MULTIPLE SPECIALIZATION USING MINIMAL-FUNCTION GRAPH SEMANTICS

WILL WINSBOROUGH

▷ Many optimizing compilers use interprocedural analysis to determine how the source program uses each of its procedures. Customarily, the compiler gives each procedure a single implementation, which is specialized according to restrictions met by all uses of the procedure. We propose a general method whereby the compiler can make the uses of each procedure implementation more uniform, enabling a greater degree of specialization. The method creates several implementations of each procedure, each specialized for a different class of use; it avoids run-time overhead by determining at compile time the appropriate procedure implementation for each call in the expanded program. The implementation suited to each call is determined by embedding in the program a deterministic finite automaton that, during execution, scans the current call path, i.e., the sequence of calls entered but not yet exited. Each automaton state has an associated class of procedure uses that includes the use made by the last call in each call path that, on input, leaves the automaton in the given state. The compiler creates one implementation for each state, using the associated class of use to specialize the implementation and the transition function to determine which implementation to invoke for each call in the expanded program. With standard automata-theory techniques, it is straightforward to merge several automaton states, in case several classes of use lead to specializations that are the same or whose differences are not substantial enough to warrant separate implementations. Thus, our method allows the compiler to perform multiple specialization where it is useful, while avoiding excessive enlargement of the generated code. We formalize the foundation of our method by constructing a general-purpose *minimal-function graph* semantics that extends and improves prior constructions of this sort. ◁

Address correspondence to Will Winsborough, Department of Computer Science, Whitmore Laboratory, The Pennsylvania State University, University Park, PA 16802.

Received May 1989; accepted February 1990.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/92/$5.00

1. INTRODUCTION

Compiler designers use program analyzers to determine when special-purpose techniques can safely be used to implement the source program. Optimizations based on interprocedural program analysis specialize the implementation of each procedure according to the procedure activations that are inferred at compile time to be reachable at run time. Traditional applications include constant propagation and common-subexpression elimination [1] and, in functional programming, strictness analysis for transforming call-by-need into call-by-value [19, 10] and liveness analysis for enabling destructive update and compile-time garbage collection [11].

In recent years investigators have proposed several optimizations of logic programs based on program analysis. Some of these optimizations are based on the modes in which each predicate can be invoked during program execution. Knowing the activation modes, the compiler can generate specialized code for unification, storage allocation, and clause indexing [3, 7, 16]. Other techniques infer when the occur check can be omitted safely from unification [21, 22], when calls are independent and so can be executed concurrently (AND-parallelism detection) [4, 8], and, as with functional languages, when destructive update can be used [18]. These applications and others motivate the study of general-purpose frameworks for the construction and verification of program analyzers [2, 13, 15].

The general-purpose compile-time technique presented in this paper seeks to improve the performance gains achievable by using any analysis-based optimization of Horn-clause logic programs. If there is similarity among the uses each procedure implementation must handle, there is more opportunity to apply optimizations to that implementation than if it must handle dissimilar uses. Our method makes the uses of each procedure implementation more uniform by creating several different implementations, each specialized for a different way the procedure is used by the program. Our method uses a notion of call-path-dependent reachability to identify for each procedure a collection of use classes that satisfies the following condition. If one specialized implementation is compiled for each identified class of use, then for each call in the expanded program there is a procedure implementation appropriate to handle that call among the compiled implementations, and the appropriate implementation can be determined at compile time. Prior frameworks either have collected just one description of the reachable uses of each procedure [13] or have not worked out a general method to create several procedure implementations and to invoke them without run-time overhead [2]. Moreover, in our framework it is straightforward to create only the implementations needed to take advantage of available opportunities for their optimization, thus avoiding excessive enlargement of the generated code.

Program analyses can be justified by using the theory of *abstract interpretation.* We construct a general abstract interpretation framework for constructing call-path-dependent reachability analyses. By providing an abstract domain with certain operations that meet certain safety requirements, a particular path-dependent analysis is constructed in our framework.

1.1. Procedure Activations

In this paper, *a logic program* is a vector of definite Horn clauses:

clauses = $\langle \texttt{clause}_1, \texttt{clause}_n \rangle$.

Definite Horn clauses have the form $h :- b_1, b_2, \ldots, b_k$, $k \geq 0$, where h and b_i are positive literals. h is called the *head*; b_1, b_2, b_k is called the *body*. Each b_i is a *call*. A clause whose head has the predicate p is called a *clause for* p. A clause whose head has the same predicate as b_i is also called a *clause for* b_i. The collection of all clauses for p in the program is called *the procedure for* p. Programs are intended to be compiled into an SLD-refutation [14] engine. The engine does a PROLOG-style search of possible derivations using the left-to-right computation rule. This operational semantics must be preserved by program optimizations.

The important characteristic of a use of a clause is the activation substitution induced by that use. (We use "activation" when referring to the input to a procedure, clause, or call, reserving "invocation" for the control event.) A *call-* or *clause-activation description* is a representation, β, of a set of substitutions over the variables in the call or clause. A *procedure-activation description* for a program **clauses** is a vector $\langle \beta_1, \ldots, \beta_n \rangle$ of substitution descriptions such that β_i is a clause-activation description for $clause_i$. So that all procedure-activation descriptions have the same type, an entry is included for each clause in the program. Clauses not in the procedure being invoked have empty entries in the corresponding procedure-activation description.

1.2. Call-Path-Dependent Analysis

Our method selects a procedure implementation to handle invocation based on the sequence of currently active procedure calls leading up to that invocation. We call that sequence a *call path*. A call path identifies a procedure invocation in the program execution. Our method constructs a collection of procedure-activation descriptions reachable along various call paths. This collection forms the set of states of a deterministic finite automaton that scans call paths. The automaton can be used to map a call invocation to a procedure-activation description (given by a state in the automaton) by running the automaton on the call path leading to that invocation. According to our method, each procedure-activation description in the collection is used to specialize a corresponding implementation. The automaton's transition function specifies how control moves from one specialized implementation to the next. At each stage of program execution, the current procedure implementation corresponds to a procedure-activation description that forms a state in the automaton. In that state, when the automaton scans the index of a clause and a call within that clause, it moves to a state given by another procedure-activation description. By compiling each call in each specialized procedure implementation so that the call invokes the procedure implementation specialized according to the procedure-activation description indicated by the automaton's transition function, we embed the automaton in the compiled program. An automaton is correct for this purpose if, for every execution of the program and every call path leading to a call invocation in that execution, when each $clause_i$ is invoked in response to the call invocation, the ensuing clause activation is in the set represented by β_i, where $\langle \beta_1, \beta_i, \ldots, \beta_n \rangle$ is the procedure-activation description given by the state of the automaton after scanning the given call path. We construct and verify such an automaton in Section 6.

1.3. Minimizing the Number of Versions

Not all procedure-activation descriptions lead to different optimizations in practice. In Section 8, we summarize a technique to eliminate redundant implementations, avoiding excessive code size. The technique is strongly related to the well-known and well-understood problem of minimization of deterministic finite automata.

1.4. Semantic Foundation

In order to determine call-path-dependent reachable activations, it is necessary to simulate the effect of executing each call on its local clause environment. This problem has been treated in [2, 13, 17]. However, we provide a new abstract interpretation framework, for two reasons.

The first reason has to do with the suitability of the framework for our purpose. The seminal framework of Mellish [17] is neither precise enough nor flexible enough. On the other hand, Bruynooghe's framework [2] is overspecified. It specifies an algorithm as the means of constructing the structures that define the abstract interpretation. While this solves problems that we do not address, we do not wish to require the reader to study those issues before gaining access to our simple proposals.

The second reason we provide a new framework is that the denotational construction we present is interesting in its own right. It is a direct application of the method of *minimal-function graph* (MFG) semantics, introduce by Jones and Mycroft [12] for applicative languages. An MFG semantics has the virtue that it constructs results for only those activations that are reachable from the entry description at hand (in contrast to the construction in [13]).

Our framework extends the work of Jones and Mycroft by using an MFG construction to provide a *core semantics* that is parametrized by a domain of substitution descriptions and two operations on that domain. Thus, analyses constructed in our abstract interpretation framework automatically receive the benefit of the MFG construction.

Other constructions for logic-programming languages have been based on minimal-function graph semantics [5, 24]. The one given here resolves several problems with [24] and is more general than the MFG semantics of Codish, Gallagher, and Shapiro [5], which, like that of Jones and Mycroft, is defined only over a concrete domain.

The MFG approach models each clause as a partial function from activation descriptions to result descriptions. The denotation of a program is a vector of such partial functions, one for each clause in the program. An analyzer constructed in our framework computes the graph of those partial functions over an abstract domain.

Part of the MFG technique is to define the function denoted by a clause only on reachable activation descriptions for that clause. Thus, analyses constructed in our framework characterize top-down aspects of the program's execution, by indicating which activations are reachable, and bottom-up aspects, by describing the results of those activations. (This is in contrast to, for example, the purely bottom-up analyses constructible in the framework of Marriott and Søndergaard [15].) In

addition to avoiding unnecessary work, this feature provides the same kind of reachability information as does a collecting semantics, such as given by Jones and Søndergaard [13]. We do not make explicit use of this top-down information here, because we seek a more subtle kind of reachability information that is sensitive to where the call occurs. But our multiple specialization method benefits from the efficiency of constructing results only for activations that are reachable.

1.5. Overview of the Abstract Interpretation Framework

In our abstract interpretation framework, a program analysis is specified by an *abstract semantics*. In traditional abstract interpretation [6], the abstract semantics is related to the standard semantics through a *collecting semantics*, which characterizes the set of execution states in which each program point can be reached. An analyzer that infers only consequences of the standard semantics must approximate from above the set of possible execution states, which is conveniently expressed and verified by using a collecting semantics. We too use a set-level intermediary, which we call the *concrete MFG semantics*, or simply the *concrete semantics*.

We follow [12, 13] in constructing a *core semantics* (see Section 4) that specifies what is common to our concrete semantics and to all analyzers constructible in our framework, but leaves undetermined the details of the representation of sets of substitutions. The concrete semantics is constructed by providing to the core semantics a concrete domain (see Section 3) consisting of sets of substitutions and operations that lift unification and substitution composition to handle sets.

One constructs an analysis in our framework by providing to the core semantics a less expressive domain of substitution descriptions, called an *abstract domain*, and operations on that domain analogous to the concrete operations. The abstract domain is chosen according to the kind of information needed for the optimization of interest, and according to the desired efficiency of the analyzer versus the desired precision of its results. Our framework requires the abstract domain to be finite. This ensures that the MFG analysis is finite and that only finitely many procedure implementations will be considered.

In Section 9, we provide sufficient conditions on the abstract domain and operations to ensure the safety of the induced analysis. These conditions relate the abstract domain and operations to the concrete domain and operations. They are based on the lattice-theoretic method of adjoined functions, as originally presented in [6] and as applied to interpretations of core semantics in, for example [12, 13]. We present theorems that use these conditions to guarantee that the abstract semantics safely approximates the concrete semantics. In Section 4.3, we verify the soundness of our concrete semantics with respect to SLD derivations as defined by Lloyd [14]. This important step is often omitted from abstract interpretation frameworks. It is because our standard semantics is an operational semantics that the intermediate stages of the computation captured in the collecting semantics are consequences of the standard semantics. A denotational semantics, such as is taken to be the standard semantics in [13], specifies only the input-output relation and does not constrain the intermediate states of computation.

Like the core semantics, the construction of the automaton for multiple specialization is parametrized by the abstract interpretation. The safety of the abstract

automaton with respect to the concrete automaton is also shown in Section 9. In Section 6 we verify the soundness of the concrete automaton with respect to SLD derivations.

1.6. Organization

The remainder of this paper is organized as follows. Section 2 presents some technical background and notation. Section 3 discusses generally the domains and operations that must be supplied to the core semantics in order to generate an analysis; Section 3 also presents the concrete interpretation. Section 4 defines the MFG core semantics and verifies the soundness of the induced concrete semantics with respect to SLD derivations. Section 5 constructs the set of call-path-dependent activations that our method constructs specialized procedure implementations for and that form the states of our automaton. Section 6 constructs the automaton's transition function and verifies the soundness of the concrete transition function with respect to SLD derivations. Section 7 presents an example illustrating the multiple specialization technique. Section 8 shows how to use finite automaton minimization techniques to minimize the number of procedure implementations created. Section 9 establishes conditions of an abstract i nterpretation (relating it to the concrete interpretation) that are sufficient to ensure the safety of the induced abstract semantics and automaton.

