Inferring Types in Smalltalk

Norihisa Suzuki
Xerox Palo Alto Research Centers
3333 Coyote Hill Rd., Palo Alto, CA 94304

1. Introduction

Smalltalk is an object-oriented language designed and implemented by the Learning Research Group of the Xerox Palo Alto Research Center [2, 5, 14]. Some features of this language are: abstract data classes, information inheritance by a superclass-subclass mechanism, message passing semantics, extremely late binding, no type declarations, and automatic storage management. Experience has shown that large complex systems can be written in Smalltalk in quite a short period of time; it is also used to teach programming to children quite effectively. Object-oriented languages like Smalltalk have begun to be accepted as friendly languages for novice programmers on personal computers.

However, Smalltalk has some drawbacks, too. Smalltalk programs are inefficient compared with Lisp or Pascal. Late binding is a major reason of this inefficiency; every time a procedure is called, its implementation in the current context has to be found.

Because of late binding, whether there is an implementation of a procedure call or not can only be found at run-time. This may be convenient in the early stages of system development; one can run a partially completed system, and when he discovers a run-time error caused by an unimplemented procedure, he can write the procedure body and proceed the computation from the point where the error was discovered. However, there is no way to guarantee that there will be no run-time errors. We found many "completed" systems which still had such run-time errors.

Another problem is that it is hard for a novice to read Smalltalk programs written by other people. The fact that there are no type declarations and the fact that the bindings are late are major causes of unreadability. All the Smalltalk procedures are so called generic procedures. Each procedure name is associated with several procedure bodies declared in different classes. Depending on the classes of the arguments of a procedure call different procedure bodies are invoked. Since the classes of the arguments may differ according to the context, it is impossible to statically predict the behavior of the procedure calls.

We observed that both inefficiency and unreadability are attributed to late binding; however, early binding can be effectively accomplished if we can tell the classes of the procedure arguments at compile time. In the long run probably Smalltalk needs to have "type" declarations——probably not rigid declarations of Pascal but rather in the form of hints to compilers and programmers. Even without changing the language it would be nice to have a tool that supplies "type" declarations to current Smalltalk or partially specified Smalltalk. This will also lead to efficient compilation.

We thus concluded that we need to introduce "types" to Smalltalk. The introduction of types is more promising in Smalltalk than in similarly declarationless language Lisp, since Smalltalk has a rich user-defined abstract classes. Therefore, the most straightforward approach to introduce types is to associate types of variables to classes that variables denote and to associate types of procedures to mappings from classes to classes. Since a variable may denote objects of different classes, we define the type of a variable to be a union of classes that the variable will ever denote.

The aim of this research is not to implement compilers for Smalltalk with type declarations. We intend to design tools to supply type declarations to current Smalltalk programs. Complete type determination is neither possible nor desirable; people do write Smalltalk programs that take advantage of late bindings. We are, therefore, interested in finding a relatively efficient method that can find types of expressions in a large number of cases.

The problem of statically assigning types to type-declarationless programs is called type-inference problem. We can find a number of work on type inference [3, 4, 7, 9, 11, 15]; these techniques are, however, either too restrictive or too inefficient for our purpose. The only technique implemented, proven to work for non-trivial cases, and used extensively was developed by Milner [7] to determine types for ML language of LCF. Even though ML language is much simpler than Smalltalk, the fact that there exists an efficient, versatile algorithm encouraged us to investigate whether we can extend the method.

The LCF type checker produces a set of equations from procedure declarations and solves them by unification [12], to obtain the types of the procedures; it can run in linear time due to a fast unification algorithm invented recently [10]. We extended Milner's method so that we can treat unions of types; in our method, we create a set of equations and inequalities and solve them by unification and a transitive closure algorithm. This technique is general and can be applied to other data-flow problems.

The advantage of Milner's method and our method is that it reduces the problems to purely mathematical domain so that we can apply various formula manipulation algorithms, without considering the execution order or side-effects. Another advantage is that these methods can handle functions with polymorphic types.

In section 2 we review earlier work on type inference. The brief introduction of the syntax and the semantics of Smalltalk is done in section 3. Then we introduce the "types" into Smalltalk in section 4. We discuss the first part of our algorithm, how to extend LCF type checking algorithm for liberal unions of types, in section 5. Then in section 6 the whole algorithm is presented. Section 7 is concerned with the implementation and experience.

Smalltalk has four major different versions of the language and implementations. The version we used for our experiments is Smalltalk-76.

2. Earlier work on type inference

There are essentially two approaches to type inference. Functional approach is used mainly for applicative languages; however, as we show in this paper, that is essentially not the limitation to this approach. Data-flow approach is used for imperative languages; state vectors consisting of the types of the variables at various locations in the programs are introduced. These state vectors are pushed through the programs until fixed-points of the types are obtained.

2.1. Functional approach

Morris and Reynolds [9, 11] independently considered the same problem at about the same time. In typeless languages like lambda calculus (Morris) or Lisp (Reynolds), it is possible to encounter run-time errors such as applying lists to arguments. So the question that they posed is: Can one infer types of functions in these typeless languages, to catch more errors at compile time?

Consider a recursive function

map(f,m) $=_{def}$ **if** null(m) **then** nil
else cons(f(car(m)), map(f,cdr(m))).

Suppose the type specifications of null, nil, cons, car, and cdr are given as follows:

null: α_1 list → Bool,
nil: → α_2 list,
cons: $\alpha_3 \times \alpha_3$ list → α_3 list,
car: α_4 list → α_4,
cdr: α_5 list → α_5 list,

where $\alpha_1, \ldots, \alpha_5$ are variables that take types as values, and list is a postfix type constructor. Then we can easily think that map has the following type:

$(\alpha \rightarrow \beta) \times \alpha$ list → β list.

They have given a way to derive types of functions such as the one shown above.

