Iterative Flow Analysis

John Plevyak Andrew A. Chien

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

Abstract

Control and data flow information is vital for the optimization of high level programming languages. Language features such as object-orientation and first class functions and selectors link data flow and control flow. For example, in an object-oriented program an object's run time type is used to determine the function (method) executed at an invocation point via dynamic dispatch. We present an iterative analysis which derives control and data flow information simultaneously. This analysis adapts to the structure of the program, efficiently deriving flow information at a cost proportional to the precision of the information obtained. The analysis results are directly applicable to such optimizations as static binding, inlining and unboxing. This analysis has been implemented in the Illinois Concert compiler, and we report quantitative results for a number of object-oriented programs.

1 Introduction

Control and data flow information is vital to optimizing compilers of high level languages. It is useful for constant, copy and lambda propagation [31], static binding, inlining and speculative inlining [9, 19], type recovery [33], safety analysis [24], customization [9], specialization [15] and cloning [18, 26] and other interprocedural optimizations [11]. In high level languages, data values can determine the code which is executed through first class functions and selectors as well as dynamic dispatch, and the code determines the data values. As a result, control and data flow must be analyzed simultaneously if precise information is to be obtained. The key to the precision of context sensitive flow analysis is the *contour* [33] representation. Contours represent the calling environments of a function; for example 0CFA uses one contour per function while 1CFA uses one for each call site [32].

Other flow analyses [20, 22] have used a fixed contour representation or adapted the representation with respect to the values of function arguments [1]. However, shallow fixed representations can require excessive amounts of memory [21] and imperative update of memory locations introduce cycles into the flow graph which can invalidate adaptive decisions after they have been made. Moreover,

---

An early version of some of this material appeared in the Proceedings of the 1994 Conference on Object-Oriented Programming Systems, Languages and Applications [27].

contours representing objects or closures can also benefit from representation adaption based on the values of their variables [27].

We present an iterative combined control and data flow analysis (IFA) which adapts the contour representation for both functions and objects to the program to derive information at a cost proportional to the amount of information obtained. The analysis constructs the flow graph locally from data flow and interprocedurally by abstract interpretation of call sites. We begin with a shallow analysis (similar to 0CFA) to produce a conservative global flow graph, then extend the contour representation where additional precision is required, repeating this process until the desired precision is reached.

We have implemented this analysis in the Illinois Concert system and applied it to a number of object-oriented programs. Our results indicate that precise information can be obtained for many common program structures including map functions, polymorphic containers and factory objects. Moreover, this additional information can be used directly for such optimizations as static binding, inlining, unboxing and data representation optimizations. We demonstrate this by showing that through IFA and cloning [26] approximately 99% of the methods in our test cases can be statically bound.

This paper is organized as follows. Section 2 describes basic flow analysis and our notation. Section 3 describes the language and the internal representation to which it is converted for analysis. The contour representation and basic analysis is presented in Section 4 and the algorithm for extending precision in Section 5. Aspects of the algorithm are discussed in Section 6. Section 7 reports our empirical results. Finally, we cover related work in Section 8, and conclude in Section 9.

2 Background

Context sensitive flow analysis of high level programming languages is a control and data flow analysis which combines elements of abstract interpretation [12] and data flow analysis [16]. Efficient implementations build the flow graph by abstract interpretation and update the values by propagation along the edges of the flow graph. Such implementations have been called *constraint-based* [23] since the flow graph resembles a constraint network, where the edges are constraints and the nodes are variables. For example, a flow analysis to determine the set of classes whose instances a variable might take on generates flow edges when an object of class `C` is created indicating that the result must be in the set containing at least `C`, {`C`}. Using $[\![v]\!]$ to denote the set of classes for variable v and N_v and N_C to denote the flow graph nodes for v and `C` respectively, the constraints and corresponding flow edges for creation and assignment would be:

program text	**constraint**	**flow edge**
$x = new\ C$	$[\![x]\!] \supseteq \{C\}$	$N_C \longrightarrow N_x$
$x = y$	$[\![x]\!] \supseteq [\![y]\!]$	$N_y \longrightarrow N_x$

When a call site is encountered during construction of the flow graph, the data flow values at the call site are used to determine the functions which may be invoked based on the dispatch semantics of the language. For example, in a single-dispatch object-oriented language, the methods (functions) are determined by the data flow values of the selector and receiver (target) object arguments. The flow graph computes a conservative approximation to these reaching problems. For example, given

a call site with a selector s with reaching selectors S and a reciever o with reaching classes C, flow edges are constructed for all methods in the cross product $S \times C$ (allowing for inheritance).

Since abstract interpretation of the call sites (construction of the flow graph) uses the data flow values (the current solution) to determine the functions called [22], the data flow values are updated concurrently with flow graph construction. The meet function for the data flow values is union and the transfer functions are derived by abstract interpretation. For instance, the transfer function for the set of class names $Name$ for the flow graph node representing the target object o at a call site with possible functions $F \subseteq S \times C$ would be:

$$Class(o)_{out} = Class(o)_{in} \cap \{c \mid (x, c) \in F\}$$

Likewise, constants, primitive functions and tests for equivalence with singleton objects (like NIL) produce transfer functions which affect the data flow values at nodes.

The context sensitivity of flow analysis follows from the *contour* representation [33, 20]. In the theory of flow analysis, the language to be analyzed is first given an exact semantics which is essentially an interpreter. The contours for such a semantics represent the call frames, include call path and determine variable bindings. For analysis, cost and precision are balanced by using abstract contours which represent some set of exact contours. A contour representation can therefore vary from coarse (one contour per function) to fine (one contour per call frame). Since unique flow graph nodes for local variables are created for each contour, a separate (context sensitive) solution is obtained for the calling contexts they represent. The selection of efficient contour representations is the subject of this paper.

3 Language

definitions	(`define` name ((variable | (variable name+))*) *expression**)
binding constructs	(`let` ((variable *expression*$_0$)*) *expression*$_1$*) (`let*` ((variable *expression*$_0$)*) *expression*$_1$*)
conditionals	(`if` *expression*$_0$ *expression*$_1$ *expression*$_2$) (`while` *expression*$_0$ *expression*$_1$*)
assignment	(`set!` *expression*$_0$ *expression*$_1$ *expression*$_2$)
variables	`self`, identifier
constants	Integer,1,2,3,... =,>,<,+,−,...

