Precise Concrete Type Inference for Object-Oriented Languages

John Plevyak **Andrew A. Chien**

Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801
{jplevyak,achien}@cs.uiuc.edu

Abstract

Concrete type information is invaluable for program optimization. The determination of concrete types in object-oriented languages is a flow sensitive global data flow problem. It is made difficult by dynamic dispatch (virtual function invocation) and first class functions (and selectors) – the very program structures for whose optimization its results are most critical. Previous work has shown that constraint-based type inference systems can be used to safely approximate concrete types [15], but their use can be expensive and their results imprecise.

We present an incremental constraint-based type inference which produces precise concrete type information for a much larger class of programs at lower cost. Our algorithm extends the analysis in response to discovered imprecisions, guiding the analysis' effort to where it is most productive. This produces precise information at a cost proportional to the type complexity of the program. Many programs untypable by previous approaches or practically untypable due to computational expense, can be precisely analyzed by our new algorithm. Performance results, precision, and running time, are reported for a number of concurrent object-oriented programs. These results confirm the algorithm's precision and efficiency.

OOPSLA 94- 10/94 Portland, Oregon USA

1 Introduction

Type information is of central importance for enabling efficient implementations of high-level languages. It can be derived from explicit programmer declarations or via *type inference*, analysis of program structure. It can be used to assist programmers to detect errors, reason about program operation, and in some cases, to optimize the implementation. However, traditional type inference systems infer *principal* or *most general* types, ensuring that the program is a legal composition of data types and their operations. While a great deal of progress has been made with respect to the inference of this type information [13, 14, 4], more precise information is required for optimization of object-oriented languages. For example, the most general type of a *max* function would take any two comparable objects and produce a comparable object result. Thus, a principal typing would ensure that the argument types matched each other and the return type. However, this level of information is inadequate for optimization since optimizing the *max* operation for integers (32 bits), complex numbers (64 bits), bignums (many bits), though they are all numbers, requires different transformations.

Concrete types distinguish implementations of data types and discriminate the actual classes or physical data layouts which occur in a program. Thus concrete type inference can provide the lower level and more specific information which is essential for program optimization. For example, in the case of *max*, concrete type information would dis-

tinguish calls on the basis of the implementation types of the arguments, allowing each to be optimized appropriately. Concrete type information enables optimizations which traditional type information cannot.

While object-oriented languages can ease the task of programming, they make optimization more difficult. Increased use of polymorphism both in modern languages and programming practice decrease the likelihood that type declarations combined with principal types will provide useful concrete type information. This is primarily because polymorphism leverages programming effort by sharing code over a number of uses, confounding concrete type information.

Concrete type inference in object-oriented languages is both especially critical for efficiency and especially difficult to obtain. Object-oriented languages use type-dependent dispatch pervasively, so concrete type information is essential to deriving accurate control flow – a prerequisite to virtually all program analysis and optimization. However, the presence of type-dependent dispatch means that the control flow, type inference, and data flow problems are coupled. Previous work [15] formulated the concrete type inference problem as a monotonic solution of a constraint network, solving all three problems simultaneously. However, it has drawbacks: 1) it does not type many common program structures, and 2) its logical extension to such structures has space and time complexity exponential in program type structure.

Our concrete type inference algorithm extends precision incrementally where needed, consequentially producing more precise type information while requiring less computation. Our algorithm exploits a shallow analysis of the type information to guide the extension of effort into program regions where imprecise results were obtained. The extension discriminates the control and data flow paths that caused imprecisions and reanalyzes the program. The process iterates until precise type information is obtained. The key to making the analysis efficient is the use of *entry sets* and *container sets* which collect similar flow histories together, reducing the cost of flow-sensitive analysis. Our analysis produces an interprocedural call graph in which functions and methods have been cloned (virtually) to eliminate polymorphism. This graph can be used to generate an implementation in which dynamically dispatched calls are statically bound or to do further analysis with more precise control flow information.

Extensible precision and efficient flow sensitive analysis allow our algorithm to precisely type deeply polymorphic structures. Examples of these include nested procedures and data structures, and recursive versions of each. Such polymorphic structures cannot be practically typed by schemes such as [15].

The major contributions of this paper are:

1. A concrete type inference algorithm which can type many previously untypable object-oriented programs.

2. An efficient algorithm which uses resources proportional to program type complexity to obtain precise information.

3. An empirical evaluation of the incremental type inference algorithm using a collection of concurrent object-oriented programs which substantiates the increased precision and practicality of the algorithm.

The basis of our algorithm is a labeling scheme which allows type variables to be distinguishing based on the dynamic program structure. This scheme is flexible, allowing appropriate levels of precision and summary in different parts of the program. Splitting for precision and summarization for efficiency are the critical issues addressed by labels. This extensible precision allows our algorithm to precisely type programs with arbitrarily deep polymorphic structures. The empirical evaluation substantiates the existence of deep polymorphic structures, and the effectiveness and practicality of the algorithm.

The remainder of the paper is organized as follows. Section 2 covers background material, nota-

tion and constraint-based type inference. In Section 3, we introduce our type inference algorithm. Subsequently, in Section 4, we illustrate some uses of the resulting information. Section 5 discusses our implementation of the incremental inference techniques and reports results for a number of programs, some as large as 2,000 lines. Discussion of our results and a summary of related work can be found in Section 6, and the paper is summarized in Section 7.

2 Background

2.1 Project Context

The type inference algorithm was developed as part of the Illinois Concert System. However, the algorithm is general and can be directly applied to a wide range of languages. The goal of the Concert System is to develop portable efficient implementations of concurrent object-oriented languages on parallel machines. This work includes a variety of research in program analysis, optimization and runtime techniques.[1] At present, the Concert system compiles the Concurrent Aggregates (CA) language [9, 10], a dynamically typed concurrent object-oriented language with single inheritance as well as first class selectors, continuations, and messages for execution on the Thinking Machines CM5 [21]. All program examples are written in Concurrent Aggregates.

