Computing Procedure Summaries for Interprocedural Analysis*

Sumit Gulwani[1] and Ashish Tiwari[2]

[1] Microsoft Research, Redmond, WA 98052
sumitg@microsoft.com
[2] SRI International, Menlo Park, CA 94025
tiwari@csl.sri.com

Abstract. We describe a new technique for computing procedure summaries for performing an interprocedural analysis on programs. Procedure summaries are computed by performing a backward analysis of procedures, but there are two key new features: (i) information is propagated using "generic" assertions (rather than regular assertions that are used in intraprocedural analysis); and (ii) unification is used to simplify these generic assertions. We illustrate this general technique by applying it to two abstractions: unary uninterpreted functions and linear arithmetic. In the first case, we get a PTIME algorithm for a special case of the long-standing open problem of interprocedural global value numbering (the special case being that we consider unary uninterpreted functions instead of binary). This also requires developing efficient algorithms for manipulating singleton context-free grammars, and builds on an earlier work by Plandowski [13]. In linear arithmetic case, we get new algorithms for precise interprocedural analysis of linear arithmetic programs with complexity matching that of the best known deterministic algorithm [11].

1 Introduction

Precise interprocedural analysis (also referred to as full context-sensitive analysis) is provably harder than intraprocedural analysis [14]. One way to do precise interprocedural analysis is to do procedure-inlining followed by an intra-procedural analysis. There are two potential problems with this approach. First, in presence of recursive procedures, procedure-inlining may not be possible. Second, even if there are no recursive procedures, procedure-inlining may result in an exponential blow-up of the program. For example, if procedure P_1 calls procedure P_2 two times, which in turn calls procedure P_3 two times, then procedure inlining will result in 4 copies of procedure P_3 inside procedure P_1. In general, leaf procedures can be replicated an exponential number of times.

A more standard way to do interprocedural analysis is by means of computing procedure summaries [20]. Each procedure is analyzed once (or a few times in

* Second author supported in part by the National Science Foundation under grant CCR-0326540.

R. De Nicola (Ed.): ESOP 2007, LNCS 4421, pp. 253–267, 2007.

```
   main(){                                    P(){
1    x := 0; y := 1; a := 2; b := 4;          1   if (*) {
2    P(); Assert(y = 2x + 1);                 2       x := x + a;
3    x := 0; y := 0; a := ?; b := 2a;         3       y := y + b;
4    P(); Assert(y = 2x);                     4   }
5    y := x + 3; a := ?; b := a;              5   else P()
6    P(); Assert(y = x + 3);                  6 }
7  }
```

Fig. 1. An example program

case of recursive procedures) to build its summary. A procedure summary can be thought of as some succinct representation of the behavior of the procedure that is also parametrized by any information about its input variables. However, there is no automatic recipe to efficiently construct or even represent these procedure summaries, and abstraction specific techniques are required.

The original formalism proposed by Sharir and Pnueli [20] for computing procedure summaries was limited to finite lattices of dataflow facts. Sagiv, Reps and Horwitz generalized the Sharir-Pnueli framework to build procedure summaries using context-free graph reachability [15], even for some kind of infinite domains. They successfully applied their technique to detect linear constants interprocedurally [17]. However, their generalized framework requires appropriate distributive transfer functions as input - and such transfer functions are not known for any natural abstract domain more powerful than linear constants.

In this paper (Section 3), we describe a general technique for constructing precise procedure summaries. This technique can be effectively used for a useful class of program abstractions (over infinite domains). We apply this technique to obtain precise interprocedural analyses for two useful abstractions - unary uninterpreted functions, and linear arithmetic (which is more powerful than the domain of linear constants used by Sagiv, Reps and Horwitz). The former (described in Section 4) gives a polynomial-time algorithm for a special case of the long-standing open problem of interprocedural global value numbering, while the latter (described in Section 5) yields a new algorithm for interprocedural linear arithmetic analysis with the same complexity as that of the best known deterministic algorithm [11].

Our procedure summaries are in the form of constraints (on the input variables of the procedure) that must be satisfied to guarantee that some appropriate generic assertion (involving output variables of the procedure) holds at the end of the procedure. A generic assertion is an assertion that involves some *context* variables that can be instantiated by symbols (or more formally, by terms with holes) of the underlying abstraction. For example, consider procedure P shown in Figure 1 with input variables x, y, a, b and output variables x, y. $\alpha x + \beta y = \gamma$ is a generic assertion in the theory of linear arithmetic involving variables x, y (and context variables α, β, γ, which denote unknown constants). Using the technique described in this paper, we compute the summary of procedure P as "$\alpha x + \beta y = \gamma$