Table 1: Language Syntax

The language we will use is a simple object-oriented language with definitions, binding constructs, conditionals, assignment, variables and constants. The syntax of this language is given in Table 1. Definitions are used to define both generic functions and classes where the list of variables represents parameters and instance variables respectively. The **name** defined by a definition is available globally and definitions sharing the same **name** form a single generic function. The run time type of the arguments is matched with the **name**s of the parameter to determine which version is executed. The variable `self` is treated specially, indicating the object defined or, if used as an argument, the object whose instance variables are scoped. For example, a simple class `A` with accessor function `get-a` and `put-a` for its instance variable `a` can be defined as:

```
(define A (a)
  (define get-a ((self A)) a)
  (define put-a ((self A) (value Integer)) (set! a value))
  self)
```

Similarly, a generic function which returns double the value of an integer or of the `a` instance variable of an object of class `A` can be defined as:

```
(define double ((a A)) (+ (get-a a) (get-a a)))
(define double ((a Integer)) (+ a a))
```

Before analysis the simple language is converted to a variant of Static Single Assignment (SSA) form [13, 34]. SSA form inserts ϕ-Nodes, essentially assignments with multiple right hand sides where control flow merges, for example after a conditional, and renames variables so that each variable appears on the left hand side of only one assignment. In addition to simplifying the construction of the flow graph, this renaming prevents interference between transfer function. For example, on the left side of Figure 1 the variable `a` variously holds instances of class `A` and `Integer`, to which are applied `get-a` and + respectively. The transfer function requires that the type of `a` contain only those classes to which both `get-a` and + can be applied. Since there are no such classes, the analysis will incorrectly report that no type can be found for a. SSA conversion prevents this problem by creating new variables for each use of `a` on the right side of Figure 1. Similarly, instance variables are converted to SSA as aliasing information permits [29], or moved to temporaries before use to prevent interference.

```
(set! a (new A))
(get-a a)
(set! a 1)
(set! a (+ a a))
```

```
(set! a1 (new A))
(get-a a1)
(set! a2 1)
(set! a3 (+ a2 a2))
```

Figure 1: Code before (left) and after (right) SSA Conversion

However, SSA conversion alone is not sufficient since it handles conflicts only for variables which are assigned. A more common problem is presented by the use (reading) of a variable under different conditions. In order to prevent these conflicts we introduce ψ-Nodes which, analogous to ϕ-Nodes rename variables which are read along different control flow paths. For example, if two different functions (`get-a` and +) are applied to a variable in the two branches of a conditional (see Figure 2) the transfer function for the flow node corresponding to variable `a` would constrain the type of `a` to those classes supported by both functions.

```
(if ...
  (get-a a)
  (... (+ a a)))
```

```
(a1,a2) = ψ(a)
(if ...
  (get-a a1)
  (... (+ a2 a2)))
a3 = φ(a1,a2)
```

Figure 2: Code before (left) and after (right) SSU Conversion

The resulting program representation is called Static Single Use (SSU) form [25]. It is similar to [4] and is computed through a simple extension of the SSA conversion algorithm [13, 34].

4 Analysis

Iterative Flow Analysis (IFA) consists of two phases: analysis and incremental precision extension (discussed in Section 5). The analysis phase constructs the flow graph, while continuously updating the node values. The simple language coupled with SSU form, which induces an explicit local data flow graph, greatly simplifies the abstract semantics over other analyses [20, 35] allowing us to concentrate on the iteration algorithm. In particular, the simple language binds all variables in the function or `self` argument's contours, preventing the capture of variables from surrounding scopes. Extension of the analysis to additional language features is discussed in Section 8.

4.1 Definitions

$$
\begin{array}{rclrcl}
n \in Node & = & Label \times Contour & N \in Nodes & = & \mathcal{P}(Node) \\
e \in Edge & = & Node \times Node & E \in Edges & = & \mathcal{P}(Edge) \\
c \in Contour & = & \mathcal{N} & C \in Contours & = & \mathcal{P}(Contour) \\
v \in Value & = & \mathcal{P}(Node) & V \in Values & = & Node \rightarrow Value \\
r \in Restrict & = & Value_1 \times ... \times Value_n & R \in Restricts & = & Contour \rightarrow Restrict \\
i \in Invoke & = & \mathcal{P}(Contour) & I \in Invokes & = & Node \rightarrow Invoke
\end{array}
$$

Figure 3: The Flow Graph

The definition of the flow graph appears in Figure 3. Each expression in the program is given a unique *Label* except variables which use the label of the expression which binds them for local variables,[1] or their definition for instance variables and definitions which use their **name**. *Contours* are unique identifiers representing abstract calling environments; we use the natural numbers $\mathcal{N}$ where 0 is the top level environment. The *Value* of a node is the set of nodes representing the values (constants, function **name**s, or object contours) which reach that node. Each contour *Restricts* the values its parameters can take on. The *Invokes* function records the abstract call graph, mapping call nodes to invoked contours.

$$
\begin{array}{rcl}
Flow(n) & = & \{m \mid (n,m) \in E\} \\
Back(n) & = & \{m \mid (m,n) \in E\} \\
Function(n) & = & \{l \mid v' \in V(n) \wedge v' = (l,c) \wedge l \in \{\text{primitive function}, \text{function}\mathbf{name}\}\} \\
Class(n) & = & \{l \mid v' \in V(n) \wedge v' = (l,c) \wedge l \in \{\text{primitive class}, \text{class}\mathbf{name}\}\} \\
Object(n) & = & \{c \mid v' \in V(n) \wedge v' = (l,c) \wedge l \in \{\text{primitive class}, \text{class}\mathbf{name}\}\} \\
Name(v) & = & \{(l,0) \mid v' \in v \wedge v' = (l,c)\}
\end{array}
$$

Figure 4: Functions on the Flow Graph

We define two functions for moving along the edges of the flow graph: *Flow* and *Back* which takes a node to the set nodes in the forward and backward flow directions respectively. We also define functions to access the set of *Function* labels (generic function names or primitive functions), the set of *Class* labels (class names or primitive classes) and the set of *Object* values (which originate from the contour of `self` of top level functions). Finally, we use definitions in the top level environment to

[1] In SSU form, local variables are assigned only once.

stand for the definitions independent their contour and access these with $Name$. These definitions represent the set of all contour therefrom derived. Conceptually:

$$\{(l,0)\} \;=\; \{(l,c') \mid c' \in C\}$$

4.2 Flow Graph Construction

Construction of the flow graph uses a worklist of call nodes. Calls are taken from the worklist, the called contours are determined, and the local contribution to the flow graph is determined. The called functions are drawn from the applicable versions of generic functions reaching the function argument of the invocation. Functions are applicable when for each argument there is a reaching $Name$ which matches one of those associated (see Table 1) with the corresponding parameter. One or more contours are then selected for each function. In Section 5.2 we discuss selection of contours in detail. Finally, those call sites $Node$s whose arguments $Value$s have changed are added to the worklist.

