Department of Computer Science, University of Illinois Urbana-Champaign
Tech Report UIUCDCS-R-93-1829

Incremental Inference of Concrete Types

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 is, in general, a flow sensitive global data flow problem. As a result, its solution is hampered by the very program structures for whose optimization its results are most critical: dynamic dispatch (as in object-oriented programs) and first class functions (including function pointers). Constraint based type inference systems are an effective way of safely approximating concrete types, but their use can be expensive and their results imprecise. We present an incremental constraint based type inference technique for extending the analysis in response to discovered imprecisions. This technique infers concrete types to high precision with a cost proportional to the information obtained. Performance results, precision and running time, are reported for a number of concurrent object-oriented programs.

1 Introduction

Strongly typed languages with type declarations and/or type inference systems enable programs to be statically type checked. Static type checking ensures there will be no runtime type errors, and can also assist programmers in finding bugs early in the development process. Such type declarations can sometimes aid program optimization, providing enough information to determine what specific data structures will be used at run time. However, modern languages and development practices include greater use of polymorphism or declarations which are parameterized over a range of concrete data types. As a result, in most cases, type declarations do not give the compiler all of the type information desirable for program optimization.

In languages with polymorphism, such as object-oriented languages, a program variable has values of distinct concrete types over the course of one program run or even across different program runs. These concrete or implementation types (the objects or functions as they occur at run time) should be distinguished from the *principle* or *most general* types which describes legal uses of a objects or functions. While a great deal of progress has been made with respect to the inference of type information [12, 13, 4], these inference techniques compute principle types, not concrete types. Concrete types can be used to support program optimization; while principle types are used primarily for reasoning about the type correctness of programs. Concrete types provide information about data structures and control flow, providing a basis for traditional optimizations such as inlining and register allocation as well as optimizations such as static binding and unboxing.

Though separate compilation is essential for extremely large programs, it is clear that separating compilation units inhibits compiler analysis and optimization. If the compiler cannot determine the exact structure of data, functions and methods, it cannot optimize the code using them. This constraint is particularly severe in object-oriented languages where polymorphic libraries are common. In particular, concrete type information depends on program data flow, so concrete type inference in general requires global flow analysis.[1]

We present a type inference algorithm based on global data flow analysis. Because of the linkage between type information and control flow and data flow caused by type-dependent dispatch, our algorithm uses and updates an incremental global data flow approximation at each step of type inference. This approach allows us to simultaneously solve the type and flow constraints.

Cast in a data flow framework, the problem of concrete type inference is characterized by the framework $\mathcal{D} =< FG, L, \mathcal{F} >$ where the flow graph $FG =< N, E, root >$ nodes (N) correspond to program statements, the edges (E) correspond to an approximation of global control flow. The elements of the lattice (L) are sets of tuples of a variable and a set of concrete types. The meet operator ($c = a \wedge b$) constructs c, the set of tuples of variables and the union of concrete types for each variable in a and b. The transfer functions $\mathcal{F}$ describe the local constraints induced by the program statements.

In this context, the major contributions of this paper are:

1. A labeling scheme for type variables based on the dynamic program structure which supports flow-sensitive analysis (Section 3.1). The scheme is flexible, supporting appropriate levels of precision in different parts of the dynamic program structure.

2. A type inference algorithm which is incrementally extendible to arbitrary precision (Section 3). This algorithm incrementally extends the global control flow approximation (E) and the dependent type variable labels.

3. Vital techniques for an efficient implementation of the incremental type inference algorithm, entry sets and container sets (Section 3.3), which limit and direct analysis effort to dynamic program structures where greater precision is required. These techniques dramatically reduce the cost of analysis by sharing inference information among similar cases.

4. An empirical evaluation of the incremental inference techniques using a collection of concurrent object-oriented programs. The evaluation shows that not only is precise type inference possible, but its cost is practical for an optimizing compiler.

5. A framework which enables type inference for languages with first class functions and continuations (Section 4).

The structure of the remainder of the paper is as follows. First, Section 2 covers background material, discussing notation, the basic idea behind constraint-based type inference,

[1] For instance, if a new piece of code assigns an object of type A' (a subtype of B) to a variable of type B which previously was only assigned variables of type A (another subtype of B), all code which uses the variable is affected.

and limitations of previous approaches. In Section 3, we introduce our incremental type inference technique which extends the precision of analysis in response to a program's type complexity. Subsequently in Section 4, we describe the framework in which the incremental analysis occurs which makes it practical for real programs. 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. Use of the type information is touched on in Section 6. Related work is briefly surveyed in Section 7 and the paper is summarized in Section 8.

2 Background

Most type checking and inference techniques [12, 4] determine types bottom up, using the types of subexpressions to form the types of expressions and finally the type of the program. At each point, a closed form for the program fragment signature is computed. This process strives to describe the type of the program fragment in all environments and then to verify that the type is legal in the environments where the fragment occurs.[2] Thus the goal of these techniques is to find the most general type. For instance, assuming that $<$ is in the interface of `OrderedObject`, such a type system might determine that the function `max` in Figure 1 can be applied to two objects of any subtype of `OrderedObject` which might include all the numeric classes, strings, and other user-defined data types.

```
function i max: j
  if i > j then
    return i
  else
    return j
main
  f (1 max: 2)
  f (1.0 max: 3.0)
```

Figure 1: Polymorphic Function

```
class A
  var i
  method i: argi
    i = argi
    return self
  method f
    return i
main
  f ((new A) i: 1)
  f ((new A) i: 1.0)
```

Figure 2: Polymorphic Container

Modern programming practice and object-oriented programming in particular encourage the use of large libraries of reusable components with deep subtype hierarchies, only a portion of which may be used at any point in the program. This makes program optimization difficult since compilers typically require specific information in order to transform the program. Optimizations like inlining, static binding and unboxing work only if a variable can be resolved to a single concrete type. Thus, in contrast to the general types of other typing techniques, our goal is to find specific type information. To enable such optimizations, we have implemented a concrete type inference algorithm in the Illinois Concert compiler [6]. For the sample code shown in Figure 1 this algorithm determines that `max` is called only on integers from the expression (`f (1 max:  2)`) and floats from the expression (`f (1.0 max:  3.0)`), enabling the `max` and `>` functions to be specialized and inlined.