Milner [7] has pursued their approach further, and shown that an efficient way to solve the problem is to use the unification algorithm. He implemented the algorithm in LCF, and demonstrated the feasibility of using such a program as an interactive programming aid: one can write programs without any type declarations and the tool fills all the details.

The characteristics of this approach are:

1. The types of functions are given as an expression with the types of parameters as free variables. Therefore, even for procedures with polymorphic types, once we compute the type of a function, no recomputation is necessary when the function is used with parameters of different types. This is particularly useful in interactive programming.

2. The domain of types can be infinite.

3. The result types of functions must be uniquely determined given the types of all the parameters. For example, determining types of a function like

$f(x) =_{\text{def}}$ **if** x **then** 1 **else** 1.1

is beyond the capability of their system.

2.2. Data-flow analysis approach

Using data-flow analysis techniques to determine properties which can be described by a finite lattice [6], various people [3,4,15] showed that one can infer types of program states at various locations in the program.

The characteristics of this approach are:
1. It can treat arbitrary union of types.
2. The domain of types must be finite.
3. Input must be constant elements of the lattice for each analysis. Therefore, it cannot handle polymorphic procedures. Unlike the functional approach, whenever a function call is encountered, the function body may have to be reanalyzed. It may be possible to apply procedural data-flow analysis techniques [13, 16], but nobody has shown how to apply them to type-inference problems of polymorphic procedures.

3. Smalltalk

There is a paper that describes fragments of Smalltalk-76 [2], but the complete language description is yet to be published [14]. Here, I will describe the syntax and the semantics of Smalltalk-76 briefly.

3.1. Syntax

The syntax is described with BNF form with the following convention: | (alternative), $\{\ \}^+$ (repetition of one or more times).

<expression>	::= <assignment> | <block> | <conditional> | <message> | <identifier>
<assignment>	::= <variable> ← <expression>
<block>	::= [<concat>]
<concat>	::= <return expr> | <expression>.<concat>
<conditional>	::= [<expression> ⇒ [<concat>]<concat>]
<return expr>	::= <expression> | ⇑<expression>
<message>	::= <expression> <unary selector> | <expression> <binary op> <expression> | <expression> {<selector> <expression>}$^+$
<selector>	::= <identifier>: | <identifier> ⦂
<unary selector>	::= <identifier>
<binary op>	::= <a sequence of operator characters>
<method>	::= {<selector> <formal parameter>}$^+$ | <temporary vars> <block> | <binary op> | <temporary vars> <block> | <unary selector> | <temporary vars> <block>

Message is equivalent to procedure call of Algol and method is equivalent to procedure declaration. A typical message

$$r\ f{:}\ e,$$

which is equivalent to the Simula [1] procedure call r.f:(e), means to invoke the procedure f: in the class of r with the parameter e.

From here on we will use the terms messages and methods instead of procedure calls and declarations.

3.2. Class structure

The only kind of entities in Smalltalk is an object. Every object belongs to one and only one class; all the objects in a class respond similarly to messages. Programmers can create classes freely, but all the classes form a tree structure; the root is Class Object.

All the objects of a class have the same internal structure. Internal structure of an object is determined by local variables and a vector of methods descriptors. There are four kinds of variables:

temporary variables: local to each method
instance variables: local to each object
class variables: local to each class
global variables: they can be accessed from every method.

A subclass is said to inherit all the properties of its superclass——all the class variables, instance variables, and a vector of methods descriptors of the superclass are implicitly defined for the subclass.

3.3. Semantics of messages

Consider a message,

$$r\ f_1{:}\ a_1\ f_2{:}\ a_2.$$

Here r is called a receiver, and a_1 and a_2 are parameters of the message. f_1: and f_2: are the selectors of the message, and the concatenation of the selectors denote the name of the method that implements the message. The execution of this message takes the following steps. First, r, a_1, a_2 are evaluated. Let the value of r be an object of class A. Then, class A is searched whether the method $f_1{:}f_2{:}$ is declared in it. If it does, the method $f_1{:}f_2{:}$ in class A is invoked with a_1, a_2 as actual parameters of $f_1{:}f_2{:}$. If $f_1{:}f_2{:}$ does not exist in class A, then its superclass is searched. This process is repeated until Class Object is searched. If $f_1{:}f_2{:}$ does not exist in Class Object, then it is a run-time error. This process is shown in Fig. 1.

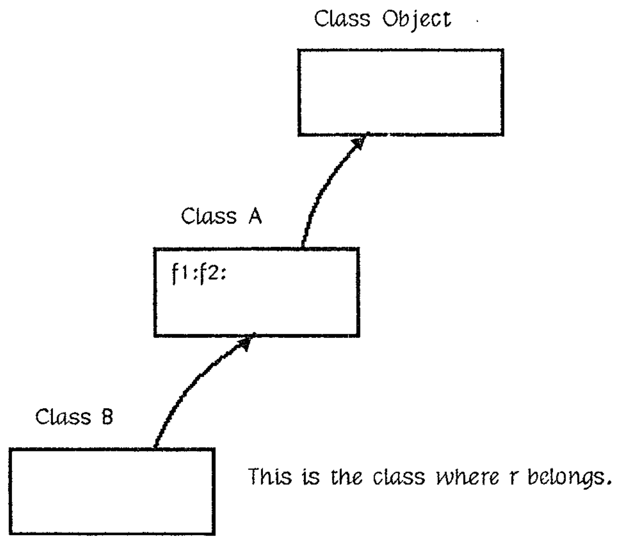

Fig. 1. The message invocation of r f1: a1 f2: a2.
The class of r is B. Since f1:f2: does not exist in Class A, f1:f2: in Class B is invoked.

