In First International Static Analysis Symposium, SAS'94, Springer-Verlag LNCS 864.

Constraint-Based Type Inference and Parametric Polymorphism

Ole Agesen
Computer Science Department
Stanford University
Stanford, CA 94305
agesen@cs.stanford.edu

Abstract. Constraint-based analysis is a technique for inferring implementation types. Traditionally it has been described using mathematical formalisms. We explain it in a different and more intuitive way as a flow problem. The intuition is facilitated by a direct correspondence between run-time and analysis-time concepts.

Precise analysis of polymorphism is hard; several algorithms have been developed to cope with it. Focusing on parametric polymorphism and using the flow perspective, we analyze and compare these algorithms, for the first time directly characterizing when they succeed and fail.

Our study of the algorithms lead us to two conclusions. First, designing an algorithm that is either efficient *or* precise is easy, but designing an algorithm that is efficient *and* precise is hard. Second, to achieve efficiency and precision simultaneously, the analysis effort *must* be actively guided towards the areas of the program with the highest pay-off. We define a general class of algorithms that do this: the adaptive algorithms. The two most powerful of the five algorithms we study fall in this class.

1 Introduction

Static analysis of programs is gaining importance: it is the core of any optimizing or parallelizing compiler [20]. Other programming tools based on static analysis are also emerging: [3] describes a tool that can extract compact applications from integrated programming environments by analyzing an image to identify a small but sufficient set of objects for a given application. Constraint-based analysis is a general technique for computing *implementation types* (or representation/concrete types). Implementation types are sets of classes. In contrast to interface types (or abstract/principal types), they deliver the low-level information that is typically needed by compilers and other tools. For example, they distinguish different implementations of the same abstract type (such as an array stack vs. a list stack).

Implementation types are perhaps most useful as a basis for further analysis. For instance, they facilitate call graph analysis which is important because many classical optimization algorithms require call graphs. Hall and Kennedy give an algorithm for computing call graphs [10]. They emphasize efficiency rather than generality so the algorithm does not handle assignable procedure-valued variables or dynamic dispatch. Thus, while useful for analyzing Fortran, it is not effective on object-oriented programs where control flow and data flow are coupled through ubiquitous use of dynamically dispatched message sends. To resolve the coupling, we need implementation types, since only by knowing possible classes of the receiver can we determine which method(s) a send may invoke.

The basic approach to constraint-based analysis was described in [16] for a simple object-oriented language. The algorithm has significant deficiencies when analyzing polymorphic code. There are two kinds of polymorphism. First, parametric polymorphism is the ability of routines to be invoked on arguments of several types. A length function that works on both singly-linked and doubly-linked lists exhibits parametric polymorphism. Second, data polymorphism is the ability to store objects of different types in a variable or slot. "Link" objects forming a heterogeneous list of integers, floats, and strings, exhibit data polymorphism because their "contents" slots contain objects of several types.

Throughout this paper we rely on the Self system [1, 24] for examples. Still, our results are language independent, and we could equally well have chosen any other polymorphic language, even one which is not object-oriented. The latter is remarkable since object-oriented dynamic dispatch and inheritance interacts with parametric polymorphism by restricting the zeroth argument (receiver) of a method to be an object which inherits the method. In [2] it is shown that the interaction can be resolved by doing *lookups* at type inference time. The approach precisely handles multiple and dynamic inheritance, and generalizes to multiple dispatch (multi-methods).

The core of this paper is a study of the basic constraint-based analysis algorithm and four refinements that different researchers have suggested to improve the analysis of polymorphic code. We study parametric polymorphism only. Data polymorphism certainly deserves an equally thorough treatment, but due to lack of space we are confined to discussing it only briefly in the conclusions. Any realistic analysis algorithm *must* deal with both, of course. Indeed, most of the algorithms that we study are part of systems which address both kinds of polymorphism. In the remainder of this paper, unless otherwise stated, "polymorphism" really means "parametric polymorphism."

The main contributions of this paper are:

- A new perspective on constraint-based analysis. By drawing direct parallels to program execution we hope it is more intuitive to practitioners and programmers than the mathematical formalisms that have hitherto been the norm.
- A coherent presentation of the basic algorithm and four improvements found in the literature. We present all the algorithms using the new perspective, thus direct comparisons are possible for the first time. We furthermore characterize when each algorithm succeeds and fails.
- A clear exposition of the difficulty of simultaneously achieving precision and efficiency. Having observed how two of the "improved" algorithms are both inefficient and imprecise, we characterize a general class of algorithms, the adaptive algorithms, which have the potential to do better. This class of algorithms is quite natural; indeed two of the five algorithms we study are adaptive.

In Section 2 we give background material, including the definition of type. In Section 3 we present and analyze the five specific algorithms. Section 4 briefly reviews related work. Finally, in Section 5 we offer our conclusions.

2 Background: Programs, Objects, and Types

We call the program being analyzed the *target program*. In Self, which is prototype-based, a program is a set of objects $\omega_1, \omega_2, ..., \omega_n$ (in a class-based language, a program is a set of classes). It may be the case that ω_{21} is `true` and ω_7 is `nil`. One of the objects, ω_1, say, is designated the "main" object and one of its methods the "main" method. The main object and method, like the `main` function in a C program, define how to *execute* the program. Conceptually, $\omega_1, \omega_2, ..., \omega_n$ are created before execution starts, thus we call them *initial objects*. All other objects that exist during execution are created by cloning either initial objects or other objects that have been recursively cloned from initial objects.

A clone family is an initial object and all objects recursively cloned from it (the class-based equivalent is a class and all its direct instances). The algorithms in this paper do not distinguish between an object and its clones. We capture this abstraction with the term *object type,* the type inference time equivalent of a clone family: object type ≅ clone family.[1] For example, at run time, several `point` objects may exist, but during type inference, they are all represented by a single `point` object type which we denote $\overline{\texttt{point}}$. In general $\overline{\omega_i}$ is the object type of all objects in ω_i's clone family.

A *type* is a subset of $U = \{\overline{\omega_1}, \overline{\omega_2}, ..., \overline{\omega_n}\}$, i.e., it is a set of object types. For instance, $\{\overline{\texttt{true}}, \overline{\texttt{false}}\}$ is the type of a boolean expression such as `3<4`, $\{\overline{\texttt{nil}}\}$ is the type of the expression `nil`, and $\{\overline{\text{int}}, \overline{\text{bigInt}}\}$ is the type of an arithmetic expression such as `11+13`. $\overline{\text{bigInt}}$ is a member because the type inference algorithms perform no range analysis, hence arithmetic overflows cannot be ruled out. The empty type, {}, describes an expression that always fails, e.g., `nil*3`.

An expression E has type $T_E \subseteq U$, if during any execution of the target program, whenever E is evaluated, it yields an object whose object type is in T_E. Informally, E can never evaluate to anything outside T_E. Similarly, a slot[2] S has type $T_S \subseteq U$, if during any execution of the target program, at any moment in time, the contents of S is an object whose object type is in T_S. Informally, S can never contain anything outside T_S.

By definition, if an expression or slot has type T, it also has type T' for any $T' \supseteq T$. The universal type U is sound for any slot or expression, but a type inference algorithm that infers type U for everything is not very useful! Indeed, the goal is to infer the most precise, yet sound, types where a more precise type is a smaller set. For brevity, when the context disambiguates, we usually drop the distinction between objects and object types and simply write types as {`true`, `false`}. The idea that sets of objects (or classes) form a useful notion of type was developed by Graver and Johnson [9, 13].

3 Type Inference Algorithms

We are now ready to analyze the basic type inference algorithm and the four improvements. We present the algorithms in order of increasing power and precision. Table 1

1. This relation must be refined to analyze data polymorphism precisely. We must distinguish integer points from float points even if they are cloned from each other.

