. This means that the groundness information is propagated intact to the recursive invocation.

Generally speaking, there is no guarantee that each reachable version will enable unique, useful optimizations. Thus, the compiler may choose to collapse versions that enable similar optimizations. This process must be performed carefully since versions that enable similar optimizations may, on the same call-path input, lead to versions that enable different optimizations. There is an obvious correspondence between this problem and the well-known (and solved) problem of minimizing deterministic finite automata. This correspondence gives a solution to the problem of collapsing similar versions, which is described in [13]. Note that, if this process collapses two entry versions, the compiler will export the same specialized implementation for both entry versions.

3.2 Automata with Tests

We now extend the notion of a call-path automaton to include tests. Recall that tests are applied to procedure arguments rather than to individual variables in a clause. Viewing an automaton as a directed graph, we incorporate tests as labels on *test arcs*. A test arc has one tail and two heads, corresponding to the test outcomes true and false. Tests incorporate additional information not already known from the version.

Each test arc is associated with a specific *Site*. This is formalized by a pool of tests given by a set $testsites \subseteq Site$ and a function $test : testsites \rightarrow Test$. The test pool determines the transition function $\hat{\Delta} : Boolean \rightarrow Site \rightarrow Version$ for the call-path automaton with tests. The new Boolean argument here indicates test outcome. Where there is no test, this Boolean is ignored.

$$\begin{array}{rcll} \hat{\Delta}\, b\, s & = & \Delta\, s & \text{if } s \notin testsites \\ \hat{\Delta}\, true\, s\, k & = & then\text{-}branch(test\, s, h_k, \Delta\, s\, k) & \text{if } s \in testsites \\ \hat{\Delta}\, false\, s\, k & = & else\text{-}branch(test\, s, h_k, \Delta\, s\, k) & \text{if } s \in testsites \end{array}$$

The entry versions and the test pool determine the versions that label the reachable nodes of the call-path automaton. For a set of entry versions V, $\widehat{ReachableVersions}(V)$ is defined by analogy to $ReachableVersions(V)$.

Example 3: Suppose that the compiler has indicated on its wish list that groundness of both arguments to `all_p` is required for some optimization. (For example, groundness of both arguments allows parallel execution of `p` and `all_p` under independent and-parallelism.) Suppose further that the program of Example 1 is embedded in a larger program containing a single call of `all_p`, located at site s, and that global analysis alone cannot guarantee that in this call both arguments are ground. Before any tests are

added, the functions $\hat{\Delta}\,true$ and $\hat{\Delta}\,false$ are each identical to Δ in Example 1. Introduction of a test for the groundness of each argument of the call at s adds versions v_4 and v_5 of Example 2 to $\widehat{ReachableVersions}$: we have $\hat{\Delta}\,true\,s = v_4$, $\hat{\Delta}false\,s = v_1$, and $\hat{\Delta}\,b\,s' = \Delta\,s'$ for all $s' \neq s$ and $b \in \{true, false\}$. Note that the result of the new test propagates intact to recursive calls. Thus, introducing the test at s and applying the call-path automaton enables repeated application of the optimized version without need for repeated testing.

As in the case without tests, different reachable versions that enable similar optimizations should be collapsed. Collapsing versions in an automaton with tests is not significantly different from collapsing versions in an automaton without tests.

4 Soundness of Δ

In this section, we argue that our specialization technique is sound. To this end, we introduce the notion of concrete call-path automata. The concrete transition functions Δ_C and $\hat{\Delta}_C$ are defined analogously to the abstract transition functions Δ_A and $\hat{\Delta}_A$ by using the concrete domains and operations $Version_C$, $Site_C$, $entry_C$, and $propagate_C$. $Version_C$ and $Site_C$ differ from $Version_A$ and $Site_A$ only in that they are constructed by using $Subst_C$ rather than $Subst_A$. Formally, $Subst_C$ is taken to be all sets of renaming-equivalence classes of substitutions:

$$Subst_C = \mathcal{P}(\{[\sigma] \mid \sigma \text{ an idempotent substitution}\})$$

where $[\sigma]$ is the renaming-equivalence class of σ. As discussed in Section 2, we assume that the concretization function γ maps each abstract domain element to the concrete domain element it represents.