Portions of this research have been previously reported in [25]. In addition to providing more thorough discussion of the previously reported material, the current report includes the following new material. Section 4.3 presents a thorough verification of our concrete MFG semantics as a basis for abstract interpretation. Section 7 presents an example illustrating our multiple specialization technique. Section 8 presents an algorithm that the compiler can use to minimize the number of procedure implementations created. Section 9 includes all intermediate results needed to justify our claim that an analysis constructed in our framework can be verified by verifying relatively simple properties of the abstract domain and operations supplied to complete that analysis.

2. PRELIMINARIES

We begin by going over some ideas that are at the foundation of both simulating clause executions and collecting call-path-dependent reachable activations. Notation not explicitly introduced is borrowed from Lloyd [14].

2.1. Syntactic Objects

`Pred`, `Func`, and `V` denote countably infinite sets of predicates, function symbols, and variables, respectively. `Term` denotes the set of logical terms over `Func` and `V`. `Literal` denotes the set of literals over `Pred` and `Term`. `Clause` is the set of definite Horn clauses. For any syntactic object t, *Vars*(t) denotes the set of variables occurring in t.

A program **clauses** $\in$ `Prog` is a vector of clauses. We denote by n the number of clauses in the fixed but arbitrary program under discussion. The ith clause in

clauses is $\texttt{clause}_i$. The body of $\texttt{clause}_i$ is *body*($\texttt{clause}_i$), and the number of calls in *body*($\texttt{clause}_i$) is k_i.

2.2. Substitutions

A substitution is a function in $\texttt{V} \rightarrow \texttt{Term}$. We use ρ, θ, σ, μ, η, and ι to denote substitutions, often with subscripts, primes, or hats. Substitution application (written postfix) extends a substitution to terms in the natural way: if t is a term, literal, or clause, $t\rho$ is derived from t by simultaneously replacing each variable occurrence in t by that variable's image under ρ.

We say a substitution *binds* a variable that it does not map to itself. If all the variables bound by some substitution ρ are in the set S, we say ρ is a *substitution over S*. (Our substitutions are total and so have no formal function domain other than V. However, formal function domains for substitutions are useful in certain abstract domains and could easily be incorporated into our framework.)

Composition of substitutions is defined by

$$\theta \circ \sigma = \lambda v{:}\texttt{V}.\, v\theta\sigma,$$

so that $t(\theta \circ \sigma) = t\theta\sigma$ for all t. The restriction of a substitution ρ to a set of variables S is defined by

$$\rho v_S = \lambda v{:}\texttt{V}.\, \text{if } v \in S \text{ then } v\rho \text{ else } v.$$

We extend this notation by writing $\rho|_t$ for $\rho|_{Vars(t)}$ where t is a term, literal, or clause. The composition operator has precedence over the restriction operator, so $\theta \circ \sigma|_S = (\theta \circ \sigma)|_S$.

The identity substitution, ϵ binds no variables. A substitution is called a *renaming* if it is invertible. We use η and ι for renamings. Two substitutions ρ and θ are *renaming-equivalent* if $\rho = \theta \circ \eta$ for some renaming η.

A most general unifier of terms t and s, when one exists, is a substitution θ *over the variables* of t and s such that (1) $t\theta = s\theta$ and (2) for all ρ, $t\rho = s\rho \Rightarrow \rho = \theta \circ \sigma$ for some σ. When it exists, such a θ is uniquely determined up to renaming equivalence. We assume that, for each t and s, *mgu*(t, s) is some fixed but arbitrary most general unifier of t and s, when one exists. Otherwise, *mgu*(t, s) is *fail*.

We need an operation to "standardize apart" the variables in the call from those in the clause used to solve the call. *StandApart* produces a renaming η such that for syntactic objects t_1 and t_2,

$$\eta = StandApart(t_1, t_2) \quad \Rightarrow \quad Vars(t_1\eta) \cap Vars(t_2) = \{\ \},$$

where η is a substitution over the variables of t_1.

2.3. Canonical Substitutions

An analyzer infers properties of the substitutions that form the local environment during execution at various points in the program. Properties of interest do not distinguish substitutions that are renaming-equivalent. For the purpose of comparison, we represent each renaming-equivalence class by a canonical representative [17]. Testing renaming equivalence is thus reduced to testing equality. We denote the set of canonical substitutions by *CanonicalSubs*. We do not require substitution

restriction, *StandApart*, or *mgu* to take on values in *CanonicalSubs*, but explicitly normalize such results. We call the operation that yields the canonical representative of a substitution's renaming-equivalence class *norm*.

2.4. *Substitution Descriptions, Domains, and Program Denotations*

An MFG semantics constructs as the denotation of a program a collection of functions from activations to results. A reasonable denotation can be given by associating activation-result maps with literals, literal instances, clauses, or procedures. For our purpose, it is convenient to characterize a program with a function for each clause. We call a vector containing one such function for each clause a *program description.*

Our concrete MFG semantics uses sets of canonical substitutions to describe precisely the reachable clause activations and their results. Each abstract semantics uses an approximate domain of substitution descriptions for this purpose. We refer to an arbitrary domain of substitution descriptions by the name Asub. To refer specifically to the concrete domain, we use $\mathsf{Asub}_{\mathrm{conc}}$. A fixed but arbitrary abstract domain is denoted $\mathsf{Asub}_{\mathrm{abs}}$. The elements of Asub represent sets of canonical substitutions. (When Asub is interpreted to be $\mathsf{Asub}_{\mathrm{conc}}$, these sets are represented by themselves.)

A domain is a pointed cpo, i.e., a partially ordered set with a least element " $\perp$ ", each of whose subsets has an upper bound in the set. The less-than relation is denoted $\sqsubseteq$. The upper-bound operation is denoted $\sqcup$. Formally, these operators are subscripted with their domains, though we omit the subscripts when no confusion results.

The most natural denotation of each clause in a logic program is a function from individual substitutions to sets of substitutions: each instance of a call has a set of solutions. To lift this denotation to handle sets of computations, the denotation of a clause becomes a map from sets of substitutions to sets of substitutions by taking the union of results obtained from each activation. However, if the input sets are all singletons, we get back essentially the standard semantics. In our core semantics we follow Jones and Søndergaard in using a basis of the domain Asub as the input domain [13]. We call the basis of Asub Csub. To say that Csub is a basis of Asub is to say that

$$\forall \beta \in \mathsf{Asub}\,.\,\beta = \bigsqcup \{\tau \in \mathsf{Csub} \mid \tau \sqsubseteq \beta\}\,.$$

In the concrete case, $\mathsf{Csub}_{\mathrm{conc}}$ will be the set of singleton substitutions, plus { }. By using this basis, Jones and Søndergaard embed the standard semantics in their collecting semantics, which helps to justify of their collecting semantics.

In contrast with Jones and Søndergaard, we relate our concrete semantics directly to SLD derivations. This approach makes the foundation of our concrete MFG semantics more firm, since the operational semantics of SLD derivations with the left-to-right computation rule determines the intermediate states of program execution, which any abstract interpretation core semantics must capture. In contrast, a denotational semantics determines only the program's input-output relation. Further, independent of abstract interpretation, the relationship between the SLD operational semantics and our concrete MFG semantics demonstrates

that our concrete MFG semantics captures the reachability aspects of pure PROLOG, which are missed by denotational semantics such as the one in [13]. This paper demonstrates only that our concrete MFG semantics reflects every activation reachable in pure PROLOG, as this is the inclusion needed for abstract interpretation. We claim without proof that the converse also holds.

3. INTERPRETATIONS

The core semantics constructed in Section 4 is parametrized by an interpretation. By providing a particular interpretation, a corresponding semantics is induced, either the concrete semantics or an abstract semantics. An interpretation has two parts: a domain, Asub, with a basis, Csub, and two operations, Call and Extend:

$$\mathsf{Call}: \texttt{Literal} \times \mathsf{Asub} \times \texttt{Clause} \rightarrow \mathsf{Asub},$$

$$\mathsf{Extend}: \texttt{Literal} \times \texttt{Clause} \times \mathsf{Csub} \times \texttt{Clause} \times \mathsf{Asub} \rightarrow \mathsf{Asub}.$$

Call is used to obtain an activation description for a clause from a call and an activation description for that call. Extend is used to update the substitution description that characterizes the clause environment during simulation of clause execution when a call in the clause is executed. Extend is used to generate the clause-environment description corresponding to all the call solutions that come from resolving a call with a given clause. Extend takes as arguments the call whose execution is to be simulated, the clause in which that call occurs, the activation description under which the call is invoked, the clause used to solve the call, and the substitution description that characterizes the result of executing the second clause according to the given activation description. The operations given by a particular interpretation are distinguished by using subscripts. For example, $\mathsf{Asub}_{\mathrm{conc}}$ is the concrete domain of substitution descriptions, and $\mathsf{Call}_{\mathrm{abs}}$ is the Call operation of the abstract domain.

The Concrete Interpretation

We now define the *concrete interpretation.* This interpretation induces the *concrete semantics* when supplied to the core semantics constructed in Section 4. In Section 9 we give safety conditions on abstract interpretations, which relate an abstract interpretation to the concrete interpretation and ensure that the abstract semantics induced by the abstract interpretation is a safe approximation of the concrete semantics.

Symbols.

σ, ρ:	*CanonicalSubs*
η, μ:	*Subs*
τ, κ:	Csub
β, δ:	Asub

Definition 3.1. The *concrete interpretation* is given by the following:

$$\mathsf{Asub}_{\mathrm{conc}} = \mathscr{P}(\mathit{CanonicalSubs}),$$

$$\mathsf{Csub}_{\mathrm{conc}} = \{\{\sigma\}, \sigma \in \mathit{CanonicalSubs}\} \cup \{\{\ \}\},$$

$$\sqsubseteq_{\mathrm{conc}} \equiv \subseteq \quad \text{(subset)},$$

$$\sqcup_{\mathrm{conc}} = \cup \quad \text{(set union)};$$

$$\mathsf{Call}_{\mathrm{conc}} : \mathtt{Literal} \times \mathsf{Asub}_{\mathrm{conc}} \times \mathtt{Clause} \to \mathsf{Asub}_{\mathrm{conc}},$$

$$\mathsf{Call}_{\mathrm{conc}}(\mathsf{c}, \beta, [\![\mathtt{h} :\!- \mathtt{b}_1, \ldots, \mathtt{b}_k]\!]) = \left\{ \begin{array}{l} \mathit{norm}(\ \mu|_{\mathtt{h} :\!- \mathtt{b}_1, \ldots, \mathtt{b}_k}) \\ |\ \mu = \mathit{mgu}(\mathtt{h}, \mathsf{c}\rho\eta),\ \rho \in \beta,\ \eta = \mathit{StandApart}(\mathsf{c}\rho, \mathtt{h} :\!- \mathtt{b}_1, \ldots, \mathtt{b}_k) \end{array} \right\};$$

$$\mathsf{Extend}_{\mathrm{conc}} : \mathtt{Literal} \times \mathtt{Clause} \times \mathsf{Csub}_{\mathrm{conc}} \times \mathtt{Clause} \times \mathsf{Asub}_{\mathrm{conc}} \to \mathsf{Asub}_{\mathrm{conc}},$$

$$\mathsf{Extend}_{\mathrm{conc}}(\mathsf{c}, \mathtt{clause}, \tau, [\![\mathtt{h} :\!- \mathtt{b}_1, \ldots, \mathtt{b}_k]\!], \beta) = \left\{ \begin{array}{l} \mathit{norm}(\ \rho \circ \mu|_{\mathtt{clause}}) \\ |\ \mu = \mathit{mgu}(\mathsf{c}\rho, \mathtt{h}\sigma\eta),\ \rho \in \tau,\ \sigma \in \beta, \\ \eta = \mathit{StandApart}((\mathtt{h} :\!- \mathtt{b}_1, \ldots, \mathtt{b}_k)\sigma, \mathtt{clause}\rho) \end{array} \right\}.$$

4. MINIMAL-FUNCTION GRAPH SEMANTICS

In this section we define the MFG core semantics and verify the soundness of the MFG concrete semantics with respect to SLD-refutation using the left-to-right (PROLOG) computation rule. First we formalize the idea of an entry description.

4.1. Entry Descriptions

We require a characterization of the possible top-level entry calls that the compiled program must be prepared to handle. We give a general mechanism that can be used in many ways, according to the application.

Definition 4.1. **Entries** $= \mathscr{P}(\mathtt{Literal} \times \mathsf{Asub})$.

Ent $\in$ **Entries** is a set of pairs consisting of a call and a substitution description for that call. Each pair specifies an activation description for the call's procedure. Our method creates one procedure implementation for each of these (except when implementations that allow identical specializations are collapsed, as described in Section 8). If at most one pair is allowed for each procedure, then the corresponding procedure implementation can be expected to handle all entry-level calls of that procedure. If more than one pair is allowed for each procedure, some application-dependent mechanism must be provided to inspect entry-level calls to find suitable implementations. In many applications it may be desirable to verify that entry-level calls conform to the entry description, even if there is only one entry-level implementation for each procedure.