2.2 Polymorphism

We differentiate *data polymorphism* and *functional polymorphism.* Data polymorphism includes polymorphic variables and polymorphic containers: objects in which an instance variable may contain other objects of more than one concrete type. Functional polymorphism refers to functions which can operate on arguments with a variety of types. Examples of both appear in Figures 1 and

[1]The Illinois Concert System including this type inference system is available from http://www-csag.cs.uiuc.edu. Interested parties can contact achien@cs.uiuc.edu for more information.

```
(function rootclass max (i j)
  (if (> i j) (reply i)
    (reply j)))

(sequential
  (max 1 2)          ;; 1a
  (max 1.0 2.0))     ;; 1b
```

Figure 1: Polymorphic Function

2. We define *level of polymorphism* as the depth of the polymorphic reference path or polymorphic function call path for data and functional polymorphism, respectively. An effective type inference system should produce accurate results in the presence of the many levels of polymorphism found in real application programs.

```
(class A a (parameters i)
  (initial (set_a self i)))
(method A geta () (reply a))

(sequential
  (geta (new A 1))      ;; 2a
  (geta (new A 1.0)))   ;; 2b
```

Figure 2: Polymorphic Container

2.3 Constraint-Based Type Inference

Constraint-based type inference techniques construct a constraint network whose solution is the desired type information. Generally, the network nodes are type variables and the directed edges are constraints. Constraints are induced by data flow, the creation of objects, and the use of variables. Type variables take on values which are sets of concrete types, and the solution for each variable is bounded by constraints from below and above. For example, when an object of type C is created, the type of the variable to which it is assigned must be of at least of type {C}, so a constraint is formed for that type variable. Using $[\![v]\!]$ to denote the type of variable v the basic constraints for creation and assignment are:

$$x = new\ C \longrightarrow [\![x]\!] \supseteq \{C\}$$
$$x = y \longrightarrow [\![x]\!] \supseteq [\![y]\!]$$

These basic constraints reflect local data flow. In addition, there are connecting constraints along the edges of the interprocedural call graph which reflect global data flow. Each method invocation generates constraints between the actual arguments (a_i) and formal parameters (p_i) of the method. The return value is also constrained in an analogous fashion. An example constraint graph and its solution are shown in Figure 3. In equational form, the connecting constraints for an invocation are:

$$x \ \mathit{selector}\ a_0\ a_1 ... a_n \longrightarrow \forall c \in [\![x]\!].$$
$$\texttt{method}\ c :: \mathit{selector}\ p_0\ p_1 ... p_n . \forall i \leq n . [\![p_i]\!] \supseteq [\![a_i]\!]$$

Solving the system of constraints is achieved by maintaining a work pile of invocations (interprocedural edges) which are processed by finding the target method or function and applying local (intraprocedural) and connecting (interprocedural) constraints. These constraints are solved by propagating the changes through all connected constraints. A nice exposition of the basic constraint technique is given in [16].

Values must be propagated through the constraints eagerly because of the coupling between types and control flow. The concrete type of the target of a message send determines the possible flow of control at that send through type-dependent dispatch. The algorithm uses the current solution to approximate the possible interprocedural control flow, putting invocation paths on the work pile as they are discovered. This works as long as the value of each type variable increases monotonically. Figure 3 illustrates the instantiation of an interprocedural constraint. When the constraint for statement `1b` (Figure 1) to the function `max` is created, the variable `i` can be a `float` as well as an `integer`, inducing the creation of an edge from the statement `(> i j)` to the `>` method for floats.

2.4 Imprecision and Type Variables

When the analysis has determined that a variable may only be of one concrete type, it knows it has precise information.[2] We say that an *imprecision* occurs when the constraint network admits a solution with a number of concrete types. At run time, variables in the program text refer to objects of different concrete types in different situations. Imprecisions result from the summarization of these *run time variables* in the dynamic program execution by the static constraint network structure. Thus, the key to resolving imprecisions is to discriminate (avoid summarizing) such variables. The inference algorithm creates a *type variable* for each set of run time variables it wishes to distinguish. Thus separate type variables, each subject to a different set of constraints, can discriminate different uses of a single variable in the program text.

```
(function rootclass leq (i j)
  (reply (or (eq i j) (< i j))))
(function rootclass max (i j)
  (if (leq i j) (reply j) ;; 3a
    (reply i)))

(concurrent
  (if (or (max 1.1 1.2)    ;; 3b
          (max 1 1))       ;; 3c
    ...
```

Figure 4: Multi-level Polymorphic Function

The critical issue for both precision and efficiency is when to use additional type variables for greater resolution. In order to handle polymorphic functions, others have proposed creating separate type variables for each call site at which the function containing the variable was invoked. Similarly, separate type variables would be created for the contents of polymorphic containers based on the point at which the object was created, its *creation point.* Unfortunately, this single level of discrimination is insufficient to infer precise types within common program structures such as polymorphic libraries with multi-level call trees, functions which create and initialize container objects, and polymorphic containers of polymorphic containers (see Figures 4

[2] Those variables which are truly polymorphic will be imprecise under all analyses.

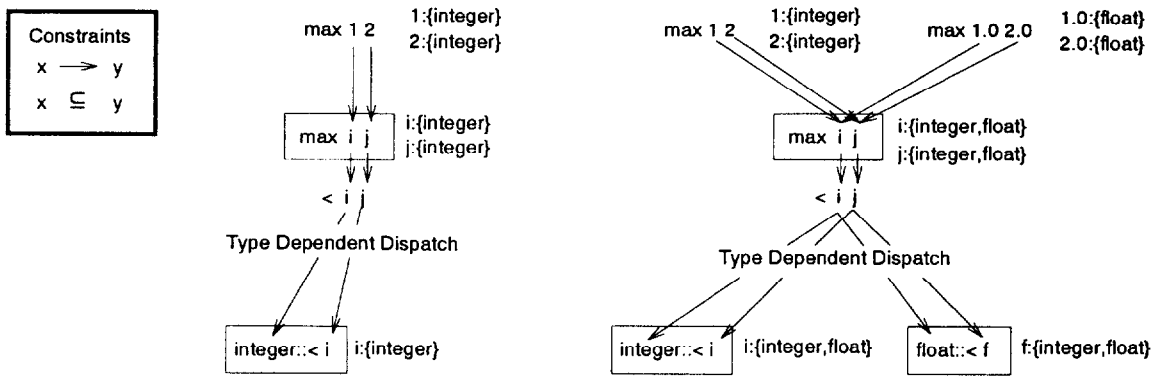

Figure 3: Constraint Graph Example for Figure 1

and 5 for illustrations of these cases).

```
(class A a (parameters i)
  (initial (set_a self i)))
(class B b (parameters i)
  (initial (set_b self i)))
(function rootclass createB (i)
  (reply (new B (* i i))))

(let ((v1 (new A (createB 1)))
      (v2 (new A (createB 1.0))))
  ...
```

Figure 5: Multi-level Polymorphic Container

Extension in the obvious manner, increasing the level of discrimination to some fixed level k, incurs a cost exponential in k, and despite that does not ensure a precise typing. Our incremental type inference algorithm not only types such multi-level polymorphic program structures, it does so efficiently, allocating effort only where necessary. This algorithm is described in detail in the next section. A detailed empirical comparison with other approaches is given in Section 5.

3 Incremental Type Inference

Precise and efficient type inference can be achieved by incrementally extending precision, focusing on detected imprecisions. This allows the typing of programs with arbitrarily complex type structure at a cost proportional to the complexity of that structure. Thus, program sections with simple type structures are typed quickly, and the algorithm concentrates effort in program sections with complex type structure; the result is an efficient type inference algorithm.