We now define $entry_C$, $propagate_C$, $entry_A$, and $propagate_A$. Following [6, 7], we introduce an operator $unify_C$ that combines the standard notions of unification, composition, and restriction.

$$unify_C : Atom \times Subst_C \times Atom \times Subst_C \to Subst_C$$
$$unify_C(a, S, a', S') = \{[|\sigma \circ \mathbf{mgu}(a\sigma, \overline{a'\sigma'})|_{\mathbf{dom}(\sigma)}] : \sigma \in S \wedge \sigma' \in S'\},$$

where some new notation must be introduced: **mgu** computes a most general unifier; using a set of variables disjoint from those in the renaming equivalence classes, the overbar renames variables to standardize apart the operands; $\mathbf{dom}(\sigma)$ is the set of variables on which σ is defined; and the vertical bars denote substitution restriction. In addition, we introduce the concrete operation $init_C : Clause \to Subst_C$, which returns the renaming-equivalence class of the identity substitution over the variables in the clause. The corresponding abstract operations, $unify_A$ and $init_A$, are required to satisfy

$$unify_C(a, \gamma(S), a', \gamma(S')) \subseteq \gamma(unify_A(a, S, a', S'))$$
$$init_C(C) \subseteq \gamma(init_A(C)).$$

By using the corresponding versions of *unify* and *init*, the concrete and abstract versions of *entry* are each defined according to

$$entry(k, a, S) = unify(h_k, init(C_k), a, S),$$

where C_k is the k^{th} clause and h_k is the head of C_k. To define propagate, we use the semantic functions of [6, 7]. These use the following domains.

$$\begin{aligned} ClauseMeaning &= Atom \times Subst \to Subst \\ RuleBaseMeaning &= \{1 \ldots n\} \to ClauseMeaning \\ BodyMeaning &= Subst \to Subst \end{aligned}$$

As above, n is the number of clauses in the rule base.

$$\begin{aligned} &\mathbf{R} : RuleBase \to RuleBaseMeaning \\ &\mathbf{C} : Clause \to RuleBaseMeaning \to ClauseMeaning \\ &\mathbf{B} : Body \to RuleBaseMeaning \to BodyMeaning \\ &F : RuleBase \to RuleBaseMeaning \to RuleBaseMeaning \end{aligned}$$

$$\begin{aligned} \mathbf{R}[\![C_1 \ldots C_n]\!] &= \mathrm{fix}(F[\![C_1 \ldots C_n]\!]) \\ \mathbf{C}[\![h\text{:-}b]\!]\, e\, (a, S) &= \text{if } a \text{ and } \overline{h} \text{ are unifiable then} \\ &\quad unify(a, S, h, \mathbf{B}[\![b]\!]\, e\, unify(h, init(h\text{:-}b), a, S)), \\ &\quad \text{else } \bot \\ \mathbf{B}[\![]\!]\, e\, S &= S \\ \mathbf{B}[\![a_1 \ldots a_i]\!]\, e\, S &= \bigsqcup \{e\, j\, (a_i, \mathbf{B}[\![a_1 \ldots a_{i-1}]\!]\, e\, S) \mid 1 \le j \le n\} \\ F[\![C_1 \ldots C_n]\!]\, e\, i &= \mathbf{C}[\![C_i]\!]\, e \end{aligned}$$

By using the corresponding versions of the semantic functions $\mathbf{B}$ and $\mathbf{R}$, we define $propagate_C$ and $propagate_A$:

$$propagate(b, S) = \mathbf{B}[\![b]\!]\, (\mathbf{R}[\![r]\!])\, S.$$

An abstract call path automaton is sound if, for every procedure invocation during execution of the subject program started with an entry version, the state reached by the automaton on scanning the current call path approximates the procedure activation. This global soundness is established as follows [13]. First, Δ_C is related to SLD-resolution semantics with Prolog's left-to-right computation rule. Second, induction on the length of the call path is used to show that, on scanning the same call path, the version reached by Δ_C is approximated by the version reached by Δ_A. The inductive proof uses a local soundness property, which we state here for completeness.