[2]Type systems like those of Pascal and C++ simply verify the programmers declaration for the fragment, modulo automatic coercion, against the environment.

We will initially consider inference in the context of strongly typed languages, and then extend the algorithm to detect run time type errors in dynamically typed languages. Program examples in the running text will be in a syntax derived from [14]. The target language of our implementation and in which the benchmarks are written is the dynamically typed concurrent object-oriented language Concurrent Aggregates [7, 8] which includes first class selectors, messages and continuations. The algorithm is very general and can be applied easily to a wide range of languages.

Before we begin, let us clarify some terms. We differentiate two types of polymorphism: data and functional. 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 2. We define *levels of polymorphism* as the depth of the polymorphic function call path and depth of the polymorphic reference path for functional and data polymorphism respectively. Effective inference requires the ability to handle the many levels of polymorphism found in real applications without losing accuracy or requiring unreasonable computational resources.

2.1 Constraint Based Type Inference

Direct implementation of the global data flow framework outlined in Section 1 would be inefficient because the transfer functions are more closely associated with variables than statements. In [9], the construction of sparse data flow evaluation graphs is proposed. The idea is to remove data flow nodes whose transfer functions are identity and to forward the results of the others directly to the nodes which use them. In terms of the type inference problem, we define the sparse flow graph $FG' =< N', E', root >$ with the nodes as the program variables and the edges as assignments between them. With this model in mind the problem can be viewed as the construction and solution of a constraint network, where the constraints are just the edges of the sparse flow graph. These constraints describe the type relations between program variables and constants.

The algorithm maintains a work pile of invocations (interprocedural edges) which are processed by finding the target method or function, and applying local constraints (intraprocedural) and connecting constraints (interprocedural). These constraints are solved by propagation where changes in the input of a node triggers recomputation of the output so as to maintain a continuously updated solution [15].

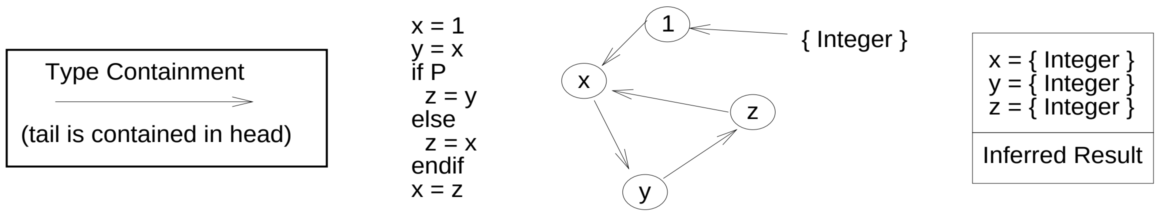
$$\begin{aligned} x = newC &\longrightarrow [\![x]\!] \supseteq \{C\} \\ x = y &\longrightarrow [\![x = y]\!] \supseteq [\![y]\!], \\ & [\![x]\!] \supseteq [\![y]\!] \end{aligned}$$

Figure 3: Basic Local Constraints

$$\begin{aligned} &x\ selector\ a_0\ a_1 ... a_n \longrightarrow \\ &\quad (\forall c \in [\![x]\!].(c\ selector\ p_0\ p_1 ... p_n \\ &\quad\quad \forall i \leq n.[\![p_i]\!] \supseteq [\![a_i]\!])) \end{aligned}$$

Figure 4: Basic Connecting Constraints

The basic local constraints reflect local data flow and special statements which create objects of known type, denoted by C (see Figure 3). The basic connecting constraints reflect global data flow along the edges of the interprocedural call graph, connecting the actual arguments (a_i) and formal parameters (p_i) as in Figure 4. An example constraint



```
x = 1
y = x
if P
  z = y
else
  z = x
endif
x = z
```

Figure 5: Constraint Graph Example

graph and its solution are shown in Figure 5. Complicating construction of the constraint system is the interaction between types and control flow vis a vis type-dependent dispatch. The concrete types of the target of a message send determine the possible flow of control at each invocation site. Therefore, the current value of the constraint system is used to continually approximate interprocedural control flow. This works only if the value of each variable increases monotonically.

2.2 Imprecision

Imprecisions in the inferred concrete type information are differences between runtime behavior and the behavior predicted by the inference system. At run time, static variables (variables in the program text) are instantiated as dynamic variables (in the running program). These dynamic variables may be used in many different situations, holding values of different types for example. To avoid imprecisions, an inference algorithm must separate the dynamic variable instances to distinguish different situations. Each set of dynamic variables separated out by the inference algorithm is called a *type variable*. This is because for the purposes of the type inference all of the dynamic variables in that set are subject to the same type constraints.

```
function i leq: j
  return i <= j
function i max: j
  if (i leq: j) then
    return j
  else
    return i
main
  if (1.1 max: 1.2) or
      (1 max: 1) then
    ...
```

Figure 6: Multi-level Polymorphic Functions

```
class A
  var b
class B
  var i
function createB: ivar
  return (new B) i: ivar
function createA: bvar
  return (new A) b: bvar
main
  a1 = createA: (createB: 1)
  a2 = createA: (createB: 1.0)
```

Figure 7: Multi-level Polymorphic Containers

In order to handle polymorphic functions, the type inference algorithm in [15] creates separate type variables for each call site at which the function containing the variable was invoked. Likewise, separate type variables are created for the contents of polymorphic containers based on the point at which the dynamic instance of the container was created, its *creation point*. Unfortunately, this single level of discrimination (caller versus caller and caller's caller) is insufficient to precisely infer types within common program structures: *i*) polymorphic libraries with multi-level call trees, *ii*) functions which create and initialize container objects, and *iii*) polymorphic containers of polymorphic containers (see Figures 6 and 7 for an illustration of these cases). However, increasing the level of discrimination to some fixed level k incurs a cost exponential in k as well as limiting the potential precision of the analysis.

3 Incremental Inference

Our solution to the limitations of the approaches described above is to incrementally extend the precision of type inference in response to detected complexities in the type structure of the program. This approach allows us to type programs with arbitrarily complex type structure with the cost of extended analysis incurred only for programs which require it for optimization. In this section, we describe the critical problems and our solutions for incremental and efficient inference.