2. Slots in Self unify instance variables, parents, formal arguments, and local variables.

lists the algorithms, the section that describes each one, and a reference for each one. The table also specifies which algorithms are “adaptive” (defined in Section 3.4).

Table 1. The five algorithms we analyze.

Name	**Section**	**Reference**	**Adaptive?**
Basic algorithm	3.1	[16]	No
1-level expansion algorithm	3.2	[16]	No
p-level expansion algorithm	3.3	[19]	No
Hash function algorithm	3.5	[2]	Yes
Iterative algorithm	3.6	[20]	Yes

3.1 The Basic Algorithm

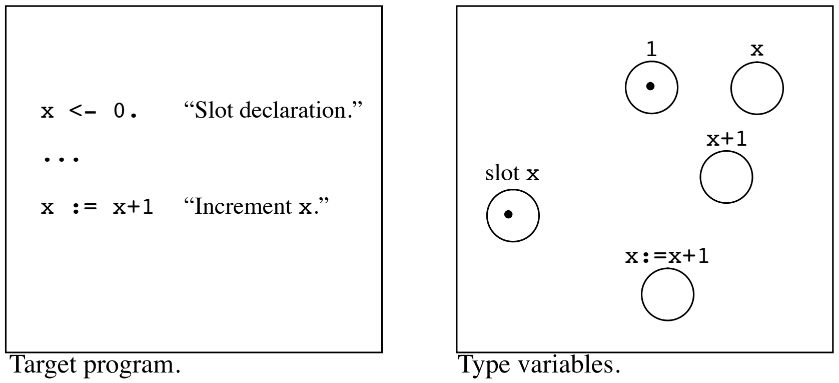
The basic algorithm for constraint-based analysis, described in [16], essentially simulates a general execution of the target program. In this regard it is like abstract interpretation [8, 23]. We describe the basic algorithm in some detail here to present a data flow perspective on constraint-based analysis.

Constraint-based type inference is a three step process. Below we describe each step as the basic algorithm executes it. Looking at the result of the three steps, in Section 3.1.1, we define “templates” which will help us explain and compare the more precise algorithms. Section 3.1.2 summarizes how analysis-time concepts correspond to well-known run-time concepts. Finally, Sections 3.1.3 and 3.1.4 return to the basic algorithm, pointing out its strengths and weaknesses.

Step 1. Allocating type variables. The first step in constraint-based analysis is to associate a *type variable* with every slot and expression in the target program. Fig. 1 shows a program fragment and the corresponding type variables. A type variable is simply a variable whose possible values are types. Initially all type variables hold the empty type, but as type inference proceeds, object types are added to them. Eventually, when the algorithm terminates, the type variables will hold sound types for the slots and expressions they are associated with. Nothing is ever removed from a type variable. Some algorithms exploit this *monotonicity* property. It can also be used to reason about complexity and termination properties of the algorithms, although we will generally refrain from doing so here.

Step 2. Seeding type variables. In the second step we add object types to certain type variables: the type variables are *seeded*. The goal is to capture the *initial state* of the target program by initializing type variables that correspond to slots or expressions where objects are initially found. For example, if a slot has an initial value (an object), we enter the corresponding object type into the type variable for the slot. Thus, in Self, the type variable for a slot `x<-`ω will have $\overline{\omega}$ added to it, i.e., it will now hold the type $\{\overline{\omega}\}$. Similarly, the type variable for a literal object such as `1` will have $\overline{\texttt{integer}}$ added to it. In Fig. 1 a seeded type variable’s member is shown as a small dot.

Step 3. Establishing constraints and propagating. In the final step, we connect the type variables into a network by adding directed edges between some of them. At the conclusion of this step, the type variables will constitute the nodes in a directed



```
x <- 0.    "Slot declaration."
...
x := x+1   "Increment x."
```

Fig. 1. In constraint-based analysis a type variable is associated with every slot and expression in the target program. The target program is shown above in the left frame, and the type variables (in no particular order) in the right frame.

graph. The edges will be *constraints*. A constraint is the type-inference-time equivalent of a run-time data flow. For example, if the target program executes the assignment `x:=y` there is a data flow from `y` to `x`. Intuitively, the significance of the data flow is that any object that can be in the `y` slot can also be in the `x` slot. When the algorithm encounters such a data flow, it will add an edge from the type variable for `y` to the type variable for `x`, to establish the fact that type(`y`) $\subseteq$ type(`x`). Fig. 2 depicts this.

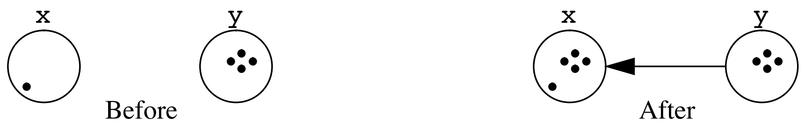


Fig. 2. The situation before and after establishing the constraint for `x:=y`.

Whenever a constraint (i.e., an edge) is added to the network, object types are *propagated* along it. This is also shown in Fig. 2 where all the object types in the `y` node are pushed along the arrow to the `x` node. As more and more constraints are added to the network, the object types that were originally only in the seeded type variables are able to flow further and further. The eager propagation of object types along the edges ensures that subset relations such as type(`y`) $\subseteq$ type(`x`) always hold: whenever an object type is added to the type variable for `y`, it is immediately propagated to the type variable for `x`, so the subset relation is undisturbed

Which constraints should be generated? The answer is "one for *every* possible data flow in the target program"; otherwise soundness of the inferred types cannot be ensured. Different languages will have different constructs that produce data flows, but some general examples can be given:

- An assignment generates a data flow from the new value expression to the assigned variable.
- A variable access generates a data flow from the accessed variable to the accessing expression.

- A message send (or function call) generates data flows from the actual argument expressions to the formal arguments of any invoked method[3]. Furthermore, there is a data flow returning the result of the invoked method to the message send.

Most languages have additional constructs that generate data flows. Self for example has a wide variety of "primitives." Fig. 3 shows the constraint for the clone primitive. Scheme has an `if` expression which is illustrated in Fig. 4. Basic data types such as integers and floats and their operators should also be taken into account. More examples are found in [2, 15, 16, 20].

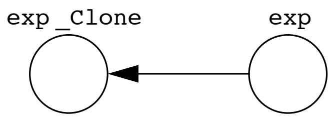


Fig. 3. Self primitives generate data flows. For the clone primitive the resulting constraint is from the expression computing the object to be cloned to the clone expression as a whole. The constraint ensures that the two types are the same, reflecting that the algorithms in this paper do not distinguish between members of the same clone family.

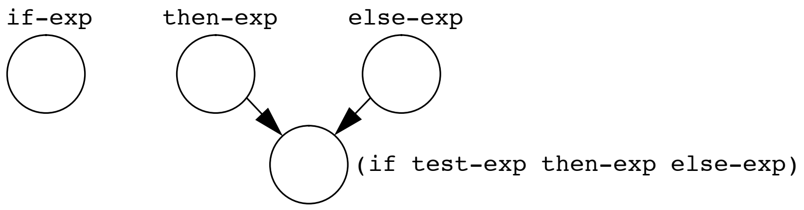


Fig. 4. The data flows generated by a Scheme `if` expression: the *value* of the whole expression is either the value of `then-exp` or `else-exp`. In terms of constraints: the *type* of the whole expression is the union of the types of `then-exp` and `else-exp`.