4.2. MFG Construction

This section constructs the semantic function, **MFG**. The MFG semantics is given by a tuple of partial functions, one per clause. To represent partial functions, the MFG technique introduces a new bottom element $\perp\!\!\perp$ into the domain of result descriptions [12]. The function has the value $\perp\!\!\perp$ on activations that are not reachable and hence undefined. We denote its addition to the domain of result descriptions by $\mathsf{Asub}_{\amalg}$. The new bottom is less than ($\sqsubseteq_{\mathsf{Asub}_{\amalg}}$) all other elements in the domain, including $\perp$. The operator # (defined below) denotes a restriction operation that uses this repres entation of partial functions.

In the following definitions we use angle brackets to construct a tuple from a sequence. We denote selection using subscripting or $\downarrow$: $\langle t_1,\ldots,t_k\rangle\downarrow_i = t_i$. We denote concatenation of sequences by concatenating their expressions, sometimes with commas. $\sqcup_{\mathsf{Asub}^n}$ denotes pointwise application of $\sqcup_{\mathsf{Asub}}$. Recall that the program is denoted by **clauses**, and the ith clause of **clauses** is denoted by $\mathtt{clause}_i$. A discussion of the intuition behind the constructions follows the definition.

Symbols.

ψ, φ: $\mathbf{Den} = (\mathsf{Csub} \to \mathsf{Asub}_{\amalg})^n$. Program descriptions.

demanded, init: Asub^n. Reachable activation descriptions for each clause.

ξ: $(\mathsf{Asub}^n)^*$. Procedure activation descriptions ensuing from each call in a given clause body.

Domain-Dependent Operations.

$$\mathbf{init}: \mathtt{Prog} \times \mathbf{Entries} \to \mathscr{P}(\mathsf{Asub}^n),$$

$$\mathbf{init}(\mathbf{clauses}, \mathbf{Ent}) = \{\langle \beta_1,\ldots,\beta_n\rangle \mid \exists\langle \mathsf{c}, \delta\rangle \in \mathbf{Ent}.\forall i.1 \le i \le n \Rightarrow \beta_i = \mathsf{Call}(\mathsf{c}, \delta, \mathtt{clause}_i)\};$$

$$\#: (\mathsf{Csub} \to \mathsf{Asub}) \times \mathsf{Asub} \to (\mathsf{Csub} \to \mathsf{Asub}_{\amalg}),$$

$$f \# \beta = \lambda\tau : \mathsf{Csub}.\ \text{if } \tau \sqsubseteq \beta \text{ then } f(\tau) \text{ else } \amalg.$$

Definition 4.2. The MFG semantics is defined as follows:

$$\mathbf{MFG}: \mathtt{Prog} \times \mathbf{Entries} \to \mathbf{Den},$$

$$\mathbf{MFG}(\mathbf{clauses}, \mathbf{Ent}) = \mathit{fix}(\mathbf{SimProg}(\mathbf{clauses}, \sqcup_{\mathsf{Asub}^n}\mathbf{init}(\mathbf{clauses}, \mathbf{Ent}))),$$

$$\mathbf{SimProg}: \mathtt{Prog} \times \mathsf{Asub}^n \to \mathbf{Den} \to \mathbf{Den},$$

$$\mathbf{SimProg}(\mathbf{clauses}, \mathbf{init})(\psi) = \langle \varphi_1,\ldots,\varphi_n\rangle,$$

where, for each i, $1 \le i \le n$,

$$\varphi_i = (\lambda\tau : \mathsf{Csub}.(\mathbf{SimBody}(\mathbf{clauses}, \mathit{body}(\mathtt{clause}_i), \mathtt{clause}_i, \psi, \tau))\downarrow 1) \# (\mathbf{demanded}_i \sqcup \mathbf{init}_i)$$

and

$$\textbf{demanded} = \sqcup_{\mathsf{Asub}^n}\left\{\left\langle\left(\textbf{SimBody}(\textbf{clauses}, \mathit{body}(\texttt{clause}_i), \texttt{clause}_i, \psi, \delta)\right)\right.\right.$$
$$\downarrow_2\rangle\downarrow_j \Big| 1 \le i \le n \;\&\; 1 \le j \le k_i \;\&\; \delta \in \mathsf{Csub}$$
$$\left.\left.\;\&\; \bot \sqsubseteq_{\mathsf{Asub}_\sqcup} \psi_i(\delta)\right\}\right.,$$

$$\textbf{SimBody}: \texttt{Prog}\times\texttt{Literal}^*\times\texttt{Clause}\times\textbf{Den}\times\mathsf{Asub}\to(\mathsf{Asub}\times(\mathsf{Asub}^n)^*),$$

$$\textbf{SimBody}(\textbf{clauses}, [\![\ ]\!], \texttt{currentclause}, \psi, \delta) = \langle \delta, \epsilon \rangle,$$

where ϵ denotes the empty sequence,

$$\textbf{SimBody}(\textbf{clauses}, [\![\texttt{b}_1, \ldots, \texttt{b}_j]\!], \texttt{currentclause}, \psi, \delta)$$
$$= \text{let } \langle \beta, \xi \rangle = \textbf{SimBody}(\textbf{clauses}, [\![\texttt{b}_1, \ldots, \texttt{b}_{j-1}]\!],$$
$$\texttt{currentclause}, \psi, \delta) \text{in}$$
$$\text{if } \beta = \bot \text{ then } \langle \bot, \xi \rangle \text{ else } \langle \beta', \xi\langle \delta_1, \ldots, \delta_n \rangle\rangle,$$

where

$$\beta' = \sqcup_{\mathsf{Asub}}\{\mathsf{Extend}(\texttt{b}_j, \texttt{currentclause}, \kappa, \texttt{clause}_i, \psi_i(\tau)) \mid 1 \le i \le n,$$
$$\kappa \in \mathsf{Csub},\ \kappa \sqsubseteq \beta,\ \tau \in \mathsf{Csub}, \tau \sqsubseteq \mathsf{Call}(\texttt{b}_j, \kappa, \texttt{clause}_i)\ \},$$

and for each i, $1 \le i \le n$, $\delta_i = \mathsf{Call}(\texttt{b}_j, \beta, \texttt{clause}_i)$.

SimBody simulates execution of a clause body, $\texttt{b}_1, \ldots, \texttt{b}_k$, on a clause-activation description δ, by using an input program description ψ. It constructs a description β of the result, plus a sequence ξ that traces the clause's execution by recording the procedure-activation description generated by each call in the clause. **SimProg** joins together all procedure activations in all traces ξ resulting from all simulations over demanded clause activations, δ [i.e., δ for which $\bot \sqsubseteq_{\mathsf{Asub}_\sqcup} \psi_i(\delta)$ or, equivalently, $\psi_i(\delta) \neq \amalg$]. The result of that join, **demanded**, together with the entry procedure activations, **init**, defines the non-$\bot\bot$ portion of the new program description generated by **SimProg**. This partia l function is constructed by using the restriction operator #, borrowed from [12]. The program analyzer can compute ψ_i in (the abstract version of) **SimProg** by applying **SimBody** to each clause-activation description τ, $\tau \sqsubseteq_{\mathsf{Asub}}$ **demanded**$_i \sqcup$ **init**$_i$, and record ing the input-output graph on that portion of the domain.

Assuming `Call` and `Extend` are monotonic and continuous, the constructed functions are also monotonic and continuous. The use of $\sqcup$ in the construction of β' in the definition of **SimBody** is necessary for continuity.

When computing an abstract instance of this semantics, techniques such as those in [20] should be employed to minimize redundant calculations. Note that the trace portion of **SimBody** provides precisely the information needed to determine which clause-activation descriptions need to be reevaluated when a result in the program description is updated.

We typically elide the arguments to **MFG**, as the arbitrary program and entry in question are fixed for the discussion. When we specialize the objects constructed in

Definition 4.2 by providing a particular interpretation, the names of the constructed objects acquire subscripts to indicate whether the interpretation is the concrete interpretation or an abstract interpretation. For example, **SimBody** becomes $\mathbf{SimBody}_{\text{conc}}$ or $\mathbf{SimBody}_{\text{abs}}$.

4.3. Soundness of MFG Concrete Semantics

Our correctness results are expressed in terms of SLD derivations that use the left-to-right computation rule, as in PROLOG execution (see Lloyd [14]). Such a derivation is shown in Figure 1. We deviate from Lloyd in our expression of resolvents by keeping the substitution separate from the goal list (separated in the figure by a semicolon). We express a derivation as a sequence of such pairs, identifying only implicitly the clauses used in each step.

Because our concrete MFG semantics operates on substitutions over the variables in the program, we must represent explicitly the variable renaming used to standardize clauses apart from resolvents. This is the source of the hats in Figure 1. For example,

$$(\texttt{clause}_i)\iota = \widehat{\texttt{clause}_i} = \hat{\texttt{h}} :\!- \hat{\texttt{b}}_1, \ldots, \hat{\texttt{b}}_k,$$

where ι is the renaming used to standardize apart this use of $\texttt{clause}_i$.

In Figure 1, the third resolvent displayed shows the invocation of call $\texttt{b}_j$ in an occurrence of $\texttt{clause}_i$. The fifth shows the invocation of the j'th call in the clause used to solve $\texttt{b}_j$. The sixth resolvent illustrates how we compress the expression of the environment using the convention $\theta_j = \theta_{j-1} \circ \theta' \circ \theta'_{k'}$, where $\theta' = mgu(\hat{\texttt{b}}_j \theta\theta_{j-1}, \hat{\texttt{h}}')$ and $\theta'_{k'}$ results from solving $\hat{\texttt{b}}'_1, \ldots, \hat{\texttt{b}}'_{k'}$ under $\theta \circ \theta_{j-1} \circ \theta'$.

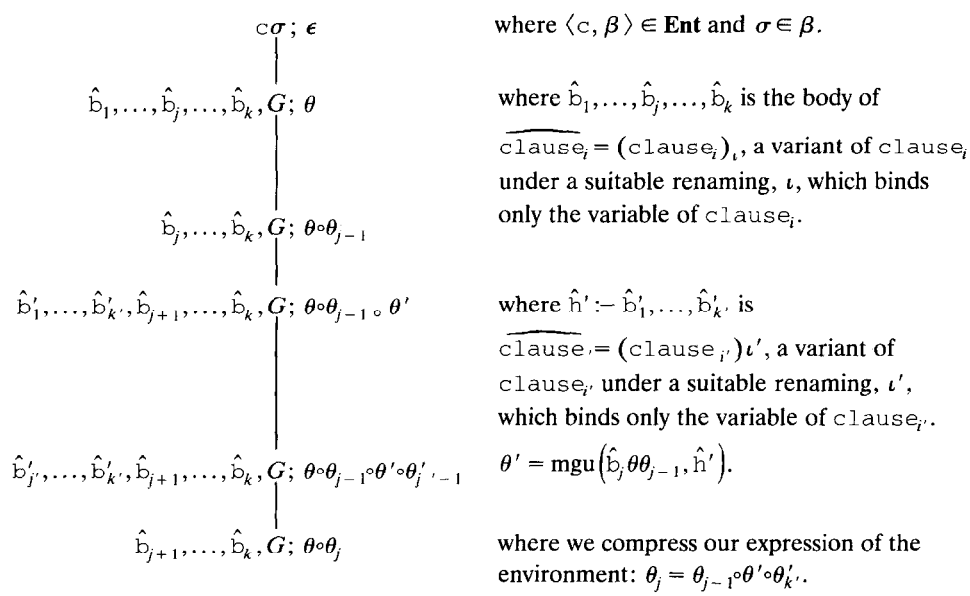

FIGURE 1. A derivation from an entry call. The naming conventions used in this figure are used throughout the article. The subderivation from the third displayed resolvent to the fifth corresponds to a call path of length 1. (See Definition 6.1).

Subderivations of the form shown in Figure 1 from the third to the fifth resolvent are the subject of several lemmas in this section. The statements and proofs of these lemmas use the names and relationships illustrated in Figure 1. For example, when we write θ' and $\hat{\mathtt{b}}_{j'}$ in the sequel, we always intend that $\theta' = mgu(\hat{\mathtt{b}}_j\theta\theta_{j-1}, \hat{\mathtt{h}}')$ and that $\hat{\mathtt{b}}_{j'}$ is a call in $(\mathtt{clause}_{i'})_{\iota'}$, where ι' is a renaming that standardizes apart according to the definition of SLD derivation [14].