As was clear from the last section, we cannot afford to globally increase the level of discrimination, so we must extend it dynamically. For this we require a way to identify the points at which to extend precision and a way to represent this selectively increased precision. Maintaining reasonable cost requires identifying a minimal set of extension points and using a representation which minimizes redundancy. The representation we will use is an extensible labeling scheme for type variables and is explained in Section 3.1. The general mechanism used to extend the labels is described in Section 3.3. A simple eager extension technique is covered in Section 3.3.1. Imprecision can result from either polymorphic functions or polymorphic containers. Discovery and extension for specific imprecisions is described in Sections 3.3.2 and 3.3.3 for functions and containers respectively.

At first it might seem that extension and inference could go on simultaneously. However, increasing the level of discrimination dynamically is complicated by the fact that the solution must increase monotonically. That is, the type estimate for each variable must only become greater (less restrictive) with time since the structure of the constraint system itself is based on the developing solution. As a result, naive modification of a developing constraint network could leave the solution in an state inconsistent with the network. On the other hand, because of the interdependencies between type information and control flow, invalidating portions of the solution can invalidate other portions of the network and so on, making such modifications costly.

Thus, our overall algorithm first infers program types, then if necessary extends the network, increasing the power of discrimination in those areas where it is needed. After the network has been extended, the solution is cleared and recomputed. This process is applied iteratively until the algorithm determines that no extensions can improve precision or a desired level of accuracy is attained. We term one cycle of constraint solution and extension an *iteration*.

3.1 Type Variables

For each program variable there is a potentially infinite set of run time variables which are generated by the execution of the program. To name these dynamically created variables, we label program variables by their execution environment. For example, in Figure 1 the function `f` is invoked in two different environments with arguments of type integer and floating point respectively. In object-oriented languages, the state of the target is also part of the execution environment. This is because the value of an instance variable can determine the return type of a method, as in Figure 2 where the return type of method `f` depends on the value of i, part of the object state. Thus, variables are labeled with invocation site and object state information.

More formally, for each program variable v we define a set of dynamic variables $\mathcal{D}(v)$ as:

$$\begin{aligned} E_i &= P \times C \times E_{i-1} \\ C &= E \\ E &= \bigcup_i E_i \\ \mathcal{D}(v) &= \{v_e \mid e \in E\} \end{aligned}$$

Where E_i denotes an execution environment and is composed of an invocation point (P), a creation point (C) and an enclosing environment. Invocation points capture the invocation environment and creation points capture the object state part of an execution environment. A creation point in turn is just the point in the program, the execution environment, where the object was created. E_0 denotes a distinguished initial environment where program execution began.

Since there are potentially an infinite number of dynamic variables in a program, a realistic analysis must partition them, grouping those which are of the same type and assigning a type variable for each group. The nodes of the constraint network then correspond to these partitions, and the value of a node (type variable) is the union of the concrete types of the partition's dynamic variables. Unfortunately, since we do not know the type of each dynamic variable a priori, we do not know how best to group them. In addition, once a partition has been constructed and used in the network solution it is difficult if not impossible to determine the values of subpartitions. The incremental analysis therefore repartitions the variable instances between iterations.

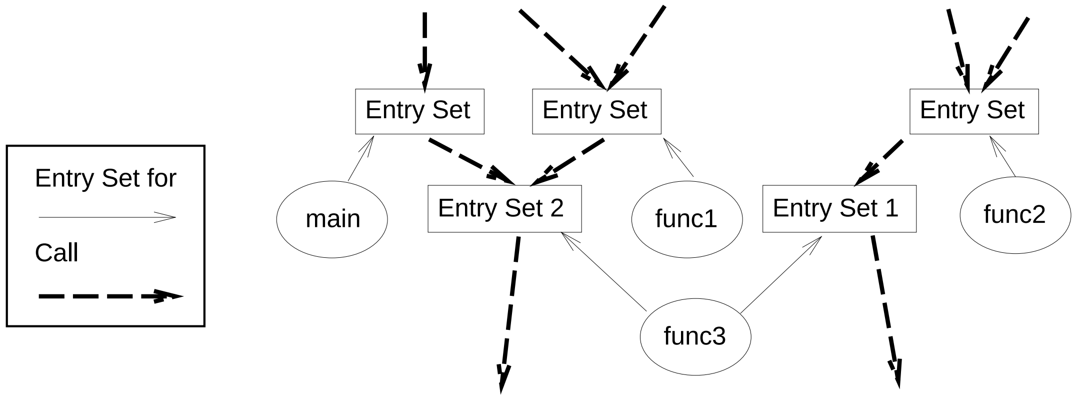


Figure 8: Entry Sets Example

3.2 Entry and Creation Sets

Since constructing partitions for each variable individually would be complex and expensive, we construct partitions for the execution environments. These in turn induce partitions over the variable instances. For every method, we maintain a collection of *entry sets*. Entry sets group execution environments by collecting edges of the interprocedural call graph incident on the method or function. The execution environment represented by an entry set is then all the environments that arrive along those edges of the call graph. A distinct set of type variables for the program variables in the method is generated for each entry set. As a result, separate type information is maintained for each entry set. An example of entry sets can be found in Figure 8.

Partitions of creation points are constructed similarly and called *creation sets*. As defined above, creation points are simply the program point and execution environment where the object was created. Creation points are used to distinguish groups of objects of the same class by the place where they are created at run time, and are propagated through the network in much the same manner as concrete types.

To simplify the exposition, in this paper we assume that an entry set is associated with only one creation set, so a type variable is uniquely determined by just the program variable and the entry set. The algorithm ensures this by construction.

Previously we labeled type variables with an invocation point, creation point, and an enclosing environment; however, introducing entry and creation sets changes this labelling. Instead of specific points, we simply label a variable by an entry set. This entry set in turn determines the creation set and a set of enclosing (calling) entry-sets. This substitution is the critical element in achieving efficient analysis. More formally, since an entry set groups together a set of invocation environments, $\mathcal{D}(v, es)$ for variable v and entry set es represents all dynamic variables:

$$\begin{aligned} \mathcal{D}(v, es) &= \{v_e \mid e \in Env(es)\} \\ Env(es) &= \{e \times e' \mid e \in Edges(es), e' \in Env(Source(e))\}) \end{aligned}$$

Edges returns the interprocedural call graph edges in an entry set or group of entry sets. An edge consists of an invocation point and the entry set from which the edge originated. *Source* returns this entry set which is at the source of the edge. The initial entry set E^0 has no edges.