4. Types in Smalltalk

We will introduce syntax of type expressions, and define how these expressions are interpreted. Then we will show how we assign type expressions to program expressions. The purpose of this is to prove that we can assign types to programs so that programs with type assignment will not create run-time errors, and to show that we can actually present such an algorithm.

4.1. Definition of types

4.1.1. Base type set

There is a finite set B, which consists of all the class names. B is called the base type set.

4.1.2. Type expressions

We will be assigning type expressions of the form $\alpha \times \beta \times \gamma \rightarrow \delta$ to functions. These type expressions are defined as follows:

1. Any subset of the base type set B is a type expression.
2. Greek letters, α_1, α_2, ... denoting type variables are type expressions.
3. $\top$ (top) and $\perp$ (bottom) are type expressions.
4. If α and β are type expressions, then so are $\alpha \times \beta$, and $\alpha \rightarrow \beta$.

{Integer, Real}, {Object}, $\alpha \rightarrow \beta$, {Integer}$\rightarrow \beta$ are all type expressions.

4.1.3. Partial order

We can also define a partial order relation (por) between type expressions. The por is defined as follows:

Constant type expressions

First, we consider por's among constant type expressions.

1. If a, b $\subseteq$ B, then $a \sqsubseteq b$ iff $a \subseteq b$.
2. If a $\subseteq$ B, then $\perp \sqsubseteq a$ and $a \sqsubseteq \top$.
3. $a \sqsubseteq g \wedge b \sqsubseteq d$ iff $a \times b \sqsubseteq g \times d$
   $a \sqsubseteq g \wedge b \sqsubseteq d$ iff $g \rightarrow b \sqsubseteq a \rightarrow d$

Variable type expressions

Variable type expressions contain type variables as parts of the expressions.

$\tau_f \sqsubseteq \tau_g$ iff there exists a substitution S such that $S\tau_g = \tau_f$.

4.1.4. Type descriptions of methods

Type expressions are not sufficient to describe types of procedures. Consider a method

```
f | [self is: Integer ⇒ [↑self addInt: 1] ↑self addReal: 1.0].
```

The receiver of this method is denoted by self in the method. If the class of the receiver is Integer, then the result of sending addInt: 1 to self is returns; otherwise, the result of sending addReal: 1.0 to self is returned. If the type of then-expression is τ_1 and the type of else-expression is τ_2, then the type of the result of the method is a union of τ_1 and τ_2. If we introduce a union operator $\sqcup$, we can associate the type of this method with a type expression

$$\tau_f = \tau \rightarrow \tau_1 \sqcup \tau_2.$$

Alternatively we can express this using por's,

$$\tau_f = \tau \rightarrow \gamma \wedge \tau_1 \sqsubseteq \gamma \wedge \tau_2 \sqsubseteq \gamma.$$

We took the latter approach for describing types of methods, because we wanted to reduce the number of symbols used.

We will describe the type of a method by a type equation and a set of por's, which may be empty. We call this the *type description* of the method.

Example
```
f: y | [self A: y ⇒ [↑self] ↑y].
```
equation: $\tau_f = \tau_x \times \tau_y \rightarrow \tau_r$
por's: $\tau_{self} \sqsubseteq \tau_r$, $\tau_y \sqsubseteq \tau_r$

Any τ_r which satisfies por's is a solution; the least solution is $\tau_x \sqcup \tau_y$.

4.2. Type assignments

We assign types to programs in a way similar to Milner [7]. We need first some notion of a type environment to give types to the free variables in an expression and assert por's obtained from function applications.

A *prefix* p is a finite sequence of variables. An *assumption* v is a finite sequence of por's. A prefixed expression (pe) has the form <p,v,e>, where every variable free in e occurs in p. If a prefix or an assumption consist of a single element, then they are represented by that element; otherwise the members are separated by commas and enclosed in parentheses.

We say that a variable x is *active* in p if no x occurs to the right of it in p.

Now a type assignment of a pe <p,v,e> is an assignment of a type expression or a type description to each element of p, and to each subexpression of e. A variable x is assigned a type expression when x is either in p or in e. A method name f is assigned a type description when f is in p, and a type expression when f is in e. Thus one type assignment of the following pe,

<,,*append:* r | [*self null* ⇒ [⇑r *copy*]
  ⇑*self car cons:* (*self cdr append:* r)]>

is:

$$\langle ,,\mathit{append:}_{\{\mathrm{List}\}\times\{\mathrm{List}\}\to\{\mathrm{List}\}}\ r_{\{\mathrm{List}\}} \mid [\mathit{self}_{\{\mathrm{List}\}}$$
$$\mathit{null}_{\{\mathrm{List}\}\to\{\mathrm{Object}\}} \Rightarrow [\Uparrow r_{\{\mathrm{List}\}}\ \mathit{copy}_{\{\mathrm{List}\}\to\{\mathrm{List}\}}]$$
$$\Uparrow \mathit{self}_{\{\mathrm{List}\}}\ \mathit{car}_{\{\mathrm{List}\}\to\top}\ \mathit{cons:}_{\top\times\{\mathrm{List}\}\to\{\mathrm{List}\}}$$
$$(\mathit{self}_{\{\mathrm{List}\}}\ \mathit{cdr}_{\{\mathrm{List}\}\to\{\mathrm{List}\}}$$
$$\mathit{append:}_{\{\mathrm{List}\}\times\{\mathrm{List}\}\to\{\mathrm{List}\}}\ r_{\{\mathrm{List}\}})_{\{\mathrm{List}\}}]_{\{\mathrm{List}\}}\rangle$$

We denote a type assignment of <p, v, e> by $\langle \bar{p}, \bar{v}, \bar{e}\rangle$, or $\langle \bar{p}, \bar{v}, \bar{e}_\sigma\rangle$ when we want to indicate the type σ assigned to e itself.

