Interprocedural type propagation for object-oriented languages

J.M. Larchevêque
INRIA*
BP 105, 78153 Le Chesnay Cedex. France
e-mail: jml@minos.inria.fr

Abstract

This paper presents a flow-sensitive interprocedural method for type propagation in an object-oriented language. The primary goal of this method is to obtain a precise call graph in the presence of late binding for function names. Thus, it can be viewed as a preliminary step for interprocedural constant propagation and/or procedure integration in an object-oriented language. It uses a new efficient form of symbolic interpretation in order to limit the amount of *intra*procedural analysis required to a single pre-pass over each function. The cost of both this pre-pass and the interprocedural propagation itself is linear in the program size. Furthermore, the output of symbolic interpretation lends itself to efficient incremental computation and can be reused for other tasks, such as constant propagation or code motion.

1 Introduction

1.1 Motivation

Late binding of function names is a crucial feature of object-oriented languages. It consists in binding a function name to an implementation at call time based on the type of a distinguished argument called the *receiver*. The set of functions whose name is thus overloaded is called a *method*. An ordering over types is specified by the programmer, and the type specified for a variable in the program text (its *static type*) is an upper bound on its actual (or *dynamic*) type. When only the static type t of the receiver is known at a call site for a method m, any implementation of m attached to t or one of its subtypes must be considered callable. Therefore, unless dynamic types are somehow inferred before building the call graph of a program, late binding will induce imprecision in interprocedural analysis and unduly inhibit procedure integration (i.e. in-lining). Furthermore, when the dynamic type of a receiver can be determined statically, a method call can be replaced by an ordinary function call, which can be considerably more efficient.

*Part of the research presented here was carried out in the Altair consortium. A preliminary version appeared as Altair report 64-90-V1 [Lar91].

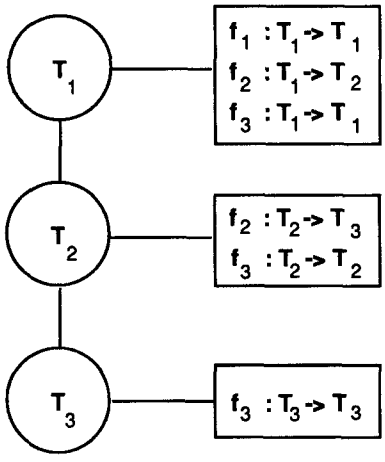
This paper proposes an efficient method for interprocedural object-oriented type propagation which supports recursion, side-effects and aliasing. It is based on the solution of standard bit-vector data flow problems and a novel form of symbolic interpretation.

While this method was designed with optimization in mind, it can be used, with a minor variation (Section 6.2.1), for type-checking a language with optional variable declarations.

The class of languages amenable to the method described is fairly large. However, it is important to note that it requires the types of method implementations to be declared. Consequently, languages with no mandatory declarations at all, like standard Smalltalk, cannot easily be handled by our algorithm. Thus, mandatory function declarations appear as the price to pay for efficient type inference. Note that, without such declarations, object-oriented type-checking in the presence of recursion is an undecidable problem, as shown in [AKW90].

1.2 Example

To illustrate motivations and desirable characteristics for object-oriented type propagation, consider the fragment of a type hierarchy and the two functions in C-like code in Figure 1.1.



body $f_1[T_1](r) : T_1 \rightarrow T_1$ {
    ⋮ /* code leaving r untouched */
    return $f_2(r)$;
}

body $f_3[T_1](r) : T_1 \rightarrow T_1$ {
    $x, y : T_1$;
    ⋮
    $x \leftarrow$ **new**$(T_2, \ldots)$;
    $y \leftarrow f_3(f_1(x))$;
    ⋮
}

Figure 1.1 Example type hierarchy and function bodies.

The diagram in Figure 1.1 represents for each type the functions attached to it and their declared types. For example, there is a method named f_3 which has 3 implementations, attached respectively to T_1, T_2 and T_3. This means that when the single argument passed in a call to f_3 —which is the receiver— has dynamic type T_2 for example, then the implementation attached to T_2 will be executed. We will note $f[T]$ the implementation of method f attached to type T. Method f_1, on the contrary, has one implementation, which is *inherited* by T_2 and T_3, meaning that $f_1[T_1]$ is called when the receiver is of type T_2 or T_3[1].

[1] An equivalent way of putting it is to say that $f_1[T_1] = f_1[T_2] = f_1[T_3]$

Now consider in Figure 1.1 the implementation of f_3 that is attached to T_1, i.e. the function $f_3[T_1]$, and suppose we want to determine possible dynamic types for y. The primitive **new** creates and initializes an object of the type passed as its first argument. So, after the assignment to x, the dynamic type of x can be inferred to be T_2, from which it follows (somewhat trivially) that the implementation of f_1 called at the next instruction is $f_1[T_1]$ (through inheritance). Then, considering the type declared for $f_1[T_1]$, namely $T_1 \rightarrow T_1$, we can infer that an upper bound on the set of possible types for $f_3(f_1(x))$, and therefore for y after the assignment, is T_1, because the type of $f_3[T_1]$ is $T_1 \rightarrow T_1$. However, this could be improved upon if, while analyzing $f_3[T_1]$, we could use information on the bodies of the functions that are called, and in particular on the body of $f_1[T_1]$. Thus, integration (in-lining) of $f_1[T_1]$ would reveal that an upper bound for the type of $f_1(x)$ is T_3 rather than T_1. Indeed, after in-lining the body of $f_1[T_1]$, the expression $f_3(f_1(x))$ becomes $f_3(f_2(x))$, with $f_2[T_2]$ of type $T_2 \rightarrow T_3$ and $f_3[T_3]$ of type $T_3 \rightarrow T_3$.