3.3 Splitting

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

Splitting can be applied to both entry and creation sets in analogous fashion. Splitting an entry set divides its edges over a number of smaller entry sets. Splitting a creation set likewise divides the creation points of the original creation set over a number of smaller creation sets. If the set to be split contains only one edge or creation point, the imprecision originated at a higher level, and the surrounding environment must be split first.

While the ultimate goal of splitting is to eliminate imprecisions in type, imprecisions of other kinds at one variable can cause imprecisions in type at another variable. These imprecisions can be of any of the data flow quantities described in Section 4 or another quantity which describes the paths which a type variable is along. This last will be described in detail in Section 3.3.3. Some of the algorithm portions in the section can operate on more than one of these quantities and are parameterized by the function $Value$ which accesses the appropriate one for the argument type variable.

In the subsequent parts of this section we consider several different types of splitting. First, we discuss eager splitting which applies in some special cases. Subsequently we discuss function splitting and container splitting which address imprecisions arising from polymorphic functions and containers respectively.

3.3.1 Eager Splitting

Eager splitting detects imprecisions as they are about to occur and splits entry sets immediately. The objective of eager splitting is to reduce the overall run time of the algorithm by exploiting partial information. In general, identifying incompatible edges requires solving the entire constraint network, but sometimes incompatibility can be detected much earlier. Specifically, if two invocation edges to a method have invocation arguments in the same position of different types (or another data flow quantity), these edges are *incompatible*. This is because the edges produce constraints which result in a *confluence* of information (a meet $a \wedge b$ where $a, b \neq \emptyset$ and $a \neq b$). Eager splitting exploits this situation splitting the information immediately (within the same iteration), creating a new entry set for the edge that would cause the confluence.

Adding the new edge to an existing entry set would be preferable, so the algorithm below is used. For simplicity we will use the function $Value$ which represents the value of a type variable with respect to a data flow quantity (e.g. concrete types: $Type$ or creation sets: $Creators$). In an implementation, for edges to be compatible they must be so at all data flow values. First, we find the compatible entry sets $CompatES(e)$; those which contain only compatible edges. If there is one or more, we select the first one arbitrarily. If no compatible entry sets exist, we create a new entry set. The function $ArgOf(e, a)$ simply finds the argument of e corresponding to a.

$$\begin{aligned}
CompatES(e) &= \{es \mid es \in EntrySets(Method(e)), \forall e' \in Edges(es), \\
&\qquad CompatibleEdges?(e, e')\} \\
CompatibleEdges?(e, e') &= \forall a \in Args(e), a' = ArgOf(e', a), \\
&\qquad Value(a) = \emptyset \vee Value(a') = \emptyset \vee Value(a) = Value(a') \\
ES(e) &= (CompatES = \emptyset) \ ? \ NewES, \ First(CompatES(e))
\end{aligned}$$

Figure 9: Finding a Compatible Entry Set

Eager splitting can handle some polymorphic functions in a single iteration, but it is not effective for polymorphic containers (splitting creation sets). This is because decision to split a creation set must be made at the creation point.[3] However, the necessity of the split cannot be known until the instance variables are actually used. This is generally much later in the analysis. In contrast, eager splitting sometimes works for entry sets because edges to polymorphic functions may have some information about their argument types when they are instantiated.

3.3.2 Function Splitting

Function splitting reduces imprecisions due to polymorphic functions. Basically, function splitting partitions an entry set, allowing a more precise typing of the function for each entry set. Function splitting is generally applied to entry sets which contain incompatible edges – those responsible for introducing imprecision in the type inference.

All the entry sets for which splitting will directly increase precision can be found by applying the *CompatibleEdges?* function (see Figure 9) pairwise to all edges in each entry sets. However, for efficiency reasons we may wish to find those entry sets which are the cause of particular imprecisions. This will also help us illustrate techniques which we will use for splitting of creation sets.

Our algorithm identifies the entry sets that must be split to resolve a particular imprecision in the constraint network by finding the sources of the imprecision. The primary sources of an imprecision are the type variables at confluence of subsets of the imprecision. Since imprecision is the unwanted mixing of information, if these type variables can be split, the mixing might not occur and the imprecision might be eliminated. To find the confluences, we follow the constraint network from the imprecise variable back to confluence points. For convenience, we define the following functions on the constraint network:

FlowVars(tv) Given a type variable tv return those type variables tv' which have direct constraints $Type(tv) \subseteq Type(tv')$. Intuitively, these are the variables along the path of data flow.

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

Using these functions, we follow the constraints back to find the primary source of the imprecision. More specifically, given a type variable tv and an imprecision im, we find those type variables which represent confluences involving im or a subset of im. The function $FV(tv, im)$ is the full value including potential imprecisions. $IsConfluence$ computes if the type variable is a confluence point. $ContinueTo$ is used to eliminate those backward edges which do not contain the imprecision, averting unnecessary work. Finally, $ConfVars'(tv, im)$ puts all of these together, following the constraints backward to all sources of the imprecision.

$$\begin{aligned}
FV(tv, im) &= Value(tv) \cup im \\
IsConfluence(tv, im) &= \exists b, b \in BackVars(tv), FV(tv, im) - Value(b) \neq \emptyset \\
ContinueTo(tv, im) &= \{b \mid b \in BackVars(tv), Value(b) \cap im\}
\end{aligned}$$

[3]We can split all creation points into separate creation sets, but not in an informed manner as with eager function splitting.

$$
\begin{aligned}
ConfVars'(tv, im) &= IsConfluence(tv, im) \cup \\
&\quad \{b' \mid b' \in ContinueTo(tv, im), ConfVars'(b, im \cap Value(b)))
\end{aligned}
$$

However, this only covers the primary cause of an imprecision. An imprecision can also arise if there is interprocedural control flow ambiguity (due to a secondary imprecision at the target of method invocation) which causes a merger at arguments or return values. This situation is handled with a slight redefinition of $ConfVars(tv, im)'$:

$$
\begin{aligned}
ConfVars(tv, im) &= \{tv' \mid tv' \in ConfVars'(tv, im)\} \cup \{tv' \mid tv'' \in ConfVars(tv, im), \\
&\quad ((Arg?(tv') \vee ReturnVar?(tv')), targ = Target(SendStmt(tv')), \\
&\quad V \in AllValues, tv'' \in ConfVars(targ, V(targ)))\}
\end{aligned}
$$