Fact 4.1 (Soundness of Δ_A) *Δ_A is sound iff for all $\langle v_C, i, j\rangle \in \mathrm{Site}_C$, $v_A \in \mathrm{Version}_A$, and $k \in \{1 \ldots n\}$ we have*

$$v_C\, i \subseteq \gamma(v_A\, i) \Longrightarrow \Delta_C\, \langle v_C, i, j\rangle\, k \subseteq \gamma(\Delta_A\, \langle v_A, i, j\rangle\, k)$$

Soundness is complicated somewhat by the presence of tests. In addition to scanning the sequence of $\langle i, j \rangle$ pairs that compose the call-path, a call-path automaton with tests scans the test outcomes that result from the execution of the tests that are given by the test pool for each call on the call path. An abstract automaton with tests is sound if, for every procedure invocation during execution of the subject program started with an entry version, the state reached by the automaton on scanning the current call path *and the corresponding test outcomes* approximates the procedure activation. As in the test-free case, global soundness is established by inductive application of a local soundness property. Local soundness says that, during execution of the subject program, if some procedure activation is approximated by the current state in the automaton, then the procedure activation reached by selecting the i^{th} clause, executing the first $j-1$ calls, and invoking the j^{th} call based on some test is approximated by the state indicated by the transition function on scanning $\langle i, j \rangle$ and the outcome of the associated test. Local soundness can be established in the manner of the case without tests.

5 Inferring Suitable Tests

In this section, we take up the problem of selecting suitable tests to incorporate into the call-path automaton. We begin our discussion with two general comments about selecting tests. First, tests should be added by starting at the outer levels of the program's call graph and working inward, recomputing the reachable versions each time. This approach is recommended because the addition of a new test can result in subsequent reachable versions that describe smaller sets of procedure activations. Second, tests that always succeed or always fail are a waste of time and should not be introduced. For site $s = \langle v, i, j \rangle$, if $S = \mathit{propagate}(a_1 \ldots a_{j-1}, v\, i)$, where $a_1 \ldots a_{m_i}$ is the body of clause i, then we would not want to add at s any test t such that $\mathit{then\text{-}branch}(t, a_j, S) = S$ or $\mathit{then\text{-}branch}(t, a_j, S) = \bot$.

Recall that our goal is to let the compiler drive the specialization process by providing a "wish list" of requirements for worthwhile optimizations. Formally, a wish list is taken to be a subset of *Requirement*, where $\mathit{Requirement} = \mathit{Proc} \times \mathit{Test}$ and *Proc* is the set of procedure names in the program. We want to incorporate tests that ensure that requirements on the wish list are met. Moreover, we want such tests to occur at outer levels of the call tree where they enable repeated inner-level optimizations. Example 3 shows that there are cases where direct incorporation of a test on the wish list can lead to specializations of the desired form. In other cases, tests will have to be derived from the wish list. One approach is to strengthen a test on the wish list so that $\hat{\Delta}$ can propagate it down the call path to other uses of the procedure.

Example 4: If the program in Example 1 used

```
1. all_p([], A, X) :- p(A, X).
```

```
2. all_p([H|T], A, X) :- p(A, X), all_p(T, H, X).
```

to define `all_p`, the compiler might suggest the requirement (ground(arg_1) ∨ ground(arg_2)) ∧ ground(arg_3) for `all_p`, enabling parallel execution of `p` and `all_p` in Clause 2. When this test is added at the procedure entry, the only information that propagates from the test arc's success branch to the recursive call is ground(arg_3). Thus the test would have to be repeated at the recursive call. On the other hand, the stronger test ground(arg_1) ∧ ground(arg_3) would propagate intact to the recursive call, eliminating the need for a repeated test. This example illustrates how helpful it can be to incorporate a test that is stronger than the requirement suggested by the compiler.

A more general approach is to position a test so that the information it provides propagates to several uses of a procedure. This position may not be at the entry of the procedure, in particular, it may lie on the call path to several uses of the procedure.

Example 5: Consider again the original program in Example 1 and suppose that the only requirement suggested by the compiler is that calls to procedure `p` satisfy ground(arg_1) ∨ ground(arg_2), enabling parallel execution of `q` and `r` in Clause 3. If this test is incorporated directly at the entry to procedure `p`, there is no way to propagate the information it gains to subsequent uses of `p`, since that site does not lie on the call path to any subsequent use. On the other hand, it is possible to recognize uses of `all_p` that lead to this requirement on `p` being met by testing ground(arg_1) ∨ ground(arg_2) (or just ground(arg_2), for that matter).