When a constraint is added, more propagation becomes possible. The reverse also holds: propagation makes new constraints necessary. For instance, if object types are propagated into a receiver expression of a send, the send may now invoke new methods, hence we need new constraints to capture this. Step 3 of constraint-based analysis consists of repeatedly establishing constraints and propagating, until no more can be done. An efficient (polynomial time) implementation using a work list is straightforward; we omit the details.

When the algorithm terminates, the constraint network has evolved into a (large) directed graph. The type of an expression or slot in the target program can be found by consulting the corresponding node in the network. A formal proof of soundness is beyond the scope of this paper, but informally the types are sound because:

- the initial state was correctly captured, i.e., the seeding of type variables accounted for all initial locations of objects, and

3. For message sends we consider the receiver expression an actual argument and the `self` slot of the invoked method a formal argument.

- a constraint was established for every possible data flow, i.e., if an object can flow from an initial location to some other place during execution, the corresponding flow path exists in the constraint network.

This informal soundness argument demonstrates the power of thinking of constraint-based analysis as a flow problem and emphasizing the direct correspondence between analysis time and run time.

3.1.1 Templates

We have described constraint-based analysis as building a large network of type variables connected by constraints. The algorithms we study are easier to understand if we impose structure on the network. A *template* is a part of the network that the basic algorithm generated from a single method. A template includes the type variables for local slots, formal arguments, and expressions in the method as well as the constraints among them. For instance, consider this definition of `max:`, found in `traits number` (the equivalent in Self of `class Number` in Smalltalk):

```
max: a = ( self > a ifTrue: [self] False: [a] ).
```

Fig. 5 shows the template that results when generating constraints for the `max:` function. The template is a box with two input type variables at the top and one output type variable at the bottom. The inputs correspond to the receiver and argument. The inner structure of the template reflects the body of `max:`. We have omitted some details to avoid clutter, but two cases are covered, corresponding to the two possible outcomes of the test `self>a`. In the true case the two inputs flow to a box that selects the first input. In the false case a similar box selects the second input.

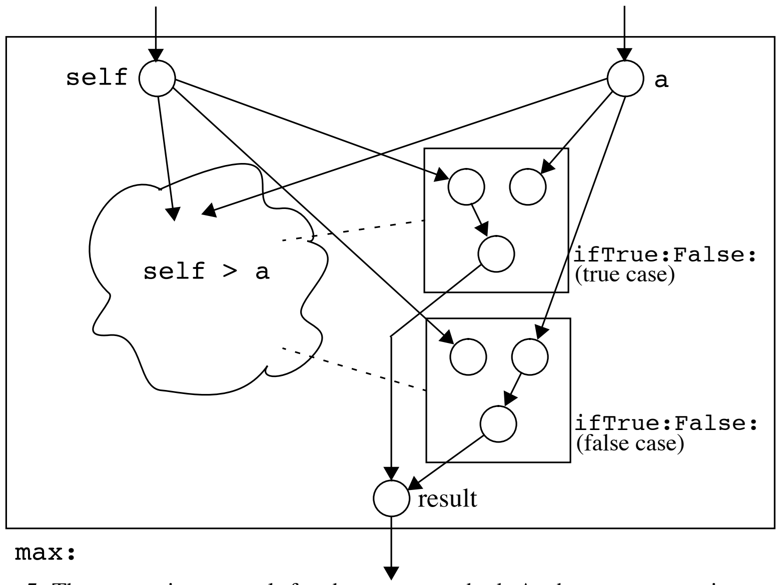


Fig. 5. The constraint network for the `max:` method. At the top are two input type variables corresponding to the receiver, `self`, and the argument, `a`. At the bottom there is a result type variable.

In the remainder of this paper, we work at the higher level of templates rather than individual constraints to make it easier to maintain the full picture. It can almost be argued that this abstraction is *necessary*, since even moderately sized programs generate constraints by the thousands [2, 20].

3.1.2 Relation Between Run Time and Type Inference Time

In constraint-based type inference there is a direct correspondence between program execution concepts and type inference concepts. We have already seen two examples: "object ≅ object type" and "data flow ≅ constraint." In addition we have "activation record ≅ template" as follows. Activation records are created when methods are invoked at run time; templates are created when methods are analyzed at type inference time. *Values* of formal argument slots and local slots can be found by inspecting the fields in activation records; analogously *types* of these slots can be found by inspecting the type variables in templates. It is also possible to establish a correspondence between specific activation records and templates. The correspondence is many to one, since an unbounded number of activation records may be created during execution, whereas only a finite number of templates can be created during type inference (else type inference would not terminate)[4]. Table 2 summarizes the relationship between run time and type inference time.

Table 2. The correspondence between run-time and type-inference-time concepts.

Run time	**Type inference time**
Object/clone family	Object type
Activation record/method	Template
Value of slot	Type of slot
Data flow	Constraint

3.1.3 When Does it Work?

Fig. 6 illustrates the basic algorithm: two different sends, each with an integer receiver and argument are connected to the template for `max:`. By propagating the object types through the network, the type inference algorithm can establish that the return type of `max:` is {integer}, i.e., that both sends have the type {integer}. For simplicity, we have omitted the inner structure of the `max:`-template. The essential property is merely that it contains a flow path from each of the receiver and argument type variables to the result type variable.

The basic algorithm works well when all uses of a given method are "similar." In particular, if there is no polymorphism in the target program, as in Fig. 6, accurate types can be inferred. The algorithm also works well in some polymorphic situations. If the `max:` sends in Fig. 6 have actual arguments of type {integer, float}, propagation through the template again infers precise types for the sends: {integer, float}.

4. This is also why we use the term "template": they are templates for method activations

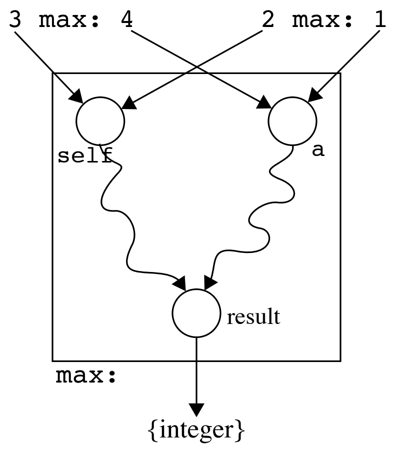


Fig. 6. By propagating integer object types from the inputs to the output, the type inference algorithm can determine that `max:` returns an integer.

3.1.4 When Does it Fail?

The basic algorithm is inaccurate when two or more uses of a given method supply different types of arguments or receivers, whether or not the uses are individually polymorphic. For example, if one send invokes `max:` to find the largest of two integers and another invokes it with floats, the types from the two monomorphic sends are mixed and both will have the imprecise type {integer, float} inferred; see Fig. 7. The key problem is that only one `max:`-template is created. Consequently `max:` has only a single type which must be general enough to describe all uses (equivalently: the single template must cover all possible activation records). The four improved algorithms we describe next, try to avoid this problem by creating several nodes from each method.