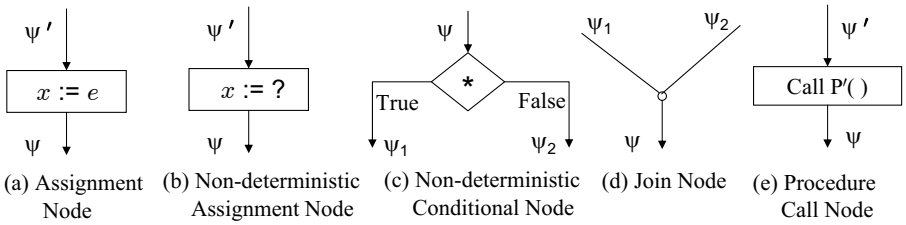

Fig. 2. Flowchart nodes in our abstracted program model

holds at the end of procedure P iff $\alpha a + \beta b = 0 \wedge \alpha x + \beta y = \gamma$ holds at the beginning of procedure P". After computing such a procedure summary for P, we can use it to verify the assertions in the `Main` procedure. To verify the first assertion $y = 2x + 1$, we first match it with the generic assertion $\alpha x + \beta y = \gamma$ to obtain the substitution $\alpha \mapsto -2, \beta \mapsto 1$ and $\gamma \mapsto 1$ for the context variables. We then instantiate the procedure summary with this substitution to obtain the precondition $b - 2a = 0 \wedge y - 2x = 1$. We then check that this precondition is satisfied in procedure `Main` immediately before the first call to procedure P. Similarly, we can verify the other two assertions.

The key idea in computing such procedure summaries is to compute weakest preconditions of generic assertions. However, a naive weakest precondition computation may be exponential in the number of operations performed (each conditional node can double the size of the precondition), and may not even terminate (in presence of loops). Hence we use some techniques for strengthening and simplifying the weakest preconditions (without any loss of precision). This simplification is based on recent connections between unification and assertion checking (described in Section 2.2). For example, consider computing the weakest precondition of the generic assertion $x = \beta y$ in the theory of unary uninterpreted functions for the procedure Q in Figure 3. (Here β represents some unknown sequence of uninterpreted functions.) The naive weakest precondition computation will not terminate and will yield $x = \beta y \wedge fx = \beta fy \wedge ffx = \beta ffy \wedge \ldots$. However, our simplification procedure will simply (and strengthen) the first two conjuncts to $x = \beta y \wedge \beta f = f\beta$, denoting that the relationship $x = \beta y$ holds at the end of procedure only if (β is of the form such that) $\beta f = f\beta$ and $x = \beta y$ holds at the beginning of the procedure. It turns out that the constraints thus obtained $\beta f = f\beta \wedge x = \beta y$ form a fixed-point, and hence our weakest precondition computation terminates immediately.

2 Preliminaries

2.1 Program Model

We assume that each procedure in a program is abstracted using the flowchart nodes shown in Figure 2. In the assignment node, x refers to a program variable while e denotes some expression in the underlying abstraction. We refer to the language of such expressions as *expression language of the program*. Following

are examples of the expression languages for the abstractions that we refer to in this paper:

- Linear arithmetic. $e \ ::= \ y \ | \ c \ | \ e_1 \pm e_2 \ | \ c \times e$
  Here y denotes some variable while c denotes some arithmetic constant.
- Unary Uninterpreted functions. $e \ ::= \ y \ | \ f(e)$
  Here f denotes some unary uninterpreted function.

A non-deterministic assignment $x :=?$ denotes that the variable x can be assigned any value. Such non-deterministic assignments are used as a safe abstraction of statements (in the original source program) that our abstraction cannot handle precisely.

A join node has two incoming edges. Note that a join node with more incoming edges can be reduced to multiple join nodes with two incoming edges.

Non-deterministic conditionals, represented by $*$, denote that the control can flow to either branch irrespective of the program state before the conditional. They are used as a safe abstraction of guarded conditionals, which our abstraction cannot handle precisely. We abstract away the guards in conditionals because otherwise the problem of assertion checking can be easily shown to be undecidable even when the program expressions involves operators from simple theories like linear arithmetic [10] or uninterpreted functions [9]. This is a very commonly used restriction for a program model while proving preciseness of a program analysis for that model.

For simplicity, we assume that the inputs and outputs of a procedure are passed as global variables. Hence, the procedure call node simply denotes the name of the procedure to be called. Also, we assume that we are given the whole program with a special entry procedure called `Main`.

2.2 Unification and Assertion Checking

A *regular* assertion is a conjunction of equalities $e = e'$ between two expressions. A substitution σ is a mapping from variables to expressions. A substitution σ is applied to an expression e (or assertion ψ), by replacing all variables x by $\sigma(x)$ in the expression (assertion). The result is denoted in postfix notation by $e\sigma$ (or $\psi[\sigma]$). A program state is a substitution on program variables. A regular assertion ψ is said to hold at a program point π if $\psi[\sigma]$ is valid (in the underlying theory) for every program state σ reached at π (along any path).

A substitution σ is a *unifier* for ψ if $\psi[\sigma]$ is valid. A substitution σ_1 is *more-general* than a substitution σ_2 if there is a substitution σ_3 s.t. $x\sigma_2 = x\sigma_1\sigma_3$ for all x. A theory is *unitary* if for all equalities $e = e'$ in that theory, there exists a unifier that is more-general than any other unifier of $e = e'$. A substitution σ can be treated as the formula $\bigwedge_x x = \sigma(x)$. For a unitary theory $\mathbb{T}$, we denote the conjunction representing the most-general unifier for ψ by $\texttt{Unif}_{\mathbb{T}}(\psi)$.