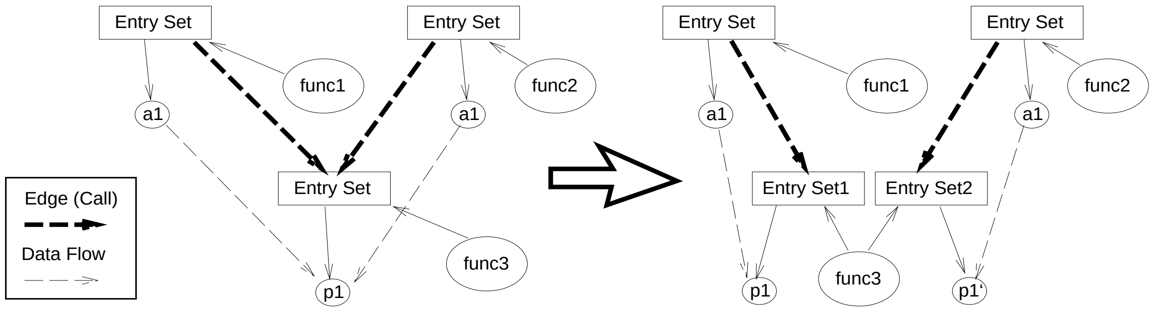


Figure 10: Function Splitting Example

We extend the old definition of $ConfVars$ to include all those variables which are $ConfVars$ for imprecisions at the target of message sends for which the primary variable is an argument or return value. Since an imprecision in any of the data flow quantities in the target may indirectly cause these imprecision, we use the set of functions $AllValues$ to indicate that we are looking for confluences at any data flow value.[4]

An example of function splitting is displayed in Figure 10 where the entry set associated with `func3` is split. Actual arguments for the formal parameter `a1` coming into `func3` from `func1` and `func2` have different concrete types, so the edges are incompatible. Splitting the entry set prevents the confluence of type information. The result is two type variables `p1` and `p1'` which have distinct types, and a more precise program typing.

3.3.3 Container Splitting

Container splitting reduces imprecisions due to polymorphic containers. Basically, container splitting partitions creation sets allowing a more precise typing for the objects represented by each set. Splitting containers is more complex then splitting functions since the point of confluence (the instance variable) is separated from the cause (the creation point). However, as before, the basic idea is to follow paths in the constraint network to find sources of the imprecision and split them.

[4]In addition, handling first class continuations requires that the function $ReturnVar?$ determine if the variable is a *Continued Value* and the function *SendStmt* find the send statement for the continued value.

There are two different types of imprecisions which are of concern with respect to containers: imprecisions in the normal data flow values and in the imprecisions in the paths which creation sets might take. Given a particular confluence at an instance variable, there will be two or more surrounding methods which assign the conflicting portions of the imprecision to the instance variable. These methods, in turn, are invoked on different type variables which are determined by the same creation set (hence the shared instance variable). By following the $BackVars$ from these 'containing' type variables back to the shared creation point we can construct the paths which, if split, would allow us to eliminate the imprecision. The appearance of a type variable in more than one of these paths is the second type of imprecision.

Container splitting involves three operations. First, we compute the paths from the type variable containing[5] the instance variable at the imprecision back to its creation points. These paths determine potential new partitions for the instance variable. Second, if there is more than one creation point along separate paths, we split the creation set. Finally, we determine where confluences occur for the paths.

First we compute the paths from the containing type variables to the creation points. As before, we parameterize the algorithm with the function $Value(v)$ which can be any imprecise quantity. In addition, we also parameterize the algorithm by the direction of which the quantity flows using $Vars$ which takes on the function values $FlowVars$ and $BackVars$. For the normal data flow quantities, the value of $Vars$ is $BackVars$, but when the algorithm is applied recursively to the paths themselves, the value of $Vars$ will be $FlowVars$. We begin with a type variable, v, which corresponds to an instance variable which is at the point of an imprecision.

$$\begin{aligned}
ImBV(v) &= \{b \mid b \in Vars(v), Value(b) \neq Value(v)\} \\
BackSets(vs) &= First(vs) \cup BackSets(\{v \mid v \in vs, Value(v) \neq Value(First(vs))\}) \\
Bs &= BackSets(ImBv(v)) \\
Containers(vs) &= \{c \mid c = Target(e), e \in Edges(EntrySet(b)), b \in vs\} \\
BL &= \{bl \mid bl = Closure(BackVars, Containers(bs)), bs \in Bs\}
\end{aligned}$$

$ImBV(v)$ finds the set of variables which carry the information which merged to cause the imprecision at v. For efficiency we then form subsets of $ImBV$ which carry identical portions of the information using the function $BackSets$. The function $Containers(vs)$ finds the 'containing' variable for each variable in a set. Finally we compute the paths back to the creation points by taking the closure of $BackVars$ over each set of containers.

BL is then the set of paths from the creation points (the origin of the creation sets) to the source of the imprecision. The paths in BL are those that would be taken by the creation sets whose existence would eliminate the imprecision. The appearance of a type variable on more than one of these paths represents a secondary imprecision. Therefore, for each type variable in BL we need to know the subset of BL in which it is contained. We define the function $TvBl(tv, BL)$ to represent the subset of BL which contains tv.

The second 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

[5]Given an instance variable, the containing type variables are those type variables which are the targets of the dynamic dispatch for methods containing the instance variable. We are assuming a language with single dynamic dispatching.

for each path or set of paths which do not share type variables. These new creation sets will cause the instance variable at the point of the imprecision to split thus removing the imprecision. We will use the type variables which describe the result of an object creation statements *cps* to stand in for the creation point at those statements.

$$\begin{aligned}
cps &= \{v \mid v \in b, b \in BL, CreationPoint(v)\} \\
TvBl(tv, BL) &= \{bl \mid bl \in BL, tv \in bl\} \\
Splittable(cp, cps, BL) &= \{cp' \mid cp' \in cps, TvBl(cp, BL) \not\cap TvBl(cp', BL)\}
\end{aligned}$$

The last step is to determine where confluences of the potential entry sets represented by the paths in BL occur. This is the second type of quantity which we alluded to above, imprecisions in which we must detect and eliminate in order to solve the type inference problem.