The incremental algorithm proceeds as follows. First, a fast analysis is done using the basic constraint-based algorithm (since it allocates a single type variable for each variable appearing statically in the program text, we call this the *static* algorithm). Second, the constraint network is analyzed for imprecisions, and extended locally in the area of the imprecision. This extension invalidates a portion of the solution which is then recomputed. The process of extension and recomputation is repeated until the algorithm determines that no more benefit may be gained. Extension and inference cannot go on simultaneously because the solution must increase monotonically. Modification of a developing constraint network could leave the solution in an state inconsistent with the network.

Since precision must be extended locally for the algorithm to be both efficient and precise, we use an extensible labeling scheme for type variables. These labels are extended in the area an imprecision occurred by tracing the imprecision from the confluence point (the place where two smaller types combine to form a larger union type) back to its source and then increasing precision along the entire path from source to confluence.

The labeling scheme for type variables is de-

scribed in Sections 3.1–3.2, and the mechanisms for extending precision are described in Sections 3.3–3.4. Section 3.5 deals with the issue of recursive procedures and data structures.

3.1 Type Variables: Sets of Run Time Variables

Type variables are used to distinguish different uses of a variable in the program text. As a result, they correspond directly to the precision of inference. Our type inference algorithm creates type variables and labels them to discriminate the run time instances of a program variable. This discrimination produces precise type inference. In this section, we first discuss run time variables and then show how to summarize them with type variables, thus ensuring a finite analysis.

For each textual program variable there may be a number of run time variables which are generated by the execution of the program. Distinguishing these run time variables is critical for precise flow-sensitive analysis. For example, in Figure 1 the function **max** is called in two different environments (`1a` and `1b`) with arguments of type **integer** and **float** respectively. Recording this flow sensitive information for **max** requires two sets of type variables: $\mathtt{i}_{1a}$,$\mathtt{j}_{1a}$ and $\mathtt{i}_{1b}$,$\mathtt{j}_{1b}$ for the textual variables `i`,`j`. We label (discriminate) run time variables by their *execution environment* (e.g. for a stack allocated variable, the call path which resulted in its allocation). In this case, the type variables can be distinguished by one level of their execution environments (call points `1a` and `1b`).

The call path is sufficient to discriminate functional polymorphism but not data polymorphism. In object-oriented languages, the state of the object on which a method is invoked is also part of the execution environment; the value of an instance variable can determine the return type of the method, as illustrated in Figure 2. In this case, the return type of `geta` depends on the type of `a` in the target object. To discriminate these cases we also label type variables with the *creation point* of the object which contains them: $\mathtt{a}_{2a}$ and $\mathtt{a}_{2b}$.

The resulting labeling scheme can be summarized as follows:

$$
\begin{aligned}
\mathit{TypeVariable} &= \mathit{ProgramVariable}_{\mathit{Environment}} \\
\mathit{Environment} &= \mathit{CallPoint} \times \mathit{CreationPoint} \mid \\
&\quad \mathit{Environment}_0 \\
\mathit{CallPoint} &= \mathit{CallStatement} \times \mathit{Environment} \\
\mathit{CreationPoint} &= \mathit{CreationStatement} \times \mathit{Environment}
\end{aligned}
$$

A type variable is a program variable labeled with an environment. The environment is determined by the statement and environment in which the method was called, and the statement and environment where the target object was created. The recursion in the definition ends with the initial environment where the program began ($\mathit{Environment}_0$).

3.2 Entry and Creation Sets

Since programs may contain a potentially infinite number of run time variables, any finite analysis must approximate these with a finite set of type variables. The choice of how to summarize is critical to the precision and efficiency of the type inference since the nodes of the constraint network are type variables, and the value of a node is the union of the concrete types of the set of run time variables it represents. We group execution environments and creation points into *entry sets* at the entry of methods and *creation sets* for each object allocation point. Type variables are now labeled with these sets inducing partitions on the run time variables.

Entry sets summarize collections of calling environments. Each entry set is a collection of interprocedural call graph edges incident on the method or function. Distinct type variables are maintained for each entry set (as in Figure 6), providing flow sensitive analysis. Because each entry set may summarize the information from a number of interprocedural edges, type information for each edge within the entry set is intermingled. This summarization enables the algorithm to be efficient since the summarized variables are only analyzed once for all the edges in the set.

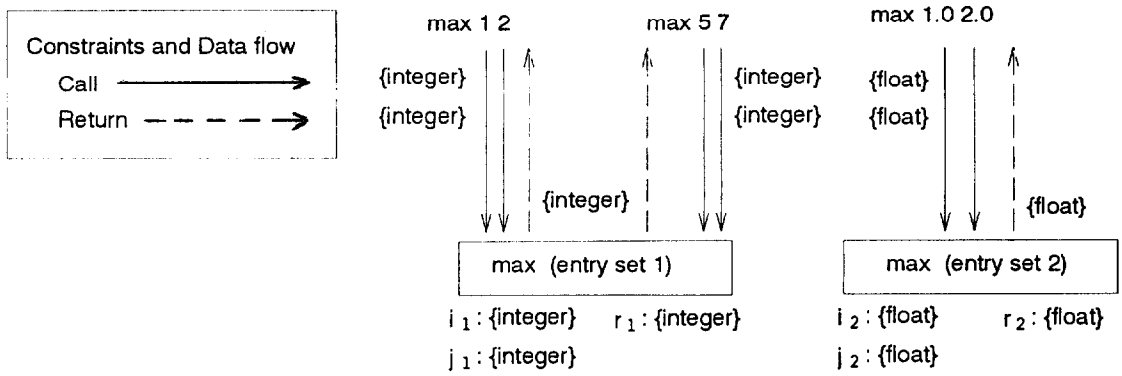

Figure 6: Entry Sets Example for Figure 1

A *creation set* summarizes the collection of run time objects created at a set of creation points. Each creation set collects outgoing data flow edges at creation points. Recall that creation points are the program statement and execution environment where an object was created. Thus, creation sets summarize all the objects created at a number of run time allocation points for the purposes of analysis. Generally, objects which have similar usage are collected together.

To simplify the exposition, in this paper we associate a single entry set and creation set with each environment. The algorithm ensures this by construction. Thus, the labeling scheme is refined as follows:

$$\begin{aligned} \mathit{EntrySet} &= P(\mathit{CallStatement} \times \mathit{Environment}) \\ \mathit{CreationSet} &= P(\mathit{CreationStatement} \times \mathit{Environment}) \\ \mathit{Environment} &= \mathit{EntrySet} \times \mathit{CreationSet} \mid \mathit{Environment}_0 \end{aligned}$$

We use P, the power set, to indicate that entry and creation sets summarize any number of call graph edges or creation points. Since an entry set is associated with a single method and appears in only one environment, it uniquely determines both its creation set and the environment. Taking advantage of this, we will use entry sets and environments interchangeably in the rest of the paper.