The formula $\texttt{Unif}(\psi)$ logically implies ψ, but it is, in general, not equivalent to ψ. Since it is often "simpler" than ψ, we may wish to replace ψ by $\texttt{Unif}(\psi)$. The basic result formally stated in Property 1 is that, *in many useful abstractions*, the formulas ψ and $\texttt{Unif}(\psi)$ are "equivalent" as far as invariance of assertions is concerned.

Property 1 ([5]). Let π be any location in a program that is specified using the flowchart nodes in Figure 2 and expressions from some unitary theory $\mathbb{T}$. An equality $e = e'$ holds at π iff $\texttt{Unif}_{\mathbb{T}}(e = e')$ holds at π.

The above property is stated and proved in [5]. The key insight is that *runs* of a program are just substitutions and if every run validates an assertion, then every run should also validate a more-general unifier of that assertion. Property 1 is used at two places in our generic weakest-precondition computation based technique for interprocedural analysis: (a) for simplification of formulas for efficiency purpose (Section 3.2), (b) for detecting fixed-point computation (Section 3.2).

Note that we present our results in the context of unitary theories for efficiency reasons; otherwise both Property 1 and our general approach of Section 3 can be generalized.

3 General Technique for Interprocedural Analysis

Our technique for interprocedural analysis uses the standard two phase summary-based approach. The two phases are described in Section 3.2 and Section 3.3.

3.1 Generic Assertions

A *generic* assertion is an assertion that involves context-variables apart from regular program variables. A context-variable represents some unknown term with holes, with the constraint that this unknown term does not involve any program variables (i.e., it only involves symbols from the underlying theory or abstraction). An important consequence of this constraint is that generic assertions are closed under weakest precondition computation across assignments to program variables.

We say that a generic assertion A_1 is more general than another generic assertion A_2 if there exists an instantiation σ of the context variables of A_1 such that $A_2 = A_1[\sigma]$. We define a set of generic assertions to be *complete* w.r.t. a given set of program variables V if for any generic assertion A_1 in the underlying theory involving program variables V, there exists a generic assertion A_2 in the set such that A_2 is more general than A_1.

For the theory of linear arithmetic, the singleton set $\{\sum_i \alpha_i x_i = \alpha\}$ constitutes a complete set of generic assertions with respect to the set of variables $\{x_i\}_i$. Here α, α_i denote unknown constants. For the theory of unary uninterpreted functions, the set $\{\alpha x_1 = \beta x_2 \mid x_1, x_2 \in V,\ x_1 \not\equiv x_2\}$ is a complete set of generic assertions with respect to the set of variables V. Here α, β represent unknown strings (applications) of unary uninterpreted functions.

3.2 Phase 1: Computing Procedure Summaries

Let P be a procedure with V as the set of its output variables. Let G be some complete set of generic assertions with respect to V for the underlying abstraction. The summary of procedure P is a collection of formulas ψ_i, one for each generic assertion A_i in G. The formula ψ_i is the weakest precondition of the

generic assertion A_i denoting that the generic assertion A_i holds at the end of procedure P only if the formula ψ_i holds at the beginning of procedure P. Each formula ψ_i itself is a conjunction of generic assertions. (Observe that weakest precondition computation involves substitution of regular variables by program expressions and performing conjunctions of formulas. Hence, conjunctions of generic assertions are closed under weakest precondition computation.)

Computing summary for procedure P requires computing the weakest precondition of each generic assertion in G one by one. The weakest precondition of a given generic assertion A across a procedure is computed by computing a formula ψ at each procedure point using the following transfer functions across flowchart nodes. The correctness of the following transfer functions is immediate.

Initialization: The formula at all procedure points except the procedure exit point is initialized to *true*. The formula at the exit is initialized to the generic assertion A.

Assignment Node: See Figure 2(a). The formula ψ' before an assignment node $x := e$ is obtained from the formula ψ after the assignment node by substituting x by e in ψ, i.e. $\psi' = \psi[x \mapsto e]$.

Non-deterministic Assignment Node: See Figure 2(b). The formula ψ' before a non-deterministic assignment node $x :=?$ is obtained from the formula ψ after the non-deterministic assignment node by universally quantifying out the variable x. However, for the case when program expressions come from a unitary theory, we can simplify $\forall x(\psi)$ to $\psi[x \mapsto c_1] \wedge \psi[x \mapsto c_2]$, where c_1 and c_2 are two distinct constants (or provably unequal terms) in the underlying theory.

Non-deterministic Conditional Node: See Figure 2(c). The formula ψ before a non-deterministic conditional node is obtained by taking the conjunction of the formulas ψ_1 and ψ_2 on the two branches of the conditional, i.e., $\psi = \psi_1 \wedge \psi_2$.

Join Node: See Figure 2(d). The formulas ψ_1 and ψ_2 on the two predecessors of a join node are same as the formula ψ after the join node, i.e., $\psi_1 = \psi$ and $\psi_2 = \psi$.

Procedure Call Node: See Figure 2(e). Let $\psi \equiv \bigwedge_{i=1}^{k} A'_i$. Let $A_i \in G$ be such that A_i is more general than A'_i and let σ_i be the instantiation such that $A'_i = A_i[\sigma_i]$. Let ψ'_i be the formula in the summary of procedure P' that represents the weakest precondition of A_i before procedure P'. Then, $\psi' = \bigwedge_{i=1}^{k} \psi'_i[\sigma_i]$.