Beginning with a type variable at a creation point *cp* we compute PC, the path confluences. The idea is the find those places where the paths that reached *cp* joined together. We use the function $TvBl(v, bl)$ which returns the subset of *bl* containing v. We then search forward along the constraint graph for those places where the size of $TvBl(v, bl)$ decreases.

$$\begin{aligned}
bl &= TvBl(cp, BL) \\
FindPC(p, v, bl) &= \{x \mid ((x = p \wedge TvBl(x, bl) \neq bl) \wedge (TvBl(v, bl) \neq \emptyset)) \\
&\qquad \vee (x = FlowVars(v) \wedge x \in FindPC(v, x, TvBl(v, bl)))\} \\
PC &= \{pc \mid x \in FlowVars(s), pc \in FindPC(cp, x, cps)\}
\end{aligned}$$

The type variables PC 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. We extend PC in the same manner as $ConfVars'$ was extended to $ConfVars$ before applying the algorithm recursively.

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 split. The new type variables for the instance variable will each have a subset of what was the original imprecision, eliminating the imprecision.

3.4 Iteration

As discussed at the beginning of the section, the incremental inference algorithm refines its approximation of the program's type structure in a series of iterations. Each iteration extends effort into regions of the constraint network which contain imprecisions. Entry and creation sets are split where appropriate and the affected portion of the network is cleared and the algorithm is rerun.

The affected portion of the constraint network includes all type variables whose entry set which was modified (split) or whose creation set was modified (split).[6] The network is

[6]An implementation may split a set by moving some of the constituents to another set with compatible constituents, both of which are modified.

cleared by removing the affected constraints and data flow values and those constraints and values dependent on them, but not the entry or creation sets. The edges are removed, but their correspondence with the entry sets is preserved, carrying the partitioning information to the next iteration. In subsequent iterations, when an edge is made for which we have remembered the entry set, that entry set is used as the target of the edge. In the same way the creation sets for each creation point are remembered. When a creation point is made, we restore the remembered creation set.

Typically, after a number of iterations, no more imprecisions will exist and the type inference algorithm will terminate. In some cases, it may be preferable to terminate the algorithm after a fixed number of iterations, inserting type checks in the generated code where imprecisions still exist. The termination and complexity of the algorithm is covered in Section 4.2.

4 Completing the Algorithm

Though we have described the general outline of a type inference algorithm, a complete concrete type inference algorithm for a language containing first class functions and continuations actually requires the simultaneous solution of three global data flow problems. These are:

Concrete Type The implementation types which a variable may take on at run time.

Creation Sets The creation points of objects a variable may refer to.

Selector Values The selectors, closures or functions a variable may refer to.

The values for each of the data flow problems travel along the global data flow approximation edges with union as the meet operator. The transfer functions include the basic constraints as well as constraints induced by dynamic dispatch. As a reflection of type-dependent dispatch, the transfer function for a formal parameter in the dynamic dispatch position (the target of the message send) is constrained to pass only the values which could cause the function to be invoked. In addition, it can only pass the creation set corresponding to the entry set of its partition.

For **Selector Values**, anonymous function are given tokens, and closures are parameterized by the variables they capture from the surrounding scope [16] allowing such variables to be treated as arguments.

In addition to the data flow problems, for each type variable there are several important attributes which must be updated:

Dependent Invocation Sites The invocation sites for which the type variable is in the dynamic dispatch or function position. This collects the dependent control flow information and is used to create new edges in the interprocedural call graph approximation on demand when the solutions for a type variable changes.

Continued Values A type variable representative of the variable's value as a continuation. That is, it represents the values to which the variable is applied as a continuation. Since the value to which a continuation is applied is returned to the continuation's creation point, the data flow for **Continued Values** is backward with respect to invocation data flow arcs. This is required to handle first class continuations.

The overall algorithm then consists of a work pile of interprocedural call edges, a network of dataflow constraints, the forward data flow problems: **Concrete Type**, **Selector Values** and **Creation Sets** and the backward data flow problem within **Continued Values**.

4.1 Type Checking

For statically typed languages, type checking is generally 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 be used to ensure the absence of runtime type errors allowing the compiler to remove type checks or to alert the programmer to possible program errors.

After type inference has been done, the only places where type error can occur is where type inference gives imprecise results. This is because there is a single solution of the type constraints everywhere a precise solution was found. Thus, for each imprecise type variable we reexamine the connecting constraints. Any type variable, a target of a message send, which includes types which fail to support any or all of the reaching selectors must be checked at runtime to ensure safety. In the current compiler these are reported to the user as warnings.

4.2 Safety and Complexity

The basic constraint-based type inference algorithm is safe because it enforces the program's data flow and invocation type constraints. Since the incremental algorithm does not change the values of the constraint network, but refines the analysis by partitioning and applying the constraints more precisely (removing confluences), it is also safe.

Termination is ensured by avoiding infinite recursive splitting of functions and containers. Before splitting we determine if the function is recursive by checking the interprocedural call graph approximation. If it is potentially recursive, the function is not split.[7] The same technique is applied to creation sets. Since the number of type variable partitions is finite the algorithm will terminate.

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.

5 Implementation

We have implemented the incremental type inference algorithm as part of the Illinois Concert project. The Concert system includes a compiler and runtime for concurrent object-

[7]A constant number of recursive calls can be split at each site without affecting termination, however our implementation allows only one recursive call.

oriented languages. The front end currently supports the language Concurrent Aggregates (CA) [7, 8], a dynamically typed concurrent object-oriented language with single inheritance as well as first class selectors, continuations, and messages.

We have tested the type inference system on more than 20,000 lines of CA code. The results on a variety of real and synthetic programs appear in Table 1. PRECISE refers to our incremental inference algorithm, PALSBERG refers to the inference algorithm in [14], and STATIC refers to a basic constraint based inference which allocates exactly one type variable per static program variable.