As motivated in Section 2.4, we demonstrate that the concrete MFG semantics of a program records the result of each clause activation reachable from an entry call. This is the inclusion needed for abstract interpretation. We claim without proof that the converse also holds.

We begin by proving three lemmas that together can be applied inductively to show that the MFG semantics constructs correct results for all clause activations reachable in the SLD-refutation semantics with PROLOG's computation rule. The first of these, Lemma 4.3, says that the MFG semantics correctly indicates reachable call activations. That is, if after executing the first $j-1$ calls of $\mathtt{clause}_i$ **SimBody** has the current substitution environment right, then for each i' **SimBody** will correctly record the activation that arises when $\mathtt{clause}_{i'}$ is used to solve the jth call of $\mathtt{clause}_i$.

Lemma 4.3. Using the naming conventions established in Figure 1, *every SLD derivation using the left-to-right computation rule satisfies*

$$
\begin{aligned}
&norm\left(\iota\circ\theta\circ\theta_{j-1}|_{\mathtt{clause}_i}\right)\\
&\quad\in \mathbf{SimBody}_{\mathrm{conc}}\left(\mathbf{clauses}, [\![\mathtt{b}_1,\ldots,\mathtt{b}_{j-1}]\!], \mathtt{clause}_i,\right.\\
&\qquad\qquad \left.\mathbf{MFG}_{\mathrm{conc}}, \{norm(\iota\circ\theta|_{\mathtt{clause}_i})\}\right)\downarrow_1\\
&\Rightarrow\quad norm\left(\iota'\circ\theta\circ\theta_{j-1}\circ\theta'|_{\mathtt{clause}_{i'}}\right)\\
&\quad\in\left\langle \mathbf{SimBody}_{\mathrm{conc}}(\mathbf{clauses}, body(\mathtt{clause}_i), \mathtt{clause}_i, \mathbf{MFG}_{\mathrm{conc}},\right.\\
&\qquad\qquad \left.\{norm(\iota\circ\theta|_{\mathtt{clause}_i})\}\right)\downarrow_2\Big\rangle\downarrow_j\downarrow_{i'}.
\end{aligned}
$$

PROOF. Referring to the definition of $\mathbf{SimBody}_{\mathrm{conc}}$, it is sufficient to show

$$
norm\left(\iota'\circ\theta\circ\theta_{j-1}\circ\theta'|_{\mathtt{clause}_{i'}}\right)\in \mathbf{Call}_{\mathrm{conc}}(\mathtt{b}_j, \beta, \mathtt{clause}_{i'}), \tag{1}
$$

where

$$
\begin{aligned}
\beta = \mathbf{SimBody}_{\mathrm{conc}}&\left(\mathbf{clauses}, [\![\mathtt{b}_1,\ldots,\mathtt{b}_{j-1}]\!], \mathtt{clause}_i, \mathbf{MFG}_{\mathrm{conc}},\right.\\
&\left.\{norm(\iota\circ\theta|_{\mathtt{clause}_i})\}\right)\downarrow_1.
\end{aligned}
$$

Since

$$
\mathtt{clause}_{i'}\iota'\theta\theta_{j-1}\theta' = \mathtt{clause}_{i'}\iota'\theta',
$$

by properties of derivations and most general unifiers, the requirement (1) reduces to

$$
norm(\iota'\circ\theta'|_{\mathtt{clause}_{i'}})\in \mathbf{Call}_{\mathrm{conc}}(\mathtt{b}_j, \beta, \mathtt{clause}_{i'}). \tag{2}
$$

Using the hypothesis of the lemma, namely,

$$norm(\iota \circ \theta \circ \theta_{j-1}|_{\texttt{clause}_i}) \in \beta,$$

in the definition of $\textbf{Call}_{\text{conc}}$, we have

$$norm(\mu|_{\texttt{clause}_{i'}}) \in \textbf{Call}_{\text{conc}}(\texttt{b}_j, \beta, \texttt{clause}_i), \tag{3}$$

where

$$\mu = mgu(\texttt{h}', \texttt{b}_j \rho\eta),$$

$$\rho = norm(\iota \circ \theta \circ \theta_{j-1}|_{\texttt{clause}_i}),$$

$$\eta = StandApart(\texttt{b}_j \rho, \texttt{clause}_{i'}).$$

Simplifying μ, we have

$$\begin{aligned}
\mu|_{\texttt{clause}_{i'}} &= mgu(\texttt{h}', \texttt{b}_j \rho\eta)|_{\texttt{clause}_{i'}} \\
&= mgu(\texttt{h}', \texttt{b}_j \iota\theta\theta_{j-1}\hat{\eta})|_{\texttt{clause}_{i'}} \\
&\quad [\text{where } \hat{\eta} = \eta' \circ \eta,\ \eta' \text{ being the renaming that normalizes } \rho: \\
&\quad\ \text{i.e., } \iota \circ \theta \circ \theta_{j-1}|_{\texttt{clause}_i} \circ \eta' = norm(\iota \circ \theta \circ \theta_{j-1}|_{\texttt{clause}_i})] \\
&= \iota' \circ \left(mgu(\hat{\texttt{h}}', \hat{\texttt{b}}_j \theta\theta_{j-1}\hat{\eta})|_{\overline{\texttt{clause}_{i'}}}\right)|_{\texttt{clause}_{i'}} \\
&\quad [\text{since } \hat{\texttt{h}} = h'\iota' \ \&\ \hat{\texttt{b}}_j = \hat{\texttt{b}}_j \iota] \\
&= \iota' \circ \left(mgu(\hat{\texttt{h}}', \hat{\texttt{b}}_j \theta\theta_{j-1})|_{\overline{\texttt{clause}_{i'}}}\right)|_{\texttt{clause}_{i'}} \\
&\quad [\text{since } \iota' \text{ was chosen to standardize } \texttt{clause}_{i'} \text{ apart,} \\
&\quad\ \text{and } \hat{\eta}|_{\overline{\texttt{clause}_{i'}}} \text{ is the identity}] \\
&= \iota' \circ (\theta'|_{\texttt{clause}_{i'}})|_{\texttt{clause}_{i'}} \\
&= \iota' \circ \theta'|_{\texttt{clause}_{i'}}.
\end{aligned}$$

Thus, (2) follows from (3). □

Lemma 4.4 says that if $\textbf{MFG}_{\text{conc}}$ considers $\texttt{clause}_i$ to be reachable (non-$\amalg$) under some activation ($norm(\iota \circ \theta|_{\texttt{clause}_i})$) and if $\textbf{SimBody}_{\text{conc}}$ correctly simulates $\texttt{clause}_i$'s body up through the $j-1$th call $(\texttt{b}_1, \texttt{b}_{j-1})$, then when $\texttt{b}_j$ is solved using $\texttt{clause}_{i'}$, $\textbf{MFG}_{\text{conc}}$ will consider $\texttt{clause}_{i'}$ to be reachable under the resulting activation.

Lemma 4.4. Using the naming conventions established in Figure 1, *every SLD derivation using the left-to-right computation rule satisfies*

$$\textbf{MFG}_{\text{conc}} \downarrow_i (\{norm(\iota \circ \theta|_{\texttt{clause}_i})\}) \neq \amalg \tag{4}$$

and

$$norm\left(\iota\circ\theta\circ\theta_{j-1}|_{\texttt{clause}_i}\right)$$
$$\in \mathbf{SimBody}_{conc}\left(\mathbf{clauses}, [\![\texttt{b}_1,\ldots,\texttt{b}_{j-1}]\!].\texttt{clause}_i, \mathbf{MFG}_{conc},\right.$$
$$\left.\{norm(\iota\circ\theta|_{\texttt{clause}_i})\}\right)\downarrow_1$$
$$\Rightarrow \quad \mathbf{MFG}_{conc}\downarrow_{i'}\left(\left\{norm\left(\iota'\circ\theta\circ\theta_{j-1}\circ\theta'|_{\texttt{clause}_{i'}}\right)\right\}\right)\neq \amalg, \tag{5}$$

where $\theta' = mgu(\hat{h}', \hat{b}_j\theta\theta_{j-1})$.

PROOF. By definition of $\mathbf{MFG}_{conc}$ and $\mathbf{SimProg}_{conc}$,

$$\mathbf{MFG}_{conc}\downarrow_{i'} = \lambda\tau : \mathsf{Csub}_{conc} .$$
$$\left(\mathbf{SimBody}_{conc}(\mathbf{clauses}, body(\texttt{clause}_{i'}), \texttt{clause}_i, \mathbf{MFG}_{conc}, \tau)\right)\downarrow_1$$
$$\#\,\mathbf{demanded}_{i'} \sqcup \mathbf{init}_{i'}.$$

So it suffices to show

$$norm\left(\iota'\circ\theta\circ\theta_{j-1}\circ\theta'|_{\texttt{clause}_{i'}}\right)\in \mathbf{demanded}_{i'}.$$

Employing the first assumption (4), the definition of **demanded** in **SimProg**, and the fact that $\{norm(\iota\circ\theta|_{\texttt{clause}_i})\}\in \mathsf{Csub}_{conc}$, it suffices to show

$$norm\left(\iota'\circ\theta\circ\theta_{j-1}\circ\theta'|_{\texttt{clause}_{i'}}\right)$$
$$\in\left\langle \mathbf{SimBody}_{conc}(\mathbf{clauses}, body(\texttt{clause}_i), \texttt{clause}_i, \mathbf{MFG}_{conc},\right.$$
$$\left.\{norm(\iota\circ\theta|_{\texttt{clause}_i})\}\right)\downarrow_2\Big\rangle\downarrow_j\downarrow_{i'},$$

which follows directly from Lemma 4.3, using the second assumption (5). □

Lemma 4.5 says that if $\mathbf{MFG}_{conc}$ considers $\texttt{clause}_i$ to be reachable under some activation, then $\mathbf{SimBody}_{conc}$ will correctly simulate $\texttt{clause}_i$'s body under that activation.

Lemma 4.5. Using the naming conventions established in Figure 1, *every SLD derivation using the left-to-right computation rule satisfies*

$$\mathbf{MFG}_{conc}\downarrow_i(norm(\iota\circ\theta|_{\texttt{clause}_i}))\neq \amalg$$
$$\Rightarrow \quad norm\left(\iota\circ\theta\circ\theta_j|_{\texttt{clause}_i}\right)$$
$$\in \mathbf{SimBody}_{conc}\left(\mathbf{clauses}, [\![\texttt{b}_1,\ldots,\texttt{b}_j]\!], \texttt{clause}_i, \mathbf{MFG}_{conc},\right.$$
$$\left.\{norm(\iota\circ\theta|_{\texttt{clause}_i})\}\right)\downarrow_1,$$

for all j, $0\le j\le k_i$.

PROOF. The proof uses induction on the length of the subderivation from $\hat{\texttt{b}}_1,\ldots,\hat{\texttt{b}}_j,\ldots,\hat{\texttt{b}}_k$, $G;\theta$ to $\hat{\texttt{b}}_{j+1},\ldots,\hat{\texttt{b}}_k$, $G;\theta\circ\theta_j$. The induction hypothesis says that the lemma holds for derivations of length less than or equal to m.

Basis: When the length of the subderivation is one, using the lemma's antecedent, we must show

$$norm(\iota \circ \theta|_{\texttt{clause}_i})$$
$$\in \mathbf{SimBody}_{\text{conc}}(\mathbf{clauses}, [\![b_1, \ldots, b_j]\!], \texttt{clause}_i, \mathbf{MFG}_{\text{conc}}, \{norm(\iota \circ \theta|_{\texttt{clause}_i})\}) \downarrow_1,$$

which follows trivially from the definition of **SimBody**.