In fact, the method proposed in this paper does not rely on procedure integration, but on a form of symbolic analysis that provides more precise information on the effects of method calls while keeping the analysis of individual functions mostly separate and avoiding any commitment to particular program transformations during the analysis phase. In this particular case, our method will discover that the type of y is T_3, but it is also capable of synthesizing information about several possible calls at sites where procedure integration is not possible.

The example just given illustrates typical opportunities for gaining precision over user-declared types through type propagation. On the one hand, the declaration of $f_1[T_1]$ announces an upper bound of T_1 on the return type, although the actual bound is T_2. Such discrepancies do make sense in so far as declared types are self-documentary features which reflect intended, but not necessary minimal, bounds. On the other hand, there is no specific declaration for $f_1[T_2]$, whose implementation and type declaration are inherited from T_1; thus, only type propagation can determine that passing a receiver of type T_2 to f_1 will produce a result of type T_3.

1.3 Algorithm outline

The algorithm to be described involves propagating upper bounds to the dynamic types of variables[2]. This algorithm consists of the following steps[3]:

Step 1 Build for each function an expression for the value returned in terms of argument values and constant values irrespective of the execution path taken inside the function (i.e. compute a *symbolic expression*).

For example, the symbolic expression for the value returned by $f_1[T_1]$ (Figure 1.1), noted $\dot{f}_1[T_1](\alpha)$, where α is the receiver's value, equals $\dot{f}_2(\alpha)$. The notation $\dot{f}$ refers to the mapping over symbolic values associated with a function f. In the context of type propagation, the symbolic values we are interested in are types. Thus, $\dot{f}_1[T_1] = \lambda\ \alpha.\dot{f}_2(\alpha)$, which we will call a *type function*, maps an input type to the result type of $f_1[T_1]$.

[2]As explained in Section 5, this need not involve any significant precision loss compared with the propagation of type sets.

[3]For an explanation of data flow analysis concepts, refer to the Appendix.

Note that $\dot{f}_2$ is the symbolic mapping associated with *method* f_2 rather than any particular implementation of it. This means that the graph of $\dot{f}_2$ (i.e. the set of pairs $\langle argument, result \rangle$ for $\dot{f}_2$) is a disjoint union of function graphs, namely $graph(\dot{f}_2[T_1]) \cup graph(\dot{f}_2[T_2])$. We will call $\dot{f}_2$ a *type method*, so as to distinguish it from its constituent *type functions* $\dot{f}_2[T_1]$ and $\dot{f}_2[T_2]$.

Step 2 Compute the graph of each type method by solving fixed-point equations.

In the example given, the type function $\dot{f}_1[T_1]$ can be defined as $\lambda\ \alpha.\dot{f}_2(\alpha)$, i.e. in terms of a type method, which itself is necessarily defined in terms of type functions (its graph being a union of type function graphs). Because of this circular dependency, it is desirable to build for a type function a representation which does not involve type methods. Now, the graph of a type function is such a representation and, since an object-oriented program will involve a finite, and comparatively small, set of types, such graphs can be computed at reasonable space and time cost.

Graphs for type methods are initialized using function declarations, and iteratively refined using a worklist algorithm (Section 4). The point of using fixed point iteration is its capacity to handle recursion.

Step 3 Compute symbolic expressions for receivers at call sites and use the function graphs built in step 2 to evaluate these expressions. Then infer sets of possible function calls so as to obtain a precise call graph.

Part of the output of step 1 can be used to build the symbolic expressions needed. These expressions are used in lieu of more conventional intraprocedural propagation techniques.

The next section defines a property of instances of the *intra*procedural type-propagation framework which we call *cyclic k-boundedness*. This property is necessary for the analysis of the *inter*procedural propagation algorithm, which is carried out respectively in Section 3, on symbolic interpretation, Section 4, on fixed-point computation of graphs for symbolic functions, and Section 5, on the computation of receiver types at call sites. The last two sections give concluding remarks and compare the results with related works.

2 Data flow framework for type propagation

This section defines type propagation as a strictly intraprocedural problem, in which all that is known about called functions is their declared types. It is shown that the framework for solving this problem has a property which we call *cyclic k-boundedness* and that this property is preserved after step 2 of the algorithm has replaced declared function types by inferred types.

This section describes a standard form of intraprocedural type propagation. The algorithm which was outlined in the previous section dispenses with this exhaustive propagation and uses instead a form of symbolic interpretation. However, the symbolic interpretation proposed (Section 3) is tailored to the cyclic bound of a particular problem, hence

the necessity of considering the problem in its standard form in order to determine this cyclic bound[4].

2.1 General description of the intraprocedural framework

The framework for intraprocedural type determination is a tuple $\langle L, \mathcal{F}, \vee \rangle$, where L is a join semilattice with ordering $\leq$ such that $T \leq T'$ if and only if T is a subtype of T', $\vee$ is the lattice join, which maps two elements of L to their closest common supertype, and $\mathcal{F}$ is a set of monotone transfer functions. Note that, under the subtype ordering, smaller means more informative, which is why we use a *join* semilattice, contrary to the convention prevailing in data flow analysis.

Considering that data flow analysis is carried out on intermediate language statements, what follows is valid for most object-oriented languages. Relevant events are assignments[5]. Expressions are built using variables, constants, object creations and method calls. Initially, declared variables are assigned their declaration types and undeclared variables the type *Object* (the lattice $\top$). At each assignment, the type of the right-hand side is computed using function declarations and current type assignments, and assigned to the left-hand side, much as was done in Section 1.2.