In any $\langle \bar{p}, \bar{v}, \bar{e}\rangle$, and any binding f_σ of a method name f in either $\bar{p}$ or $\bar{e}$, a type variable in σ which does not occur in any enclosing y_τ binding of a variable y is called a *generic* type variable for the binding f_σ. A *generic instance* of σ is an instance of σ in which only generic type variables are instantiated.

We now define the notion of a *well-typed* (wt) pe as follows:

(i) $\langle \bar{p}, \bar{v}, x_\tau\rangle$ is wt iff either

(a) x is a variable, and x_τ is active in $\bar{p}$, or

(b) x is a method name, and x_σ is active in $\bar{p}$ and, if σ consists of a type equation $\tau_X = \tau$ and por's $\mathcal{P}_X$, then τ is a generic instance of σ, and the instantiation of $\mathcal{P}_X$ by this substitution is asserted by $\bar{v}$.

(ii) $\langle \bar{p}, \bar{v}, (\bar{e}_\rho f_\sigma)_\tau\rangle$ is wt iff $\langle \bar{p}, \bar{v}, \bar{e}\rangle$, $\langle \bar{p}, \bar{v}, \bar{f}\rangle$ are wt, and $\sigma = \alpha\to\beta$ and $\rho\sqsubseteq\alpha$ and $\beta\sqsubseteq\tau$ are asserted by $\bar{v}$.

(iii) $\langle \bar{p}, \bar{v}, [\bar{e}_\rho \Rightarrow [\bar{e}'_\sigma]\ \bar{e}''_\gamma]_\tau\rangle$ is wt iff $\langle \bar{p}, \bar{v}, \bar{e}\rangle$, $\langle \bar{p}, \bar{v}, \bar{e}'\rangle$, and $\langle \bar{p}, \bar{v}, \bar{e}''\rangle$ are all wt, and $\sigma\sqsubseteq\tau$ and $\gamma\sqsubseteq\tau$ are both asserted by $\bar{v}$.

(iv) $\langle \bar{p}, \bar{v}, f\!:_\sigma r_\delta \mid v_\gamma [\bar{s}]_\tau$ *declared in class A*$\rangle$ is wt, iff $\langle \bar{p}$ *conc* $(f\!:_\sigma, r_\delta, v_\gamma), \bar{v}, \bar{s}\rangle$ is wt, and $\sigma = \alpha\times\delta \to \tau$, where α is a union of class names of A and subclasses of A that does not have the declaration of f: and its superclass is in α.

If there is a way to well-type an expression, then the execution of the expression in the type environment will not create any run-time errors. This can be easily proved.

The rest of the paper deals with the algorithm to assign well-typing.

5. Extending the type-inference algorithm of LCF

We first show that we can extend the type-inference algorithm of LCF so that we can infer the types of procedures that have liberal unions of types for conditional expressions. Then we extend further to allow temporary variables and assignments. The algorithm in this section are not yet the complete algorithm for the type-inference problem of Smalltalk.

5.1. Type inference algorithm for union of types

Let us consider the following *append:* function, which takes two lists as arguments and returns a list:

append: r | [*self null* ⇒ [⇑r *copy*] ⇑*self car cons:* (*self cdr append:* r)].

We assume that all the functions in the definition of *append:* have the type descriptions as follows:

null	$\{\mathrm{List}\} \to \{\mathrm{Object}\}$
copy	$\{\mathrm{List}\} \to \{\mathrm{List}\}$
car	$\{\mathrm{List}\} \to \top$
cdr	$\{\mathrm{List}\} \to \{\mathrm{List}\}$
cons:	$\top \times \{\mathrm{List}\} \to \{\mathrm{List}\}$.

We develop an algorithm that can handle this example.

Algorithm L: (Milner's algorithm to infer types for ML language of LCF).

Algorithm A: (We allow liberal unions of types. Namely, conditional expressions may evaluate to objects of different classes according to whether then-branch is taken or else-branch is taken.)

Step 1: Let the function definition to which we are assigning a type be

f: r | A(x).

Assign new type variables to this declaration as follows.

First we introduce an equation:

$$\tau_f = \tau_x \times \tau_r \rightarrow \rho_0.$$

This formula represents that the types of the function, the arguments, and the result are τ_f, τ_x, τ_r, and ρ_0 respectively. Then we will assign fresh type variables to all the expressions and argument positions of method in A(x).

For the case of append:, it is done as in Fig.2 by introducing type variables ρ_1 to ρ_{19}.

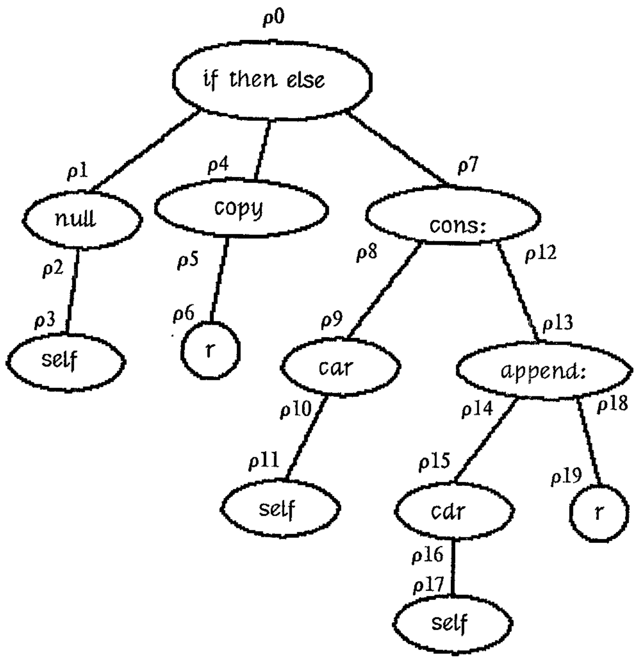

Fig. 2. Assignment of type variables to expression of append:

The meaning of this assignment is that the type of each expression is represented by the type variable attached to it. Also all the messages have type variables attached to their argument positions. They are types expected for the arguments.