The flow graph nodes for local variables and expressions are defined by their label and the selected contour. The nodes for instance variables are defined by their label and the *Object* value(s) of the `self` argument. The flow graph edges are the SSU assignments, the flow from arguments to parameters and from the function result to the call result. The transfer functions for parameter nodes impose the dispatch constraints (Section 2). For example, values are restricted to those having the **name**s associated with the parameter. The transfer functions for a contour c also restrict the values flowing from argument a_i to be $V(a_i) \cap R(c)_i$, enabling the use of separate contours for different combinations of values. Since any given variable can only hold one value at one time, separate analysis is safe so long as each element of the cross product of values is represented by some contour (the cross product rule). This is achieved in the alternative contour representation of [20] by single-value based analysis of curried functions.

4.3 Imprecision and Polymorphism

```
(define power (x y)
  (if (> y 0)
    (* x (power x (- y 1)))
    (one x)))

(let ()
  (power 1 2)
  (power 1.0 2))
```

Figure 5: Functional Polymorphism

```
(define tuple (l r)
  (define left ((self tuple)) l)
  self)

(let ()
  (left (tuple 1 2))
  (left (tuple 1.0 2)))
```

Figure 6: Polymorphic Objects

To simplify the exposition of the iterative algorithm, we differentiate *function imprecision* from *data imprecision*. Imprecisions are nodes whose values are not singleton sets. Function imprecisions are those of nodes defined by the surrounding function's contour. Data imprecisions refer to nodes defined by object contours (instance variables). Imprecisions result from incomplete input, flow insensitivity, and (for mutable locations) temporal insensitivity. This flow analysis focuses on the second sort which often results from the use of polymorphism functions or objects. Intuitively, the *level of polymorphism* is the depth of the polymorphic function call path or polymorphic reference

path (see Figures 5 and 6). Flow analyses typically produces precise results for up to a fixed level of polymorphism; for instance, 0CFA handles no polymorphism while 1CFA [31] handles one level. Since real programs use varying levels of polymorphism in different places, efficient analyses adapt locally to those levels. In the next section we present Iterative Flow Analysis (IFA) which uses the results from simpler analyses at lower levels to adapt the contour representation for successive iterations.

5 Iterative Flow Analysis (IFA)

Iterative Flow Analysis (IFA) uses the results of the previous iteration (starting with 0CFA) to extend the contour representation for the next iteration. Iteration is required because during analysis assignment to mutable locations (instance variables) can cause the value of nodes to change after their contours have been selected. After each iteration, the contour representation is extended by *splitting* the set of invocations associated with a contour (see Figure 3) to differentiate uses of the function or object it represents. A new analysis iteration starts by clearing the values V and the edges E which make up the flow graph. However, the abstract call graph I which captures the local levels of context sensitivity is preserved. In this way, the analysis adapts to the structure of the program. The result is an efficient allocation of analysis resources to the many levels of polymorphism in programs.

5.1 Splitting

Splitting divides contours, increasing the number of flow graph nodes and potentially eliminating imprecisions from the analysis results. Splitting polymorphic functions (*function splitting*) divides the invocations associated with a function contour over a number of smaller of more specific contours. Splitting polymorphic objects (*data splitting*) divides the invocations associated with the creation of objects of a particular class over a number of contours representing subsets of the instances which are used in different ways.

In its simplest form, splitting relies on the values of arguments, selecting a contour the values of whose parameters most closely match those of the arguments. The calls are processed in depth first fashion so the arguments have approximations of their final values when the contours for the call are selected. In order to minimize the number of iterations, the partial information is used to eagerly split function contours. Similarly, we can eagerly split contours representing objects. However, since the selection of these contours occurs at the point where the objects are created and before the instance variables are used, it generally is less effective. Eager splitting occurs as part of contour selection.

5.2 Selecting Contours

When a call is encountered, the set of applicable functions is determined and then contours are selected. For a given target function, the transfer functions for the dispatch are applied to the values of the arguments to determine the values which will flow into the parameters for this call (Section 4.2). While a contour could be created for each element of the cross product of entering values ($w = \prod_i v_i$), this would be expensive and, in general, prevent termination (see Section 6). Instead we select contours based on information from the last iteration and then eagerly split contours based on the **name**s of the argument values, leaving splitting based on the contour component ($Object$) of the values to be done non-eagerly.

The contours for a call from node n are selected in three steps. First, from the cached contours $I(n)$ we select those whose restriction cross product $\prod_i r_i$ intersects w, favoring those which intersect the smallest number of elements, and remove those elements from w. For any remaining elements of w we select from all contours associated with the function those whose restrictions intersect w. Finally, we form subsets out of any remaining elements by applying $Name$ to each parameter value ($\prod_i Name(v_i)$) and create contours for each identical result with the singleton $Name$s as restrictions. Intuitively these contours are insensitive to particular contours reaching their parameters, but are (eagerly) differentiated with respect to the names of the functions or classes reaching those parameters.

5.3 Function Splitting

Function splitting partitions contours, enabling separate information to be obtained for different uses of the function. In its simplest form, we examine the values of the arguments of all the invocations for a particular contour, and if one of the argument's value is a strict subset of the corresponding parameter value, a new contour is created for that invocation. In practice, there may be many object contours for a particular class definition which distinguish subsets of the class's instances important to only a fraction of functions. So instead we start from a specific imprecision which we wish to eliminate (e.g. where a type check or dynamic dispatch would be required) and look for the possible causes.