We now address the problem of finding tests of this sort: given a requirement and a test site, is there a test that, when satisfied at the site, ensures the requirement is met, and if so, what is the weakest such test? A suitable test has the following property. If, at some stage in some execution, the activation of the call at the test site satisfies the test, then the requirements must be met for all clause activations that are reached during the execution of the test-site call. A test that, when satisfied at the test site, guarantees satisfaction of a requirement is said to guard that requirement. Here, we seek the weakest guarding tests, with the understanding that it may often be desirable to use less costly, stronger tests.

We call the process of identifying suitable tests for a given test site and a given requirement *test abduction* because the problem is to find a test whose success would allow us to infer that the requirement is met. We begin our discussion of test abduction by showing how to determine whether the success of a given test incorporated at the test site leads to the satisfaction of a requirement. This specification of the abduction problem requires some new machinery.

Recall that a site consists of a version, a clause whose activation is described by that version, and a call in that clause. For (concrete) site

$s = \langle v, i, j \rangle$ and test t at s, the function $success_C(s, t)$ gives the set of substitutions that, at call j of clause i, satisfy t and that arise at site s, i.e., that arise at the point of invoking call j of clause i when clause i is executed starting with a substitution give by $v\,i$. For abstract and concrete alike,

$$\begin{aligned} success &: Site \times Test \rightarrow Subst \\ success(\langle v, i, j \rangle, t) &= then\text{-}branch(t, a_j, propagate(a_1 \ldots a_{j-1}, v\,i)), \end{aligned}$$

for $a_1 \ldots a_{m_i}$ the body of clause i and $j \leq m_i$. Safety of $success_A$ follows from safety of $then\text{-}branch_A$.

When our technique is to be used to specialize unification code, it is necessary to test the requirement before performing the unification. Thus, our specification of the abduction problem applies the requirements at the point of call, rather than in the clause heads. Consequently, we need some machinery to construct a description of the call activations possible for each call in the program. For given atom a and $A \in Subst$, the set of call activations reachable from the call a activated according to A is given by the function

$$ReachableCalls : (Atom \times Subst) \rightarrow Atom \rightarrow Subst$$

where, under the pointwise order, $ReachableCalls(a, A)$ is the least $R : Atom \rightarrow Subst$ such that

$$\begin{aligned} &A \sqsubseteq R\,a, \text{ and} \\ &\text{for all } a' \text{ in the program,} \\ &propagate(a_1 \ldots a_{j-1}, entry(i, a', R\,a') \sqsubseteq R\,a_j, \end{aligned}$$

where $a_1 \ldots a_{m_i}$ is the body of the i^{th} clause and $j \leq m_i$.

Throughout the process of inferring tests, the test pool is omitted from calculations. This enables the condensing technique we discuss below. Only the candidate test for meeting the requirement is considered. A test that is found to satisfy the requirement when the test pool is ignored will also satisfy the requirement when the test pool is used, because tests lead to no loss of precision.

Using *ReachableCalls*, we can specify, for each call in the program, the activations of that call that are guarded by the test t at site s in the current automaton: $ReachableCalls(a, success(s, t))$, where a is the call at site s. These are the call activations that must be considered to determine whether t satisfies a given requirement.

The satisfaction of a test *req* by $A \in Subst_C$ and an atom a is expressed by $satisfies_C(req, a, A)$, which must obey

$$satisfies_C(req, a, A) \text{ iff } then\text{-}branch_C(req, a, A) = A.$$

To be reasonable, $satisfies_A$ must obey $A \sqsubseteq_A B \Longrightarrow (satisfies_A(i, a, B) \Rightarrow satisfies_A(i, a, A))$. The safety requirement for $satisfies_A$ is

$$satisfies_A(i, a, A) \Rightarrow satisfies_C(i, a, \gamma(A)).$$

Now, formally, a test t at test site s guards a requirement $\langle p, req \rangle$ from the wish list if,

$$\begin{array}{l}\text{for all calls } a \text{ to procedure } p, \\ \mathit{satisfies}(req, a, \mathit{ReachableCalls}(a', \mathit{success}(s,t))\, a),\end{array}$$

where a' is the call at site s. This formula specifies the abduction problem. We need a computable approximation for it. We might imagine using the abstract analogue of this formula to try out hypothesized tests. However, we wish to find a test that is as weak as possible. Thus, we must either try numerous tests to see whether they satisfy the requirement, or we must find some domain-dependent way to compute good tests directly. In either case, it is useful to consider ways to simplify this formula that do not depend on t.