It can be shown that the transfer functions thus defined are monotone. Informally, this is due to the fact that the return type and the receiver's type are covariant in the declarations of function types. In other words, if $T_2 \leq T_1$ and the return type declared for $f[T_1]$ is T_i, then the return type declared for $f[T_2]$ is necessarily less or equal to T_i. Therefore, the transfer function for an assignment of the form $x \leftarrow E(y, z)$ (where $E(y, z)$ is an arbitrary expression with input variables y and z) assigns a new type to x monotonically in terms of the types of y and z.

2.2 Boundedness of the function space

2.2.1 k-boundedness

The concept of k-boundedness was introduced in [Tar81] to express a bound on the length of useful execution paths in the presence of loops.

Let F be a monotone function in a join semilattice[6] and $F^{[i]}$ be defined as follows:

$$F^{[i]} = \bigvee_{j=0}^{i} F^j$$

Intuitively, if transfer functions are associated not only with edges, but also with paths, and the ascending chain $\{F^{[i]}\}$ has an upper bound $F^* = \bigvee_{j=0}^{\infty} F^j$, this upper bound is

[4]The framework-dependent character of the symbolic expressions found by our method sets it off from previous approaches and allows to replace problems which —in their full generality— are undecidable by restricted problems solvable in linear time (Section 7).

[5]Where necessary, dummy assignments can be introduced, for example after branching tests or before method calls (see Section 6.2.1).

[6]The original definition supposed a meet semilattice; but switching between the two perspectives might be confusing.

the optimal transfer function for a loop whose body has transfer function F. This is the motivation for the concept of k-boundedness, which can be defined as follows:

Definition 2.1 *A function space $\mathcal{F}$ is k-bounded if and only if, for any F in $\mathcal{F}$, the ascending chain $\{F^{[i]}\}$ admits an upper bound $F^{[k-1]}$.*

It can be proved [Mar89] that, F being monotone, $F^{[k-1]}$ is an upper bound on $\{F^{[i]}\}$ if an only if

$$F^k \leq F^{[k-1]} \tag{2.1}$$

Equation (2.1) is actually the definition given by Tarjan for k-boundedness [Tar81].

2.2.2 Cyclic k-boundedness

For the purpose of building symbolic expressions tailored to the properties of the type-propagation framework, we are interested in a weaker property, which we will call *cyclic k-boundedness.* This concept does not apply to functions in general, but specifically to the transfer functions of data flow frameworks.

We define an *instance* of a framework $\langle L, \mathcal{F}, \vee \rangle$ as a tuple $\langle L_G, \mathcal{F}_G, \vee \rangle$ in which the semi-lattice L_G and the function space $\mathcal{F}_G$ are contained respectively in L and $\mathcal{F}$ and include only the elements necessary to analyze the control flow graph G [7].

In order to define cyclic k-boundedness for a framework instance, we consider statements of the form $x \leftarrow E(x)$, in which $E(x)$ is an expression involving arbitrary functions and operators occurring in the flow graph to analyze. Let C_G be the set of such assignments for a flow graph G; let $\mathcal{F}_{C_G}$ be the set of transfer functions for single-statement blocks containing such assignments; cyclic k-boundedness can be defined as follows:

Definition 2.2 *A framework instance for flow graph G is cyclically k-bounded if and only if, for any transfer function F_C in $\mathcal{F}_{C_G}$, ${F_C}^k \leq {F_C}^{[k-1]}$.*

Note that the height of the lattice in a framework instance is always a cyclic bound on the function space (owing to monotonicity), but it is not necessarily a standard bound. For example, constant propagation is $2 \times |V| + 1$-bounded, where $|V|$ is the number of variables in the program to analyze, but cyclically 3-bounded. Indeed, each variable can change its value twice in the constant-propagation lattice; and, as each function in $\mathcal{F}_{C_G}$ involves a single variable, $\mathcal{F}_{C_G}$ is 3-bounded, i.e. $2 \times |V| + 1$-bounded with $|V| = 1$.

2.2.3 Cyclic bound of the type propagation framework

Definition 2.2 indicates that, in order to find a cyclic bound for a framework instance, it is enough to find the maximal number of useful iterations for propagating information through a loop of the form

```
while cond do
    x ← E(x) ;
od ;
```

[7] Precise rules for building the lattice and function space of a framework instance can be found for example in [GW76].

Let $\dot{E}$ be the type mapping associated with E in $E(x)$. We note that possible subexpressions of $E(x)$ are pre-loop values, constants, object creations, method calls, and x itself. Only information contributed by the last two items, viz. method calls and x, are sensitive to input information and so can possibly require more than one application of $\dot{E}$ in order to reach a fixed point solution. Therefore, we can restrict our attention to expressions $E(x)$ involving "interesting method calls", i.e. method calls whose receiver is either x or the result of an interesting method call.

We can further restrict the class of expressions to consider if we are content to prove a sufficient condition for cyclic k-boundedness, which can be expressed as

$$\forall F_C \in \mathcal{F}_{C_G} : \; {F_C}^k \leq {F_C}^{k-1} \tag{2.2}$$

One can observe that any bound found using (2.2) for a single method is also valid for a composition of methods, for monotonicity implies the following:

$$\forall f, g \; : \; f^k \leq f^{k-1} \text{ and } g^k \leq g^{k-1} \Rightarrow (f \circ g)^k \leq (f \circ g)^{k-1}$$

Therefore, equation (2.2) translates into the following theorem:

Theorem 2.1 *If for any method f occurring in a flow graph G, $\dot{f}^k \leq \dot{f}^{k-1}$, then the instance $\langle L_G, \mathcal{F}_G, \vee \rangle$ of the type propagation framework is cyclically k-bounded.*