Using the functions described in Figure 4 we traverse the flow graph. Starting from the point of imprecision we look back for a set of *confluences*[2] (in data flow terms, meets $a \wedge b$ where $a, b \neq \emptyset$ and $a \neq b$). Given a node n with imprecise Val (one of $Function$, $Object$ or $Class$) we find the least set of confluences $Conf(n, Val)$ which obey:

$$\begin{aligned} Conf(n,Val) &\supseteq \begin{cases} \{n'\} & \text{if } \exists n' \in Back(n) \wedge Val(n) \neq Val(n') \\ \emptyset & \text{otherwise} \end{cases} \\ Conf(n,Val) &\supseteq Conf(n',Val) \text{ where } n' \in Back(n) \end{aligned}$$

The contours of argument nodes in $Conf(n, Val)$ represent the first order contribution to the imprecision, and splitting them is the first way in which the imprecision may be eliminated. Imprecision can also arise from interprocedural control flow ambiguity due to secondary imprecisions in other arguments. For example, since the result value is the meet (union) of the results of all the possible functions invoked, if the set of functions reaching a call is imprecise, the result can be imprecise. Similarly, the parameters of the invoked functions might be imprecise as a result of the extraneous flow edges. Lastly, imprecisions in object contours can lead to imprecise results of instance variable accesses. We extend $Conf(n, Val)$ to $Conf'(n, Val)$ to handles these cases, where i is an invocation:

$$\begin{aligned} Conf'(n,Val) = Conf(n,Val) \cup \\ \{(n'' \mid n' \in Conf(n,Val) \vee |Val(n)| > 1) \wedge \\ (n' \text{ is an argument or return variable of } i \ \wedge \\ (n'' \in Conf'(SelfArgument(i), Object) \vee \\ n'' \in Conf'(FunctionArgument(i), Function) \ \vee \\ n'' \in Conf'(DispatchArgument(i), Class)))\} \end{aligned}$$

[2]This is not to be confused with the Church-Rosser property. Instead, we draw on definition 1 from the Oxford English Dictionary (second edition): *A flowing together; the junction and union of two or more streams or moving fluids.*

The three occurrences of *Conf'* lower down account for the effects of imprecise object contours, imprecise reaching functions and imprecise generic function dispatch respectively. The newly split contours are distinguished in the next analysis phase through the changes to abstract call graph I or additional restrictions R. For confluences ($Conf(n, Val)$ is non-empty) at an argument node (l, c), we create new contours C' with identical restrictions $\forall c' \in C'.R(c') = R(c)$, but with I mapping callers with identical values to seperate contours (i.e. $\bigcup V(v' \in Back((l, c')) = V((l, c'))$). For an imprecision ($|Val(n)| > 1$) at argument node n at position i, we create new contours with identical invokes ($\forall l \in Label.I(l, c') = I(l, c)$) and modify the restrictions $r = R(c')$ to differentiate the elements of $Val(n)$ (e.g. $|V(r_i)| = 1$). Different contours will them be selected in the next iteration.

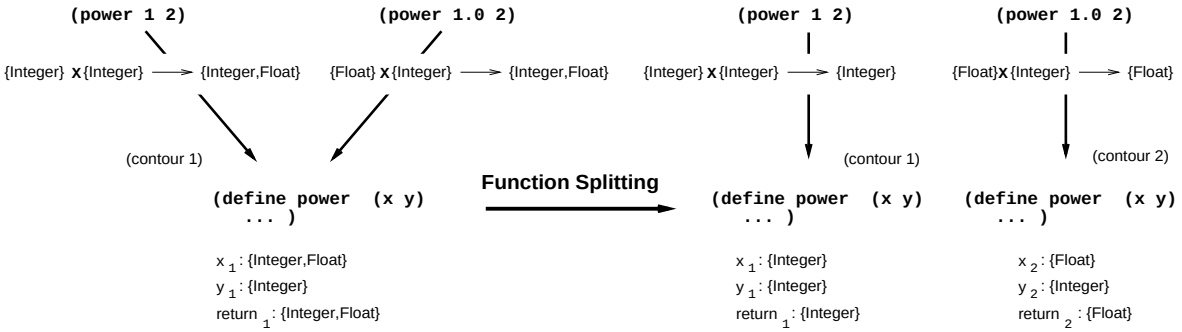


Figure 7: Function Splitting for Integers and Floats

Figure 7 illustrates function splitting involving the `power` function from Figure 5. At the left, the actual arguments for the formal parameters `x` and `y` coming from `(power 1 2)` and `(power 1.0 2)` have different *Class*es, so there is a confluence. The imprecision manifests itself in a confluence at the return value {`Integer`,`Float`}, when it is clear that the value for the first call is `Integer` and for the second `Float`. Splitting introduces two sets of nodes $\mathtt{x}_1$,$\mathtt{y}_1$ and $\mathtt{x}_2$,$\mathtt{y}_2$, eliminating the confluence.

5.4 Data Splitting

Data splitting partitions contours based on the usage of the objects they represent. It is more complex than function splitting because the point of confluence (the instance variable) is separated from the point at which the contour was created in the flow graph. In fact, since object contours flow through the graph, splitting the object contour alone is not enough; we must ensure new contours remain distinct as they flow through the flow graph.

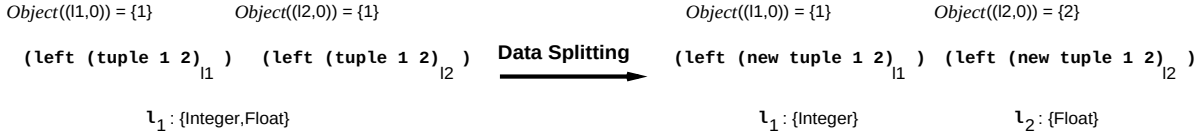


Figure 8: Data Splitting for Imprecision at `l`

Figure 8 is an example of data splitting based on the program example in Figure 6. On the left, the two creation points, `(tuple 1 2)` and `(tuple 1.0 2)` produce the same contour. As a result,

the value of the instance variable `l` is {`Integer`,`Float`}. Splitting the object contour discriminates the two cases, produce precise results for both cases.

Data splitting involves four basic operations.

1. Identifying the assignments to the instance variable which give rise to the imprecision.
2. Identifying the paths in the flow graph which the instance variable's contour took from its creation point to the assignments.
3. Ensuring that these paths are distinct.
4. Dividing the contour into a set of contours.