3.3 Data Flow Values

The basic constraint-based approach maintains two different data flow values: concrete types and creation points. That is, for each variable, it records the estimated type of that variable, and the places where the objects it contains may have been created (creation sets in our algorithm). These values flow forward in the data flow graph. Our algorithm also maintains the set of selectors or function pointers which each variable may contain since, like the type of a variable, these may influence control flow. Imprecision any of these three values can result in an imprecision in type, and require extension of the analysis. There is an additional type of data flow value which refers to paths through the network itself, but we will defer discussion of it to Section 3.4.2. Some of the algorithm portions described in the following section can operate on more than one of these data flow values and are therefore parameterized by the function $Value$.

3.4 Splitting

Splitting divides entry and creation sets, allocating additional inference effort and increasing the precision of analysis. Each split introduces more type variables, potentially eliminating imprecisions from the inferred types. Choosing the best set to split is important because splitting at the wrong place or choosing partitions that are too small wastes effort. On the other hand, choosing partitions that are not small enough can incur additional iterations of the type inference algorithm.

Splitting an entry set (*function splitting*) divides its edges over a number of smaller entry sets. Splitting a creation set (*container splitting*) likewise di-

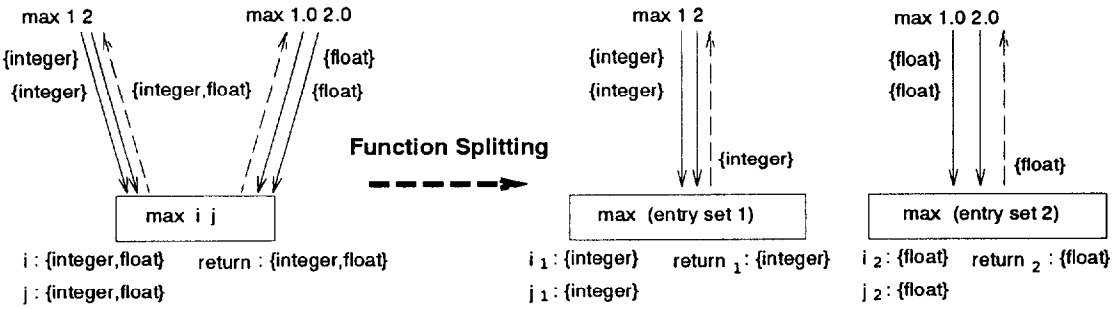

Figure 7: Function Splitting for integers and floats.

vides the creation points of the original creation set over a number of smaller creation sets. While some polymorphic functions can be handled by eagerly splitting entry sets, eager splitting of creation sets is not effective for polymorphic containers. This is because the decision to split a creation set must be made at the creation point. However, the necessity of the split cannot be known until the instance variables are actually used, generally much later in the analysis. A full discussion involves heuristics and termination issues which are beyond the scope of this paper. Our implementation includes eager splitting, and interested readers are referred to [17] for more information. We discuss non-eager function and container splitting in the following sections.

3.4.1 Function Splitting

Function splitting partitions an entry set, separating the type inference for the execution environments in each partition. Our algorithm finds the entry sets that must be split to resolve a particular imprecision, then splits them. Identifying the appropriate entry sets involves tracing back through the constraint network from the imprecision to its primary source. Typically, this is a confluence of type information (a meet $a \wedge b$ where $a, b \neq \emptyset$ and $a \neq b$). Splitting entry sets between the confluence and the imprecision is sufficient to eliminate the imprecision.

In the following paragraphs, we describe the identification of type confluences and entry sets which must be split in detail. First, we define the functions $FlowVars(tv)$ and $BackVars(tv)$ on the constraint network:

FlowVars(tv) Given a type variable tv return those type variables tv' which have direct constraints $Type(tv) \subseteq Type(tv')$.

BackVars(tv) Similar to $FlowVars$ but with $Type(tv) \supseteq Type(tv')$.

These functions are used to follow constraints back to the sources of the imprecision.

$$ConfVar(tv, Value) = \begin{cases} \{tv\} & \text{if } \exists b \in BackVars(tv) \wedge \\ & \quad Value(tv) \neq Value(b) \\ \emptyset & \text{otherwise} \end{cases}$$

$$ConfVars(tv, Value) = ConfVar(tv) \cup \{b \mid b \in BackVars(tv) \wedge ConfVars(b, Value)\}$$

To find the sources of an imprecision in tv, we find the type variables at confluences involving some portion of $Value(tv)$. $ConfVar(tv, Value)$ indicates that the type variable tv is a confluence point with respect to $Value(tv)$, and thus a possible source of the imprecision. The function $ConfVars(tv, Value(tv))$ finds all type variables which are at confluences contributing to the final imprecision.

Imprecision can also arise from interprocedural control flow ambiguity due to imprecision in the selector or the type of the target at a message send. With such imprecisions, we trace back through the constraint network to find the confluences which cause the imprecision. Also, imprecision in the creation set of the target of a message can result in imprecision in instance variables

within the method. We extend $ConfVars(tv, im)$ to $ConfVars'(tv, im)$ to handles these three cases.

$$
\begin{aligned}
&ConfVars'(tv, Value) = ConfVars(tv, Value) \cup \\
&\quad \{tv'' \mid tv' \in ConfVars(tv, Value) \wedge \\
&\quad ((tv' \text{ is an argument or return variable of } send) \wedge \\
&\quad (tv'' \in ConfVars'(TargetOfSend(send), Type) \vee \\
&\quad tv'' \in ConfVars'(TargetOfSend(send), CreationSets) \vee \\
&\quad tv'' \in ConfVars'(SelectorOfSend(send), Selectors))
\end{aligned}
$$

The three occurrences of *ConfVars'* on the bottom trace back imprecisions in target type, creation sets, and the message selectors respectively. This identifies all causes of an imprecisions which can be resolved with function splitting.

Figure 7 illustrates function splitting involving the `max` function from Figure 1. At the left, the actual arguments for the formal parameters `i` and `j` coming from `max 1 2` and `max 1.0 2.0` have different concrete types, so there is a type confluence. The imprecision manifests itself in the imprecise return type {`integer`,`float`}, when it is clear that the return type for the first call is `integer` and for the second call it is `float`. Splitting the entry set introduces two sets of type variables i_1,j_1 and i_2,j_2, eliminating the confluence and producing a precise typing.

3.4.2 Container Splitting

Container splitting partitions creation sets, separating the type information for the creation points in each partition. Container splitting is necessary when there is an imprecision in type at an instance variable. Partitioning the creation points allows a more precise typing for the objects represented by each partition, reducing imprecisions caused by data polymorphism. The term container splitting is used because we must split the type information for the object which "contains" a polymorphic reference.

Figure 8 is an example of container splitting based on the program example in Figure 2. On the left, the two creation points, `(new A 1)` and `(new A 1.0)` are part of the same creation set. As such, they constrain the value of the instance variable `a` and consequentially the return type of `geta` to be `integer` or `float`. Splitting the creation set discriminates the two cases, allowing a function splitting on `geta` to produce a precise typing of `geta` for both cases.