Simplification

Property 1 says that we do not need to distinguish between two regular assertions that have the same set of unifiers. We can generalize this to generic assertions. We say two formulas (conjunctions of generic assertions) ψ and ψ' are *essentially equivalent*, denoted by $\psi \rightleftharpoons \psi'$, if $\psi\sigma$ and $\psi'\sigma$ have the same set of unifiers for *every* substitution σ that assigns every context variable in ψ, ψ' to a term with a hole (in the signature of the underlying theory). We denote by $\psi \rightharpoonup \psi'$ the fact that every unifier of $\psi\sigma$ is also a unifier of $\psi'\sigma$ (for every σ).

We can simplify ψ at any program point by replacing it by another essentially equivalent formula ψ'. The soundness and completeness of this transformation follows from Property 1. This simplification is needed to bound the size of the formula ψ because otherwise a naive computation of weakest precondition may lead to an exponential blowup in the number of operations performed. In case of linear arithmetic, this simplification simply involves removing linearly dependent equations. In case of unary uninterpreted functions, this simplification involves strengthening the formula.

Observe that the number of conjuncts in the formula computed before any node (in particular the procedure call node) is at most quadratic in the maximum number of conjuncts in any simplified formula. Hence, the time required to simplify any such formula can be bounded by $T_{\mathbb{T}}(k)$, which is as defined below.

Definition 1 (Simplification Cost $T_{\mathbb{T}}(k)$). *For any theory $\mathbb{T}$, let $S_{\mathbb{T}}(k)$ denote the maximum number of conjunctions (of generic assertions) in any simplified formula over k program variables. Let $T_{\mathbb{T}}(k)$ denote the time required to simplify a formula over k program variables with at most $(S_{\mathbb{T}}(k))^2$ generic assertions.*

Fixed-Point Computation

In presence of loops (inside procedures as well as in call-graphs), we iterate until fixed-point is reached. The standard way to perform such an iteration is to maintain a worklist that stores all program points whose formulas have changed with respect to the formulas in the previous iteration, but whose change has not yet been propagated to its predecessors.

Let ψ be the formula computed at some program point π, and let ψ' be the formula at π in the previous iteration. If ψ and ψ' are logically equivalent, then it is intuitive that the formula at π has not changed from the previous iteration (and hence does not require any further propagation to the predecessors of π). However, it follows from Property 1 that we can strengthen this notion to conclude that the formula at π has not changed even if $\psi \rightleftharpoons \psi'$. This observation is important because it allows to detect fixed-point faster. In case of unary uninterpreted functions, this makes significant difference (E.g., for the loop in procedure Q in Figure 3, fixed-point is not even reached with the former intuitive notion of change, while it is reached in 2 steps with the latter stronger notion of change, as explained on Page 255). The number of times the formula ψ at each point inside a procedure gets updated is bounded by the *maximum unifier chain length* of the underlying theory as defined below.

Definition 2 (Maximum Unifier Chain Length $M_{\mathbb{T}}(k)$). *We define the* maximum unifier chain length *of any theory $\mathbb{T}$ for k variables, denoted by $M_{\mathbb{T}}(k)$, to be the maximum length of any chain $\psi_1, \psi_2, \ldots$ (where each ψ_i is a conjunction of generic assertions over k variables) such that $\psi_i \rightharpoonup \psi_{i+1}$ but $\psi_{i+1} \not\rightharpoonup \psi_i$.*

Computational Complexity

The number of updates performed during phase 1 is bounded above by $n \times M_{\mathbb{T}}(k)$, where n is the total number of program points and k is the maximum

number of program variables that are live at any program point (This follows from Definition 2). The cost of each update is bounded above by $T_{\mathbb{T}}(k)$. Hence, the cost of Phase 1 is $O(n \times M_{\mathbb{T}}(k) \times T_{\mathbb{T}}(k))$.

3.3 Phase 2: Using Procedure Summaries

We now show how to use the procedure summaries computed in phase 1 to verify and discover assertions at different program points. The correctness of this phase is easy to observe, while its computational complexity is bounded above by that of phase 1.

Verifying a given assertion at a given program point. For this purpose, we can perform the weakest precondition computation of the given assertion as in Phase 1. However, there are two main differences. The formula computed at each program point is a regular assertion instead of a generic assertion. Secondly, the preconditions computed at the beginning of the procedures are copied before the call sites of those procedures. When the process reaches a fixed-point, we declare the assertion to be true iff the precondition computed at the beginning of `Main` procedure is `true`.

Computing all invariants at a given program point. Instead of computing the weakest precondition of a given assertion at a program point π (as described above), we can also compute the weakest preconditions of a complete set of generic assertions. The preconditions obtained at the beginning of `Main` procedure for each of these generic assertions will be in the form of constraints on the context variables. These constraints exactly characterize the invariants that hold at π.