The first step is to identify the conflicting assignments to the same instance variable with the same contour. Next we find the flow paths from the creation of the contour c which defines the instance variable node n (e.g. $n = (l, c)$) to the assignments. These paths must be disjoint to propagate the distinct contours we introduce by data splitting, or we will fail remove the imprecision. We ensure this by function and data splitting along the flow paths where necessary. With paths in hand, we resolve the conflicting values assigned into different contours splitting the original contour.

```
(define A (a)
  (define set_a ((self A) value)
    (set! a value))
  self)

(let ((x (A)_c)              ;; I((c,0)) = {1})
      (y (A)_d)              ;; I((d,0)) = {1})
  (set_a x 1)_e              ;; I((e,0)) = {2})
  (set_a y 1.0)_f            ;; I((f,0)) = {3})
```

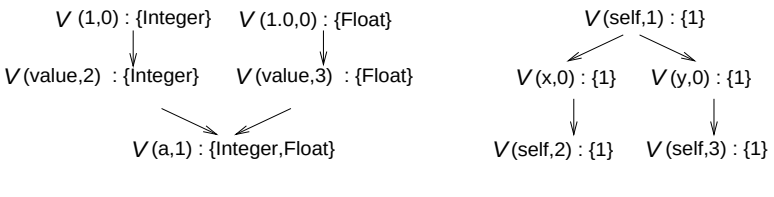
Figure 9: Data Splitting Example

To illustrate some of the equations we will use the example in Figure 9. In this example, two instances of class `A` are created. The function `set_a` is then used to set the `a` instance variable of each to a different type of number.

Identifying the Assignments First, we find the nodes which are assigned (have a flow edge to) the imprecise instance variable. These are grouped so that all the nodes in each group have identical values with respect to the type of the imprecision, indicated by the parameterizing function Val. We define the function $AssignSets(n, Val)$ which takes a node n, an imprecise instance variable, and return a set s of sets of nodes s_i, each of which represents a different use of the instance variable.

$$AssignSets(n, Val) \quad = \quad s \text{ where } \bigcup_i s_i = Back(n) \wedge \forall s_i \in s, n' \in s_i, n'' \in s_i . Val(n') = Val(n'')$$

The nodes in $Back(n)$ are the right hand sides of assignments to the imprecise instance variable. Figure 10 shows the flow graph for our example, and the assignment sets derived.



$$\mathbf{AssignSets}((\mathtt{a},1),\mathbf{Name}) \quad = \quad \{\{(\mathtt{value},2),(\mathtt{value},3)\}\}$$

Figure 10: Data Flow and Assignment Sets Example

Identifying the Paths Next, we compute the flow paths which the instance variable's contour took from its creation point to the assignments. For each element a of $AssignSets(n, Val)$ we find the `self` nodes $Self(a)$ of the functions containing the assignments a. The $Object$ values of these nodes contain the contour c which defines the instance variable node $n = (l, c)$. Then we compute the paths taken by the contour from its creation point (the node $(\mathtt{self}, c)$) to $Self(a)$. These paths must to be distinct in order to eliminate the imprecision. Intuitively, if the contour's paths merge they will be applied to the same functions with the same values (by the cross product rule of Section 4.2).

$$\begin{aligned} a &\in AssignSets(n, Val) \\ Self(a) &= \{(\mathtt{self}, c) \mid (l, c) \in a\} \\ Path(a) &= Closure(Back, Self(a)) \end{aligned}$$

We compute the paths $Path(a)$ for each assignment a by taking the closure of the function $Back$ over the set containing the `self` nodes for the functions containing the assignments. The `self` nodes are found with the function $Self(a)$ by extracting the function contour from a.

The paths $Path(a)$ are those which would be taken by a new contour created to eliminate the portion of the imprecision $Val(a)$. For example, in Figure 11 *1* travels to $\mathtt{value}_2$ through $\mathtt{self}_2$ and $\mathtt{x}_0$. Since this path must be distinct from the other paths the appearance of a node on more than one of these paths represents a secondary imprecision. For each node we need to know the subset paths in which it is contained.

$$\begin{aligned} AllPaths(n, Val) &= \{p \mid p = Path(a) \wedge a = AssignSet(n, Val)\} \\ NodePaths(n', n, Val) &= \{ps \mid n' \in ps \wedge ps \in AllPaths(n, Val)\} \end{aligned}$$

We define the function $AllPaths(n, Val)$ to be all the paths for all the assignment sets. Further, we define $NodePaths(n, Val)$ to be the subset of all the paths which a particular node n' is on.

Ensuring Discrimination Using the paths determined above, we now apply the confluence finding algorithm recursively to determine the confluences of the potential contours represented by these paths. However, the paths are defined by the assignments, and join at the creation point whereas the other values are distinct when created but join at merges in the flow graph. Thus the path can

be thought of as flowing backward in the data flow graph. This requires modification of the $Conf$ function:

$$FlowOrBack(Val) = \begin{cases} Flow & \text{if } Val = Path \\ Back & \text{otherwise} \end{cases}$$

$$Conf(n, Val) = \begin{cases} \{n\} & \text{if } \exists n' \in (FlowOrBack(Val))(n) \ \wedge Val(n) \neq Val(n') \\ \emptyset & \text{otherwise} \end{cases}$$

The new $Conf$ uses the $FlowOrBack(Val)$ function which is either $Back$ as before or $Flow$ when Val refers to the paths. $AssignSet$ requires a analogous change, and the rest of the algorithm is identical.

$$\begin{aligned} Path(\{(\texttt{value}, 2)\}) &= \{(\texttt{self}, 2), (\texttt{x}, 0), (c, 0)\} \\ Path(\{(\texttt{value}, 3)\}) &= \{(\texttt{self}, 3), (\texttt{y}, 0), (d, 0)\} \end{aligned}$$

$$Splittable(0, (\texttt{a}, 1), Class) = \{\{(c, 0)\}, \{(d, 0)\}\}$$

Figure 11: Paths and Splittability Example

Resolving the Imprecision The last step is the actual splitting of the object contours. When two or more paths do not share any nodes, the contour can be split and a new contour created for each path or set of paths not sharing nodes. Figure 11 provides an example of a contour is determined to be splittable. The new contour will cause the node representing the instance variable at the point of the imprecision to split, removing the imprecision. The function $Splittable(c, n, Val)$ determines the subsets of creation points for contour c which can be profitably split for the imprecision at node n of type Val:

$$\begin{aligned} Splittable(c, n, Val) &= \{t \mid t \subset s \wedge \bigcup NodePaths(n' \in s, n, Val) \neq AllPaths(n, Val) \wedge \\ &\qquad \forall n' \in t, n'' \in t, NodePaths(n', n, Val) \cap NodePaths(n'', n, Val) \neq \emptyset\} \\ \text{where } s &= \{n \mid n \in p \wedge p \in AllPaths(n, Val) \wedge Back(n) = \{(\texttt{self}, c)\}\} \end{aligned}$$

Using s the nodes which represent the creation points for the object contour c, we determine the subsets of creation points whose paths are not disjoint. Since the creation points are the end of the paths, non-disjointness implies equality and that the union of these subsets will be s. Further, since creation points correspond to invocations on the class (object creation) function, $Splittable(c, n, Val)$ computes the sets the invocations for the new contours. Thus, if

$$Splittable(c, n, Val) = AllPaths(n, Val)$$

the object contour cannot be profitably split. When the object contour is split, the newly created contours are substituted for the original in the restrictions for argument nodes along the corresponding paths. Thus the new contours will follow the distinct paths in the next iteration, and their instance variables will be assigned a subset of the values of the original, removing the imprecision.