Container splitting is more complex than function splitting because the point of confluence (the instance variable) is linked to the creation points by data flow, not control flow. As a result, the creation set which must be split may be distant from the imprecision. Thus splitting the creation set is not enough, we must ensure the additional discrimination introduced by the splitting is maintained to the point of the imprecision.

Container splitting involves four basic operations.

1. Identifying the assignments to the instance variable which give rise to the imprecision.
2. Identifying the paths from the origin(s) of the creation set to the methods containing the assignments.
3. Ensuring a container set split will increase discrimination at the imprecision by splitting along these paths.
4. Resolving the imprecision via container set splitting.

Container splitting addresses type imprecision for instance variables, so the first step is to identify the assignments which produce the imprecision. These are conflicting assignments to the same instance variable of objects from the same creation set. After we have the assignments, we find data flow paths from the creation points in the creation set to the assignments. These paths must propagate any discrimination we introduce by container splitting, or we will fail to refine the imprecision. We ensure this discrimination will be preserved along the data flow paths by splitting functions and containers where necessary. With paths in hand, we resolve the assignments into different creation sets by splitting the container sets. This overall algorithm is what we term container splitting.

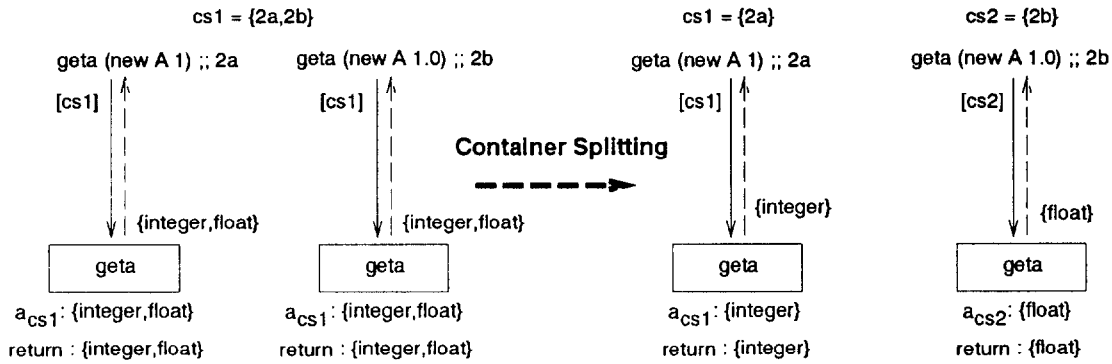

Figure 8: Container splitting for imprecision at a.

Identifying the Assignments First we find the type variables which carry the imprecise information to the instance variable with the function $AssignSets(v, Value)$. As before, it is parameterized with the function $Value$ which can be any imprecise data flow value. We begin with a type variable, v, which corresponds to an instance variable at the point of an imprecision and end with a set of sets of type variables, each of which represents a different use of the instance variable.

$$
\begin{aligned}
AssignSets(v, Value) &= AS(BackVars(v), Value) \\
AS(vs, Value) &= \texttt{if } vs = \emptyset \texttt{ then } \emptyset \\
&\quad \texttt{else } \{ASet(vs, Value)\} \cup \\
&\quad\quad AS(vs - ASet(vs, Value), Value) \\
ASet(vs, Value) &= \\
&\quad \{v \mid v \in vs \wedge Value(v) = Value(First(vs))\}
\end{aligned}
$$

The variables in $BackVars(v)$ correspond to the right hand sides of the assignments to the imprecise instance variable. For efficiency, we use $AS(vs, Value)$ to form subsets of the type variable in vs which are identical with respect to $Value$. The function $ASet(vs, Value)$ finds one subset of vs with the same $Value$ using the function $First(vs)$ which selects the first type variable in a set vs.

Identifying the Paths For element as of $AssignSets(v, Value)$ representing a different use of the instance variable v, we compute the path from the type variables representing the *containing* objects to their creation points. First, we find the objects whose instance variables were assigned from as a portion of $Value(v)$. Then we compute the path between these objects and their creation points. A new creation set would have to take this path in order split the instance variable v for the assignment set as and eliminate the imprecision.

$$
\begin{aligned}
CPath(as) &= Closure(BackVars, ContainingVars(as)) \\
ContainingVars(as) &= \{c \mid c = Target(e) \wedge \\
&\quad e \in Edges(es), es \in EntrySet(v) \wedge v \in as\}
\end{aligned}
$$

We compute the path $CPath(as)$ back to the creation point for the variables in the set as by taking the closure of $BackVars$ over the set of containers. The function $ContainingVars(as)$ finds the type variables which represent the containers of instance variables assigned from the elements of as. It uses $EntrySet(v)$ which returns the entry set which determines v,[3] and $Edges(es)$ which returns the interprocedural call graph edges summarized by the entry set es. The $Target(e)$ of edge e is the type variable on which the method was invoked.

The path $CPath(as)$ is that which would be taken by a new creation set whose existence would eliminate the portion of the imprecision $Value(as)$ (see Figure 9) This path must be distinct from the other paths computed for each element of $AssignSets(v, Value)$ because the union meet of the $CreationSets(v)$ data flow value would make it impossible to separate out the uses of the new

[3]Each type variable is determined by its environment, and that environment is determined by an entry set.

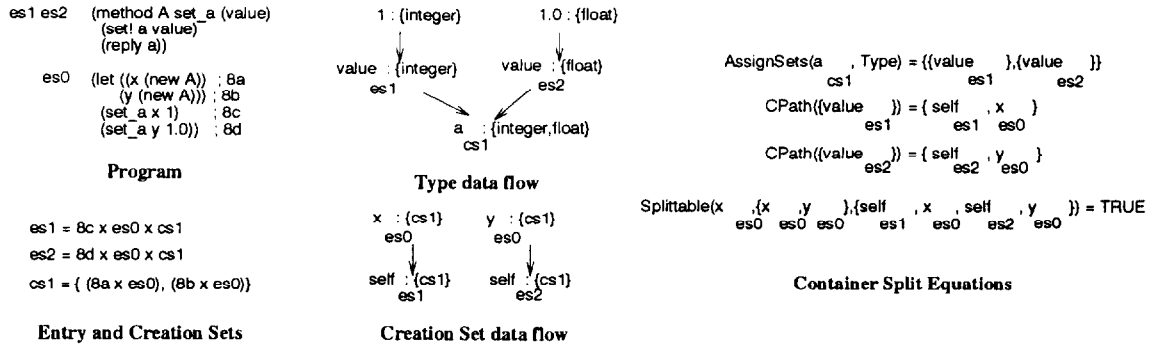

Figure 9: Example of Container Splitting Equations

creation set. Since the appearance of a type variable on more than one of these paths represents a secondary imprecision, for each type variable we need to know the subset paths in which it is contained.

$$\begin{aligned} TvCPaths(tv, ps) &= \{cpath \mid cpath \in ps, tv \in cpath\} \\ AllCPaths(v, Value) &= \{cpath \mid cpath = CPath(as) \wedge \\ &\quad as \in AssignSets(v, Value)\} \end{aligned}$$

We define the function $TvCPaths(tv, ps)$ to represent the subset of the paths ps which contains tv. For example, given all the paths which carry an imprecision of a particular $Value$ at a particular variable v, the set going through tv is $TvCPaths(tv, AllCPaths(v, Value))$.