Step: Assume the lemma holds for derivations of length m or less. Consider an arbitrary derivation of length $m+1$. We assume the lemma's antecedent and derive its consequent. The subderivation from $\hat{b}_1, \ldots, \hat{b}_k, G; \theta$ to $\hat{b}_j, \ldots, \hat{b}_k, G; \theta \circ \theta_{j-1}$ has length less than or equal to m. So the induction hypothesis applies, yielding

$$norm(\iota \circ \theta \circ \theta_{j-1}|_{\texttt{clause}_i})$$
$$\in \mathbf{SimBody}_{\text{conc}}(\mathbf{clauses}, [\![b_1, \ldots, b_{j-1}]\!], \texttt{clause}_i, \mathbf{MFG}_{\text{conc}}, \{norm(\iota \circ \theta|_{\texttt{clause}_i})\}) \downarrow_1. \quad (6)$$

Now, using (6) and the lemma's antecedent, we apply Lemma 4.4, deriving

$$\mathbf{MFG}_{\text{conc}} \downarrow_{i'} (\{norm(\iota' \circ \theta \circ \theta_{j-1} \circ \theta'|_{\texttt{clause}_{i'}})\}) \neq \amalg. \quad (7)$$

Let

$$\tau = norm(\iota' \circ \theta \circ \theta_{j-1} \circ \theta'|_{\texttt{clause}_{i'}}).$$

The subderivation from $\hat{b}'_1, \ldots, \hat{b}'_{k'}, \hat{b}_{j+1}, \ldots, \hat{b}_k, G; \theta \circ \theta_{j-1} \circ \theta'$ to $\hat{b}_{j+1}, \ldots, \hat{b}_k, G; \theta \circ \theta_j$ has length less than or equal to m. So the induction hypothesis applies and, combined with (7), yields

$$norm(\iota \circ \theta \circ \theta_{j-1} \circ \theta' \circ \theta'_{k'}|_{\texttt{clause}_{i'}})$$
$$\in \mathbf{SimBody}_{\text{conc}}(\mathbf{clauses}, [\![b'_1, \ldots, b'_{k'}]\!], \texttt{clause}_{i'}, \mathbf{MFG}_{\text{conc}}, \{\tau\}) \downarrow_1. \quad (8)$$

From (8) we derive

$$norm(\iota' \circ \theta \circ \theta_j|_{\texttt{clause}_{i'}}) \in \mathbf{MFG}_{\text{conc}} \downarrow_{i'} (\{\tau\}), \quad (9)$$

using $\theta_j = \theta_{j-1} \circ \theta' \circ \theta'_{k'}$. Now, by construction,

$$\mathbf{MFG}_{\text{conc}} \downarrow_{i'} (\{\tau\})$$
$$= \mathbf{SimProg}(\mathbf{clauses}, \mathbf{init})(\mathbf{MFG}_{\text{conc}} \downarrow_{i'} (\{\tau\}))$$
(by definition of **MFG**)
$$= (\lambda \kappa : \mathsf{Csub} . (\mathbf{SimBody}_{\text{conc}}(\mathbf{clauses}, body(\texttt{clause}_{i'}), \texttt{clause}_{i'}, \mathbf{MFG}_{\text{conc}}, \kappa)) \downarrow_1)$$
$$\#(\mathbf{demanded}_i \sqcup \mathbf{init}_i)(\{\tau\}),$$
(by definition of **SimProg**)
$$= \mathbf{SimBody}_{\text{conc}}(\mathbf{clauses}, body(\texttt{clause}_{i'}), \texttt{clause}_{i'}, \mathbf{MFG}_{\text{conc}}, \{\tau\}) \downarrow_1,$$

since $\tau \in \mathbf{demanded}_i$, by Lemma 4.3.

Combining (6) and (9) in the definition of **SimBody**, we obtain

$$\begin{aligned}
&\mathbf{SimBody}_{\mathrm{conc}}\big(\mathbf{clauses}, [\![\mathtt{b}_1,\ldots,\mathtt{b}_j]\!], \mathtt{clause}_i, \mathbf{MFG}_{\mathrm{conc}},\\
&\qquad \{norm(\iota\circ\theta|_{\mathtt{clause}_i})\}\big)\downarrow_1\\
&\supseteq \mathbf{Extend}_{\mathrm{conc}}\Big(\mathtt{b}_j, \mathtt{clause}_i, \{norm(\iota\circ\theta\circ\theta_{j-1}|_{\mathtt{clause}_i})\}, \mathtt{clause}_{i'},\\
&\qquad norm(\iota'\circ\theta\circ\theta_j|_{\mathtt{clause}_{i'}})\}\Big) \qquad \{\\
&= \{norm(\rho\circ\mu|_{\mathtt{clause}_i})\}, \qquad (10)
\end{aligned}$$

where

$$\rho = norm(\iota\circ\theta\circ\theta_{j-1}|_{\mathtt{clause}_i}),$$

$$\mu = mgu(\mathtt{b}_j\rho, \mathtt{h}'\sigma\eta),$$

$$\sigma = norm(\iota\circ\theta\circ\theta_j|_{\mathtt{clause}_{i'}}),$$

$$\eta = StandApart(\mathtt{h}'\sigma, \mathtt{b}_j\rho).$$

Letting η' be the renaming that normalizes ρ (i.e., $\iota\circ\theta\circ\theta_{j-1}|_{\mathtt{clause}_i}\eta' = norm(\iota\circ\theta\circ\theta_{j-1}|_{\mathtt{clause}_i})$), we have

$$\begin{aligned}
\{norm(\rho\circ\mu|_{\mathtt{clause}_i})\} &= \{norm((\iota\circ\theta\circ\theta_{j-1}|_{\mathtt{clause}_i}\circ\eta'\circ\mu)|_{\mathtt{clause}_i})\}\\
&= \{norm(\iota\circ\theta\circ\theta_j|_{\mathtt{clause}_i})\}, \qquad (11)
\end{aligned}$$

because

$$\begin{aligned}
\mu &= mgu(\mathtt{b}_j\iota\theta\theta_{j-1}\eta', \mathtt{h}'\iota'\theta\theta_j\eta)\\
&= mgu(\hat{\mathtt{b}}_j\theta\theta_{j-1}\eta', \hat{\mathtt{h}}'\theta\theta_{j-1}\theta'\theta'_{k'}\eta)\big(\text{since } \hat{b}_j = b_j\iota \text{ and } \hat{h}' = h'\iota\big)\\
&\sim (\eta')^{-1}\theta'\theta' B_{k'}\eta,
\end{aligned}$$

where $\sim$ denotes renaming equivalence, since $\hat{\mathtt{b}}_j\theta\theta_{j-1}\eta'(\eta')^{-1}\tilde{\theta}' = \hat{\mathtt{b}}_j\theta\theta_{j-1}\theta' = \hat{\mathtt{h}}'\theta\theta_{j-1}\theta'$. Thus, combining (10) and (11), we obtain

$$\begin{aligned}
&norm(\iota\circ\theta\circ\theta_j|_{\mathtt{clause}_i})\\
&\quad \in \mathbf{SimBody}_{\mathrm{conc}}\big(\mathbf{clauses}, [\![\mathtt{b}_1,\ldots,\mathtt{b}_j]\!], \mathtt{clause}_i,\\
&\qquad\qquad \mathbf{MFG}_{\mathrm{conc}}, \{norm(\iota\circ\theta|_{\mathtt{clause}_i})\}\big)\downarrow_1
\end{aligned}$$

concluding the induction step. □

Lemma 4.6 summarizes Lemmas 4.3–4.5.

Lemma 4.6. Using the naming conventions established in Figure 1, *every SLD derivation using the left-to-right computation rule satisfies*

$$\mathbf{MFG}_{\text{conc}} \downarrow_i (norm(\iota \circ \theta|_{\texttt{clause}_i})) \neq \amalg$$

$$\Rightarrow \quad normalize\left(\iota' \circ \theta \circ \theta_{j-1} \circ \theta'|_{\texttt{clause}_{i'}}\right)$$

$$\in \langle \mathbf{SimBody}_{\text{conc}}(\mathbf{clauses}, body(\texttt{clause}_i), \texttt{clause}_i, \mathbf{MFG}_{\text{conc}}, \{norm(\iota \circ \theta|_{\texttt{clause}_i})\}) \downarrow_2 \rangle \downarrow_j \downarrow_{i'},$$

and

$$\mathbf{MFG}_{\text{conc}} \downarrow_{i'} \left(\left\{ norm\left(\iota' \circ \theta \circ \theta_{j-1} \circ \theta'|_{\texttt{clause}_{i'}}\right) \right\} \right) \neq \amalg,$$

and

$$norm\left(\iota \circ \theta \circ \theta_j|_{\texttt{clause}_i}\right)$$

$$\in \mathbf{SimBody}_{\text{conc}}(\mathbf{clauses}, [\![\texttt{b}_1, \ldots, \texttt{b}_j]\!], \texttt{clause}_i, \mathbf{MFG}_{\text{conc}}, \{norm(\iota \circ \theta|_{\texttt{clause}_i})\}) \downarrow_1$$

for all j, $0 \leq j \leq k_i$.

PROOF. Follows directly from Lemmas 4.3–4.5. □

The following theorem says that the concrete MFG semantics of a program records the result of each clause activation reachable from an entry call.

Theorem 4.7. Using the naming conventions established in Figure 1, *every SLD derivation using the left-to-right computation rule and starting from* $\texttt{c}\sigma$, *where* $\langle \texttt{c}, \beta \rangle \in \mathbf{Ent}$ *and* $\sigma \in \beta$, *satisfies*

$$\mathbf{MFG}_{\text{conc}} \downarrow_i (\{norm(\iota \circ \theta|_{\texttt{clause}_i})\}) \neq \amalg \tag{12}$$

and

$$norm(\iota \circ \theta \circ \theta_k|_{\texttt{clause}_i}) \in \mathbf{MFG}_{\text{conc}} \downarrow_i (\{norm(\iota \circ \theta|_{\texttt{clause}_i})\}) \tag{13}$$

for all resolvents $\hat{\texttt{b}}_1, \ldots, \hat{\texttt{b}}_k$, G; θ, *where* $\hat{\texttt{b}}_1, \ldots, \hat{\texttt{b}}_k = (body(\texttt{clause}_i))\iota$.

PROOF. Let $\texttt{clause}_{i''} = \texttt{h}'' :- \texttt{b}''_1, \ldots, \texttt{b}''_{k''}$ be the first clause used in the derivation, and let $\theta'' = mgu(h''\iota'', \texttt{c}\sigma)$. We show

$$\mathbf{MFG}_{\text{conc}} \downarrow_{i''} (\{norm(\iota'' \circ \theta''|_{\texttt{clause}_{i''}})\}) \neq \amalg. \tag{14}$$

By definition of **SimProg**, $\mathbf{MFG}_{\text{conc}} \downarrow_{i'}(\tau) \neq \amalg$ holds for all $\tau \in \mathbf{Csub}_{\text{conc}}$ such that

$$\tau \subseteq \left(\bigsqcup_{\mathbf{Asub}^n} \mathbf{init}_{\text{conc}}(\mathbf{clauses}, \mathbf{Ent}) \right) \downarrow_{i'}.$$

Thus, using the definition of **init**, it is sufficient to show

$$\{norm(\iota'' \circ \theta''|_{\texttt{clause}_{i''}})\} \subseteq \bigcup_{\langle \texttt{c}, \beta \rangle \in \mathbf{Ent}} \mathbf{Call}(\texttt{c}, \beta, \texttt{clause}_{i''}).$$

In the proof of Lemma 4.3 we showed that (2),

$$norm(\iota'' \circ \theta''|_{\texttt{clause}_{i''}}) \in \mathbf{Call}_{\mathbf{conc}}(\texttt{c}, \beta, \texttt{clause}_{i''}),$$

follows from the definition of $\mathbf{Call}_{\mathbf{conc}}$. This proves (14).

The first proposition (12) of the theorem follows from (14) by using Lemma 4.6 and induction on the length of the subderivation from $\hat{\texttt{b}}''_1, \ldots, \hat{\texttt{b}}''_{k''}; \theta''$. By using (12) in Lemma 4.6 we obtain

$$norm(\iota \circ \theta \circ \theta_k|_{\texttt{clause}_i})$$
$$\in \mathbf{SimBody}_{\mathbf{conc}}(\mathbf{clauses}, [\![\texttt{b}_1, \ldots, \texttt{b}_k]\!], \texttt{clause}, \mathbf{MFG}_{\mathbf{conc}}, \{norm(\iota \circ \theta|_{\texttt{clause}_i})\}) \downarrow_1,$$

where $\texttt{b}_1$,$\texttt{b}_k$ and θ are as in the statement of the theorem. The second part (13) of the theorem now follows by definitions of **SimProg** and **MFG**. □

5. PATH-DEPENDENT REACHABLE ACTIVATIONS

This paper formalizes multiple specialization at the level of procedures. This formalization is appropriate for procedure-level compiler optimizations, such as those based on mode analysis. The multiple-specialization technique can easily be extended to accommodate clause-level optimizations and optimizations based on the execution state at intermediate points of clause-body execution.

Throughout, we assume that the compiler generates procedure implementations that are correct for the activation descriptions supplied by our method. The aspect of specialization we address here is how to select appropriate procedure implementations to handle the calls in the expanded program.

As discussed in Section 1.1, we formalize the notion of a procedure-activation description as a vector in $\mathbf{Asub}^n$, having one clause-activation description for each clause in the program. (By including an element for every clause in the program, rather than for every clause in the procedure, we avoid having a different type of version for each procedure.) We call such a vector a *version*. Given a version, **version** $= \langle \beta_1, \beta_n \rangle$, β_i is a clause-activation description for $\texttt{clause}_i$. All versions we construct will have non-$\perp$ elements only in positions corresponding to clauses in the procedure being invoked.