Is it possible that the basic algorithm works reasonably well on real programs, but just not on artificial examples that are constructed to show its weakness? The answer is "no." The mixing of types and subsequent imprecision of inference occurs extremely often. We present an example from the Self system. Self has no built-in control structures. Instead they are defined using objects and methods. For example, conditional statements are implemented by a pair of methods with selector `ifTrue:False:` in the `true` and `false` objects respectively:

```
true = (|
  ifTrue: blk1 False: blk2 = (blk1 value). "Invoke true block."
|).
false = (|
  ifTrue: blk1 False: blk2 = (blk2 value). "Invoke false block."
|).
```

Even code this simple cannot be analyzed with a useful degree of precision. Specifically, if the algorithm infers that some particular use of `ifTrue:False:` may return a horse object, suddenly *all* uses of `ifTrue:False:` start returning horses! Fig. 8

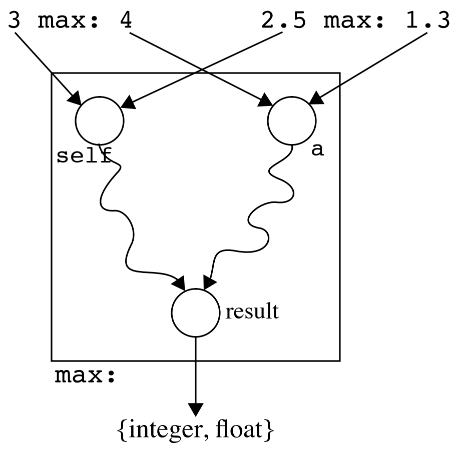


Fig. 7. In the basic algorithm each method has only a single template and thus only a single type. The types inferred for the `max:` sends above are {integer, float} even though one computes the maximum of two integers and the other the maximum of two floats.

shows the constraint network corresponding to the following two conditional statements:

```
b1 ifTrue: [horse] False: [donkey].   "conditional 1"
b2 ifTrue: [tiger] False: [jaguar].   "conditional 2"
```

`b1` and `b2` are boolean receiver expressions. Each conditional connects to two templates: the template for the `ifTrue:False:` method in `true`, and the template for the `ifTrue:False:` method in `false`. The type of each conditional, obtained by combining the contributions from the templates it connects to, is the disappointingly weak {`horse`, `donkey`, `tiger`, `jaguar`}. In general, `ifTrue:False:` will accumulate every type returned in any conditional block in the target program.

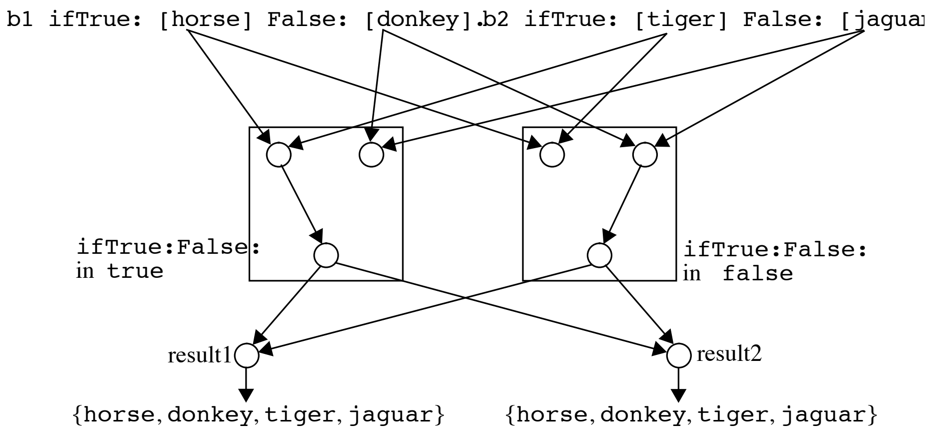

b1 ifTrue: [horse] False: [donkey].b2 ifTrue: [tiger] False: [jagua
ifTrue:False:
in true
ifTrue:False:
in false
result1
result2
{horse, donkey, tiger, jaguar}
{horse, donkey, tiger, jaguar}

Fig. 8. The basic type inference algorithm does not keep different uses of conditional statements distinct. As a result, the inferred types are imprecise.

We have illustrated the shortcomings of the basic algorithm with a specific example from Self. However, the situation that defeats the algorithm is so simple that analogous structures occur frequently in almost any real program, no matter what language it is written in. The `car` and `cdr` functions in LISP, the `new` method in Smalltalk, and iterators on user-defined data structures in any language are prime examples.

3.2 The 1-Level Expansion Algorithm

Palsberg, Schwartzbach, and Oxhøj acknowledged that high-precision type inference requires avoiding the mixing of types [15, 16]. To this end they proposed retyping each method for every send invoking it. Consequently, if the `max:` method is used by two different sends, types are inferred for it twice. This improvement[5] enabled them to type small test programs that had previously proven too hard for other type inference algorithms such as [11].

We express the idea by creating several templates from each method in the program; in contrast, the basic algorithm creates only one template per method. Specifically, the improved algorithm connects each send S that may invoke a given method M to its own M-template. Fig. 9 illustrates this for two different sends that invoke `max:`. Now, instead of having a single M-template that corresponds to all M-activation records, we have several M-templates, each of them corresponding to the M-activation records that some particular send may generate. Equivalently, each method has a fresh type inferred for each send invoking it.

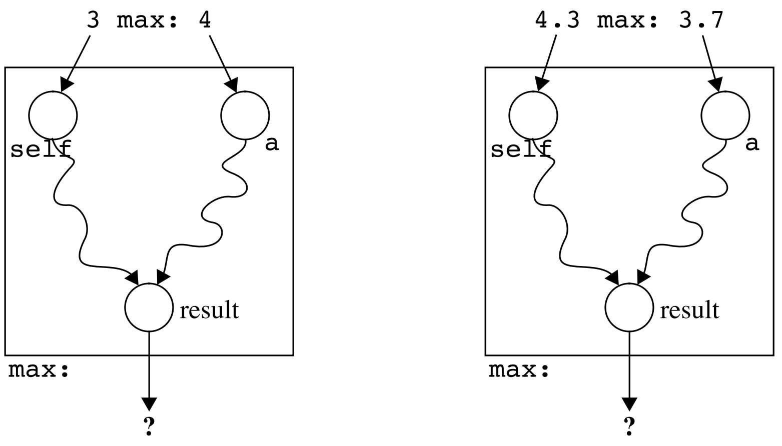


Fig. 9. The 1-level expansion algorithm never shares a template between different sends.

Palsberg and Schwartzbach presented this improvement as a semantics-preserving source to source transformation (expansion) of the target program, followed by appli-

5. Palsberg and Schwartzbach also eliminated inheritance from the target program by copying down all inherited methods. We ignore it for now, but note that the examples we give are not affected by the presence or absence of inheritance. In Section 3.5 we present a more elegant way to achieve the same benefits as the copy down elimination of inheritance.

cation of the basic algorithm to the transformed program. Assume the target program contains n message sends, s_1, s_2, ..., s_n, with some selector S, and k methods, m_1, m_2, ..., m_k, with the same selector. The expansion creates n copies of each method m_j: m_{j1}, m_{j2}, ..., m_{jn}. The i'th copy, m_{ji}, is reserved exclusively for use by the i'th send s_i. (Of course, in most programs every S-send will not invoke every S-method in which case some of these copies are never used).

We call the resulting algorithm the "1-level expansion algorithm," because the expansion of the target program is done only once. Interestingly, expansion during type inference has much in common with inlining during compilation. Expansion allows a method to be typed in a specific context, hopefully improving the inferred types. Inlining allows a method to be compiled in a specific context, hopefully improving the generated code (elimination of the call overhead is often less significant than optimizations made possible by compiling in a more specific context [12]).

3.2.1 When Does it Work?

Below we show that the 1-level expansion is only a slim improvement over the basic algorithm, but first we give an example where the expansion actually helps:

```
    b1 ifTrue: [horse] False: [donkey].
    b2 ifTrue: [tiger] False: [jaguar].
```

Fig. 10 shows that the 1-level expansion algorithm handles such conditional statements better than the basic algorithm: it uses separate pairs of templates for each of the two `ifTrue:False:` sends so the mixing of predators and prey is avoided.