In the intraprocedural problem being considered, $\dot{f}$ is directly derived from the type declarations for f. Therefore, if in a framework instance all declared function types $t_i \rightarrow t_j$ are such that $t_j \leq t_i$, all type methods $\dot{f}$ are descending, meaning that the framework instance is cyclically 1-bounded ($\dot{f} \leq \iota$). If some function declarations are not descending, then the framework instance is cyclically k+1-bounded, where k is the maximal length of a chain of non-descending function types $\langle \langle T_1 \rightarrow T_2 \rangle, \langle T_2 \rightarrow T_3 \rangle, \ldots, \langle T_{k-1} \rightarrow T_k \rangle \rangle$. For example, with the type declarations represented in Figure 2.2, $\dot{f}^2 = \dot{f}^3$, which implies

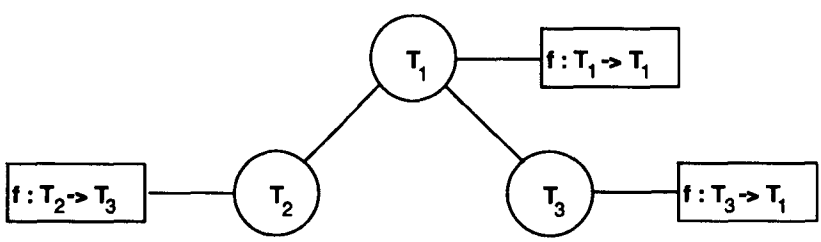


Figure 2.2

that the framework instance is 3-bounded. And indeed, the length of the chain of non-descending function declarations $\langle f[T_2] : T_2 \rightarrow T_3, f[T_3] : T_3 \rightarrow T_1 \rangle$ equals 2. This gives rise to the following theorem:

Theorem 2.2 *If the chains of non-descending function declarations associated with a flow graph G have maximal length $k-1$, then the instance $\langle L_G, \mathcal{F}_G, \vee \rangle$ of the type propagation framework is cyclically k-bounded.*

In fact, since nontrivial chains of non-descending function declarations are likely to be extremely rare *in practice, most instances of the type-propagation framework will be cyclically bounded to* 2.

Now, to show that cyclic k-boundedness is preserved when fixed-point graphs of symbolic functions are used instead of function declarations, consider that the fixed-point algorithm described in Section 4 finds return types less or equal to the return types declared. Therefore, all descending functions will remain so and the actual cyclic bound is necessarily less or equal to the cyclic bound inferred from function declarations.

3 Symbolic interpretation

3.1 General principle

The general idea consists in *(i)* building a set of use-definition edges (or *ud*-edges for short) for each function, *(ii)* considering each *ud*-edge as a reduction rule that replaces a variable occurrence by the join of its reaching definitions, and (*iii*) transforming ("normalizing") the resulting reduction system using cyclic k-boundedness in order to give it the property of termination, so that a symbolic expression for any variable occurrence can be obtained by deriving a normal form in a finite number of steps.

Many advantages accrue from this approach

1. Other problems, like constant propagation, can reuse *ud*-edges [Ken81]. In addition, for the purposes of our algorithm, *ud*-edges can indifferently be replaced by SSA edges, which are necessary for propagating constants efficiently and accurately [WZ91]. Furthermore, some steps involved in the process of putting a program into SSA form can be reused by our algorithm for normalizing reduction systems (footnote to page 10).

2. Use-definition edges can be built by solving the Reaching Definition problem which, being partitionable, is amenable to a form of incremental analysis particularly well suited to the requirements of a Language-Based Editor [Zad84].

3. No exhaustive intraprocedural type analysis is needed at all, for reduction systems enable one to solve the intraprocedural problem for selected statements. So intraprocedural analysis and flow-sensitive interprocedural analysis can be combined without any redundant computations.

Note that special measures must be taken in order to accommodate side-effects and aliasing (Section 3.5).

3.2 Representing *ud*-edges as reduction rules

To illustrate the process, we will consider the control flow graph in Figure 3.3.

We construct a set of rules in which x_s represents the symbolic value assigned to the use of x at site s. When several definitions reach a given use, as at site s in the example, the representation merges the corresponding expressions through the confluence operator ($\vee$ in our framework)[8]. So, the *ud*-edges for the example will be represented as:

[8] If SSA edges are used, then explicit conditions rather than joins can be used in reductions.

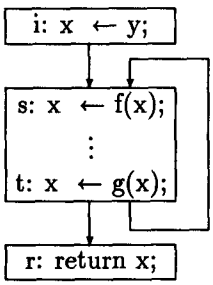


Figure 3.3 Cyclical use-definition dependences

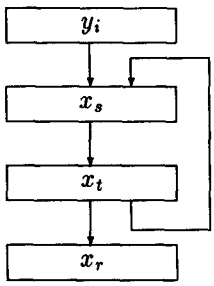
$$x_r \rightarrow \dot{g}(x_t) \tag{3.3}$$
$$x_t \rightarrow \dot{f}(x_s) \tag{3.4}$$
$$x_s \rightarrow \dot{g}(x_t) \vee y_i \tag{3.5}$$

3.3 Identifying self-embedding occurrences

The system is self-embedding, meaning that it produces derivations like $x_t \Rightarrow \dot{f}(\dot{g}(x_t) \vee y_i)$ or $x_s \Rightarrow \dot{g}(\dot{f}(x_s)) \vee y_i$. It is important to note that such circularity is necessarily due to the presence of a loop in the Control Flow Graph. Supposing the framework instance to be cyclically k-bounded, we can handle it easily if the flow graph is reducible[9].