Computing all invariants at all program points. We can repeat the above process for all program points to compute all invariants at all program points. However, when the expression language of the program comes from a unitary theory (e.g., linear arithmetic and uninterpreted functions), we can perform a more efficient analysis based on a forward intraprocedural analysis for that abstract domain. For this purpose, we simply run a forward intraprocedural analysis on each procedure. The invariant at the entry point of `Main` procedure is initialized to `true`, while for all other procedures, it is obtained as the join of the invariants before all call sites of that procedure. We only need to describe the transfer function for the procedure call node. Let F be the invariants computed before the procedure call node. Let $\sigma = \texttt{Unif}(F)$ be the substitution representing the most-general unifier of F. (Note that unitary theories have a single most-general unifier). Let V be the set of variables that do not have a definition in σ, but are the inputs to procedure P. Let the summary of procedure P be: "the assertion ψ_i holds at the end of procedure iff the constraints ψ'_i hold at the beginning of procedure" (for all generic assertions ψ_i from some complete set G). The transfer function for the procedure call node then is: $F'_i = \bigwedge_i \texttt{Normalize}(\forall V \psi'_i[\sigma], \psi_i)$.

The key idea here is to instantiate each of the constraints ψ'_i with σ and universally quantify out the remaining input variables V (by using the same technique described in weakest precondition computation across non-deterministic

assignment nodes). There is no precision loss in quantifying out V since, by assumption, there are no invariants on V. The resulting constraints on context variables describe all relationships of the form ψ_i that hold among the output variables of procedure P after the procedure call node. The function `Normalize` translates these constraints into the desired invariants. `Normalize`(C, ψ_i) takes as input some constraints C on the context variables corresponding to some generic assertion ψ_i and returns the assertions obtained by eliminating the context variables. (Eg., `Normalize`$(a + b = 0 \wedge c - d = 0, ax + by + cz = d)$ returns $x = y \wedge z = 1$, which is obtained by eliminating a, b, c, d from $\forall a, b, c, d(a + b = 0 \wedge c - d = 0 \Rightarrow ax + by + cz = d)$).

4 Unary Uninterpreted Functions

In this section, we instantiate the above general framework for performing interprocedural analysis over the abstraction of unary uninterpreted functions. As a result, we obtain a PTIME algorithm for computing all equality invariants when the program is specified using the flowchart nodes described in Figure 2, and the expression language of the program involves unary uninterpreted functions.

Unary uninterpreted functions can be used to model fields of structures and objects in programs, as well as deterministic function calls with one argument– this is useful when the function body is unavailable or is too complicated to analyze. Yet another motivation for studying the unary uninterpreted abstraction comes from the long-standing open problem of interprocedural global value numbering. This problem seeks to analyze programs whose expression language contains uninterpreted functions of *any* arity. A brief history of this problem is given in Section 6. The results in this section, thus, make progress toward solving this open problem.

Apart from the general ideas mentioned in Section 3, our results in this section also rely on another key idea of representing large strings succinctly via singleton context-free grammars [13].

Notation. Terms constructed using unary function symbols can be represented as strings. For example, the term $f(g(x))$ can be treated as the string fgx. The expressions $f(\_)$ and $f(g(\_))$, (respectively strings f and fg) are terms with a hole $\_$. Variables that take terms with a hole as values, or equivalently context variables, will be denoted by α, β, etc. The concrete terms with holes are denoted by C, D, E, F with suitable annotations.

4.1 Simplification

We compute procedure summaries by backward propagation of all the generic assertions in the set $\{\alpha x_1 = \beta x_2 \mid x_1, x_2 \in V,\ x_1 \not\equiv x_2\}$, where V is the set of output variables of the corresponding procedure. The assertions generated in the process are simplified to one of the following forms:

$$(1)\ \alpha C x_i = \beta C' x_j \qquad (2)\ \alpha C \alpha^{-1} = \beta C' \beta^{-1} \qquad (3)\ \alpha = \beta C$$

```
P(){                          Q(){                        main(){
1  x := fgx;                  1  while (*) {              1  y := a;
2  y := gfy;                  2     x := fx;             2  x := fa;
3  if (*) { Q(); }            3     y := fy;             3  P();
4  else { P(); }              4  }                        4  assert(x = fy);
5 }                           5 }                         5 }
```

Fig. 3. Program

Thus, every ψ is simply a conjunction of assertions of these forms. The *inverse* operator, $^{-1}$, satisfies the intuitive axioms: $(\alpha\beta)^{-1} = \beta^{-1}\alpha^{-1}$, $\alpha\alpha^{-1} = \epsilon$, and $(\alpha^{-1})^{-1} = \alpha$.[1] The strings C, C' in Form 2 are allowed to contain the inverse operator, whereas strings C, C' in Form 1 and Form 3 do not contain the inverse operator. Equations of Form 2 are an elegant way of encoding constraints on the context-variables α and β that are generated by the backward analysis.

We show now that weakest precondition computation across the various program nodes maintains assertions in one of these forms. We consider the case of a Procedure Call node "Call P()" (the other cases are easy to verify). At any stage of the fixpoint computation, the (partially computed) summary of a procedure P will be given as: "$\alpha' x_i = \beta' x_j$ holds at the end of procedure P if ψ''_{ij} holds at the beginning" for each pair $x_i, x_j \in V$. Equations of Form 2 and Form 3 are unchanged in the weakest precondition computation. The weakest precondition of an equation $\alpha C x_i = \beta C' x_j$ is obtained by instantiating ψ''_{ij} by $\{\alpha' \mapsto \alpha C, \beta' \mapsto \beta C'\}$. Applying this replacement in equations of Form 1 or Form 2 in ψ''_{ij} gives back equations of the same form. When applied on equations of Form 3, we get equations of the form $\alpha C = \beta C'$. We remove the largest common suffix of C, C' and if the equation does not reduce to Form 3, then the weakest precondition is *false*.