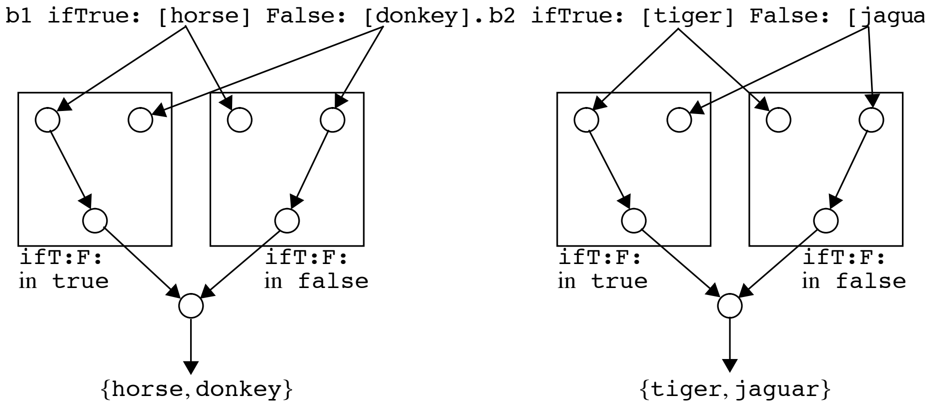


Fig. 10. The 1-level expansion algorithm avoids mixing different uses of `ifTrue:False:` by connecting different sends to distinct templates.

3.2.2 When Does it Fail?

The 1-level expansion only works well when the polymorphic call chain is at most one call deep. Realistically, this is not a significant improvement over the basic algorithm. In particular, precise types still can not be inferred for `max:`. Fig. 11 shows how one use of `max:` with floats still mixes with another use of `max:` with integers. Now the mixing happens in a pair of `ifTrue:False:`-templates that the 1-level expansion algorithm regrettably shares between the `max:`-templates it worked hard to keep separate in the first place.

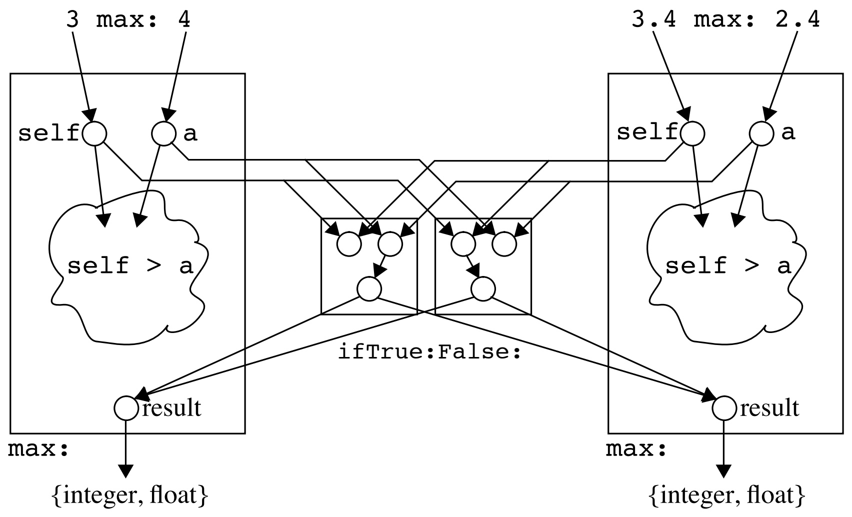


Fig. 11. The 1-level expansion algorithm creates separate `max:`-templates for two different sends that invoke `max:`. The types still mix, though, since the two `max:` templates share a pair of `ifTrue:False:`-templates.

The general problem exposed by the `max:` example is a multi-level polymorphic call chain. First, `max:` may be invoked on either integers or floats. Then, from within `max:`, the `ifTrue:False:` methods in `false` and `true` may be invoked with blocks that return integers or floats. Since there is only a single send of `ifTrue:False:` in the target program, the 1-level expansion algorithm only creates one pair of `ifTrue:False:`-templates. Both the integer- and float-blocks are propagated into this pair of templates and the types mix.

Perhaps the 1-level expansion algorithm works reasonably well on real programs, but just not on artificial examples that are constructed to show its weakness? Again, the answer is “no.” Polymorphic call chains of depth greater than one are common in real programs [20]: all it takes to create such a chain is one method receiving a polymorphic argument and passing it to another method. In the Self system the most compelling example is the method `ifTrue:`. It is inherited by `true` and `false` and implements a one armed conditional statement by sending `ifTrue:False:` to `self` (which is `true` or `false`):

```
ifTrue: blk = ( self ifTrue: blk False: [] ).
```

Since `ifTrue:` is polymorphic, the single send in its body is polymorphic. Hence, every time `ifTrue:` is used in a Self program, the 1-level expansion algorithm will fail to keep the particular type distinct from other uses of `ifTrue:`. Even if a language has built-in control structures, deep polymorphic call chains remain common. Take quicksort: an implementation will likely consist of a polymorphic routine, quickSort that invokes another polymorphic routine, partition. A two level polymorphic call chain results.

Another serious problem with the 1-level expansion algorithm is its inefficiency. In the worst case, the source to source transformation squares the size of the target program. While the full algorithm runs in polynomial time, like the basic algorithm, it can be significantly slower. In practice, performance is quite low because a lot of redundant work is done. For example, every place in the target program that adds two integers causes the integer addition method to be reanalyzed (in Self this method is not simple since it must handle overflows and type coercions). Even a medium-sized program may add integers in hundreds of different places, and the cost of analyzing the integer addition method every time, just to discover again and again that the sum of two integers is an integer, can quickly add up.

3.3 The p-Level Expansion Algorithm

The p-level expansion algorithm is a generalization of the 1-level algorithm: it repeats the expansion step p times before applying the basic algorithm. Phillips and Shepard implemented this algorithm for Smalltalk [19]. In addition to p, which controls how precisely parametric polymorphism is analyzed, their algorithm has a second integer parameter, k, that affects analysis of data polymorphism. The p-level expansion algorithm is also discussed in [17] where it is applied to the untyped lambda calculus.

We consider the p-level expansion unrealistic. The calamity is that the size of the transformed program is exponential in p, effectively ruling out even moderately high values of p. Even p=1 carries a high computational cost yet improves precision only marginally, as we saw in Section 3.2. Furthermore, common code sequences keep showing up that will baffle any feasible value of p. For instance, in the Self system the following family of messages is found in defaultBehavior, a common ancestor for almost any object:

```
defaultBehavior = (|
  value: a With: b With: c With: d = (self value: a With: b With: c).
  value: a With: b With: c         = (self value: a With: b).
  value: a With: b                 = (self value: a).
  value: a                         = (self value)
  value                            = (self).
|).
```

These messages allow blocks to strip unwanted arguments and are executed very frequently. Blocks in Self are invoked by sending value:With:... as in Smalltalk. If a 1-argument block is sent value:With:, the value:With: method inherited from defaultBehavior will execute and send value: to self, effectively stripping off the unwanted extra argument.[6] Consider now the following simple block and two different invocations of it:

```
someBlk := [|:arg| arg].   "This block returns its argument."
someBlk value: 777 With: nil With: nil With: nil.   "Returns 777."
someBlk value: 2.5 With: nil With: nil With: nil.   "Returns 2.5."
```

The first invocation returns an integer, but only after passing the argument 777 through four message sends (`value:With:With:With:` → `value:With:With:` → `value:With:` → `value:`). The second invocation returns a float after a similar call sequence. Since the two invocations supply different types as the first argument, we have a four level polymorphic call chain. To avoid mixing the types during type inference, a prohibitively expensive four level expansion is needed.