To this end, we draw up the *Inverse Dependency Graph* of the reduction system, defined as $\langle N, E \rangle$ where N is the set of variable occurrences appearing in the reduction system and E the set of all pairs $\langle a_i, b_j \rangle$ such that there is a rule defining b_j in terms of an expression involving a_i. In the example, the IDG is

[9]Intraprocedural flow graphs (as opposed to call graphs) for structured languages are almost invariably reducible.



Intuitively, an arrow in the diagram can be read as "defines". It can be shown that the IDG can be derived from the flow graph by reducibility-preserving transformations; therefore it is possible to assume the IDG reducible whenever the flow graph is. Consequently, cycles in the IDG are regions. The header of a strongly connected region will be termed a *self-embedding occurrence.* In our example, x_s can be seen to be a self-embedding occurrence.

3.4 Eliminating self-embedding occurrences

A self-embedding occurrence is a variable occurrence whose defining rule in the reduction system contains loop-induced definitions and definitions occurring prior to the loop. In the example, the rule $x_s \rightarrow \dot{g}(x_t) \vee y_i$ assigns to the self-embedding occurrence x_s the loop-induced definition $\dot{g}(x_t)$ and the pre-loop definition y_i[10].

Therefore, cyclic k-boundedness can be used as follows: A *k-bounded normal form* for x_s is computed by deriving x_s exhaustively using the original rule for x_s only $k-1$ times and thereafter replacing it by a rule incorporating only pre-loop definitions, namely $x_s \rightarrow y_i$.

Supposing the example to be a cyclically 2-bounded problem, this method yields the 2-bounded normal form $\dot{g}(\dot{f}(y_i)) \vee y_i$ for x_s. Then, the rule for x_s can be modified using this form, which yields the following 2-bounded reduction system.

$$x_r \rightarrow \dot{g}(x_t) \quad (1')$$
$$x_t \rightarrow \dot{f}(x_s) \quad (2')$$
$$x_s \rightarrow \dot{g}(\dot{f}(y_i))) \vee y_i \quad (3')$$

[10]If no pre-loop definition appears, a used-before-defined error can be diagnosed and a fake assignment of a typed *error* value inserted in a preheader to the function. Here, if x had not been assigned any value prior to the loop, a fake assignment would have been added at the beginning of the function, conferring to x its declaration type and the value *error*.

Also note that the algorithm described in [RWZ88] for computing SSA forms involves identifying pre-loop values ("landing-pad definitions") in a reducible graph, which offers opportunities for reuse if our algorithm is made to work with SSA edges.

Note that self-embedded occurrences must be processed in reverse order of nesting level in the IDG. This ensures that, whenever a self-embedded occurrence is processed, it is the header of an inner loop of the IDG associated with the current reduction system.

3.5 Dealing with side-effects and aliasing

A strong point of the algorithm presented here is that the only intraprocedural problem that has to be solved is a bit-vector problem (typically Reaching Definitions), in order to build *ud*-edges. If some form of flow-sensitive alias analysis was factored into the construction of *ud*-edges, this feature would be lost, and the interprocedural problem might become intractable [Mye81].

On the other hand, there exist a number of algorithms for collecting the side-effects of procedures in a flow-insensitive way (for example [CK89]). Therefore, a simple idea consists in using the *MOD* information obtained through such an algorithm to determine if reduction rules for a variable v occurring in a function f are to be built using *ud*-edges or a pessimistic estimate, namely the declared type. More precisely, given *MOD* information for each function, i.e. a set $MOD(f)$ of global variables and reference parameters which may be modified by a function f, we compute in a flow-insensitive way for each function f the set $SMOD(f)$ of all variables —both local and global— which may be modified by side-effect at call sites in f. Then, a reduction rule of the form

$$x_s \rightarrow E$$

will take the form $x_s \rightarrow$ *if* $x \in SMOD$ *then* $decl(x)$ *else* E, where $decl(x)$ is the type declared for x and *SMOD* is the *SMOD* information for the function being analyzed.

This means that a normal form will contain conditionals, and so does not need to be updated when the *SMOD* information changes. A similar solution is described in [CCKT86] for a representation of "jump functions" supporting incremental changes (see Section 7).

The form of aliasing taken into account by algorithms computing *MOD* information for procedures is aliasing through reference parameters. Another form of aliasing is aliasing through pointers. An elegant treatment of their incidence on interprocedural analysis can be found in [Wei80]. It is based on closures, inversions, and compositions of copy and alias relations throughout a program. Applied to the problem at hand, it could be used to compute a set *ALIAS* of aliasing relations induced by pointers (and taking reference passing into account). Such a set being computed for the whole program rather than a particular function, its use will lead to more pessimistic assumptions than the use of *MOD* information. Improvements are certainly possible but lie outside the scope of this paper[11]. A simple, but pessimistic, approach consists in assigning its declaration type to any variable participating in the *ALIAS* relation using the same mechanism as for the *SMOD* sets.

Solutions yielding more accurate (partly flow-sensitive) analysis of side-effects and aliasing are considered in Sections 6.3 and 6.4. These solutions involve the whole interprocedural algorithm rather than just symbolic interpretation.