6 Discussion

In this section we discuss termination and complexity issues, and how features of less restrictive languages than that of Section 3 can be handled. Since the contour representation used by IFA is not static and recursively defined, recursion in the program being analyzed requires special handling.

6.1 Recursion

Since the definition of contours is recursive, ensuring termination requires limiting the number of contours produced by recursive program structures. There are three types of these structures:

- Recursive functions
- Recursive data structures
- Function-data recursion

The first two types are normal recursive functions and recursive types. The third type represents the case where a recursive function creates objects on which it is later invoked. This is the case for such common programming idioms as insertion into a linked list. While contours are represented by unique identifiers, their uniqueness is determined by their callers I and their restrictions R. The first two types of recursive structures induce other contours by invocation I while the third induces them through restrictions R.

After each iteration and before splitting, we identify the strongly connected components (SCCs) in the graph whose nodes are the contours and whose edges are:

- a contour c and the contours it invokes $\{c' \mid c' \in I((l, c))\}$
- a contour c and the contours it restricts $\{c' \mid (l, c) \in R(c')_i\}$

The SCCs in this graph contain the contours which have a part in defining each other. To prevent non-termination we do not allow invocations or restrictions between contours in the same SCC to cause splitting. Furthermore, invocations into recursive cycles can also lead to non-termination as recursive cycles are successively "peeled". These invocations are also prohibited from splitting beyond a constant level (in our implementation, two levels). Allowing invocations on the cycle to split to a constant level enables analysis of recursive structures with a period less than or equal to the constant since contours can form cycles up to that length.

6.2 Complexity

Termination is ensured by limiting the number of contours produced by recursion. However, since $Node$s and $Value$s are defined by labels (program points) and contours, which in turn can be distinguished by their values at each argument, the theoretical complexity is exponential. Nevertheless, in practice, we have found the complexity to be related to both the size and levels of polymorphism of the analyzed program. Furthermore, we have found that the level of polymorphism in programs increases relatively slowly with program size, and the complexity of analysis along with it (see Section 7).

6.3 Additional Language Features

The simple language defined in Section 3 does not have some features which are present in sufficiently related languages to be of interest here. In particular, it does not allow variables captured from surrounding lexical scopes (closures), and has neither first class continuations nor arrays. Closures can be handled by extending the notion of contours to include a map from the *Label*s of those variables to *Contour*s [20] and by using data splitting to extend the precision of captured variables. First class continuations are handled by building a second flow graph with nodes for each primary flow graph node containing a continuation and edges which are the inverse of those in the primary graph. The values in the secondary graph are those to which the continuations are applied. Appropriate transfer functions pass values between the two graphs [25] when the continuation is used. Finally, the contents of arrays can be analyzed homogeneously as a single instance variable, using a special *Label* to represent array contents.

7 Implementation and Empirical Results

We have implemented the analysis algorithm and tested it on more than 40,000 lines of Concurrent Aggregates (CA) [10]. CA is a single dispatch and single inheritance object-oriented language similar to the simple language described in Section 3, but extended for concurrency and including first class continuations and messages. The implementation is fully integrated into the compiler and complete; no language features were excluded. In this section we present an empirical study on a selection of programs. These programs were chosen to represent a cross section of applications, library functions and test cases. All but the smallest three were written by different authors.

Program	ion	network	circuit	pic	mandel	tsp	mmult	poly	test
Lines	1934	1799	1247	759	642	500	139	41	39

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 queuing 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).[3]

7.1 Analysis

We implemented three different analysis algorithms: 0CFA with one flow graph node per program variable, OPS [22] with contours distinguished by their immediate caller (i.e. one level of caller-based splitting for functions and objects), and Iterative Flow Analysis (IFA). We compared these algorithms based on precisions, time and space complexity.

[3]The compiler, language manual [10] and codes are available via at http://www-csag.cs.uiuc.edu.

Algorithm	*Progs Typed*	*Progs Failed*	*Type Checks*	*Runtime (secs)*
IFA	9	0	0	199
OPS	3	6	99	150
0CFA	0	9	718	34

7.1.1 Precision

We use two criteria for precision: typing (assignment of types such that run time type checks are not required) and elimination of dynamic dispatch. In this section we cover the former, leaving the latter for Section 7.2. The table above shows that 0CFA was unable to type even simple programs. OPS fared little better, typing only three of nine programs. However, IFA was able to type all the programs. The times are for our implementation in CMU Common Lisp/PCL on a Sparc10/31.

Program	*Lines*	OPS *Typed?*	*Time Sec.*	IFA/ OPS
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 12: Efficiency of Type Inference Algorithms

7.1.2 Time Complexity

Figure 12 shows the time taken by the three algorithms which were implemented in the same framework using identical data structures. Note that the speed of IFA compares favorably to that of OPS in two of the three cases where the were both able to type the program. This is because IFA focuses its effort only on areas of the program where it is required. However, when IFA produces better information, it often required more time.

7.1.3 Space Complexity

We compare the space complexity by examining the number of contours used per method (the number of nodes used by each algorithm are reported in the Appendix A). Figure 13 show the number of contours required IFA and OPS as a multiple of the methods in the program. IFA requires 1.5 and 2.5 per method while OPS requires 2.5 - 4. While additional contours can result in greater precision, IFA's goal directed splitting reduces number required for a given level of precision.