Ensuring Discrimination Using the paths determined above, the next step is to determine where confluences of the potential creation sets represented by these paths occur. This is the additional type of data flow value which we discussed above, imprecisions in which we must detect and eliminate. We can extend our definition of $ConfVar$ to include imprecisions in these potential creation set paths:

$$ConfVar'(tv, Value) = \begin{cases} \{tv\} & \text{if } \exists b \in Vars(tv, Value) \wedge \\ & \quad Value(tv) \neq Value(b) \\ \emptyset & \text{otherwise} \end{cases}$$

The new $ConfVar'$ uses the $Vars(tv, Value)$ function which is either $BackVars(tv)$ as before or $FlowVars(tv)$ when $Value$ refers to the paths themselves. Since the paths are separate at the imprecision and converge at the instance variable, they can be thought of as flowing backward in the data flow graph. *AssignSet* requires a analogous change, and the rest of the algorithm is identical.

The type variables which are $ConfVar's$ for these paths need to be split, either by splitting the entry sets of the enclosing functions (for normal type variables) or by splitting the creation set (for instance variables) before we can split the creation set for which *cp* is a creation point. As with function splitting, it is important to consider those variables which might indirectly contribute to the imprecision by way of another imprecision.

Resolving the Imprecision The last step is the actual splitting of creation sets. When two or more paths do not share any type variables, then the creation set can be split. A new creation set is created for each path or set of paths which do not share type variables. Figure 9 provides an example of using these equations to determine that a creation set can be split. The new creation set will cause the instance variable at the point of the imprecision to split thus removing the imprecision. We will use the type variable which describes the result of an object creation statement *cp* to stand in for the creation point at those statements.

$$\begin{aligned} ps &= AllCPaths(v, Value) \\ cps &= \{cp \mid cp \in cpath \wedge cpath \in ps \wedge CreationResult(cp)\} \\ Splittable(cp, cps, ps) &= \forall cp' \in cps \wedge \\ &\quad cp' \neq cp \wedge TvCPaths(cp, ps) \not\cap TvCPaths(cp', ps) \end{aligned}$$

We will use ps, the set of all type variables on any path between the imprecision and a creation point, and cps, the set of all type variables resulting from a creation statement as indicated by $CreationResult(cp)$. The function $Splittable(cp, cps, ps)$ then determines if a creation set with creation points cps can be split, creating a new creation set containing cp (a member of cps). This is the case when no type variables occur on both the path between cp and the imprecision and the path between any other creation point cp' in cps and the imprecision. Since the paths flow backward in the data flow graph with a union meet, $TvCPaths(cp, ps)$ summarizes the $TvCPaths$ for all type variables between cp and the imprecision. Hence, when the intersection of $TvCPaths(cp, ps)$ and all other $TvCPaths(cp', ps)$ is empty, a new creation set containing cp can be split from that containing cps.

Once we have both removed all of the intervening confluence points between the creation points and the imprecision point and have split the creation set, the instance variable at the imprecision point will be split. The new type variables for the instance variable will each have portions of the original data flow value, eliminating the confluence and consequently the imprecision.

3.5 Recursion

Recursion in functions or data requires careful handling to ensure that our algorithm terminates and does so with precise type information. Splitting some recursive functions is required to type polymorphic recursive functions precisely. However, splitting them in all cases where precision may be increased can lead to non-termination. We distinguish three types of recursion: 1) recursive data structures (container recursion), 2) function recursion, and 3) function-creation recursion. The first case is the easiest. Creation sets are only split when the algorithm can find a distinct path for the new creation set, ensuring that an imprecision will be eliminated. Recursion is bounded in the path finding algorithm by determining paths only once for each creation point.

For the other cases, we prevent non-termination by identifying edges which are part of recursive cycles. After each iteration and before splitting, we identify the strongly connected components (SCCs) in the graph where nodes are the entry and creation sets and arcs are 1) interprocedural calls from entry set to entry set, 2) creation set to the environment they determine (which uniquely determine an entry set) and 3) entry sets to the creation sets, one of whose creation points they determine. The SCCs in this graph contain the sets of entry sets that are recursive or that create an object on which they are then invoked. Edges between entry sets in the same SCC are not split. In addition, splitting edges which point to recursive cycles can also lead to infinite execution as it may successively "peel" recursive cycles. Thus, these edges are also prohibited from splitting beyond a constant level. Note that allowing edges entering the cycle to split to a constant level is enough to enable typing of recursive structures with a period less than or equal to the constant. These techniques are discussed in detail in [17].

3.6 Safety, Termination and Complexity

The basic constraint-based type inference algorithm is safe because it enforces the program's data flow and invocation type constraints [15]. Since the incremental algorithm does not change the values of the constraint network, but only refines the analysis by partitioning and applying the constraints more precisely it is also safe. This remains true so long as the connecting constraints represent a conservative approximation of the interprocedural call graph, which the algorithm also ensures. A more detailed discussion of these issues can be found in [17].

Termination is ensured because there is only finite unfolding of a program without recursion and recursion is blocked beyond a constant level (see Section 3.5). While the complexity of the algorithm is bound by the finite number of type variables, this number is exponential if the level of polymorphism

in a program grows linearly in program size. In practice we do not expect and have not found such programs. In fact, our measurements show that the level of polymorphism in programs increases relatively slowly with program size.

4 Use

This analysis produces a wealth of information about type information, data and control flow. In the Concert System this information is used for global constant propagation, removing unreachable methods (tree shaking), and cloning as well as for debugging and the insertion of type checks. We will cover cloning and inserting type checks in greater detail.

4.1 Cloning

Cloning makes new copies of a method for different invocation contexts, such as the concrete types of its arguments. This information is then used directly to optimize the cloned method as well as any dependent calls. The resulting implementation can leverage a few dynamic dispatches to execute large tracts code with few if any dynamic dispatches. Of course, these tracts are now candidates for a variety of classical optimizations.

The organization of the type inference results are particularly well-suited for eliminating dynamic dispatches, as they contain entry sets which indicate productive clones of methods. By using these entry sets to direct code replication, we can control replication, and direct it to where it will do the most good. The Concert compiler produces method clones using entry sets as discussed in [17].

4.2 Type Checking

For statically typed languages, type checking can be done before type inference, so we know that all messages and functions will resolve legally during type inference. For dynamically typed languages, we have no such guarantee. However, the results of concrete type inference can ensure the absence of run time type errors allowing the compiler to remove type checks or to alert the programmer to possible program errors.