We now construct **ReachableVersions**, which is a set of call-path-dependent reachable procedure-activation descriptions. Like the core MFG semantics, this construction is parametrized by an interpretation. A compiler using our method creates one procedure implementation for each element of **ReachableVersions**. For finite abstract domains there will be finitely many reachable versions. In the concrete case there may be infinitely many reachable versions. However, since we construct the concrete set only as a theoretical intermediary towards verifying the abstract set, this is not a problem. Recall Definition 4.1: **Entries** $= \mathscr{P}(\texttt{Literal} \times \mathbf{Asub})$.

Symbols.

χ, ζ:	$\mathscr{P}(\mathbf{Asub}^n)$
reachable:	$\mathbf{Asub}^n$

Definition 5.1.

$$\textbf{ReachableVersions}: \texttt{Prog} \times \textbf{Entries} \to \mathscr{P}(\mathsf{Asub}^n),$$

$$\textbf{ReachableVersions}(\textbf{clauses}, \textbf{Ent}) = \mathit{fix}(\textbf{CollectVersions}(\textbf{clauses}, \textbf{MFG}, \textbf{init}(\textbf{clauses}, \textbf{Ent}))),$$

$$\textbf{CollectVersions}: \texttt{Prog} \times \textbf{Den} \times \mathscr{P}(\mathsf{Asub}^n) \to \mathscr{P}(\mathsf{Asub}^n) \to \mathscr{P}(\mathsf{Asub}^n),$$

$$\textbf{CollectVersions}(\textbf{clauses}, \psi, \chi)(\zeta) = \chi \cup \left\{ \left\langle (\textbf{SimBody}(\textbf{clauses}, \mathit{body}(\texttt{clause}_i), \texttt{clause}_i, \psi, \textbf{reachable}_i)) \downarrow_2 \right\rangle \downarrow_j \mid 1 \le i \le n, 1 \le j \le k_i, \textbf{reachable} \in \zeta \right\}.$$

6. THE CALL-PATH AUTOMATON

A compiler using our method creates a procedure implementation for each element of **ReachableVersions**. The compiler selects implementations for each call according to an automaton that has **ReachableVersions** as its set of states. This automaton scans a call path, given by a sequence of pairs, each a clause index and the index of a call in that clause. The automaton's transition function, Δ, is the central subject of this section. For each clause-call pair it scans, the automaton changes state to a procedure-activation description that covers all possible procedure activations arising when the given clause and call are used in a derivation conforming to the prior state.

The initial state of the automaton is given by a procedure-activation description satisfying the entry description. If, in a given application, the entry description is permitted to have at most one element for any predicate in the program, then the initial state, and hence the procedure implementation to invoke, is uniquely determined for each entry-level call. If the entry description is allowed to have more than one element for a given predicate, some other mechanism must be employed to characterize top-level calls, as discussed in Section 4.1.

Each state of the automaton corresponds to a regular set of call paths, each of which leads to a call activation matching the description given by that state. Before the compiler generates the expanded program from the automaton, it has the option of minimizing the number of automaton states for which it must generate implementations by using the technique presented in Section 8. This option allows the compiler to merge versions that enable the same or similar compiler optimizations, avoiding excessive code size.

This section is organized as follows. First, we define call paths. Second, we construct the transition function Δ. Third, we define what we mean when we say a derivation *conforms* to a call path. This constrains the derivation to select the clauses given by the call path for the calls along that path. Fourth, we show that Δ always selects an implementation that is correct for the call. That result ensures that a compiler using our method will safely select specialized implementations for calls in the program's expanded implementation. Finally, we observe that **ReachableVersions(clauses, Ent)** is closed under the transition function. That result ensures that if a compiler creates a specialized implementation for every activation description in **ReachableVersions(clauses, Ent)**, it will be able to find a safe implementation for *every* call in the program's expanded implementation.

Intuitively, a call path identifies a branch in an AND/OR tree for the program. Within an AND/OR tree, each procedure call is represented by an OR node. The downward path from one OR node to another is given by a clause number and the number of a call within that clause. We consider this path to have length one, and represents it by a pair of integers. Longer call paths are formed by concatenation.

Definition 6.1. $\{P_l\}_m = \langle i_1, j_1\rangle \ldots \langle i_m, j_m\rangle$ is a *call path* for **clauses** and **Ent** if

(1) for every l, $1 < l \le m$, $\texttt{clause}_{i_l}$ is a clause for $\texttt{b}_{j_{l-1}}$, where $\texttt{clause}_{i_{l-1}} = \texttt{h:-}\ \texttt{b}_1, \ldots, \texttt{b}_{j_{l-1}}, \ldots, \texttt{b}_k$, and
(2) $\forall l,\ 1 \le l \le m,\ 1 \le j_l \le k_{i_l}$.

(Recall that k_i is the number of calls in $\texttt{clause}_i$.)

We write

$$\{P_l\}_m = \{P_l\}_{m-1}\langle i_m, j_m\rangle = \langle i_1, j_1\rangle \ldots \langle i_{m-1}, j_{m-1}\rangle\langle i_m, j_m\rangle.$$

In the definition of Δ, we assume that **clauses** and **Ent** are fixed but arbitrary. If $\mathbf{SimBody}_{\text{conc}}$ and $\mathbf{MFG}_{\text{conc}}$ are used, we get Δ_{conc}, and likewise for Δ_{abs}.

Symbols.

version: Asub^n

Definition 6.2.

$$\Delta : \mathsf{Asub}^n \times ([1..n] \times [1..k]) \to (\mathsf{Asub}^n \cup \{\mathit{undefined}\}),$$

$$\Delta(\mathbf{version}, \langle i, j\rangle) = \text{if } k_i < j \text{ then } \mathit{undefined} \text{ else}$$

$$\langle \mathbf{SimBody}(\mathbf{clauses}, \mathit{body}(\texttt{clause}_i), \texttt{clause}_i, \mathbf{MFG}, \mathbf{version}_i)\downarrow_2\rangle \downarrow_j.$$

Theorem 6.3 shows that Δ_{conc} selects safe implementations within the activations that $\mathbf{MFG}_{\text{conc}}$ considers reachable.

Theorem 6.3. Using the naming conventions for resolvents established in Figure 1, *each SLD derivation using the left-to-right computation rule and starting from* $\texttt{c}\sigma$, *where* $\langle \texttt{c}, \beta\rangle \in \mathbf{Ent}$ *and* $\sigma \in \beta$, *satisfies*

$$\mathit{norm}(\iota\circ\theta|_{\texttt{clause}_i}) \in \mathbf{version}_i$$

$$\Rightarrow \quad \mathit{norm}\left(\iota'\circ\theta\circ\theta_{j-1}\circ\theta'|_{\texttt{clause}_{i'}}\right) \in \Delta_{\text{conc}}(\mathbf{version}, \langle i, j\rangle)\downarrow_{i'}.$$

PROOF. From Theorem 4.7, we have

$$\mathbf{MFG}_{\text{conc}}\downarrow_i(\{\mathit{norm}(\iota\circ\theta|_{\texttt{clause}_i})\}) \neq \amalg.$$

The theorem follows from this by Lemma 4.6. □

The rest of this section is devoted to showing that an automaton using Δ selects safe implementations for all calls that are reachable along any call path from an initial goal given by **Ent**. We now characterize the derivations in which a given call path is taken.

Definition 6.4. Using the naming conventions established in Figure 1, an SLD derivation D, using the left-to-right computation rule, *conforms* to a call path $\{P_l\}_m$ from $\langle \mathtt{c}, \beta \rangle \in$ **Ent** if

(1) $\{P_l\}_m$ is empty and D has length one: $D = \langle \mathtt{c}; \sigma \rangle$, where $\sigma \in \beta$, or

(2) using the abbreviations $j_{m-1} = j$, $i_{m-1} = i$, $j_m = j'$, and $i_m = i'$, there is a prefix of D,

$$D' = \langle \mathtt{c}; \sigma \rangle \ldots \langle \hat{\mathtt{b}}_j, \hat{\mathtt{b}}_{(j+1)}, \ldots, \hat{\mathtt{b}}_k, G; \theta \circ \theta_{j-1} \rangle$$

where

$$(\mathtt{clause}_i)\iota = \hat{\mathtt{h}} :\!- \hat{\mathtt{b}}_1, \ldots, \hat{\mathtt{b}}_j, \hat{\mathtt{b}}_{(j+1)}, \ldots, \hat{\mathtt{b}}_k,$$

such that D' conforms to $\{P_l\}_{m-1}$ from $\langle \mathtt{c}, \beta \rangle$, where

$$\{P_l\}_{m-1} = \langle i_1, j_1 \rangle \ldots \langle i, j \rangle,$$

$$\{P_l\}_m = \langle i_1, j_1 \rangle \ldots \langle i, j \rangle \langle i', j' \rangle,$$

and

$$D = D'.\langle \hat{\mathtt{b}}'_{j'}, \ldots, \hat{\mathtt{b}}'_{k'}, \hat{\mathtt{b}}_{(j+1)}, \ldots, \hat{\mathtt{b}}_k, G; \theta \circ \theta_{j-1} \circ \theta' \circ \theta'_{j'-1} \rangle,$$

where

$$(\mathtt{clause}_{i'})\iota' = \hat{\mathtt{h}}' :\!- \hat{\mathtt{b}}'_1, \ldots, \hat{\mathtt{b}}'_{j'}, \ldots, \hat{\mathtt{b}}'_{k'}.$$

We extend the automaton's transition function to arbitrary call paths in the natural way.

Definition 6.5.

$$\Delta^* : (\mathsf{Asub}^n \cup \{\mathit{undefined}\}) \times ([1..n] \times [1..k])^* \to (\mathsf{Asub}^n \cup \{\mathit{undefined}\}),$$

$$\Delta^*(\mathbf{version}, \{P_l\}_m)$$

$$= \text{if } m = 0 \text{ then } \mathbf{version} \text{ else}$$

$$\text{if } \Delta^*(\mathbf{version}, \{P_l\}_{m-1}) = \mathit{undefined} \text{ then } \mathit{undefined} \text{ else}$$

$$\text{if } k_{i_m} < j_m \text{ then } \mathit{undefined} \text{ else}$$

$$\Delta(\Delta^*(\mathbf{version}, \{P_l\}_{m-1}), \langle i_m, j_m \rangle).$$

We now formalize the central result of this section, that the call invocation at the end of each call path will, in all conforming derivations, lead to a procedure activation given by the behavior of Δ_{conc} on that call path. Theorem 6.6 guarantees that call invocations will be handled by implementations that are suitable for the activation.

Theorem 6.6. Let $\{P_l\}_m = \langle i_1, j_1 \rangle \ldots \langle i_m, j_m \rangle$ *be a call path. Let*

$$D = \langle \mathtt{c}; \sigma \rangle \ldots \langle \hat{\mathtt{b}}_{j_m}, \ldots, \hat{\mathtt{b}}_{k_{i_m}}, G; \theta \circ \theta_{j-1} \rangle,$$

where $\mathtt{b}_1, \mathtt{b}_{j_m}, \ldots, \mathtt{b}_{k_{i_m}} = \mathit{body}(\mathtt{clause}_{i_m})$, *be any derivation from* $\langle \mathtt{c}, \beta \rangle$ *using*

clauses *and conforming to* $\{P_l\}_m$. *Then, for each* $i' \in [1..n]$ *such that* $\theta' = mgu(\hat{b}_{j_m}\theta\theta_{j-1}, \hat{h})$ *is not fail, where* $(\texttt{clause}_{i'})\iota' = \hat{h}' :- \hat{b}'_1, \ldots, \hat{b}'_{k'}$, *it follows that*

$$norm\left(\iota' \circ \theta \circ \theta_{j-1} \circ \theta' \big|_{\texttt{clause}_{i'}}\right) \in \Delta^*_{\text{conc}}(\textbf{version}, \{P_l\}_m)\downarrow_{i'},$$

where {**version**} = **init**(**clauses**, {⟨c, β⟩}).

PROOF. Follows by induction on the length of $\{P_l\}_m$ from Theorems 4.6 and 6.3. □

Theorem 6.7 ensures that **ReachableVersions** contains all the procedure-activation descriptions for which implementations need be created.

Theorem 6.7. **ReachableVersions**(**clauses**, **Ent**) *contains* **init**(**clauses**, **Ent**) *and is closed under* λ **version**. Δ(**version**, $\langle i, j \rangle$) *for all* $\langle i, j \rangle$ *such that* $1 \le i \le n$ *and* $1 \le j \le k_i$.