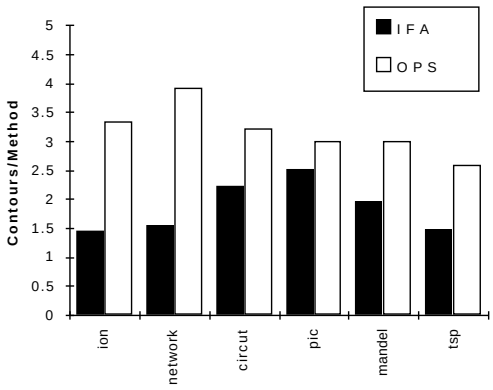


Figure 13: Contours per Method

7.2 Optimization Opportunities

Here we examine the usefulness of the additional information provided by IFA for the purpose of optimization. We compare code analyzed with IFA and optimized to that analyzed with 0CFA with and without optimization. We neglect OPS for two resons: 1) we are aware several implementations of 0CFA used for optimization and non of OPS and 2) in our test cases OPS produced little additional information at much higher cost. The unoptimized code represents the lower bound on efficiency, indicating the number of methods and messages required by a naive implementation. The optimized 0CFA version uses customization [6] to create specialized versions of methods for each receiver (target object) class. The optimized IFA version further clones and specializes methods based on the classes of all arguments [26, 14, 15].

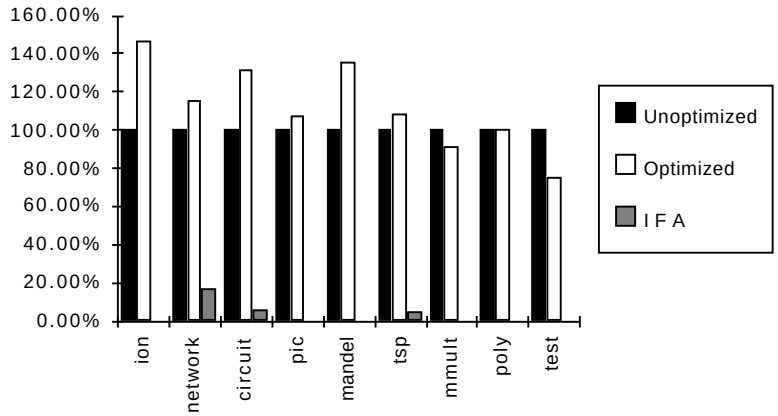


Figure 14: Dynamic Dispatch Sites In Code

7.2.1 Dynamic Dispatch Sites Removed

Figure 14 presents the effects of optimization and the IFA analysis on the percentage of dynamic dispatch sites in the program code. The unoptimized code provides the baseline. In most cases optimization increases the number of dynamic dispatch sites by inlining methods containing them in

more than one place, but in two cases, optimization eliminates unreachable code containing dynamic dispatches. The IFA analysis combined with cloning is able to remove 80 to 100 percent of the dynamic dispatch sites in all cases.

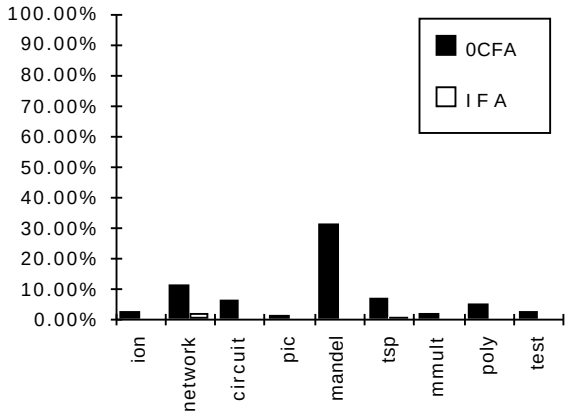


Figure 15: Percentage of Total Invocations Statically Bound (Dynamic Counts)

The dynamic effectiveness of dispatch removal was measured by running the programs on sample input. Figure 15 reports the number of dynamic dispatches occurring during runs for code analyzed with 0CFA and IFA. These percentages are with respect to the total number of invocations executed for the unoptimized version of the code. In most cases, the 0CFA algorithm statically binds better than 90 percent of the invocations, and the IFA algorithm better than 99 percent. However, many of the statically bindable invocations will be eliminated by optimization, as we explain below.

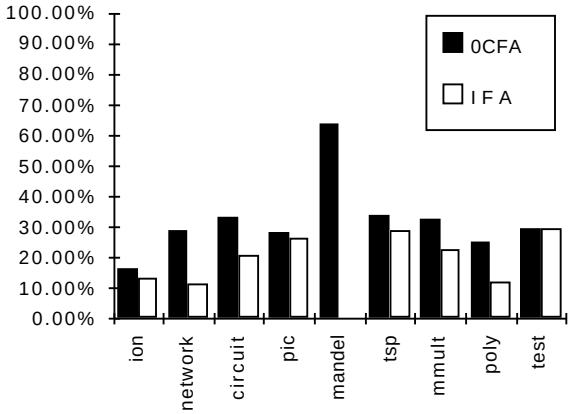


Figure 16: Percentage of Invocations Remaining after Optimization

7.2.2 Effect of Optimization

Standard optimizations including inlining can have a dramatic effect on the number of invocations made during program execution. We inline statically bound methods using heuristics based on static

frequency estimation [38], the size of both the caller and callee method, and the inline depth. In Figure 16 we report the number of total invocations (static and dynamic) remaining after optimization. For 0CFA, optimizations eliminate all but an average of about 30 percent, while for IFA the result is closer to 20 percent.

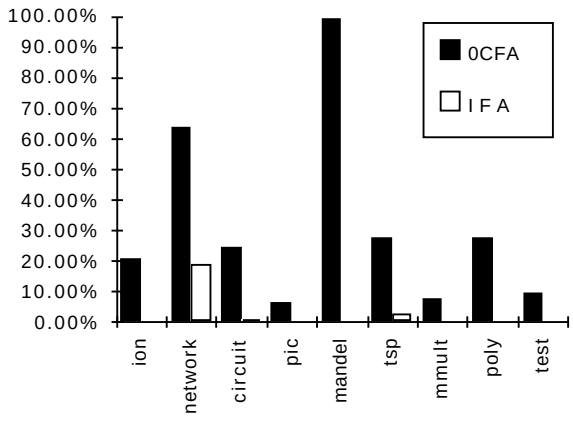


Figure 17: Percentage of Remaining Invocations Dynamically Bound

From the percentage of invocations statically bound with respect to the total in unoptimized code it might seem that 0CFA performed satisfactorily. However, since inlining decreases the number of invocations dramatically, that conclusion is unjustified. Figure 17 reports the number of dynamic dispatches with respect to the number of invocations remaining in the optimized code. For 0CFA, better than 20 percent in most cases, and in some cases better than 50 percent of all invocations require dynamic dispatch. For IFA with cloning, in all but one case less than 3 percent of the remaining invocations require dynamic dispatch.[4]

8 Related Work

The use of non-standard abstract semantic interpretation for flow analysis in Scheme by Olin Shivers [33] 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. Recently, Stefanescu and Zhou [35] as well as Jagannathan and Weeks [20] have provided simplified frameworks for flow analysis.