Program	*Lines*	*P*	*TVs*	*Tot Cts*	*Cts*	*E*	*ESs*	*CPts*	*CSs*	*Im*	*Im-N*	*Sec*
PRECISE												
mandel.ca	642	2	9695	12359	4618	617	183	24	24	0	0	44.52
simple.ca	2035	5	43982	122508	21583	3428	782	71	71	445	0	903.21
p-i-c.ca	759	11	37125	186223	16820	1692	355	34	34	65	0	793.22
titest7.ca	35	4	1105	1679	403	96	60	11	11	0	0	7.42
mmult.ca	139	3	9373	13691	5254	418	157	12	12	0	0	40.21
poly.ca	41	3	2781	4043	1344	150	73	9	9	0	0	13.84
tsp.ca	500	3	8631	15616	4898	558	189	15	15	141	0	55.02
quicksort.ca	152	2	2411	2702	2702	169	59	7	7	0	0	10.66
queens.ca	121	2	3256	4216	1694	232	82	9	9	0	0	13.15
fft.ca	260	2	3283	4161	1573	234	80	8	8	0	0	13.05
PALSBERG												
mandel.ca	642	1	21922	17940	17940	1301	767	27	25	0	0	64.60
simple.ca	2035	1	88074	73503	73503	7096	2935	73	70	889	0	600.44
p-i-c.ca	759	1	72164	67322	67322	4344	1191	43	25	405	310	261.61
titest7.ca	35	1	917	635	635	85	75	9	9	9	9	2.47
mmult.ca	139	1	9275	7719	7719	504	258	8	7	125	124	22.47
poly.ca	41	1	3245	2462	2462	218	144	7	7	83	83	7.32
tsp.ca	500	1	17422	14319	14319	1238	663	18	16	250	46	51.30
quicksort.ca	152	1	7669	6150	6150	511	258	7	7	0	0	17.79
queens.ca	121	1	6957	5427	5427	486	319	11	11	0	0	17.53
fft.ca	260	1	8265	6485	6485	533	325	11	11	0	0	19.27
STATIC												
mandel.ca	642	1	8535	8369	8369	860	119	24	19	606	488	25.50
simple	2035	1	36862	40388	40388	5318	433	71	9	2921	2672	255.04
p-i-c.ca	759	1	25989	25760	25760	2255	253	32	9	612	492	85.55
titest7.ca	35	1	785	622	622	86	36	9	7	19	9	1.89
mmult.ca	139	1	3577	3140	3140	216	57	7	5	89	74	7.47
.ca	41	1	1669	1367	1367	125	44	7	5	55	43	3.6
tsp.ca	500	1	7329	6793	6793	603	141	16	10	263	179	25.86
quicksort.ca	152	1	2202	1881	1881	168	49	7	5	12	0	4.78
queen.ca	121	1	3066	2695	2695	264	69	9	6	25	12	9.55
fft.ca	260	1	3180	2718	2718	262	70	8	5	16	8	9.8

Table 1: Results of Incremental Type Inference

The **man.ca** program computes the Mandelbrot set using a dynamic algorithm. **simple.ca** is the SIMPLE hydrodynamic simulation and **p-i-c.ca** is a particle-in-cell code. **titest7.ca** is a synthetic code designed to illustrate the algorithm's effectiveness and appears in Appendix A. The **mmult.ca** program multiplies integer and floating point matrixes using a polymorphic library. **poly.ca** evaluates integer and floating point polynomials. The program **tsp.ca** solves the traveling salesman problem. **quicksort.ca** implements the quicksort algorithm. queens.ca solves the N-queens problem, and **fft.ca** computes a Fast Fourier Transform using a butterfly network. All test cases were compiled with the standard CA prologue (240 lines of code) and are available along with the language manual [8] and the standard prologue from anonymous ftp.[8]

The columns are defined as follows. **P** refers to the number of passes (iterations) which the algorithm required. This number is determined automatically by the algorithm which terminates when it has determined that the best precision has been reached. **TVs** reports the number of type variables created by the algorithm. Our compiler translates the program into a variation on Static Single Assignment form [10] which greatly increases this number for a given program. **Tot Cts** and **Cts** refer to the total constraints formed over all passes and the constraints in the final pass respectively. **E** and **ESs** refer to edges and entry sets respectively, just as **CPts** and **CSs** refer to creation points and creation sets. **Im** refers to the number of imprecisions (type variables with more than one possible type) and **Im-N** refers to the number of imprecisions if NULL is ignored (not counted as an imprecision). Using **Im-N** as a basis of judgement is justified in part because the occurrence of NULL, while requiring a runtime check, does not prevent inlining and other optimizations and also because uses of such variables are usually conditioned by tests which our algorithm does not as yet use to constrain types in the subsequent code.

The problem size solved in any one pass can be seen in the columns labeled **TVs**, **Cts**, **ESs** and **CSs**, but the last three are especially indicative since some type variables are only used in one pass. In many cases, PRECISE required less work in a single pass than PALSBERG. This is because PALSBERG is doing a lot of redundant work. By reducing the overhead of our implementation (by not verifying the entire network each pass) we should be able to further reduce the total cost.

The precision of the type inference solution is best characterized by the column *Im-N*. PRECISE is able to determine precise type information for all the programs in the test suite. That is, the analysis was able to find a finite static unfolding of the program which required no dynamic type dispatch. In contrast, both PALSBERG and STATIC give significant numbers of imprecisions.

The implementation is approximately 2000 lines of Common Lisp/CLOS and the timings are for CMU Common Lisp/PCL on a Sparc10/31. While the gross execution times are large, they are not unreasonable for an optimizing compiler. Further, because of the relatively long run times for the STATIC algorithm, we believe the long execution times reflect inefficiency in the implementation, not fundamental limitations of the algorithm. Different data structures including constructing templates [14] or using a sparse evaluation graph to remove the replication caused by Static Single Assignment form [9] should reduce the gross times.

6 Uses

There are many uses for type information both for compilation and program development. For example, the type information provides precise interprocedural control flow, an essential prerequisite to virtually all traditional program analyses. It can also increase the availability of interprocedural constants.[9] In this section, we first describe a use of the type inference information to eliminate dynamic dispatches, then describe a more limited replication scheme which may be more attractive in some cases.

[8] At a.cs.uiuc.edu: /pub/csag.