PROOF. Follows immediately from the definition of **ReachableVersions** and of Δ. □

Together Theorems 6.6 and 6.7 demonstrate the soundness of our multiple-specialization framework.

7. EXAMPLE

To illustrate our multiple-specialization technique, consider the program for list reversal given in Figure 2, suggested for this purpose by M. Bruynooghe in a private correspondence. Suppose **Ent** = {⟨rev(X, Y), ⟨X ↦ *ground*, Y ↦ *free*⟩⟩}. Then **ReachableVersions** = {**ver**$_1$, **ver**$_2$, **ver**$_3$}, where

$$\textbf{ver}_1 = \langle\langle \texttt{L} \mapsto \textit{ground}, \texttt{R} \mapsto \textit{free}\rangle, \langle \texttt{X} \mapsto \textit{ground}\rangle, \perp_{\text{Asub}}, \perp_{\text{Asub}}\rangle,$$

$$\textbf{ver}_2 = \langle \perp_{\text{Asub}}, \perp_{\text{Asub}},$$
$$\langle \texttt{X} \mapsto \textit{ground}, \texttt{T} \mapsto \textit{ground}, \texttt{L} \mapsto \textit{ground}, \texttt{T1} \mapsto \textit{free}\rangle,$$
$$\langle L \mapsto \textit{ground}\rangle\rangle,$$

$$\textbf{ver}_3 = \langle \perp_{\text{Asub}}, \perp_{\text{Asub}},$$
$$\langle \texttt{X} \mapsto \textit{ground}, \texttt{T} \mapsto \textit{free}, \texttt{L} \mapsto \textit{free}, \texttt{T1} \mapsto \textit{ground}\rangle,$$
$$\langle \texttt{L} \mapsto \textit{ground}\rangle\rangle.$$

```
(1) rev(L,R) :- append(L1, L2, L), rev(L1, R1),
                rev(L2, R2), append(R2, R1, R).
(2) rev([X], [X]).
(3) append([X|T], L, [X|T1] :- append(T, L, T1).
(4) append([ ], L, L).
```

FIGURE 2. When `rev` is used with its first argument ground and its second argument free, the two uses of `append` use different modes, which enable different opptimizations.

Δ will be

$(\mathbf{ver}_1, \langle 1,1 \rangle) \mapsto \mathbf{ver}_3$,

$(\mathbf{ver}_1, \langle 1,2 \rangle) \mapsto \mathbf{ver}_1$,

$(\mathbf{ver}_1, \langle 1,3 \rangle) \mapsto \mathbf{ver}_1$,

$(\mathbf{ver}_1, \langle 1,4 \rangle) \mapsto \mathbf{ver}_2$,

$(\mathbf{ver}_2, \langle 3,1 \rangle) \mapsto \mathbf{ver}_2$,

$(\mathbf{ver}_3, \langle 3,1 \rangle) \mapsto \mathbf{ver}_3$.

The compiler uses this information to generate one version of `rev`, which we call $\texttt{rev}_1$, and two versions of `append`, which we call $\texttt{append}_2$ and $\texttt{append}_3$. The generated code corresponds to the following expansion of the program:

```
(1) rev1 (L, R) :- append3(L1, L2, L), rev1(L1, R1),
                   rev1(L2, R2), append2(R2, R1, R).
(2) rev1([X], [X]).
(3) append2([X|T], L, [X|T1]) :- append2(T, L, T1).
(4) append2([ ], L, L).
(5) append3([X|T], L, [X|T1]) :- append3(T, L, T1).
(6) append3([ ], L, L).
```

The implementation of $\texttt{append}_2$ can be specialized under the assumption that the first two arguments are ground and the third argument is free; the implementation of $\texttt{append}_3$ can be specialized under the assumption that the first two arguments are free and the third argument is ground.

More realistic examples can be imagined. For instance, it is not difficult to write a procedure that, in one mode of use, parses an input program and, reversing the input–output roles of the arguments, "pretty-prints" a parse tree (generating less attractive printouts on backtracking). Such a procedure can be used to convert an unformatted program into a "pretty-formatted" program:

```
prettyprint(UglyText, PrettyText)
                                :- prettyparse(UglyText, ParseTree),
  prettyparse(PrettyText, ParseTree).
```

Our multiple-specialization technique generates one specialized version of `prettyparse` for each of these two uses.

8. MINIMIZING THE NUMBER OF VERSIONS

Not all procedure-activation descriptions lead to different optimizations in practice. We would prefer not to create two implementations of the same procedure with the same optimizations. However, we cannot eliminate all such duplication. In addition to implementing its procedure, each different procedure implementation implements a different state in the automaton for call paths. Therefore, we must be cautious about collapsing different states when their corresponding implementations employ the same optimizations: two automaton states that permit the same optimizations may, on the same input call path, lead to states that permit different

optimizations. There is an obvious correspondence between this problem and the well-known (and solved) problem of minimizing deterministic finite automatons. This correspondence gives a solution to the problem of minimizing the number of implementations.

Essentially, the procedure is as follows. First, partition **ReachableVersions**, grouping versions that enable the same specializations. Then, incrementally refine this partition by dividing each equivalence class into subsets, each containing the versions that, pointwise, have the same image under the transition map, modulo the current partition. These steps can be formalized as follows:

1. $\mathsf{Partition} = \{\{v' \mid \mathit{SameSpecializations}(v', v)\} v \in \mathbf{ReachableVersions}\}$.
2. $\mathsf{Partition}' = \{\{v' \mid \exists_{1,2} \in \mathsf{Partition}\,.\, v, v' \in p_1 \;\&\; \forall i, j\,.\, \Delta(v', \langle i, j\rangle) \in p_2 \;\&\; \Delta(v, \langle i, j\rangle) \in p_2\} \mid v \in \mathbf{ReachableVersions}\}$.

Step 1 is applied to create the initial partition. Step 2 is then applied iteratively until no changes result. We see from step 2 that Partition′ is a refinement of Partition. Consequently, in the abstract case, the refinement process must terminate. The sets in the final partition indicate how to merge implementations to avoid excessive code size.

Of course, if code size is an especially pressing issue, it would be possible to implement the automaton state separately and to implement all procedure calls with a level of indirection, using that state to determine the appropriate implementation. Clearly, that would introduce run-time overhead, which would offset the benefits of our method somewhat. Since time is generally a scarcer resource than space, we concentrate on techniques that implement each automaton state by a different procedure implementation and thus avoid all time overhead, at the expense of some space.

9. SAFE ABSTRACT INTERPRETATIONS

We now take up the issue of ensuring that an abstract interpretation induces an abstract semantics that safely approximates the concrete semantics. Although this is an important aspect of our method, most of the techniques used in this section are well known. Therefore, discussion and proofs are brief.

We need the abstract MFG semantics to approximate from above the set of reachable activations and their results. That way, anything the analyzer tells the compiler cannot happen during program execution really cannot, as required for correct program optimization. This safety relation is formalized by means of adjoined functions [6].

Definition 9.1. Let D_{conc} and D_{abs} be domains, and let

$Abs : D_{\text{conc}} \to D_{\text{abs}}$,

$Conc : D_{\text{abs}} \to D_{\text{conc}}$.

Then *Abs* and *Conc* are *adjoined* if

(1) *Abs* and *Conc* are monotonic and continuous,
(2) for all $x \in D_{conc}$, $x \sqsubseteq_{conc} Conc(Abs(x))$, and
(3) for all $x \in D_{\text{abs}}$, $x = Abs(Conc(x))$.

Using this definition, we give sufficient conditions for the safety of an abstract interpretation.

Condition 9.2. An abstract interpretation satisfies Condition 9.2 if Call_{abs} and Extend_{abs} are monotonic and continuous and there exist adjoined functions

$$\mathsf{Abs}: \mathsf{Asub}_{conc} \to \mathsf{Asub}_{abs},$$

$$\mathsf{Conc}: \mathsf{Asub}_{abs} \to \mathsf{Asub}_{conc}$$

such that for $c \in \mathtt{Literal}$, $\beta_{abs} \in \mathsf{Asub}_{abs}$, $\beta_{conc} \in \mathsf{Asub}_{conc}$, $\tau_{abs} \in \mathsf{Csub}_{abs}$, $\tau_{conc} \in \mathsf{Csub}_{conc}$, and $\mathtt{clause}_1, \mathtt{clause}_2 \in \mathtt{Clause}$, where c is a call in $\mathtt{clause}_1$,

$$\beta_{conc} \sqsubseteq_{\mathsf{Asub}_{conc}} \mathsf{Conc}(\beta_{abs})$$

$$\Rightarrow \quad \mathsf{Call}_{conc}(c, \beta_{conc}, \mathtt{clause}_2) \sqsubseteq_{\mathsf{Asub}_{conc}} \mathsf{Conc}(\mathsf{Call}_{abs}(c, \beta_{abs}, \mathtt{clause}_2)),$$

and

$$(\beta_{conc} \sqsubseteq_{\mathsf{Asub}_{conc}} \mathsf{Conc}(\beta_{abs}) \text{ and } \tau_{conc} \sqsubseteq_{\mathsf{Asub}_{conc}} \mathsf{Conc}(\tau_{abs}))$$

$$\Rightarrow \quad \mathsf{Extend}_{conc}(c, \mathtt{clause}_1, \tau_{conc}, \mathtt{clause}_2, \beta_{conc})$$

$$\sqsubseteq_{\mathsf{Asub}_{conc}} \mathsf{Conc}(\mathsf{Extend}_{abs}(c, \mathtt{clause}_1, \tau_{abs}, \mathtt{clause}_2, \beta_{abs})).$$

Lemma 9.3.

$$\tau_{conc} \in \mathsf{Csub}_{conc} \quad \Rightarrow \quad \mathsf{Abs}(\tau_{conc}) \in \mathsf{Csub}_{abs}.$$

Definition 9.4.

$$\mathbf{Conc}(\psi_{abs}) = \lambda \tau_{conc} : \mathsf{Csub}_{conc} . \mathsf{Conc}(\psi_{abs}(\mathsf{Abs}(\tau_{conc}))).$$

We use $\sqsubseteq_{\mathbf{Den}_{conc}}$ to relate functions in $\mathbf{Den}_{conc}$ by applying $\sqsubseteq_{\mathsf{Asub}_{\sqcup conc}}$ pointwise. In the remaining results of this section we assume that the abstract interpretation given by Asub_{abs}, Csub_{abs}, Call_{abs}, and Extend_{abs} satisfies Condition 9.2.

Lemma 9.5. For all $\tau_{conc} \in \mathsf{Csub}_{conc}$ *and* $\tau_{abs} \in \mathsf{Csub}_{abs}$, *and for all* $\psi_{abs} \in \mathbf{Den}_{abs}$ *and* $\psi_{conc} \in \mathbf{Den}_{conc}$,

$$\psi_{conc} \sqsubseteq_{\mathbf{Den}_{conc}} \mathbf{Conc}(\psi_{conc})$$

$$\Rightarrow \quad (\tau_{conc} \sqsubseteq_{\mathsf{Asub}_{conc}} \mathsf{Conc}(\tau_{abs})$$

$$\Rightarrow \psi_{conc}(\tau_{conc}) \sqsubseteq_{\mathsf{Asub}_{\sqcup conc}} \mathsf{Conc}(\psi_{abs}(\tau_{abs}))).$$

Lemma 9.6. For all $\texttt{clause} \in \texttt{Clause}$, *for all prefixes of the body of* $\texttt{clause}, \texttt{b}_1, \ldots, \texttt{b}_j$, *for all* $\psi_{\text{abs}} \in \mathbf{Den}_{\text{abs}}$, $\psi_{\text{conc}} \in \mathbf{Den}_{\text{conc}}$, $\delta_{\text{conc}} \in \mathsf{Asub}_{\text{conc}}$, *and* $\delta_{\text{abs}} \in \mathsf{Asub}_{\text{abs}}$, *and for all* j', $1 \le j' \le j$, i, $1 \le i \le n$,

$$\psi_{\text{conc}} \sqsubseteq_{\mathbf{Den}_{\text{conc}}} \mathbf{Conc}(\psi_{\text{abs}}) \textit{ and } \delta_{\text{conc}} \sqsubseteq_{\mathsf{Asub}_{\text{conc}}} \mathsf{Conc}(\delta_{\text{abs}})$$

$$\Rightarrow \mathbf{SimBody}_{\text{conc}}(\mathbf{clauses}, [\![\texttt{b}_1, \ldots, \texttt{b}_j]\!], \texttt{clause}, \psi_{\text{conc}}, \delta_{\text{conc}}) \downarrow_1$$

$$\sqsubseteq_{\mathsf{Asub}_{\text{conc}}} \mathsf{Conc}(\mathbf{SimBody}_{\text{abs}}(\mathbf{clauses}, [\![\texttt{b}_1, \ldots, \texttt{b}_j]\!], \texttt{clause}, \psi_{\text{abs}}, \delta_{\text{abs}}) \downarrow_1)$$

and

$$\langle \mathbf{SimBody}_{\text{conc}}(\mathbf{clauses}, [\![\texttt{b}_1, \ldots, \texttt{b}_j]\!], \texttt{clause}, \psi_{\text{conc}}, \delta_{\text{conc}}) \downarrow_2 \rangle \downarrow_{j'} \downarrow_i$$

$$\sqsubseteq_{\mathsf{Asub}_{\text{conc}}} \mathsf{Conc}(\langle \mathbf{SimBody}_{\text{abs}}(\mathbf{clauses}, [\![\texttt{b}_1, \ldots, \texttt{b}_j]\!], \texttt{clause}, \psi_{\text{abs}}, \delta_{\text{abs}}) \downarrow_2 \rangle \downarrow_{j'} \downarrow_i).$$