[11] Possible improvements to Weihl's approach in the context of an object-oriented language are described in [Lar92, Sec. 9.2.6]

4 Fixed-point graphs for symbolic methods

Once a reduction system has been built for each function, chaotic fixed point iteration can be used to compute graphs of type methods as shown in Algorithm 4.1.

```
initialize the graphs of symbolic functions and methods
    to the types declared;
place all symbolic functions in the work list ;
until the work list is empty do
    remove an arbitrary symbolic function f̀[T] from the work list ;
    compute graph(f̀[T]) using the symbolic expression
        for f̀[T] and the types found so far ;
    if this value has changed then
        update the corresponding method graph graph(f̀) ;
        insert all symbolic functions
            whose definition involves f̀ into the work list ;
    fi ;
od ;
```

Algorithm 4.1 Fixed-point iteration for graphs of type methods

The step summed up as "update the corresponding method graph $graph(\dot{f})$" involves ensuring that 2 conditions are met by constituent function graphs $graph(\dot{f}[T])$: *(i)* the result type for $\dot{f}[T]$ is the join of the result types of all $\dot{f}[T']$, $T' \leq T$, and *(ii)* the graph for $\dot{f}[T]$ is extended to types inheriting an implementation of f from T.

Dependence relations between symbolic functions are determined using a "raw" call graph, in which all possible calls implied by declared types are considered.

Chaotic (i.e. worklist-based) iteration is preferable to regular iteration because the visit order is given straightforwardly by the call graph, whereas regular iteration would additionally require finding a topological order of the call graph so as to keep down the number of iterations [ASU86, HU73]. Section 6.1 shows that the time bound for this step is on the average proportional to the number of functions in a program.

5 Computing receiver types

Once a reduction system has been obtained for each function and the graphs of type methods are available, computing the type of the receiver at a call site is immediate. A symbolic expression is derived from the reduction system, and the effect of method calls in this expression is interpreted using the type method graphs[12]. For example, if we consider the statement $y \leftarrow f_3(f_1(x))$ in function $f_3[T_1]$ (Figure 1.1), the symbolic expression for the receiver of f_3 is $\dot{f}_1(T_2)$, which —using the graph of symbolic method $\dot{f}_1$— will be found to equal T_3, which shows $f_3[T_3]$ to be the only implementation of f_3 that can be called at this site.

[12]The solutions thus found are acceptable in the sense of [GW76], as proved in [Lar92, Sec. 8.3.5].

In [Lar92, Sec. 8.5], we show that added precision can be obtained by introducing one-element type sets to represent *(i)* the type of an object-creation expression and *(ii)* the initial type of the first formal parameter (receiver) of a function. This does not change the construction and use of reduction systems in any essential way and can be made transparent to the interprocedural step. In addition, the resulting precision is practically comparable to the precision obtained by propagating type sets throughout.

6 Concluding remarks

6.1 Cost

6.1.1 Fixed-point computation of graphs for symbolic methods

In the iterative determination of type method graphs, the number of visits for a given type function is 1 plus the number of times a predecessor in the call graph changes. Therefore, if F is the dependence factor, i.e. the average number of predecessors in the dependence graph for each type function, and H is the height of the subtype semilattice, the cost is $O(F \times N \times (H-1))$. If we assume that F in practice does not depend on the size of a program, we can write this cost $O(N \times (H-1))$. If on the other hand, H can be expected to grow very slowly with the size of a program and be asymptotically constant, we find a cost essentially proportional to the number of functions, which in turn can be estimated proportional to the size of a program for a given language.

6.1.2 Symbolic interpretation

If we assume the derivation of a normal form is cheap and performed in constant time on the average, the significant part in the cost of symbolic interpretation is attributable to the construction of a k-bounded reduction system.

This cost breaks down into the following items:

1. Computation of *ud*-edges or SSA edges: Almost linear algorithms exist for both. In addition, the output of this computation is eminently reusable.

2. Determination of a cyclic bound k: This requires exploring sequentially each function declaration in subtype order, and can be performed in time linear in the number of functions.

3. Detection of self-embedding occurrences, sorted by order of nesting level in the Inverse Dependency Graph: most of this operation consists in building a spanning tree for the IDG, which can be achieved in time linearly related to the number of nodes, i.e. of variable occurrences.

4. Derivation of a k-bounded form for each self-embedding occurrence, in reverse order of nesting level. The cost of this operation is absorbed in the cost of detecting self-embedding occurrences.

If the average number of variable occurrences in a function and the number of functions are considered proportional to program size, then the cost of symbolic interpretation, like that of interprocedural propagation, turns out to be linearly related to program size.

6.2 Other applications

6.2.1 Type-checking

If the algorithm just described was used for type-checking a language with no variable declarations, then —in order to obtain enough precision— it would be necessary to insert after each method call a statement whose effect is to assign to the receiver the closest common *sub*type between its current type assignment and the maximal type for a receiver of the given method. It does not seem that this would affect the essential properties of the type propagation framework, apart from the fact that we would need to introduce a lattice $\perp$, so that a closest common subtype can always be computed.

6.2.2 Constant propagation

In order to compute autonomous representations of symbolic functions, we use method graphs, exploiting the finiteness of the type propagation semilattice. In the constant propagation framework, this solution is obviously not available. One way in which symbolic functions could be used, however, is by performing β-reductions, i.e. expanding calls to symbolic methods for those parts of the call graph which are not recursive. Note that the expansion of a *method* may be the join of several expressions. The symbolic representations obtained after the expansion process can be used to determine if the value returned by a function being passed constant parameters is constant. Wegman and Zadeck [WZ91] note that the passing of constant parameters occurs very frequently, so the profitability of such an approach could be fairly high.

6.3 Accurate side-effect analysis

If a function returns values not only through its result, but also through side-effects to global variables and value-return parameters, this is readily amenable to the kind of symbolic analysis performed for function results, provided there is no aliasing. Note by the way that value-return as well as reference passing do not fare well with object-oriented languages, as these can only be performed in a semantically safe way if the static types of actual and formal parameters match exactly, which runs counter to the philosophy of subtype polymorphism. This is one of the reasons why we will consider here only global variables, the other reason being that they illustrate all the problems which may arise in dealing with alias-free side-effects.