Phillips and Shepard's measurements give a feeling for the computational cost of increasing the expansion level. The time needed to infer types for the expression `3+4` is 30 minutes for p=k=0, but over 10 hours for p=k=1 (`3+4` in Smalltalk is quite complex because overflow cannot be ruled out, and thus the bigInt code is also analyzed). Caution should be taken with these numbers, because they do not separate the effects of incrementing p and k. We believe their numbers are indicative, if not in absolute terms at least relatively. (The p=k=0 algorithm applied to and coded in Self is roughly 100 times faster for the `3+4` example; see also [2] which presents extensive measurements of a more advanced algorithm applied to Self programs).

3.4 Introduction to Adaptive Algorithms

Precise type inference must avoid mixing types when mapping run time to type inference time. Types mix if two incompatible activation records are represented by the same template. This observation suggests creating lots of templates from each method so that each template may represent fewer activation records, hopefully lowering the chance of unifying activation records that should preferably be kept separate. *Efficient* type inference, on the other hand, requires processing as few templates as possible, since every template created carries a computational cost. We observed this dilemma with the 1-level expansion algorithm which, sadly, delivered the *worst* of both worlds: it did not create enough templates so types were still mixing, yet it created so many templates that performance was miserable.

To achieve the *best* of both worlds, a good type inference algorithm must walk a fine balance between creating too few templates (resulting in low precision) and creating too many (resulting in slow type inference). Adaptive type inference algorithms strive to achieve this by creating many templates when it appears necessary and sharing templates when it seems safe. A good adaptive algorithm tries to lump similar activation records into a shared template and keep disparate activation records in distinct templates. If successful, the result is precise type inference at a cost proportional to the "amount" of polymorphism in the target program.

6. Throwing away arguments may seem weird. Why pass them in the first place? We chose the example for its conciseness. It is, however, easy to find other examples which do not rely on this particular style of programming. Indeed, the "opposite" situation, successively supplying default arguments by forwarding calls, also yield long polymorphic call chains. Consider this chain: `display`→`displayOn:`→`displayOn:InColor:`→`displayOn:InColor:Scale:`. To display an object on the default display, in the default color and scale, the object can be sent `display`. This method is implemented by a call to the more general method `displayOn:`, etc.

3.4.1 The Critical Situation in Adaptive Type Inference

Before describing specific algorithms let us make the critical situation clear. Consider the send

```
    rcvrExp1 max: argExp1.
```

Assume the type inference algorithm is analyzing this send in the context of a template N_1 and that the send invokes our old friend, the `max:` method in `traits number`. If this is the first time a use of the `max:` is encountered, the strategy is simple: create a new template for `max:` and establish constraints from the above send in N_1 to it. Later the type inference algorithm may encounter a second use of `max:`

```
    rcvrExp2 max: argExp2.
```

The second use of may either be a different send in the target program or it may be the same send as the first use, but now being analyzed in the context of a different template N_2. Given these two uses of `max:` the crucial question is:

Can they share a `max:`*-template or should they each have their own?*

Using the types of the receiver and argument expressions we can express when a template can be shared without impairing precision of type inference:

$$\text{Share a template} \quad \Leftrightarrow \quad \begin{cases} \text{type}(\texttt{rcvrExp1}, N_1) = \text{type}(\texttt{rcvrExp2}, N_2) \\ \text{type}(\texttt{argExp1}, N_1) = \text{type}(\texttt{argExp2}, N_2) \end{cases}$$

In words, the two uses can share a template if they have the same receiver and argument types respectively. The problem, however, is, that we do *not* know these types, since they are the very types we are trying to infer!

In this light, the basic algorithm in its eternal optimism always answers: “share a template.” And the 1-level expansion algorithm says: “share a template if and only if the two *sends* are the same.” In both cases the answer is based entirely on static information such as the program structure. In contrast, an *adaptive* type inference algorithm will base the answer on more than static information, using partial type information or whatever else is available at the decision point.

In summary, the advantage that adaptive algorithms have is that they can fine-tune the analysis process while it is in progress. They can thereby avoid the unfavorable trade-off between efficiency and precision that, e.g., the p-level expansion algorithm exhibited. We are now ready to study two specific adaptive algorithms.

3.5 Hash Function Algorithm

The hash function algorithm was the first adaptive algorithm to be developed [2]. It dramatically improved precision *and* efficiency over the non-adaptive algorithms. As a result, useful types could be inferred for many Self programs for the first time. Today the hash function algorithm appears to be outperformed by the algorithm described in Section 3.6. Central to the algorithm is a hash function that can be computed at the time when a use of a method is processed. The hash function maps a use (i.e., a send in the context of a template) to a hash value:

$$\text{hash: Send} \times \text{Template} \rightarrow \text{HashValue}$$

The hash function is used to decide if templates should be shared: two uses can share a template if and only if they have the same hash value. Different trade-offs between efficiency and precision of type inference can be obtained by applying different hash functions. The specific hash function described below offers a reasonable compromise for most Self programs.

Assume a send S is being analyzed in the context of a template N, i.e., N is one of the templates for the method containing S. This situation actually results in not one but a whole family of hash values, one for each possible receiver of the send:

$$\text{hash\_family}(S, N) = \{(\rho_1,S), (\rho_2,S), ..., (\rho_k,S)\}$$

where

$$\text{type}(S.\text{receiver}, N) = \{\rho_1, \rho_2, ..., \rho_k\}.$$

Consider a hash value (ρ_i,S): the first component, ρ_i, is a possible receiver. Thus, two uses of a given method can share a template only if they provide the same receiver object, ensuring that the type of `self` is a singleton set (this is similar to customization [5]). This improves type inference precision in two ways. First, inherited methods are reanalyzed in the context of every object that inherits them. Second, sends to self, which tend to be common, can be analyzed more precisely because a single target can be found. Palsberg and Schwartzbach obtained the same benefits by expanding away inheritance from the target program [16].

The second component of each hash value is the send itself. Thus, different sends, which often supply different types of arguments, connect to different templates. This is essentially an implementation of the 1-level expansion. One further refinement was used: if a send has no arguments, it is dropped from the hash values, thereby buying back some efficiency by allowing more sharing when it is obviously safe.

One complication remains to be explained: the receiver type is generally not fully known during type inference, so how can it be used in the hash function? The answer is that although types are not fully known during inference, they are known to be monotonically growing sets, hence as long as the type inference algorithm goes back and extends the analysis whenever a receiver type grows, everything works out fine.

3.5.1 When Does it Work?

The hash function does a good job at controlling polymorphism in the receiver. For example, the cascading `value:With:` methods in `defaultBehavior` (see Section 3.3), cause no problems when they fall through all the way to the bottom and return `self`. Consider these two sends:

```
444 value: nil With: nil With: nil.   "This send returns 444."
3.5 value: 100 With: 100 With: 100.   "This send returns 3.5."
```

The first send has `444` as the receiver. During type inference, the hash function forces the send to be connected to a `value:With:With:`-template, N, for which type(`self`, N) = {integer}. This pattern repeats for the `value:With:`, `value:`, and `value` sends until, finally, the type of the result is determined to be the type of `self` in the last template, i.e., {integer}. Since the second send of `value:With:With:` has `3.5` as the receiver it will yield a distinct set of templates in which the type of `self` is {float}, thus there will be no interference between the two sends, and their types can be precisely inferred.

3.5.2 When Does it Fail?

While the hash function controls polymorphic receivers well, it does not excel on polymorphic arguments. In fact, the hash function does not even depend on argument types. Polymorphic arguments appear to be less common in Self than polymorphic receivers, but there are nevertheless important cases where the polymorphism is in the arguments and not the receiver, e.g., `ifTrue:False:` as we have previously seen. Another example of polymorphic arguments is found in methods that implement arithmetic operators such as "+-*/". These operators are "doubly-dispatched," a standard trick to ensures that the types of both the receiver and argument are known in the method that implements the arithmetic operations. Without a precise handling of polymorphic arguments, the type inference algorithm will lose information when analyzing the double-dispatching code, since it is unable to keep distinct types separate when they appear as arguments.