Iterative type analysis and message splitting using run time testing are conceptually similar techniques developed in the SELF compiler [6, 7, 8]. However, iterative type analysis does not type an entire program, only small regions. Later work by Hölzle [19] on the SELF-93 compiler uses the results of polymorphic inline caches to determine likely run time types, inserting type tests to ensure that the expected actually occurs.

Type inference in object-oriented languages in particular has been studied for many years [36, 17].

[4]This normalization is somewhat misleading since the absolute number of invocations within the optimized IFA code is less than that within optimized 0CFA code. As a result, the relative frequency of dynamic dispatches within optimized code exaggerates the absolute number in IFA relative to OPS. The absolute numbers are reported in Appendix A.

Constraint-based type inference is described by Palsberg, Schwartzbach and Oxhøj in [23, 22]. Their approach was limited to a single level of discrimination and motivated our efforts to develop an extendible approach. Agesen [1, 2] extended the basic one level approach to handle the features of SELF [37]. However, this approach uses a single pass, limiting it to eager splitting.

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 [30], again for purely functional languages.

9 Conclusion

We have developed and implemented an algorithm for context sensitive flow analysis of high level programming langugages. This algorithm uses a novel contour representation which is iteratively extended, enabling efficient analysis of many common programming structures. We have implemented this algorithm as as part of the Illinois Concert System whose goal is to develop portable efficient implementations of concurrent object-oriented languages on parallel machines. Our empirical demonstrate that the algorithm produces information at a cost proportional to the amount obtained and that the information is valuable for optimizing compilers.

Our compiler currently uses flow information with cloning [26] to eliminate dynamic dispatch, inline functions and methods, unbox variables, as well as for interprocedural constant propagation and locality approximation. This information has enabled us to achieve C-like performance for object-oriented programs on numerical codes [29] and to specialize the calling conventions for distributed programs [28]. We are currently expanding the framework for more interprocedural analyses, and looking at ways to enable summarization [3] and to characterize the precision of the algorithm.

10 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 and Urs Hölzle. Type feedback vs. concrete type analysis: A comparison of optimization techniques for object-oriented languages. Technical Report TRCS 95-04, Computer Science Department, University of California, Santa Barbara, 1995.

[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] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 257–271. ACM SIGPLAN, June 1990.

[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*, pages 1–15, May 1991.

[9] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, Stanford, CA, March 1992.

[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] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the R^n environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[12] Patrick Cousot and Radia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth Symposium on the Principles of Programming Languages*, pages 238–252, 1977.

[13] 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.

[14] Jeffrey Dean, Craig Chambers, and David Grove. Identifying profitable specialization in object-oriented languages. Technical Report TR 94-02-05, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, February 1994.

[15] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programmin g Language Design and Implementation*, pages 93–102, La Jolla, CA, June 1995.

[16] Kildal G. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

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

[18] M. W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, 1991.

[19] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336, June 1994.

[20] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Twenty-second Symposium on Principles of Programming Languages*, pages 393–407. ACM SIGPLAN, 1995.

[21] Suresh Jagannathan and Andrew Wright. Effective flow analysis for avoiding run-time checks. Technical Report 95-3, NEC Research Institute, May 1995.

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

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

[24] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference for partial types. *Information Processing Letters*, 43:175–180, 1992.

[25] John Plevyak. *Optimization of Object-Oriented and Concurrent Programs.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois. In Preparation.

[26] John Plevyak and Andrew Chien. Efficient cloning to eliminate dynamic dispatch in object-oriented languages. Submitted for Publication, 1995.

[27] John Plevyak and Andrew A. Chien. Precise concrete type inference of object-oriented programs. In *Proceedings of OOPSLA*, 1994.

[28] John Plevyak, Vijay Karamcheti, Xingbin Zhang, and Andrew Chien. A hybrid execution model for fine-grained languages on distributed memory multicomputers. Submitted for Publication, 1995.

[29] John Plevyak, Xingbin Zhang, and Andrew A. Chien. Obtaining sequential efficiency in concurrent object-oriented programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 311–321, January 1995.

[30] 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.

[31] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University Department of Computer Science, Pittsburgh, PA, May 1991. also CMU-CS-91-145.

[32] Olin Shivers. The semantics of scheme control-flow analysis. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–198, June 1991.

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

[34] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Twenty-second Symposium on Principles of Programming Languages*, pages 62–73. ACM SIGPLAN, 1995.

[35] Dan Stefanescu and Yuli Zhou. An equational framework for the flow analysis of higher-order functional programs. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 318–327, 1994.

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

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

[38] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Orlando, Florida USA, June 1994.

A Experimental Results

The results of analysis for the three algorithms on a variety of complete, kernel and synthetic programs appear in Table 2. IFA refers to our incremental inference algorithm, OPS refers to the inference algorithm in [22], and 0CFA refers to a standard flow insensitive algorithm which allocates exactly one type variable per static program variable.

Program	*Lines*	*Passes*	*Nodes*	*Invokes*	*Contours*	*Typed?*	*Checks*	*Im*	*Time*
IFA									
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
OPS									
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
0CFA									
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 2: Results of Iterative Flow Analysis

The number of **Passes** is determined by the algorithms automatically when it determines that no run time type errors are possible. **Nodes** is the number of flow graph nodes used by the algorithm. **Invokes** is the number of invocations (abstract calls) analyzed. **Contours** is the number of contours. In 0CFA this corresponds to the number of methods. A program can be **Typed?** by an algorithm if it can prove an absence of run time type errors. **Checks** is the number of type checks which must be made to ensure such an absence. The number of imprecisions *Im* indicates number of nodes which were not resolved to a singleon value. The implementation is approximately 2600 lines of largely unoptimized Common Lisp/CLOS and **Time** in seconds is reported for CMU Common Lisp/PCL on a Sparc10/31.