After each type inference iteration has completed we determine where the typing is not adequately precise to ensure that no run time type errors will occur. These points of imprecision occur where any type variable, a target of a message send, includes types which fail to support any or all of the selectors which may be sent to it. By applying function and container splitting to these imprecisions, we type check the program. For programs which do not type check, we can use the same information to insert run time type checks. The Concert compiler reports the insertion of type checks to the user as warnings which often indicate programming errors.[4]

5 Implementation and Empirical Results

We have implemented the incremental type inference algorithm and tested it on more than 35,000 lines of Concurrent Aggregates (CA) programs. The implementation is fully integrated into the compiler and complete; no language features were excluded. In this section, we present excerpts from our empirical studies; a concise table appears in the appendix, with a complete report in [17].

Our test suite spans a range of program sizes between 40 and 2000 lines. The **ion** program simulates the flow of ions across a biological membrane. **network** simulates a queueing network. **circuit** is an analog circuit simulator. **pic** is a particle-in-cell code. The **man** program computes the Mandelbrot set using a dynamic algorithm. **tsp** solves the traveling salesman problem. The **mmult** program multiplies integer and floating point matrices using a polymorphic library. **poly** evaluates integer and floating point polynomials. **test** is a synthetic code designed to test the algorithm's effectiveness. All programs were compiled with the standard CA prologue (240 lines of code).

[4] This enables safe debugging of programs written in a development mode since type inference with type checks

Algorithm	*Progs Typed*	*Progs Failed*	*Type Checks*	*Average Seconds*
PRECISE	9	0	0	199
PALSBERG	3	6	99	150
STATIC	0	9	718	34

Figure 10: Precision of Type Inference Algorithms

We implemented three different algorithms: STATIC with one type variable per program variable, PALSBERG with one level of constant function and container splitting, and PRECISE which is our algorithm. Figure 10 shows that STATIC was fast, but unable to type even simple programs. PALSBERG fared little better, typing only three of nine programs. In contrast, PRECISE was able to type all the programs. Furthermore, the type information produced by PRECISE eliminated the need for any run time type checks while PALSBERG and STATIC required many in the final code. All run times given are for our CMU Common Lisp/PCL implementation on a Sparc10/31.

Figure 11 shows that our algorithm not only produces better type information, it generally does so faster. In two of the three cases, where both PRECISE and PALSBERG were able to type the program, the PRECISE algorithm was much faster. The reason for this is that PRECISE focuses its effort on regions of the program where it is productive. Of course, when PRECISE returned greater type information, it often required much longer run times.

Not only does PRECISE produce precise typings, the entry set and container set mechanisms produce a concise typing.[5] That is, the incremental type inference algorithm does not unnecessarily split type variables. This is especially important when the result of type inference is used with cloning to eliminate dynamic dispatches. The "conciseness" of a precise typing reduces the number of clones required to produce output code without dynamic dispatches. In Figure 12 we see that if we produced new clones for the type variables required by the algorithm, PRECISE would produce a program with between 1.5 and 2.5 as many methods while eliminating almost all dynamic dispatches. This is much better than the PALSBERG typing (not precise), even ignoring the fact that run time type checks are still required. Using the PALSBERG typing would produce a 2.5 - 4 times code expansion but eliminate many fewer dynamic dispatches. The number of dynamic dispatches eliminated and the actual effect on code size using a more efficient algorithm is covered in detail in [17].

catches all run time type errors.

[5]One measure of this is the number of type variables required (see appendix).

Program	*Lines*	PALSBERG *Typed?*	*Time Sec.*	PRECISE/ PALSBERG
ion	1934	NO	714	1.2
circuit	1247	NO	290	2.1
pic	759	NO	363	2.5
tsp	500	NO	56	1.4
mmult	139	NO	78	3.5
test	39	NO	15	5.1
network	1799	YES	234	.65
mandel	642	YES	25	.42
poly	41	YES	18	2.2

Figure 11: Efficiency of Type Inference Algorithms

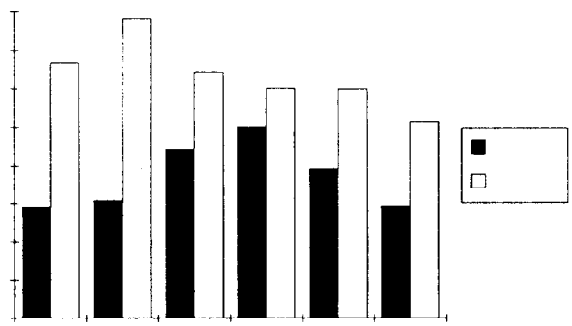

Figure 12: Clones per Method by Algorithm

6 Discussion and Related Work

While in general, the static typing of all programs which will not produce run time type errors is unde-

cidable, the PRECISE algorithm was able to type all of our application programs. The empirical studies indicate that our incremental type inference algorithm significantly extends the range of program behaviors that can be typed. However, there are some possible program structures which will require run time type checks, even with our improved algorithm. These include: 1) programs which store a variety of types in a single array, 2) programs which build variant records and compute the tags, and 3) programs which reuse storage to store different types (such as a program and its garbage collector).

The use of non-standard abstract semantic interpretation for type recovery in Scheme by Olin Shivers [19] provides a good basis for this and other work on practical type inference. In particular, the ideas of a call context cache to approximate interprocedural data flow and the reflow semantics to enable incremental improvements in the solution foreshadow this work.

Iterative type analysis and message splitting using run time testing are conceptually similar techniques developed in the SELF compiler [6, 7, 8]. Iterative type analysis uses structures similar to entry sets, but never attempted to accurately type an entire program. Instead it recovers information from small regions. Run time tests are used to select optimized code sequences when a particular alternative is considered likely. We expect that these techniques and virtually all other optimization of object-oriented languages will benefit greatly from the precise type information generated by our improved inference techniques.

Type inference in object-oriented languages in particular has been studied for many years [20, 12]. Constraint-based type inference is described by Palsberg and Schwartzbach in [16, 15]. Their approach was limited to a single level of discrimination and motivated our efforts to develop an extendible inference approach. Recently Agesen has extended the basic one level approach to handle the features of SELF [22] (see [1]). However, the problems with precision and cost inherent in a single pass approach are tackled by exploiting specialized knowledge about the SELF language [2].

The soft typing system of Cartwright and Fagan [5] extends a Hindley-Milner style type inference to support union and recursive types as well as insert type checks. To this Aiken, Wimmers, and Lakshman [3] add conditional and intersection types enabling the incorporation of flow sensitive information. However, these systems are for languages which are purely functional where the question of types involving assignment does not arise and extensions to imperative languages are not fully developed. Lastly, our algorithm shares some features of the closure analysis and binding time analysis phases used in self-applicative partial evaluators [18], again for purely functional languages.