To address these shortcomings, the algorithm described in [2] supplements the hash function with a small number of additional rules. The rules specify that templates should *never* be shared for a certain small group of methods, including the double-dispatching methods, `ifTrue:False:`, and a few others. The computational cost of never sharing these templates is affordable since the methods are small and few. However, the rules affect the robustness of the type inference algorithm, making it sensitive to how certain things in the Self world are coded.

3.6 The Iterative Algorithm

Plevyak and Chien [20] developed an iterative adaptive type inference algorithm and applied it to the Concurrent Aggregates language in the Illinois Concert System language [6], a dynamically typed, single inheritance, object-oriented language. Their algorithm improved the precision for both parametric and data polymorphism, but we will only consider the former.

The iterative algorithm gains precision over the basic algorithm by repeatedly inferring types for the target program. The first iteration is simply the basic algorithm which, for any method, creates a single template and shares it between all uses of the method. In subsequent iterations less will be shared. The idea is to use the type information computed in the preceding iteration to decide whether or not to share templates in the current iteration. For example, considering these two uses of `max:`

```
rcvrExp1 max: argExp1.      (in context N1)
rcvrExp2 max: argExp2.      (in context N2)
```

In iteration i sharing of a template is allowed if and only if it does not result in any loss of precision according to the types inferred in iteration i-1, i.e., if and only if:

$$\begin{cases} \text{type}_{i\text{-}1}(\texttt{rcvrExp1}, N_1) = \text{type}_{i\text{-}1}(\texttt{rcvrExp2}, N_2) \\ \text{type}_{i\text{-}1}(\texttt{argExp1}, N_1) = \text{type}_{i\text{-}1}(\texttt{argExp2}, N_2) \end{cases}$$

The advantage of iterating the type inference is that complete type information is available from the previous iteration to guide the present iteration. Compared with the hash function algorithm, the iterative algorithm has more information available when making the crucial decision. The hash function algorithm could only employ the receiver types in the decision; the iterative algorithm can use the types of both the receiver and arguments, albeit from an earlier and less precise iteration.

Plevyak and Chien described their algorithm using the concepts of "entry sets" and "splitting." It is helpful to know, when reading their paper, that entry sets are in one to one correspondence with templates, and that splitting corresponds to the creation of additional templates for a given method.

When should the iteration stop? Plevyak and Chien suggest two alternatives: stopping after a fixed number of iterations or stopping when a fix-point is reached. The problem with the former is that it may be difficult to select the number of iterations: too few may impact precision and too many may impact efficiency. Typically, though, it seems that 5-7 iterations suffice. The problem with the latter alternative is that a fix-point may never be reached when analyzing recursive routines; see [20] for a discussion of termination issues. Assuming that a fix-point will be reached, the resulting algorithm has the nice property of creating enough templates to avoid mixing of types, but no more. Consequently, it should be expected, as Plevyak and Chien point out, that the time required to analyze a program is proportional to the "amount" of polymorphism in it.

3.6.1 When Does it Work/Fail?

Given enough iterations, the iterative algorithm can precisely analyze polymorphic call chains of any length, whether the polymorphism is in the receiver, the arguments, or both. In particular, it can infer precise types for the cascading `value:With:With:` sends, double-dispatching methods, conditional statements such as `ifTrue:`, and, of course, the `max:` method.

3.7 Summary

Table 3 condenses the most important attributes of the five algorithms that we have discussed. The efficiency column is informal. It is not based on complete measurements or theoretical complexity analyses, although we do have measurements comparing some of the algorithms.

4 Related Work

A significant part of the related work was surveyed while describing the five algorithms in the previous section. Here we focus on work done elsewhere, much of it in the context of analyzing dynamically typed, higher-order functional languages for the purpose of optimizing compilation, elimination of run-time type checks, and partial evaluation.

Static analysis of functional programs has been based on inference of *principal types* where an untyped program is annotated with type declarations that are derived

Table 3. Informal summary of the five type inference algorithms.

Algorithm	Efficiency	Precision	Main drawback	Main advantage
Basic	Fast	Inadequate	Each method has only one type	Simple
1-level Expansion	Slow	Meager	Fails if polymorphic call chain length > 1	Has germ of ideas leading to better algorithms
p-level Expansion	Too slow	Good	Gets too slow before getting precise	Parameterized cost/ precision trade-off
Hash	Fast	Good	Polymorphic arg's need extra rules	Handles polymorphic receivers well
Iterative	Fast	Best	When stop iteration?	No ad-hoc rules

over a grammar of type expressions [14]. Wand has shown how inference of principal types can be formulated as a constraint problem [25]. Wand's constraints have a different interpretation than the constraints found in this paper. His constraints are over the domain of type expressions, allowing him to build type expressions using unification. Our constraints capture data flow between different run-time entities. A further comparison of the two systems, and a proof that the flow-based approach in some sense is more powerful, can be found in [18].

Closer to the constraint-based analysis described here, is Sestoft's *closure analysis* [21, 22] which Bondorf reviews in [4]. The analysis computes, for each application point in the target program, a superset of the closures that may be applied at that point. Closure analysis for higher-order functional languages face similar difficulties as inference of implementation types for object-oriented programs. In both cases, there is a coupling between data flow and control flow. In the functional case, the challenge is to accurately track closures from their creation points to application points that invoke them. In the object-oriented case, the challenge is to accurately track objects that "carry along methods" from their creation points to points they are sent messages.

A distinction has been made between *monovariant* and *polyvariant* analysis by people working on partial evaluation, see Consel [7]. In monovariant analysis each function is analyzed once, analogously to the basic algorithm described in Section 3.1. A polyvariant analyzer, on the other hand, may analyze a function several times to obtain more precise sub-results than is possible if a single analysis must cover all the contexts in which the function can be invoked. The four improved algorithms we presented can thus be considered polyvariant because they may all create multiple templates from a single method. Both monovariant and polyvariant analysis have been based on other approaches than constraints and for other purposes than type inference. Consel describes a polyvariant binding time analysis (binding time analysis determines which expressions in a program can be evaluated at compile time, possibly given a partial specification of the inputs). His analysis is parameterized by a function that determines the degree of polyvariance. In this regard it is similar to the hash function algorithm.

In [17] a constraint-based closure analysis of the untyped lambda calculus is given. On top of the closure analysis a binding time analysis and a safety analysis is defined. The latter determines where type checks must be inserted in the target program to ensure type-safe execution. Palsberg and Schwartzbach proceed by first defining a monovariant analysis which is simply an application of the basic algorithm (Section 3.1) to the lambda calculus. Next they obtain a polyvariant analysis by preprocessing the target program in the same manner as in Section 3.2 (1-level expansion), but tailored for the lambda calculus. They use a special primitive, "select-apply," to succinctly express the result of the expansion. They point out the benefits of iterating the expansion to obtain ever better results as in Section 3.3 (p-level expansion), but do not offer any guidelines on how many times the expansion should be repeated.

5 Conclusions

We have presented constraint-based analysis as a flow problem. This perspective differs from previously used mathematical formalisms by emphasizing the direct correspondence between analysis-time concepts and run-time concepts.

Using the flow perspective, we have studied the basic algorithm and four particular improvements, for the first time presenting these algorithms in a coherent way. This allowed direct comparisons, often giving code fragments on which one algorithm failed but another succeeded. Our comparison and analysis of the algorithms revealed serious weaknesses in how well the basic, the 1-level expansion, and the p-level expansion algorithms deals with parametric polymorphism: they are not precise enough, and furthermore the latter two are too inefficient.