According to Fig.2 *self car cons: (self cdr append: r)* gets the type ρ_7, and their arguments *self car* and *self cdr append: r* are assigned ρ_9 and ρ_{13} respectively.

Step 2: Obtain type equations and por's from the fresh type variable assignments. Type equations are created from messages. We also obtain por's from the comparisons between actual parameters and argument positions. If there is a message e_1 f: e_2 and the fresh variable assignments are

$\rho_i \rightarrow e_1$ f: e_2
$\rho_j \rightarrow$ the first argument position of f:
$\rho_k \rightarrow$ the second argument position of f:
$\rho_l \rightarrow e_1$
$\rho_m \rightarrow e_2$,

then the equation produced is

$\tau_{f:} = \rho_j \times \rho_k \rightarrow \rho_i$,

and the por's produced are $\rho_l \sqsubseteq \rho_j$, $\rho_m \sqsubseteq \rho_k$.

Por's are also created from conditional expressions and from references to formal parameters. If there is a conditional expression

$$[e \Rightarrow [s_1]\, s_2]$$

and the fresh variables are assigned as follows:

$\rho_s \rightarrow [e \Rightarrow [s_1]\, s_2]$
$\rho_1 \rightarrow s_1$
$\rho_2 \rightarrow s_2$,

then the por's produced are

$\rho_1 \sqsubseteq \rho_s$, $\rho_2 \sqsubseteq \rho_s$.

From Fig.2 we obtain the following equations and por's:

$\tau_{null} = \rho_2 \rightarrow \rho_1$
$\tau_{copy} = \rho_5 \rightarrow \rho_4$
$\tau_{car} = \rho_{10} \rightarrow \rho_9$
$\tau_{cdr} = \rho_{16} \rightarrow \rho_{15}$
$\tau_{append:} = \rho_{14} \times \rho_{18} \rightarrow \rho_{13}$
$\tau_{cons:} = \rho_8 \times \rho_{12} \rightarrow \rho_7$

$\tau_{self} \sqsubseteq \rho_3$, $\rho_3 \sqsubseteq \rho_2$, $\tau_r \sqsubseteq \rho_6$, $\rho_6 \sqsubseteq \rho_5$, $\tau_{self} \sqsubseteq \rho_{11}$, $\rho_{11} \sqsubseteq \rho_{10}$, $\rho_9 \sqsubseteq \rho_8$, $\tau_{self} \sqsubseteq \rho_{17}$, $\rho_{17} \sqsubseteq \rho_{16}$, $\rho_{15} \sqsubseteq \rho_{14}$, $\tau_r \sqsubseteq \rho_{19}$, $\rho_{19} \sqsubseteq \rho_{18}$, $\rho_{13} \sqsubseteq \rho_{12}$, $\rho_4 \sqsubseteq \rho_0$, $\rho_7 \sqsubseteq \rho_0$.

Step 3: Unify the equations of functions created in Step 2, against the definitions of functions using Algorithm L.

From the unification of the equations for append:, we obtain the following most general unifier:

$(\{\text{Object}\}/\rho_1)(\{\text{List}\}/\rho_2)(\{\text{List}\}/\rho_4)(\{\text{List}\}/\rho_5)(\top/\rho_9)$
$(\{\text{List}\}/\rho_{10})(\{\text{List}\}/\rho_{15})(\{\text{List}\}/\rho_{16})(\top/\rho_8)(\{\text{List}\}/\rho_7)$
$(\{\text{List}\}/\rho_{12})$

Step 4: Collect all the por's. Then solve them to obtain all the relations among parameters and results. The way to solve these por's is discussed in 5.2. Finally substitute the por's using the most general unifier obtained in Step 3.

In the above example, the por's are:

$\tau_{self} \sqsubseteq \rho_3$, $\rho_3 \sqsubseteq \rho_2$, $\tau_r \sqsubseteq \rho_6$, $\rho_6 \sqsubseteq \rho_5$, $\tau_{self} \sqsubseteq \rho_{11}$, $\rho_{11} \sqsubseteq \rho_{10}$, $\rho_9 \sqsubseteq \rho_8$, $\tau_{self} \sqsubseteq \rho_{17}$, $\rho_{17} \sqsubseteq \rho_{16}$, $\rho_{15} \sqsubseteq \rho_{14}$, $\tau_r \sqsubseteq \rho_{19}$, $\rho_{19} \sqsubseteq \rho_{18}$, $\rho_{13} \sqsubseteq \rho_{12}$, $\rho_4 \sqsubseteq \rho_0$, $\rho_7 \sqsubseteq \rho_0$.

Therefore, the type description of *append:* is,

$\tau_{append:} = \tau_{self} \times \tau_r \rightarrow \rho_0$,
$\tau_{self} \sqsubseteq \{List\}$, $\tau_r \sqsubseteq \{List\}$, $\{list\} \sqsubseteq \rho_0$.

This says, the arguments of append: are of class List and the result must be of class List.

End of Algorithm

5.2. Solving the set of partial order relations

We will explain the algorithm to solve a set of por's required in Step 4 of the previous algorithm.

The partial order relations are all of the form $e \sqsubseteq b$, where e and b are either constants or variables. These variables can be divided into two classes: variables representing parameters, which we call terminal variables, and results and variables used to represent subexpressions, which we call non-terminal variables.

The input of the algorithm is the set of por's and the output is the set of por's $c \sqsubseteq d$, where c and d are either constants or terminal variables, such that they represent all the relations among constants and terminals inferred from the input.

This can be solved by transitive closure algorithm.

Step1: Replace each constant by each fresh variable e_i.

Step2: Apply transitive closure algorithm to por's.