Bounding the size of ψ. We will show that any conjunction of equations of Form 1, Form 2, and Form 3 over k variables can be simplified to contain *at most* $k(k-1)/2 + 1$ equations. Specifically,

- for each pair x_i, x_j of variables, there is at most one equation of Form 1; and
- either there is at most one equation of Form 2, or there is at most one equation of Form 3.[2]

The Simplification procedure uses unification to simplify the equations and keeps the result *essentially equivalent* to the original set. It performs two main steps. For a fixed pair x, y of variables, let ψ_{xy} denote the set containing all equations of Form 1 in ψ. First, by repeated use of Lemma 1 ψ_{xy} is simplified to a set containing at most one equation of Form 1 and either one equation of Form 3 or

[1] Note that the inverse operator implicitly builds in simplification using unification. For instance, while $fx = fy$ does not logically imply $x = y$, using the inverse axioms we have $fx = fy \Rightarrow f^{-1}fx = f^{-1}fy \Rightarrow x = y$.

[2] Note that an equation of Form 3 essentially gives a concrete solution, since we can assume, by Property 1, that one of α, β is ϵ.

Ite	Proc	Current Summary for $\alpha x = \beta y$	Comment
0	P, Q	*true*	Init
1	Q	Simp$(\alpha x = \beta y, \alpha f x = \beta f y) = (\alpha x = \beta y, \alpha f \alpha^{-1} = \beta f \beta^{-1})$	
2	P	$\alpha f g x = \beta g f y, \alpha f \alpha^{-1} = \beta f \beta^{-1}$	Use Q's summary
3	Q	$\alpha x = \beta y, \alpha f \alpha^{-1} = \beta f \beta^{-1}$	fixpoint for Q
4	P	Simp$(\alpha f g f g x = \beta g f g f y, \alpha f g x = \beta g f y, \alpha f \alpha^{-1} = \beta f \beta^{-1})$	Use P's summary
5	P	$\alpha f = \beta, \alpha f g x = \beta g f y$	fixpoint for P

Fig. 4. This figure illustrates summary computation for interprocedural analysis over the unary abstraction. In Column 3, the summary consists of the constraints that must hold at the beginning of the procedure P/Q for $\alpha x = \beta y$ to be an invariant at the end of the procedure.

finitely many equations of Form 2. For example, in iteration 2 of Figure 4, the set of equations $\{\alpha x = \beta y, \alpha f x = \beta f y\}$ is simplified to $\{\alpha x = \beta y, \alpha f \alpha^{-1} = \beta f \beta^{-1}\}$.

Lemma 1. *The equation set* $\{\alpha C_i x = \beta C_i' y : i = 1, 2\}$ *either has no solutions, or it has the same solutions as a set containing either one of these two equations and at most one equation of Form 2 or Form 3.*

Next, if there is an equation of Form 3 then it can be used to simplify an equation of Form 2 to either *false* or *true*. Otherwise, a set $\{\alpha C_i \alpha^{-1} = \beta C_i' \beta^{-1}, i = 2, \ldots, k\}$ containing multiple equations of Form 2 is simplified by repeated use of Lemma 2.

Lemma 2. *The equation set* $\{\alpha C_i \alpha^{-1} = \beta C_i' \beta^{-1}, i = 1, 2\}$ *is either unsatisfiable, or has the same solutions as a set containing at most one equation of either Form 2 or Form 3.*

For example, in iteration 4-5 of Figure 4, $\{\alpha f \alpha^{-1} = \beta f \beta^{-1}, \alpha f g \alpha^{-1} = \beta g f \beta^{-1}\}$ is simplified to $\{\alpha f = \beta\}$. In this way, any conjunction ψ of equations of Form 1, Form 2, and Form 3 is simplified to a conjunction with at most $k(k-1)/2 + 1$ equations.[3]

The algorithms used in the proof of Lemma 1 and Lemma 2 use a constant number of string operations. Assuming the basic string operations take time T_{base}, the time taken to simplify $S_{\text{uu}}(k)^2 = O(k^4)$ assertions is $O(k^4 T_{\text{base}})$.

Maximum Unifier Chain Length. It is easy to see that the maximum unifier chain length for k variables is bounded by $k(k-1)/2 + 2$. This is because the number of equations in ψ can increase only $k(k-1)/2 + 1$ times, and beyond that the formula either becomes unsatisfiable, or it is forced to have a unique solution for its variables. Note that it is not possible for the number of equations to remain the same and the formula to get stronger. This is a consequence of Lemma 1.

[3] The observation that we need to keep only a small number of equations $Cx_i = \alpha C' x_j$ intuitively means that we keep only a few runs. However, these runs in the *simplified* formula may not correspond to any real runs, but some equivalent hypothetical runs.

Hence, for the case of unary uninterpreted (uu) abstraction, we have:

$$S_{\text{uu}}(k) = \frac{k(k-1)}{2} + 1 \qquad T_{\text{uu}}(k) = O(k^4 T_{\text{base}}) \qquad M_{\text{uu}}(k) = \frac{k(k-1)}{2} + 2$$

4.2 Computational Complexity: Efficient Representations

We note that the time complexity of interprocedural analysis for the unary uninterpreted abstraction is polynomial *assuming* that the string operations can be performed efficiently. However, the length of strings can be *exponential* in the size of the program, as the following example shows.