The idea consists in replacing a method call $x \leftarrow f(\ldots)$ by a series of assignments of the form

$$
\begin{aligned}
v_1 &\leftarrow f_{v_1}(\ldots)\ ; \\
v_2 &\leftarrow f_{v_2}(\ldots)\ ; \\
&\vdots \\
v_n &\leftarrow f_{v_n}(\ldots)\ ; \\
x &\leftarrow f_{\rho}(\ldots)\ ;
\end{aligned}
$$

The v_i's denote the variables global to the program portion under analysis. Note that the assignment to x must come last, because x might be a global variable. The function noted f_{v_i} is a function which makes no side-effect (a "pure" function) and returns the value of v_i as updated by f. The function f_ρ is a pure version of the original function. Of course, these functions do not have to be built, and the substitution is performed only for the sake of symbolic interpretation. The symbolic function associated with each of the functions f_{v_i} is derived by adding a dummy use of v_i to the exit block of f before *ud*-edges are built, and then computing a normal form for this occurrence.

Note that by adding one assignment per global variable v_i and equating if needed f_{v_i} to the identity function, we avoid having to modify all call sites when a variable becomes affected or unaffected by a given function.

6.4 Feedback possibilities

The algorithm described in this paper could fit into a scheme like the following:

1. Carry out flow-insensitive alias and side-effect analysis.
2. Put the program into SSA form and build SSA edges.
3. Build k-bounded reduction systems for functions, where k is the cyclic bound for the type propagation problem at hand. Reduction rules should contain conditionals rather than join operations (footnote to page 8).
4. Build graphs for type methods and compute receiver types so as to improve the call graph.
5. Perform conditional constant propagation, as in [WZ91]. Improve the call graph as dead code is eliminated.
6. Use constant propagation interprocedurally, as described in [WZ91], to provide flow-sensitive feed-back to alias analysis.
7. If the call graph has been improved, iterate, skipping redundant computations.

Considerable precision could be derived from such a scheme, but opportunities for incremental processing are diminished if feed-back is used between the different steps. Note that the bound on the maximum number of iterations can be made arbitrarily constant and all the operations involved can be performed in almost linear time, so that this scheme is practically linear in the program size.

7 Comparison with related works

Interprocedural type propagation is performed in the Self optimizer [CU90], where type analysis relies on procedure integration and code duplication. The method described here, on the contrary, allows to keep the analysis phase separate from the transformation phase, which results in more flexibility, more modularity, and therefore more potential for incrementality.

Suzuki [Suz81] describes a very powerful method for interprocedural type propagation which combines intraprocedural data flow analysis and inequality propagation through transitive closure and unification. The emphasis of this approach is on type-checking, so the scheme described is not intended to fit in the global context of an optimizer. A main drawback of the algorithm described is that intraprocedural analysis has to be carried out exhaustively each time a function is visited during interprocedural propagation.

Borning and Ingalls [BI82] describe a very practical scheme for flow-insensitive type determination inside a procedure for a version of Smalltalk with function declarations and optional variable declarations. It does not include any interprocedural type propagation.

[CCKT86] contains the description of a scheme for interprocedural constant propagation. It involves symbolic interpretation to compute "jump functions" (symbolic expressions for actual parameters) and "return jump functions" ("symbolic functions" in our terminology). Most of the paper, however, is devoted to the use of jump functions, and the properties of return jump functions —which seem much more interesting— are only given cursory, if insightful, consideration.

Wegman and Zadeck [WZ91] describe an algorithm for constant propagation which shares many objectives with the present paper: this algorithm propagates constants to branching conditionals, so as to eliminate superfluous flow graph edges, just as the present scheme propagates types to eliminate superfluous call graph edges; it uses sparse representations (SSA edges) for the sake of efficiency (which is an improvement over [Weg75]), just as we use such representations to avoid redundant computations (which is an improvement over [Suz81]).

Reif and Lewis [RL86] describe a linear-time algorithm for symbolic interpretation. It is not equivalent in effect to the method used here, however. The symbolic value of an expression is expressed in terms of functions and particular variable occurrences. A set of symbolic values for all variable occurrences is called a *cover* of the program. A *minimal cover* relates the value of each occurrence to its earliest definition points in the program. Reif and Lewis show that finding a minimal cover is generally undecidable, and accordingly propose an algorithm to find the best possible cover in linear time. However, expressing, as we do, the type of a returned expression in terms of values on entry to the function involves computing a *minimal cover* for the function. Thus, at the cost of restricting the validity of the cover found to cyclically k-bounded problems, our approach makes it possible to compute a minimal cover in linear time. In other words, the originality of the approach presented in this paper consists in trading off generality for expressive power.

Acknowledgements

I wish to thank all the people who helped me with their suggestions and encouragements while I worked on this paper. In particular, I owe special thanks to Thomas Marlowe for his detailed review of an earlier version of the paper, and to Martin Jourdan and Bernard Lang for their advice and moral support. I am also indebted for insightful and stimulating remarks to Ken Zadeck, François Rouaix, François Thomasset, Véronique Benzaken, Françoise Gire and Claude Delobel.

References

[AKW90] S. Abiteboul, P. Kanellakis, and E. Waller. Method schemas. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems (Nashville, TN)*, pages 16–27, April 1990.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers : principles, techniques and tools.* Addison-Wesley, 1986.

[BI82] Alan H. Borning and Daniel H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Conference record of the Ninth Annual ACM Symposium on the Principles of Programming Languages, Albuquerque (NM)*, pages 133–141, January 1982.

[Cal88] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, Georgia*, pages 47–56. ACM, June 1988.