Step3: For each terminal variable, obtain all the terminal variables and constants related by the closure of $\sqsubseteq$ relation.

5.3. Type inference for temporary variables and assignments

The second step is to introduce local variables, assignments, and statement series into the language. This requires a flow analysis to determine definition points and application points of variables.

Consider the following program:

```
x ← e.                  (1)
until: x f do:          (2)
    [x ← g].            (3)
```

The value of x at (2) is defined by the values of x assigned at both (1) and (3). Algorithms L and A require that any name, whether it may denote a function or an object, retains the same type whenever it is referred. On the other hand it is actually possible that a variable denote objects of different classes at different locations. Therefore, if we need finer details of type information, we give different names to the different occurrences of a local variable. We then use the partial order relations among these different type variables of the same temporary variable to give the data-flow information among these variables.

This will certainly increase the complexity of the algorithm. On the other hand if we assign one type variable to one local variable we may not obtain fine details we may need for some procedures. Therefore, we have to choose a method according to the requirements on the time of computation, the ease of implementation, and the degree of accuracy.

Algorithm B: (Assigning types to the language with temporary variables and assignments).

Step 1: Put suffix to temporary variables for each occurrence.

In the above example, we suffix as follows:

```
x1 ← e.                 (1)
until: x2 f do:         (2)
    [x3 ← g].           (3)
```

Step 2: Obtain type equations and type formulas for the program using Algorithm A; except, for an assignment

$$y \leftarrow e$$

we create a por

$$\tau_e = \tau_y.$$

From our example the por's obtained are:

$$\tau_e = \tau_{x_1},\ \tau_g = \tau_{x_3},\ \tau_{x_2} \sqsubseteq \tau_{f_{pre}}.$$

Step 3: Obtain por's describing the data-flows.

Now at the application instance (2), x_2 can be both x_1 and x_3. Therefore, we create two por's:

$$\tau_{x_1} \sqsubseteq \tau_{x_2},\ \tau_{x_3} \sqsubseteq \tau_{x_2}.$$

Step 4: Solve the por's.

In the example we obtain

$$\tau_e \sqsubseteq \tau_{f_{pre}},\ \tau_g \sqsubseteq \tau_{f_{pre}}.$$

End of Algorithm

6. Algorithm to infer types in Smalltalk

The features that Smalltalk has and LCF does not have are:

1. temporary variables and assignments,
2. arbitrary union of types,
3. late bindings, and
4. global variables and assignments.

We have shown algorithms for 1 and 2. We will develop the rest of the algorithms in this section.

6.1. Late bindings

All the algorithms we presented require that we know the types of messages used in a method, except the type of itself if it has a recursive call. We have not described how we can treat multiple recursion, but we can deal with multiple recursion by a trivial extension of Algorithm L.

The problem that the late binding causes for the previous algorithms is that we do not know the association between a message and methods so that we cannot tell the types of the messages. Furthermore, the association may change dynamically. Therefore, all we can do at the beginning is to assume the worst case and assume that each message may invoke all the possible methods. Therefore, we assume the type of the message to be some kind of a union of all the types of all the methods invoked.

Using these types of the messages we can use Algorithm B to obtain the type description of the method. Then, we may obtain more accurate information about the type of the method; hence, more information about bindings of each message.

We iterate this process until we can no longer obtain more accurate information. This iteration should terminate since the number of different type descriptions is finite and the iteration will always decrease the ordering of the type description.

The data structure used to perform this algorithm is a table with message names as keys. The values of this table are lists of pairs of class names and the data structures called the type trees. A type tree consists of a list of the parameter types, the temporary-variable types, the result type, and a list of por's. This is shown in Fig. 3.

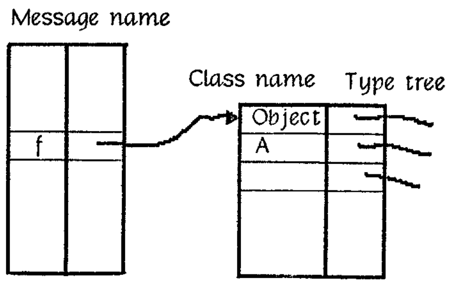

Fig. 3. Type table

There is also a table of back references with a key of a message name; each value of the table consists of a list of pairs of a class and a method where this message is referred.

Following is the type-inference algorithm for late binding.

Algorithm C: (Late binding)

Step 1: Initialize the type trees.

The first parameter, which is the receiver of the method, is initialized to a set of class names; class names included in this set are the class name of the method and class names of subclasses in which the method is not declared and their superclasses are in this set. Then the rest of parameters are initialized to $\perp$ and the result is to $\top$.

Step 2: For each method do the following:

For each message in the method, obtain the types by taking type unions of methods involved. How to take type unions of methods are described in 6.2. The methods involved may be all the methods if the receiver is $\top$.

Then apply Algorithm B to determine types of parameters, temporaries, and results. If the computation yields smaller types than the types at the beginning for any of the entities, then we replace them by the new types, and mark the method that it is updated.

If the types of parameters or results change, then all the methods that calls the method being computed must also be marked for recomputation. However, if only the temporaries change, then we only need to recompute the current method.

Step 3: After scanning all the methods, repeat Step 2 for each method marked for recomputation.

End of Algorithm

6.2. Type unions of methods

In Step 2 of Algorithm C, we had to take unions of types of methods to obtain the type of a message. How should this union be taken?

Consider a message f and the corresponding methods in Class A and B. We distinguish these methods by putting suffixes using class names, then the types are τf_A and τf_B. The type union must behave as follows: if we assign $\tau f_A \oplus \tau f_B$ to f and the program is well-typed, then the program should still be well-typed even if τf_A is assigned to f and τf_B is assigned to f.