Example 1. Consider the n procedures $P_0, \ldots, P_{n-1}$ defined as

$$P_i(x_i)\ \{\ t := P_{i-1}(x_i);\ y_i := P_{i-1}(t);\ \mathit{return}(y_i);\}$$
$$P_0(x_0)\ \{\ y_0 := f x_0;\ \mathit{return}(y_0);\}$$

The summary of procedure P_i is: $y_i = \alpha x_i$ iff $\alpha = f^{2^i}$.

Hence, if we use a naive (explicit) representation, the size of ψ can grow exponentially (when we apply substitutions during transfer function computation across procedure call nodes). Instead we appeal to shared representation of strings using *singleton context-free grammars* (SCFG). An SCFG is a context-free grammar where each nonterminal represents *exactly* one (terminal) string. An SCFG can represent strings in an exponentially succinct way. The strings C_i's that arise in the equations can be represented succinctly using SCFGs in size that is linear in the size of the program (because the program itself is an implicit succinct representation of these strings using SCFGs).

Example 2. Following up on Example 1, we note that the string f^{2^n} can be represented by the SCFG with start symbol A_n and productions $\{A_{i+1} \rightarrow A_i A_i \mid 1 \leq i \leq n\} \cup \{A_0 \rightarrow f\}$. In particular, the summaries of the procedures can be represented as: $y_i = \alpha x_i$ iff $\alpha = A_i$.

A classic result by Plandowski [13] shows that equality of two strings represented as SCFGs can be checked in polynomial time. Apart from this, the simplification procedure implicit in the proofs of Lemma 1 and Lemma 2 require largest common prefix/suffix computation and substring extraction. It is an easy exercise to see that these string operations can also be performed on SCFG representations in polynomial time. Hence, the computational procedure outlined above can be implemented in polynomial time using the SCFG representation of strings. In conclusion, this shows that summaries can be computed in PTIME on the abstraction of unary symbols. We remark here that Plandowski's result has been generalized to trees [19] suggesting that it may be possible to generalize our result to the interprocedural global value numbering problem (over binary uninterpreted functions).

5 Linear Arithmetic

The technique described in Section 3.2 can also be used effectively to compute procedure summaries for the abstraction of linear arithmetic. We compute the weakest precondition of the generic assertion $\alpha_1 x_1 + \cdots + \alpha_k x_k = \alpha$ (which constitutes a complete set by itself) where $x_1, \ldots, x_k$ are the output variables of the corresponding procedure.

The conjunction ψ of equations thus obtained at any point in the procedure during the weakest precondition computation can be seen as *linear* equations *over* the $k^2 + k + 1$ variables: k^2 variables representing the products $\alpha_i x_j$ and the $k + 1$ variables α_i and α. We can simplify the equations thus obtained by maintaining only the linearly independent (non-redundant) equations. We know that there can not be more than $k^2 + k + 1$ linearly independent equations and hence ψ can have at most $k^2 + k + 1$ equations. This shows that for the linear arithmetic (la) abstraction,

$$S_{\text{la}}(k) = k^2 + k + 1 \qquad T_{\text{la}}(k) = O(T_{\text{base}} k^8) \qquad M_{\text{la}}(k) = k^2 + k + 1,$$

where T_{base} denotes the time to perform an arithmetic operation. Since constants can become large (programs can encode large numbers succinctly), we use modulo arithmetic and randomization to get a true PTIME procedure, as in [11].

Müller-Olm and Seidl also gave a precise interprocedural algorithm for linear arithmetic of similar complexity [11]. However, their algorithm is different and is based on the the observation that *runs* of a procedure correspond to linear transformations and there can be only quadratic many linearly-independent transformations. In a certain sense, this is the *dual* of our approach.

6 Related Work and Discussion

Forward vs. Backward Analysis. The approach presented in this paper for computing procedure summaries is based on backward propagation of generic assertions. It is presently unclear how the dual approach, namely forward propagation of a complete set of generic assertions, can be effectively used. A forward propagation involves developing context-sensitive or distributive transfer functions for assignment nodes (usually involves existential quantifier elimination) and join nodes. Giving a general procedure for such operations appears to be hard for regular assertions (intraprocedural case) and would be significantly more difficult for generic assertions.

Nevertheless, these difficulties may be overcome for very specific abstractions, such as linear arithmetic [11,8]. In this case, the authors essentially look at a procedure as a linear transformation and compute in the $(k+1)^2$-dimensional vector space of these linear transformations. This allows them to perform abstract interpretation using either backward or forward analysis [11,8]. However, this general approach of developing interprocedural analysis by describing program behaviors as transformations (in a finite dimensional vector space) is applicable only on arithmetic abstractions. In contrast, our approach promises to be simpler, and more generally applicable.