[9] For this, constants are considered subtypes (e.g. 1 is a subtype of `Integer`).

The organization of the type inference results are particularly well-suited for the elimination of dynamic dispatches since it involves partial unfolding the program's dynamic call graph. By using the entry sets to direct code replication, we can efficiently control the amount of replication, increasing code size only when it is necessary. For each method in the program, we replicate the code for each entry set, connecting the entry edges accordingly. Within this new call graph we can statically bind all method invocations except those whose target has an imprecise type. If, as in our experiments, all of the types can be resolved precisely, all dynamic dispatch will be eliminated.

In some cases, less code replication may be desirable. The basic replication scheme can be extended to exploit "pass through" situations. In effect, this eliminates splitting along a call path where the type distinctions do not affect control flow, then resumes splitting at a later point, based on a single type-dependent dispatch. For example, if we use polymorphic matrix library in a program with integers, floating point, double precision floating point and complex numbers, we may only multiply matrices of like types. The basic replication scheme would produce four sets of matrix multiplication functions, splitting the library from top to bottom. By "passing" the polymorphic matrices "through" the upper levels of the matrix library, the more limited replication scheme would only split the methods deeper in the library, selecting amongst the split and optimized algorithm kernels based on the type of a matrix argument.

7 Related Work

Type inference in object-oriented languages has been studied for many years [18, 11]. Constraint-based type inference is described by Palsberg and Schwartzbach in [15, 14]. The limitations of their algorithm to a single level of discrimination has motivated this research. Recently Agensen has extended the basic one level approach to handle the features of Self [19] (see [1]). However, the problems with precision and cost inherent in a single pass approach are tackled by exploiting specialized knowledge about Self [2].

The Self compiler [5] employs a speculative optimization techniques based on runtime tests. These tests select an optimized code sequence from a small number of such sequences speculatively compiled. If the compiler can narrow the type of an expression to a few concrete types, these techniques can be very effective. This and virtually all other optimization of object-oriented languages will benefit from the superior information generated by our improved type inference techniques.

The constraint based type system of Aiken, Wimmers, and Lakshman [3], adds conditional types unions and intersections to an ML-style type inference allowing the incorporation of flow sensitive information. Our algorithm also shares some features of the closure analysis and binding time analysis phases used in self-applicative partial evaluators [17]. However, both these systems are for languages which are purely functional where the question of types involving assignment does not arise. The extension of the former system to imperative languages is not fully developed.

8 Summary and Future Work

We have developed and implemented an algorithm for the incremental inference of concrete types. This algorithm is based on novel techniques which direct type inference effort to where it is fruitful. We have presented techniques for discriminating between critical run time variables while minimizing redundant work and for extending the power of discrimination incrementally. These techniques make it possible to efficiently infer concrete types in programs with deeply polymorphic libraries and data structures.

We have implemented these techniques in the Illinois Concert compiler,[10] and have used them to infer concrete types in many 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. Our compiler currently uses the type information to inline and statically bind functions and methods, unbox variables and for interprocedural constant propagation. We are extending this to include speculative compilation (splitting), and the techniques discussed in Section 6.

In the future, we intended to improve the efficiency of our implementation. First, we are looking at ways of collecting creation points into creation sets efficiently. Our current implementation can either split each creation point into a new creation set or split only those which cause imprecision. While the latter produces fewer creation sets and therefore is potentially more efficient, it increases the number of passes and so far it is slower for most of our test cases (the reported statistics are for the former). Second, we wish to decrease the cost of successive iterations by not verifying constraints unnecessarily. Finally, we are looking at ways of summarizing the information for methods and functions. This would reduce the cost of splitting and successive iterations.

9 Acknowledgements

We would like to thank Vijay Karamcheti, Xingbin Zhang, Julian Dolby and Mahesh Subramaniam for their work on the Concert System. We would also like to thank Tony Ng, Jesus Izaguirre and Doug Beeferman for writing CA applications and for putting up with the early versions of the type inference implementation.

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. Agensen, 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 Agensen. Personal communication, 1993.

[10]The Illinois Concert System including this type inference system is available. Interested parties can contact achien@cs.uiuc.edu for more information.

[3] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Sort typing with conditional types. In *Twenty First Symposium on Principles of Programming Languages*, 1994. To appear.

[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] 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.

[6] Andrew Chien, Vijay Karamcheti, and John Plevyak. The concert system – compiler and runtime support for efficient fine-grained concurrent object-oriented programs. Technical Report UIUCDCS-R-93-1815, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1993.

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

[8] 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, September 1993.

[9] 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.

[10] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

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

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

[13] 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*, June 1993. To appear.

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

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

[16] Simon Peyton-Jones. *Implementation of Functional Languages.* Prentice-Hall International, 1987.

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

[18] Norihisa Suzuki. Inferring types in smalltalk. In *Eigth Symposium on Principles of Programming Languages*, pages 187–199, January 1981.

[19] 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 Code for titest7.ca

This code represents the various cases of polymorphism handled by the incremental type inference algorithm. The class `D` has the polymorphic instance variable `d` which is assigned an instance of the class `C` with polymorphic instance variable `c` containing either an instance of `A` or `B`. In addition, the method `create::D` allocates the instance of both `D` and `C`. In order to distinguish the two different uses of `c`, we need to split the entry sets for methods: `create::D`, `new:D`, `initial_D:D`, `new::C`, `initial_C::C`, as well as the various accessor methods: `d::D`, `set_d::D`, `c::C`, and `set_c::C`. The creation sets for `new::D` and `new::C` must also be split.

```
(class A a
       (parameters arg)
       (initial
(set_a self arg)))

(class B b
       (parameters arg)
       (initial
(set_b self arg)))

(class C c)

(class D d
   (parameters arg)
   (initial
      (set_d self arg)))

(method D create ()
(let* ((c1 (new C))
      (d1 (new D c1)))
  (reply d1)))

(method osystem initial_message ()
(let* ((va (new A 1))
       (vb (new B 2))
       (vd1 (create D))
       (vd2 (create D)))
   (set_c (d vd1) va)
   (set_c (d vd2) vb)
           (let ((theA (c (d vd1)))
                 (theB (c (d vd2))))
       (OUT (global console) (a theA))
       (OUT (global console) (b theB))
  (reply nil))))
```