Consider an example,

$$\tau f_A = \tau_1 \rightarrow \tau_2,\ \tau_1 \sqsubseteq \{A\},\ \{C, D\} \sqsubseteq \tau_2,$$
$$\tau f_B = \tau_1 \rightarrow \tau_2,\ \tau_1 \sqsubseteq \{B\},\ \{D, E\} \sqsubseteq \tau_2.$$

Then the message f can be successfully sent to objects of both class A and B. The result may be C, D, or E. Therefore,

$$\tau f_A \oplus \tau f_B = \tau_1 \rightarrow \tau_2,\ \ \tau_1 \sqsubseteq \{A, B\},\ \{C, D, E\} \sqsubseteq \tau_2.$$

In the case of multiple parameters, the treatment of the parameters other than the first one is different.

Consider,

$$\tau g_A = \tau_1 \times \tau_2 \rightarrow \tau_3, \tau_1 \sqsubseteq \{A\},\ \tau_2 \sqsubseteq \{C, D\},\ \{E, F\} \sqsubseteq \tau_3,$$
$$\tau g_B = \tau_1 \times \tau_2 \rightarrow \tau_3, \tau_1 \sqsubseteq \{B\},\ \tau_2 \sqsubseteq \{D, G\},\ \{F, H\} \sqsubseteq \tau_3.$$

Let the union be

$$\tau g_A \oplus \tau g_B = \tau_1 \times \tau_2 \rightarrow \tau_3.$$

The message can be sent to objects of class A and B, therefore,

$$\tau_1 \sqsubseteq \{A, B\}.$$

However, when the assignment of $\tau g_A \oplus \tau g_B$ produces well-typing, the assignments of τg_A and τg_B should also produce well-typing. So we have to take simple intersections of types for the rest of the parameters.

Therefore,

$$\tau_2 \sqsubseteq \{D\},\ \{E, F, H\} \sqsubseteq \tau_3.$$

6.3. Example for Algorithm C

Now let us look at how Algorithm C works with an example. There are two classes A and B, where the following methods are defined.

```
Class A
   null
   add:
   car
   cdr
   append: x | r
      [r ← x.
      until: r null do:
         [self ← self add: r car. r ← r cdr].
      ↑self].

Class B
   add:
   car
   cdr.
```

We will show that we can infer that the type of *append:* is

$$\{A\}X\{A\} \rightarrow \{A\}.$$

We assume that we know the types of all the other methods as listed below.

Class A

null	$\{A\} \rightarrow \{\text{Object}\}$
add:	$\tau_1 \times \top \rightarrow \tau_1,\ \tau_1 = \{A\}$
car	$\{A\} \rightarrow \top$
cdr	$\tau_1 \rightarrow \tau_1,\quad \tau_1 = \{A\}.$

Class B

add:	$\tau_1 \times \top \rightarrow \tau_1,\ \tau_1 = \{B\}$
car	$\{B\} \rightarrow \top$
cdr	$\tau_1 \rightarrow \tau_1,\quad \tau_1 = \{B\}.$

Step 1

Let us first examine the body of append:. There are four messages:

null, add:, car, and cdr.

We compute the types of messages by taking type unions:

null	$\{A\} \rightarrow \{\text{Object}\}$
add:	$\tau_1 \times \top \rightarrow \tau_1,\ \tau_1 = \{A, B\}$
car	$\{A, B\} \rightarrow \top$
cdr	$\tau_1 \rightarrow \tau_1,\quad \tau_1 = \{A, B\}$
append:	$\tau_{\text{self}} \times \tau_x \rightarrow \tau_3.$

Step 2

We put suffixes to temporary variables and obtain the following version of *append:*,

```
append: x | r
   [r1 ← x.
   until: r2 null do:
      [self2 ← self3 add: r3 car. r4 ← r5 cdr].
   ⇑self4].
```

Now obtain the partial order relations by unification.

Step 3: We solve this set of por's and obtain

$$\tau_x = \tau_{r_1} \sqsubseteq \tau_{r_2} = \tau_{r_3} = \tau_{r_4} = \tau_{r_5} \sqsubseteq \{A\},$$
$$\{A\} = \tau_{\text{self}_1} \sqsubseteq \tau_{\text{self}_2} = \tau_{\text{self}_3} \sqsubseteq \{A,B\},\ \tau_{\text{self}_2} \sqsubseteq \tau_{\text{self}_4} \sqsubseteq \tau_3$$

Now we can assign as follows:

$$\tau_{r_1} = \tau_{r_2} = \tau_{r_3} = \tau_{r_4} = \tau_{r_5} = \{A\} \text{ and}$$
$$\{A\} = \tau_{\text{self}_1} = \tau_{\text{self}_2} = \tau_{\text{self}_3}.$$

Then compute the types using the message-types of Class A. The result is

$$\tau_x \sqsubseteq \{A\},\ \{A\} \sqsubseteq \tau_3.$$

6.4. Global and own variables.

Finally, we have to consider global and own variables. Here we mean own variables to be variables whose values at the beginning of the computation of methods depend on the previous history of computation.

There are three kinds of global and own variables: instance variables, class variables, and global variables. Actually there are pool variables, but they behave exactly like global variables except that they have to be explicitly imported to classes.

Instance variables are local to each object. They are created whenever an object is created, and their values can be only accessed from the methods of the class. Class variables are local to each class. They are created whenever a class is created, and they can be accessed from every object of the class. Global variables are accessed from every object of Smalltalk. We will treat all of them in a similar manner; we will only present an algorithm which works for all of them.

The difficulty of handling global variables is that unlike temporary variables the flow of control among all the occurrences of global variables is totally unpredictable. Therefore, in general we have to consider a value of a global variable at one location to be affected by all the assignments to that variable in the entire system.

Therefore, what we will do is to assume the type of a global variable to be the union of all the types of objects that the variable may ever denote in the system.

Algorithm D: (Handling of global variables).