PROOF. Follows from Lemma 9.5 and properties of $\sqcup$ and monotonic functions. □

Lemma 9.7. For all $\psi_{\text{abs}} \in \mathbf{Den}_{\text{abs}}$ *and* $\psi_{\text{conc}} \in \mathbf{Den}_{\text{conc}}$,

$$\psi_{\text{conc}} \sqsubseteq_{\mathbf{Den}_{\text{conc}}} \mathbf{Conc}(\psi_{\text{abs}}) \Rightarrow \mathbf{SimProg}_{\text{conc}}(\psi_{\text{conc}}) \sqsubseteq_{\mathbf{Den}_{\text{conc}}} \mathbf{Conc}(\mathbf{SimProg}_{\text{conc}}(\psi_{\text{conc}})).$$

PROOF. Follows from Lemma 9.6. □

Theorem 9.8 says that it is safe to use the abstract MFG semantics to simulate call execution.

Theorem 9.8. $\mathbf{MFG}_{\text{conc}} \sqsubseteq_{\mathbf{Den}_{\text{conc}}} \mathbf{Conc}(\mathbf{MFG}_{\text{abs}})$.

PROOF. Follows from Lemma 9.7 by induction on the Kleene sequence of **SimProg**, which can be used to compute the fixpoint that defines **MFG**. □

Theorem 9.9 tells us that, so long as the first call is handled by an appropriate implementation, each subsequent call will also be handled by an appropriate implementation. In the statement of Theorem 9.9, Conc^n and $\sqsubseteq_{\mathsf{Asub}^n_{\text{conc}}}$ are Conc and $\sqsubseteq_{\mathsf{Asub}_{\text{conc}}}$ applied pointwise.

Theorem 9.9. For all i, $1 \le i \le n$, *for all* j, $1 \le j \le k_i$, *and for all* $\mathbf{version}_{\text{abs}} \in \mathsf{Asub}^n_{\text{abs}}$ *and* $\mathbf{version}_{\text{conc}} \in \mathsf{Asub}^n_{\text{conc}}$,

$$\mathbf{version}_{\text{conc}} \sqsubseteq_{\mathsf{Asub}^n_{\text{conc}}} \mathsf{Conc}^n(\mathbf{version}_{\text{abs}}) \Rightarrow \Delta_{\text{conc}}(\mathbf{version}_{\text{conc}}\langle i, j\rangle) \sqsubseteq_{\mathsf{Asub}^n_{\text{conc}}} \mathsf{Conc}^n(\Delta_{\text{abs}}(\mathbf{version}_{\text{abs}}, \langle i, j\rangle)).$$

PROOF. Follows from Lemma 9.6 and Theorem 9.8. □

10. RELATED WORK

In the abstract interpretation presented in [13], Jones and Søndergaard construct for each clause a total function over substitution descriptions. (A second compo-

nent of their construction collects, for each procedure, one procedure-activation description that covers all the possible activations of that procedure for a given entry. However, this part is not useful for our purpose, since it collects only one activation description for each procedure.) In practice, one wishes to avoid computing the transform over all activation states and to limit one's efforts to activations that are reachable from a given entry description. This minimization of effort can be addressed by using a demand-driven evaluation. Alternatively, a minimal-function graph semantics has a demand-driven construction.

Bruynooghe has given a detailed abstract-interpretation framework for logic programs in [2]. There he mentions the possibility of generating several implementations:

> A compiler using this abstract and-or tree as input has the option to generate code for several versions of the clauses. Of course, care must be taken that each call uses the appropriate version.

Our method works out how to be sure that each call uses the appropriate version. Moreover, our method makes it straightforward to generate a collection of implementation versions that is *minimal* among those that utilize all available opportunities for optimization. This is because our method is expressed in terms of a deterministic finite automaton to which we can apply well-known minimization techniques, according to the opportunities for optimization that arise for each procedure in response to each procedure-activation description computed using our method.

The method given in [9] provides multiple specialization based on a *source-level* partial evaluation. Our technique provides multiple specialization based on compiler optimizations, such as optimization of unification code and storage allocation, choice-point elimination, occur-check reduction, or discovery of goal-independent AND parallelism.

11. CONCLUSION

The primary purpose of this paper is to show how analysis-based compiler optimizations can take advantage of call-path-dependent reachability to incorporate multiple specialized procedure implementations without run-time overhead, although slightly increasing the code size.

Our method constructs a collection of call-path-dependent procedure-activation descriptions. From these descriptions, a compiler using our method can create a collection of specialized procedure implementations. For each procedure implementation in the collection and each call in that implementation, our method finds another procedure implementation in the collection suitable to handle that call. With finite automaton minimization techniques, it is straightforward to minimize the size of the collection without losing any opportunity for optimization. This minimality is relative to the particular optimization being applied and the particular abstract domain over which the program analysis is done.

Our method selects the procedure implementation used for each call invocation based on the path to that call in the computation's AND-OR tree. Previous techniques have selected the implementation of procedure invocations with coarser resolution: all invocations of each procedure are given the same implementation.

That makes the implementation selection trivial, but it results in missed opportunities for optimization. The implementation of procedure invocations could be differentiated with finer resolution by using more information about the computation's history than the current call path to distinguish among procedure invocations. This would require the maintenance of explicit state information and some inspection of that state when invoking affected procedures. We have developed the approach that, among techniques that require no time overhead during execution, achieves the finest possible resolution in selecting implementations for procedure invocations.

Although we have not measured the performance improvement obtainable in particular applications, the lack of overhead in all but space-constrained systems seems to rule out the possibility of degraded performance in practice. (Although the expansion in code size could in principle diminish locality exhibited by the code, logic programs do not exhibit significant locality in the first place [23].) It remains open to assess the performance benefits of particular applications on a case-by-case basis.

The secondary purpose of this paper is to give a compositional core semantics that constructs results only for reachable activations. We have done this by giving a minimal-function graph semantics [12] that is parametrized by an abstract domain. To construct an analysis in this framework, one simply provides an abstract domain and two operations on it. To verify that analysis, one verifies that the abstract domain and operations obey Condition 9.2.

Our method is general and can be applied to virtually any optimization based on interprocedural program analysis. We have not specified abstract domains—only their correctness requirements. Previous research has demonstrated the viability of providing a general framework and subsequently specializing it by supplying the abstract domain [2, 13, 17]. This paper also does not provide algorithms to evaluate our constructions. O'Keefe has demonstrated the existence of simple, efficient algorithms for evaluating fixpoint constructions [20].

Although we do not treat them explicitly, negated calls (using negation as failure) cannot change the substitution environment, except to kill it off, which, if ignored, leads to an upwards approximation of the true set of possible environments. Therefore, negated calls can be tolerated and ignored safely by our analyses.

This work was supported in part by the National Science Foundation under grants CCR 87-06329 and CCR 88-05503. It was also supported in part 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. Aho, A. and Ullman, J., *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.
2. Bruynooghe, M., A Practical Framework for the Abstract Interpretation of Logic Programs, Dept. of Computer Science Report CW 62, Katholieke Univ. Leuven, 1988; *J. Logic Programming*, 10:91–124 (1990).

3. Bruynooghe, M., Janssens, G., Callebaut, A., and Demoen, B., Abstract Interpretation: Towards the Global Optimization of Prolog Programs, in: *Proceedings of the 1987 Symposium on Logic Programming*, IEEE Computer Soc. Press, Los Angeles, 1987, pp. 192–204.
4. Chang, J.-H., Despain, A. M., and DeGroot, D., AND-Parallelism of Logic Programs Based on a Static Data Dependency Analysis, in: *Proceedings of the IEEE 1985 Spring CompCon*, IEEE Computer Soc. Press, Los Angeles, 1985, pp. 218–225.
5. Codish, M., Gallagher, J., and Shapiro, E., Using Safe Approximations of Fixed Points for Analysis of Logic Programs, in: *Meta Programming in Logic Programming*, MIT Press, Cambridge, Mass., 1989.
6. Cousot, P. and Cousot, R., Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, ACM Press, New York, 1977, pp. 238–252.
7. Debray, S. K. and Warren, D. S., Automatic Mode Inference for Logic Programs, *J. Logic Programming* 5(3):207–229 (1988).
8. Debray, S. K., Synthesizing Control Strategies for AND-Parallel Logic Programs, Dept. of Computer Science Technical Report 87-12, Univ. of Arizona, Tucson, 1987.
9. Gallagher, J., Codish, M., and Shapiro, E., Specialisation of Prolog and FPC Programs Using Abstract Interpretation, *New Generation Computing* 6:159–186 (1988).
10. Hudak, P. and Young, J., Higher-Order Strictness Analysis in Untyped Lambda Calculus, in: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, ACM Press, New York, 1986, pp. 97–109.
11. Hudak, P., A Semantic Model of Reference Counting and Its Abstraction, in: S. Abramsky and C. Hankin (eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, Chichester, 1987, pp. 45–63.
12. Jones, N. D. and Mycroft, A., Data Flow Analysis of Applicative Programs Using Minimal Function Graphs: Abridged Version in: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, ACM Press, New York, 1986, pp. 296–306.
13. Jones, N. D. and Søndergaard, H., A Semantics-Based Framework for the Abstract Interpretation of Prolog, in: S. Abramsky and C. Hankin (eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood, Chichester, 1987, pp. 123–142.
14. Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1984.
15. Marriott, K. and Søndergaard, H., Bottom-Up Abstract Interpretation of Logic Programs, in: R. Kowalski and K. Bowen (eds.), *Fifth International Conference on Logic Programming*, MIT Press, Cambridge, Mass., 1988, pp. 733–748.
16. Mellish, C. S., The Automatic Generation of Mode Declarations for Prolog Programs, Dept. of Artificial Intelligence Research Report 163, Univ. of Edinburgh, Edinburgh, Scotland, 1981.
17. Mellish, C. S., Abstract Interpretation of Prolog Programs, in: *Third International Conference on Logic Programming*, Lecture Notes in Comput. Sci. 225, Springer-Verlag, Berlin, 1986, pp. 463–474.
18. Mulkers, A., Winsborough, W., and Bruynooghe, M., Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs, in: *Seventh International Conference on Logic Programming*, MIT Press, Cambridge, Mass., 1990.
19. Mycroft, A., Abstract Interpretation and Optimizing Transformations for Applicative Programs, Ph.D. Dissertation, Univ. of Edinburgh, Edinburgh, Scotland, 1981.
20. O'Keefe, R., Finite Fixed-Point Problems, in: J.-L. Lassez (ed.), *Fourth International Conference on Logic Programming*, MIT Press, Cambridge, Mass., 1987, pp. 729–743.
21. Plaisted, D., The Occur-Check Problem in Prolog, in: *Proceedings of the International Symposium on Logic Programming*, IEEE Computer Soc. Press, 1984, pp. 272–280.

22. Søndergaard, H., An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction, in: *Proceedings of the European Symposium on Programming*, Lecture Notes in Comput. Sci. 213, Springer-Verlag, Berlin, 1986, pp. 327–338.
23. Tick, E., Studies in Prolog Architectures, Ph.D. Dissertation, Stanford Univ., Palo Alto, Calif., 1987.
24. Winsborough, W., A Minimal Function Graph Semantics for Logic Programs, Computer Sciences Dept. Technical Report 711, Univ. of Wisconsin–Madison, 1987.
25. Winsborough, W., Path-Dependent Reachability Analysis for Multiple Specialization, in: E. Lusk and R. Overbeek (eds.), *Proceedings of the North American Conference on Logic Programming*, MIT Press, Cambridge, Mass., 1989, pp. 133–153.