In research that will be reported elsewhere, we have developed a method for condensing the abstract execution of the program from the test site up to the various requirement sites. The method is based on [6, 7]. Suppose the call at the test site is a. For each call c to the procedure with the requirement, the condensed execution represents the unification constraints that are imposed on the variables in a and c by the execution of a up to the invocation of c. These constraints are given by an element B of Subst_A describing a substitution over the variables in c as well as the variables in the call a at the test site. Through a single unification of a under B with a under $\mathit{success}(s,t)$, we obtain $\mathit{ReachableCalls}(a, \mathit{success}(s,t))\, c$. This reduces the general abduction problem to abduction for a single unification. This simpler problem can, in principle, be solved by using a generate-and-test approach. However, we have worked out a direct solution for two nontrivial domains, including *Prop*, for groundness analysis. These, too, will be reported elsewhere.

6 Concluding Remarks

This paper has presented a framework for multiple specialization of logic programs that incorporates run-time testing. The starting point for this work was the multiple specialization technique of Winsborough [13], which introduced the notion of a call-path automaton. The transition function for a call-path automaton specifies which procedure version should be invoked by each call in each version of each procedure. The compiler uses the call-path automaton, together with a set of entry versions derived from entry mode declarations, to determine which versions of procedures should be created and how versions should call one another. This paper extended call-path automata to include test arcs, which lead to different versions depending on the outcome of the test. In this way, tests provide additional information about procedure activations, thereby setting up new specializations. We presented an algorithm for constructing an automaton containing a given

set of tests at given test sites. This is the first technique of which we are aware that propagates the results of run-time tests across clause boundaries.

This leaves open the problem of generating suitable tests and test sites. Our goal is to derive tests from a compiler-generated "wish list" of requirements that enable useful optimizations. In addition, since testing can increase run-time overhead, we want tests to occur at outer levels of the call tree where they enable repeated inner-level optimizations. One approach is to insert tests, perhaps suitably strengthened, directly at the requirement site given on the wish list. We have given examples where such tests will be propagated by the automaton across clause boundaries, enabling repeated inner-level optimizations. More generally, one may wish to abductively derive a test at a given site that, upon success, ensures some property at the requirement site. We presented initial work on a technique for deriving such tests. Using the notion of condensing introduced in [6, 7], we discussed how to collapse the general abduction problem to abduction for a single unification. We have worked out domain-dependent solutions to this simpler problem for two nontrivial domains, including *Prop*, for deriving groundness information. In our future work, we plan to experiment with specialization heuristics based on abduction.

Acknowledgments

This research was supported in part by NSF grant CCR–8807171 and by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[2] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: Towards the global optimization of Prolog programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 192–204. IEEE, 1987.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[4] S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229, September 1988.

[5] D. DeGroot. Restricted and-parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 471–478. ICOT, 1984.

[6] D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In E. Lusk and R. Overbeek, editors, *Proc. North Americam Conf. on Logic Programming*, pages 154–165. MIT Press, 1989.

[7] D. Jacobs and A. Langen. Static analysis of logic programs for independent and-parallelism. *Journal of Logic Programming*, 1990. to appear.

[8] N. D. Jones and H. Sondergaard. A semantics-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 6, pages 124–142. E. Horwood, 1987.

[9] K. Marriott, H. Sondergaard, and N. Jones. Denotational abstract interpretation of logic programs. in preparation, 1990.

[10] C. S. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2(1):43–66, January 1985.

[11] C. S. Mellish. A semantics-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 6, pages 181–198. Ellis Horwood, 1987.

[12] R. Warren, M. Hermenegildo, and S. K. Debray. On the practicality of global flow analysis of logic progams. In *Proceedings of the Fifth Conference on Logic Progamming*, pages 684–699. MIT Press, 1988.

[13] W. Winsborough. Path-dependent reachability analysis for multiple specialization. In E. Lusk and R. Overbeek, editors, *Proc. North Americam Conf. on Logic Programming*, pages 133–153. MIT Press, 1989. (full paper to appear in *J. Logic Programming*).