Observing that the p-level expansion algorithm is both slow and imprecise because it does not direct its efforts towards the parts of the target program where it pays off most, we defined a general class of algorithms that have the potential to do better: adaptive algorithms. This is not an artificial class of algorithms. The existing literature describe two algorithms that fall in this class: the hash algorithm and the iterative algorithm. Adaptive algorithms face a difficult task of trying to make use of type information while in the process of computing it. Indeed, despite their successes, the aforementioned two algorithms are somewhat hampered by the difficulty of harvesting type information timely: the hash algorithm loses precision because it can employ receiver type information only, and the iterative algorithm loses efficiency because it resorts to iteration to make type information available.

Watching the five algorithms unfold, seeing how one fails on a code fragment that the next handles comfortably, clearly demonstrates that the real challenge is not in the basics of constraint-based analysis. Rather, the challenge is in the secondary issue of how to guide the analysis towards maximal precision for a given effort spent. Poor guidance yields a slow and imprecise algorithm; good guidance gives a fast and precise algorithm. It furthermore testifies to the importance of accurate guidance that the two adaptive algorithms achieved superior results despite the difficulties they faced in timely obtaining the needed type information.

We have studied parametric polymorphism only. The other side of the coin, data polymorphism, is no less important and probably a harder challenge, given the difficulties that state and assignment cause in other contexts such as semantics. In the literature, proposals exist for how to improve the analysis of data polymorphism over the basic algorithm. These proposals parallel the algorithms for parametric polymorphism:

there is the basic algorithm (which does nothing particular), there is the 1-level expansion where the expansion is on object types rather than methods [15] and its generalization to p levels [19]. An adaptive algorithm, the data polymorphism equivalent of the iterative algorithm, has also been developed [20]. Based on the poor performance of the non-adaptive algorithms on parametric polymorphism, we do not hold much faith in non-adaptive algorithms for data polymorphism. It is not surprising then, that the good results presented in [20] were obtained using an algorithm which is adaptive on both parametric and data polymorphism.

We are currently working on a new adaptive algorithm. It has the potential to be more precise than the algorithms surveyed here, but its main advantage is that it is non-iterative. In principle, this permits a faster algorithm because it does not "throw away" the effort expend in all but the last iteration.

Acknowledgments. Bay-Wei Chang, Urs Hölzle, Jens Palsberg, and David Ungar offered valuable comments on earlier versions of this paper. The author has been generously supported by The Natural Science Faculty of Aarhus University, The Danish Research Academy, and Sun Microsystems Laboratories. The Self project has been supported by the National Science Foundation Presidential Young Investigator Grant #CCR-8657631, Sun Microsystems, IBM, Apple Computer, Cray Laboratories, Tandem Computers, NCR, Texas Instruments, and DEC.

References

1. Agesen, O., L. Bak, C. Chambers, B.W. Chang, U. Hölzle, J. Maloney, R.B. Smith, D. Ungar, M. Wolczko. *How to use Self 3.0* & *The Self 3.0 Programmer's Reference Manual*. 1993. Sun Microsystems Laboratories, 2550 Garcia Avenue, Mountain View, CA 94043, USA. Available by anonymous ftp from self.stanford.edu or www: http://self.stanford.edu/.

2. Agesen, O., J. Palsberg, and M.I. Schwartzbach, Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance. In *ECOOP '93, Seventh European Conference on Object-Oriented Programming*. 1993. Kaiserslautern. Springer-Verlag (LNCS 707).

3. Agesen, O. and D. Ungar, Sifting Out the Gold: Delivering Compact Applications from an Object-Oriented Exploratory Programming Environment. To be presented at OOPSLA'94.

4. Bondorf, A., Automatic Autoprojection of Higher Order Recursive Equations. In *Science of Computer Programming*, 17(1-3), 1991, p. 3-34.

5. Chambers, C., D. Ungar, and E. Lee, An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89, Object-Oriented Programming Systems, Languages and Applications*. 1989. New Orleans, LA. Also published in *Lisp and Symbolic Computation* 4(3), Kluwer Academic Publishers, June, 1991.

6. Chien, A.A., V. Karamcheti, and J. Plevyak, The Concert System — Compiler and Run–time Support for Efficient, Fine-Grained Concurrent Object-Oriented Programs. Department of Computer Science, University of Illinois Urbana-Champaign, Technical Report UIUCDCS-R-93-1815, 1993.

7. Consel, C., Polyvariant Binding-Time Analysis For Applicative Languages, *ACM-PEPM'93*, Copenhagen, Denmark.

8. Cousot, P. and R. Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symposium on Principles of Programming Languages*. 1977.

9. Graver, J.O. and R.E. Johnson, A Type System for Smalltalk. In *Seventeenth Symposium on Principles of Programming Languages*. 1990. ACM Press.

10. Hall, M.W. and K. Kennedy, Efficient Call Graph Analysis. *ACM Letters on Programming Languages and Systems*, 1992. 1(3) p. 227-242.

11. Hense, A.V., Polymorphic Type Inference for a Simple Object Oriented Programming Language With State. Tech. Bericht Nr. A 20/90 (Technical Report), Universität des Saarlandes, 1990.

12. Hölzle, U. and D. Ungar, Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. To appear in PLDI'94, June 1994.

13. Johnson, R.E., Type-Checking Smalltalk. In *OOPSLA '86 Object-Oriented Programming Systems, Languages and Applications*. 1986.

14. Milner, R., A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences,* 17, 1978, p. 348-375.

15. Oxhøj, N., J. Palsberg, and M.I. Schwartzbach, Making Type Inference Practical. In *ECOOP '92, Sixth European Conference on Object-Oriented Programming*. 1992. Utrecht, The Netherlands. Springer-Verlag (LNCS 615).

16. Palsberg, J. and M.I. Schwartzbach, Object-Oriented Type Inference. In *OOPSLA '91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*. 1991. Phoenix, Arizona.

17. Palsberg, J. and M.I. Schwartzbach, Polyvariant Analysis of the Untyped Lambda Calculus. Technical Report, Daimi PB-386, Computer Science Department, Aarhus University, Denmark, 1992.

18. Palsberg, J. and M.I. Schwartzbach, Safety Analysis versus Type Inference for Partial Types. *Information Processing Letters,* 43, 1992, p. 175-180.

19. Phillips, G. and T. Shepard, Static Typing Without Explicit Types. Submitted for publication. Dept. of Electrical and Computer Engineering, Royal Military College of Canada, Kingston, Ontario, Canada, 1994.

20. Plevyak, J. and A.A. Chien, Incremental Inference of Concrete Types, Department of Computer Science, University of Illinois Urbana-Champaign, Technical Report UIUCDCS-R-93-1829, 1993.

21. Sestoft, P., Replacing Function Parameters by Global Variables, M.Sc. thesis 88-7-2, DIKU, University of Copenhagen, Denmark, 1988.

22. Sestoft, P., Replacing Function Parameters by Global Variables. *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture,* London, UK, p. 39-53, ACM Press, September 1989.

23. Shivers, O., Control-Flow Analysis of Higher-Order Languages, Ph.D. thesis. School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213. 1991.

24. Ungar, D. and R.B. Smith, SELF: The Power of Simplicity. *Lisp and Symbolic Computing,* 4(3), Kluwer Academic Publishers, June 1991. Originally published in *OOPSLA '87, Object-Oriented Programming Systems, Languages and Applications,* p. 227-241, 1987.

25. Wand, M., A Simple Algorithm and Proof for Type Inference. *Fundamenta Informaticae,* X, 1987, p. 115-121.