[CCKT86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, June 86*, pages 152–161, June 1986.

[CK89] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages, Austin (TX)*, pages 49–59, January 1989.

[CU90] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting. In *Proceedings of the ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation, White Plains, NY*, pages 150–164, June 1990. published as SIGPLAN Notices, Vol. 25, Num. 6.

[GW76] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

[HU73] Matthew S. Hecht and Jeffrey D. Ullman. Analysis of a simple algorithm for global data flow problems. In *Conference Record of the ACM Symposium on the Principles of Programming Languages*, pages 207–217, September 1973.

[Ken81] Ken Kennedy. A survey of data flow analysis techniques. In S. Muchnick and N. Jones, editors, *Program Flow Analysis, Theory and Applications*, chapter 1, pages 5–54. Prentice-Hall, 1981.

[Klo87] Jan Willem Klop. Term rewriting systems: a tutorial. *Bulletin of the EATCS*, 32:143–182, 1987.

[KU77] J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–318, 1977. Originally published as Research Report TR-169, Computer Sciences Laboratory, Princeton University.

[Lar91] J.-M. Larchevêque. Interprocedural type propagation for object-oriented languages. Rapport Technique 64–90–V1, GIP Altaïr, Rocquencourt, France, September 1991.

[Lar92] J.-M. Larchevêque. *Compilation techniques for incremental development in a persistent object-oriented environment.* PhD thesis, LRI, Université de Paris-Sud, January 1992.

[Mar89] T.J. Marlowe. *Data Flow Analysis and Incremental Iteration.* PhD thesis, Rutgers University, New Brunswick, New Jersey 08903, October 1989. Report DCS-TR-25.

[Mye81] Eugene W. Myers. A precise inter-procedural data flow algorithm. In *Conference record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, pages 219–230, January 1981.

[RL77] John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *Conference record of the Fourth Annual ACM Symposium on the Principles of Programming Languages, Los Angeles (CA)*, pages 104–118, January 1977.

[RL86] John H. Reif and Harry R. Lewis. Efficient symbolic analysis of programs. *Journal of Computer and System Sciences*, 32:280–314, June 1986.

[Rou90] François Rouaix. Safe run-time overloading. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages, San Francisco (CA)*, pages 355–366, January 1990.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego (CA)*, pages 12–27, January 1988.

[Suz81] Norihisa Suzuki. Inferring types in Smalltalk. In *Conference record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, pages 187–198, January 1981.

[Tar81] R. Endre Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):576–593, 1981.

[Weg75] Ben Wegbreit. Property extraction in well-founded property sets. *IEEE Trans. Software Eng.*, SE-1(3):270–285, September 1975.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference record of the Seventh Annual ACM Symposium on the Principles of Programming Languages*, pages 83–94, June 1980.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[Zad84] F. K. Zadeck. Incremental data flow analysis in a structured program editor. *SIGPLAN Notices*, 19(6):132–143, June 1984. Proceedings of the SIGPLAN Symposium on Compiler Construction (Montreal, Canada).

Appendix

Data-flow theoretical definitions

I Framework

Solving a global (or intraprocedural) data flow analysis problem consists in decorating nodes of a control flow graph, which is essentially a flow chart, with information on a program's behavior. Nodes in the flow graph are basic blocks, i.e. single-entry single-exit sequences of statements. The information associated with a basic block represents assertions which hold on entry to the block. This information is modeled as a semilattice, usually a meet semilattice, but this paper uses a join semilattice for conformity with the subtype ordering. If a join semilattice is used, information on entry to a block B located at the confluence of several paths is the result of applying the join operator $\vee$ among information items coming from each path. Information on entry to a block B is mapped to information on entry to each successor S of B by a *transfer function*, which can be associated either with block B or with each edge $\langle B, S\rangle$. A *monotone framework* $\langle L, \vee, \mathcal{F}\rangle$ is made up of a join semilattice L with join operator $\vee$, and a set $\mathcal{F}$ of monotone transfer functions. Monotonicity is important to prove the termination of most data flow analysis algorithms. Beside monotonicity, a useful property of $\mathcal{F}$ is closure under join and composition, which makes it possible to extend the definition of transfer functions from edges to paths in the natural way.

II Use-definition edges, SSA edges

A use-definition edge, or *ud*-edge for short, for variable occurrence x_s is a pair $\langle x_s, t\rangle$, where t is a possible definition site for x_s.

SSA edges are essentially information edges (use-definition or definition-use edges) for a program in SSA form. A program is in SSA form if variables are renamed so that there is only one assignment to each variable in the program text. At a join point, a statement of the form $v_i \leftarrow \phi(v_j, v_k)$ is inserted for each variable v with different renamings in the branches that are joined. The ϕ function returns the value of v_j or v_k according as control comes from the branch where v_j or v_k is defined. One advantage of using SSA edges is that the effect of two confluent SSA edges like $\langle v_i, v_j \rangle$ and $\langle v_i, v_k \rangle$ can be predicated on the value of a branching condition (as part of the static evaluation of $v_i \leftarrow \phi(v_j, v_k)$). On the contrary, when several *ud*-edges exist for a variable occurrence, the corresponding definitions can only be related through the confluence operator of the framework, which may lead to more pessimistic results.

III Symbolic interpretation

Symbolic interpretation consists in assigning values to variables at each point in a program irrespective of the possible execution paths leading to this point. To solve the problems induced by joins and circular definitions, the domains of program operations are replaced by sets of symbols with adequate properties. For example, in the type propagation problem described in the paper, the domain of program operations (methods) is replaced by sets of upper bounds on types and a join operation is defined on them.