7 Summary and Future Work

We have developed and implemented an algorithm for precise concrete type inference in object-oriented languages. This algorithm uses entry sets and creation sets to incrementally extend precision and direct type inference effort to where it is fruitful. These techniques make efficient inference of concrete types in programs with many levels of polymorphism in functions and data structures practical.

We have implemented these techniques in the Illinois Concert compiler and have used them to infer concrete types on a number of programs. These programs contain first class selectors, continuations, and messages and are written in the dynamically typed concurrent object-oriented language Concurrent Aggregates. Our empirical results indicate that the incremental type inference algorithm is viable, practical, and productive. Not only is the resulting concrete type information precise, the run time of the algorithm is reasonable for use in an optimizing compiler.

Our compiler currently uses the type information with cloning to eliminate dynamic dispatch, inline functions and methods, unbox variables, as well as for interprocedural constant propagation and locality approximation. In the future we will also explore more efficient implementations of the type

inference algorithms, including templates [15] and sparse evaluation graphs [11] which may reduce the memory and compute time requirements. We are expanding the framework for more interprocedural analyses, and increasing the type domain for summarization [3] and to include integer ranges.

8 Acknowledgements

We would like to thank Vijay Karamcheti, Xingbin Zhang, Julian Dolby and Mahesh Subramaniam for their work on the Concert System and Tony Ng, Jesus Izaguirre and Doug Beeferman for writing applications and for working with early versions of the algorithm's implementation. We would also like to thank our reviewers for their comments.

The research described in this paper was supported in part by National Science Foundation grant CCR-9209336, Office of Naval Research grants N00014-92-J-1961 and N00014-93-1-1086, and National Aeronautics and Space Administration grant NAG 1-613. Additional support has been provided by a generous special-purpose grant from the AT&T Foundation.

References

[1] O. Agesen, J. Palsberg, and M. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. In *Proceedings of ECOOP '93*, 1993.

[2] Ole Agesen. Personal communication, 1993.

[3] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Twenty First Symposium on Principles of Programming Languages*, pages 151–162, Portland, Oregon, January 1994.

[4] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, and Robert van Gent. Safe and decidable type checking in an object-oriented language. In *Proceedings of OOPSLA'93*, pages 29–46, 1993.

[5] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, Ontario, Canada, June 1991.

[6] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–60, 1989.

[7] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 150–60, 1990.

[8] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *OOPSLA '91 Conference Proceedings*, 1991.

[9] Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs.* MIT Press, Cambridge, MA, 1993.

[10] Andrew A. Chien, Vijay Karamcheti, John Plevyak, and Xingbin Zhang. Concurrent aggregates language report 2.0. Available via anonymous ftp from cs.uiuc.edu in /pub/csag or from http://www-csag.cs.uiuc.edu/, September 1993.

[11] J. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse dataflow evaluation graphs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1990.

[12] J. Graver and R. Johnson. A type system for smalltalk. In *Proceedings of POPL*, pages 136–150, 1990.

[13] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[14] John C. Mitchell, Furio Honsell, and Kathleen Fisher. A lambda calculus of objects and method specialization. In *1993 IEEE Symposium on Logic in Computer Science*, pages 26–38, June 1993.

[15] N. Oxhøj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In *Proceedings of OOPSLA '92*, 1992.

[16] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proceedings of OOPSLA '91*, pages 146–61, 1991.

[17] John Plevyak and Andrew Chien. Precise object-oriented type inference and its use in program optimization. To be issued as a technical report, 1994.

Program	*Lines*	*Passes*	*Type Vars*	*Edges*	*Entry Sets*	*Typed?*	*Checks*	*Im*	*Time*
PRECISE									
ion	1934	5	50779	3470	760	YES	0	0	713.70
network	1799	3	29090	2228	730	YES	0	31	234.15
circuit	1247	6	34505	1801	430	YES	0	7	289.52
pic	759	6	40284	2128	357	YES	0	0	363.18
mandel	642	1	17257	1011	442	YES	0	0	25.48
tsp	500	3	10290	627	207	YES	0	0	56.24
mmult	139	7	11518	543	147	YES	0	0	78.35
poly	41	4	3819	234	90	YES	0	0	18.12
test	39	7	1581	130	76	YES	0	0	15.11
PALSBERG									
ion	1934	1	115800	7098	2817	NO	19	264	577.51
network	1799	1	73864	6018	2296	YES	0	87	357.47
circuit	1247	1	49849	2646	1097	NO	44	679	136.03
pic	759	1	48420	2783	1068	NO	28	196	144.16
mandel	642	1	26280	1442	562	YES	0	0	60.78
tsp	500	1	18203	1150	472	NO	2	31	40.78
mmult	139	1	10928	595	216	NO	4	104	22.36
poly	41	1	4233	250	137	YES	0	0	8.25
test	39	1	1353	123	100	NO	2	0	2.94
STATIC									
ion	1934	1	34729	3380	396	NO	260	1096	131.16
network	1799	1	18874	1804	407	NO	132	926	58.77
circuit	1247	1	15491	976	190	NO	111	405	28.93
pic	759	1	16065	1300	180	NO	119	390	37.68
mandel	642	1	8755	760	116	NO	59	524	16.52
tsp	500	1	7006	571	130	NO	27	225	15.79
mmult	139	1	3842	231	61	NO	4	89	7.60
poly	41	1	1848	138	48	NO	4	55	3.84
test	39	1	1001	108	44	NO	2	19	2.92

Table 1: Raw Results for Three Type Inference Algorithms

[18] Bernhard Rytz and Marc Gengler. A polyvariant binding time analysis. Technical Report YALEU/DCS/RR-909, Yale University, Department of Computer Science, 1992. Proceedings of the 1992 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation.

[19] Olin Shivers. *Topics in Advanced Language Implementation*, chapter Data-Flow Analysis and Type Recovery in Scheme, pages 47–88. MIT Press, Cambridge, MA, 1991.

[20] Norihisa Suzuki. Inferring types in Smalltalk. In *Eighth Symposium on Principles of Programming Languages*, pages 187–199, January 1981.

[21] Thinking Machines Corporation, Cambridge, Massachusetts. *CM-5 Technical Summary*, October 1991.

[22] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *Proceedings of OOPSLA '87*, pages 227–41. ACM SIGPLAN, ACM Press, 1987.

A Experimental Results

Table 1 contains raw data from our tests. The *Passes* column indicates how many passes were required for each algorithm to terminate. *Type Vars* is the number of type variables created by each algorithm. *Edges* is the number of interprocedural call graph edges and *Entry Sets* is the number of (virtual) method clones created by each algorithm. For STATIC this is the number of methods used by the programs. *Checks* is the number of type checks required by each algorithm to ensure that no undetected run time type errors occur. A program is *Typed?* when it requires no run time type checks. *Im* is the number of imprecise type variables remaining after the algorithms terminate. **Time** is reported in seconds.