Step 1: We assign two types to each global variable. One is access type and the other is assign type. Initialize access types to $\top$ and initialize assign types to $\perp$.

Step 2: Compute the types of methods using Algorithm C. Whenever, global variables are accessed, we use access types as their types. On the other hand we accumulate all the assignments and argument restrictions to assign types. Namely, if a global variable is assigned an object with type τ, we replace the type of the assign type of that global variable by the union of τ and the previous value of the assign type.

Step 3: After the computation, compare access types and assign types. If assign types are smaller than access types, replace access types by assign types. Then repeat Step 2 on all the methods which refer the global variables whose access types are changed.

7. Implementation

The type-inference algorithm has been implemented in Smalltalk-76. It is intended to type-check the Smalltalk kernel system, which consists of about 60 classes and 1300 methods. Out of these methods 98 are implemented outside of the system by microcode and Bcpl programs. The types of these primitive methods are hand coded and fed into the type-checking system.

We have actually implemetned a simpler version of the algorithms presented in this paper; we treated temporary variables (local variables) to take the same value in the entire method in Algorithm B, and we assumed all global variables to have $\top$ as their values in Algorithm D. This decision was made from the following reasons:
a) we wanted the system to run fast,
b) we needed some data on how well a simple algorithm will work before implementing an elaborate algorithm, and
c) most of Smalltalk methods are simple and straightforward so that they do not have the behavior that their types differ depending on the locations.

One of the most useful features of this algorithm is that it detects whether the method returns self, NIL, or one of the parameters. Since many Smalltalk methods return self or NIL, this capability was very important.

We have not yet tested the type inference on the entire Smalltalk kernel, because Smalltalk-76 has a severe limitation on the number of objects to be created and the entire data structure does not fit into the memory. The system is planned to be expanded shortly and we can report the result of the entire system. So far we have been testing on the subset of the kernel that is concerned with numbers.

8. Conclusion

After embarking on this project, Al Perlis suggest to me another approach to obtain more information on types. The approach is, we run the system against some examples and record all the types of arguments and the results. This will probably converge quite quickly and we can obtain information close to the actual types of the methods. This idea is also found in the thesis by Mitchell [8].

It is possible to implement an efficient compiler using this technique. We gather not only the types of arguments but also the frequency and the distribution of these types. Suppose a message invokes one method very often, say 90% of the time, we can create the following code: The class of the receiver is checked to see whether that represents the most frequent case. If so, it jumps to the corresponding method directly; otherwise, it searches methods by the standard way.

However, these information we obtain from statistics can never be better than the actual type. What we obtain is the lower bound of the actual type.

On the other hand we obtain the upper bound of the actual type by type inference program. If both agree, then we are sure of the accuracy of our algorithm. It is always important to obtain both these information in a type inference system.

Furthermore, the statistics can never give us the information on polymorphic types. Many Smalltalk programs can be simply analyzed that they return one of the parameters as the results, and this fact is very important for various purposes.

Acknowledgements

We appreciate various help and suggestions of our collegues at the Xerox Palo Alto Research Centers. We are indebted to Adele Goldberg, Jim Horning, Jim Morris, Larry Masinter for reading this paper.

References

1. Dahl, O.-J., Nygaard, K. Simula——an Algol-Based Simulation Language. Comm. of the ACM, Vol.9, No.9, pp.671-678.

2. Ingalls, D.H.H. The Smalltalk-76 Programming System Design and Implementation. Conf. Rec. Fifth ACM Symp. on Principles of Programming Languages, Tucson, Arizona, 1978, pp.9-16.

3. Jones, N.D., and Muchnick, S. Binding time optimization in programming languages. Conf. Rec. Third ACM Symp. on Principles of Programming Languages, Atlanta, Ga., 1976, pp.77-94.

4. Kaplan, M., and Ullman, J.D. A Scheme for the automatic inference of variable types, Journal of the ACM, No.1, Vol.27, pp.128-145, 1980.

5. Kay, A.C. Microelectronics and the Personal Computer. Scientific American, 237, No.3, pp230-244, September 1977.

6. Kildall, G.A. A Unified Approach to Global Program Optimization. Conf. Rec. ACM Symp. on Principles of Programming Languages, Boston, Mass., 1973, pp.194-206.

7. Milner, R.. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17, 348-375 (1978).

8. Mitchell, J.G. The Design and Construction of Flexible and Efficient Interactive Programming Systems. Ph.D. thesis, Carnegie-Mellon University, 1970. Reprinted in Outstanding Dissertations in the Computer Sciences series. Garland Publishing Co., N.Y., 1980.

9. Morris, J.H. Lambda-Calculus Models of Programming Languages, Ph.D. Thesis, MAC-TR-57, MIT, 1968.

10. Paterson, M.S. & Wegman, M.N. Linear Unification. Proc. Eighth ACM Symp. on Theory of Comp., Hershey, Pa., 1976, pp.181-186.

11. Reynolds, J.C. Automatic Computation of Data Set Definitions. Information Processing 68, pp.456-461, North-Holland Publishing Company, Amsterdam, 1969.

12. Robinson, J.A. A machine-oriented logic based on the resolution principle, Journal of the ACM, No.1, Vol.12, pp.23-41, 1965.

13. Rosen, B.K. Data flow analysis for procedural languages, Journal of the ACM, No.2, Vol. 26, pp.322-344, 1979.

14. Goldberg, A., Robson, D., Ingalls, D., & Tesler, L. Dreams & Schemes, Part One: The Language & its Implementation. To be published.

15. Tennenbaum, A. Type determination for very high level languages. Rep. NSO-3, Courant Inst. Math. Sci., New York U., New York, 1974.

16 Weihl, W.E. Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables. Conf. Rec. Seventh ACM Symp. on Principles of Programming Languages, Las Vegas, Nevada, 1980, pp.83-94.