Weakest Precondition of Generic Assertions vs. Regular Assertions. To ensure termination of weakest precondition computation over generic assertions, we used some connections between unification and assertion checking. Similar connections have been used earlier for weakest precondition computation for regular assertions in the intraprocedural case [5,6]. However, in the intraprocedural case, we just need to solve unification problems over regular assertions. These problems are well-studied and efficient algorithms are known for several theories. In the interprocedural case, we now have to solve unification problems over *generic* assertions. In the theorem proving community, these are studied under the name of "second-order unification" and "context unification". These problems are known to be more difficult than their first-order counterparts. Thus, while our approach of backward analysis based on generic assertions provides a uniform framework for developing interprocedural analyses, it also helps to ascertain the difficulty of interprocedural analysis over intraprocedural analysis by drawing connections with the complexity of second-order unification vs. standard unification in theorem proving. Templates, which are similar to generic assertions, have been used to generate invariants, but only in the context of intraprocedural analysis and without any completeness guarantees [18].

History of Global Value Numbering. Since checking equivalence of program expressions is an undecidable problem, in general, program operators are commonly abstracted as uninterpreted functions to detect expression equivalences. This form of equivalence is also called *Herbrand equivalence* [16] and the process of discovering it is often referred to as *value numbering*. Kildall [7] gave the first intraprocedural algorithm for this problem based on performing abstract interpretation [2] over the lattice of Herbrand equivalences in exponential time. This was followed by several PTIME, but imprecise, intraprocedural algorithms [1,16,3]. The first PTIME intraprocedural algorithm was given by Gulwani & Necula [4], and then by Müller-Olm, Rüthing, & Seidl [9]. However, PTIME interprocedural global value numbering algorithm has been elusive. There are some new results, but only under severe restrictions that functions are side-effect free and one side of the assertion is a constant [12]. Neither of these assumptions is satisfied by the program in Figure 3. The technique described in this paper yields a PTIME algorithm for the special case of unary uninterpreted functions.

7 Conclusion

Proving non-trivial properties of programs requires analyzing programs over rich abstractions. The scalability of such program analyses depends upon the possibility of constructing efficient and precise summaries of procedures over such abstractions. In this paper, we have described a new technique for computing procedure summaries for a class of program abstractions over infinite domains, thereby adding to some limited piece of work known in this area.

In the description of our technique, we assume at some places that conditionals are non-deterministic and expression language of the program comes from a

unitary theory. These assumptions are needed to prove that our technique computes the most precise procedure summary in an efficient manner. We believe that the general ideas in our technique can be extended to reason about predicates in conditionals and handle expressions that are not from a unitary theory (e.g., as suggested in [6]), albeit with some (unavoidable) precision loss because the problem is undecidable in general.

References

1. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *15th Annual ACM Symposium on POPL*, pages 1–11, 1988.
2. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM Symposium on POPL*, pages 234–252, 1977.
3. K. Gargi. A sparse algorithm for predicated global value numbering. In *PLDI*, volume 37, 5, pages 45–56. ACM Press, June 17–19 2002.
4. S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. In *Static Analysis Symposium*, volume 3148 of *LNCS*, pages 212–227, 2004.
5. S. Gulwani and A. Tiwari. Assertion checking over combined abstraction of linear arithmetic & uninterpreted functions. In *ESOP*, volume 3924 of *LNCS*, Mar. 2006.
6. S. Gulwani and A. Tiwari. Assertion checking unified. In *Proc. VMCAI*, LNCS 4349. Springer, 2007. See also Microsoft Research Tech. Report MSR-TR-2006-98.
7. G. A. Kildall. A unified approach to global program optimization. In *1st ACM Symposium on POPL*, pages 194–206, Oct. 1973.
8. M. Müller-Olm, M. Petter, and H. Seidl. Interprocedurally analyzing polynomial identities. In *STACS*, volume 3884 of *LNCS*, pages 50–67. Springer, 2006.
9. M. Müller-Olm, O. Rüthing, and H. Seidl. Checking Herbrand equalities and beyond. In *VMCAI*, volume 3385 of *LNCS*, pages 79–96. Springer, Jan. 2005.
10. M. Müller-Olm and H. Seidl. A note on Karr's algorithm. In *31st International Colloquium on Automata, Languages and Programming*, pages 1016–1028, 2004.
11. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *31st ACM Symposium on POPL*, pages 330–341, Jan. 2004.
12. M. Müller-Olm, H. Seidl, and B. Steffen. Interprocedural Herbrand equalities. In *ESOP*, volume 3444 of *LNCS*, pages 31–45. Springer, 2005.
13. W. Plandowski. Testing equivalence of morphisms on context-free languages. In *Algorithms - ESA '94*, volume 855 of *LNCS*, pages 460–470. Springer, 1994.
14. T. Reps. On the sequential nature of interprocedural program-analysis problems. *Acta Informatica*, 33(8):739–757, Nov. 1996.
15. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd ACM Symposium on POPL*, pages 49–61, 1995.
16. O. Rüthing, J. Knoop, and B. Steffen. Detecting equalities of variables: Combining efficiency with precision. In *SAS*, volume 1694 of *LNCS*, pages 232–247, 1999.
17. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *TCS*, 167(1–2):131–170, 30 Oct. 1996.
18. S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using grbner bases. In *POPL*, pages 318–329, 2004.
19. M. Schmidt-Schauß. Polynomial equality testing for terms with shared substructures. Technical Report 21, Institut für Informatik, November 2005.
20. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.