Analysis for Concurrent Objects

Paolo Di Blasio
Dipartimento di Informatica e Automazione, Università di Roma Tre Via della Vasca Navale 84, I-00146 Roma, Italy, ph. +39-6-55177052, fax: +39-6-5573030, email: `diblasio@inf.uniroma3.it`

Kathleen Fisher
AT&T Labs Research
600 Mountain Ave, Murray Hill, New Jersey, 07974, USA, ph. +1-908-582-7584, fax: +1-908-582-4113, email: `kfisher@research.att.com`

Carolyn Talcott
Computer Science Department, Stanford University
Stanford, California, 94305, USA, email: `clt@sail.stanford.edu`

Abstract
We present a set-based control flow analysis for an imperative, concurrent object calculus extending the Fisher-Honsell-Mitchell functional object-oriented calculus described in (Fisher *et al.* 1993). The analysis is shown to be sound with respect to a transition-system semantics.

Keywords: concurrent object-oriented, control-flow analysis, soundness, prototype-based

1 INTRODUCTION

A well-designed set of computation abstractions for concurrent object-oriented programming, equipped with a well-developed formal semantics, is important because it (*i*) helps write simpler, easier to manage programs; (*ii*) facilitates correct and efficient implementations; (*iii*) provides a basis for tools to help programmers; and (*iv*) aides program specification and verification. As we move to the world of mobile code and agents, a clear formal semantics becomes even more crucial (cf.(Magic 1996, McGraw *et al.* 1996)).

A well-developed formal semantics includes: operational semantics with clear computational meaning; abstract compositional semantics upon which to base specifications and transformations; and sound formal analyses such as type systems, effect systems, and data and control flow analyses. Program analyses are particularly important and difficult to perform in languages which provide first-class objects and possibly first-class functions in addition to concurrency and dynamic process creation. For example, in the presence of first-class objects, the target of a method invocation is generally not known until run-time. Thus the code to be executed is dynamically determined. Similarly, in the case of first-class functions, the function part of an application is determined at run-time. Hence analyses and program manipulations in this situation may require a control flow analysis. In the case of higher-order programs such an analysis determines (an upper bound on) the set of functions (closures) that may be returned as values at each program point. In the case of first-class objects, a control flow analysis means determining the objects (and their possible method tables) that may be returned as values from each program point. Extensive work has been done to develop techniques to perform control flow analysis for sequential higher-order (*e.g.*, (Shivers 1991, Flanagan *et al.* 1995)) and object-oriented (*e.g.*, (Palsberg *et al.* 1991, Holzle *et al.* 1995)) languages. Formal analysis methods for concurrent programs typically do not treat dynamic creation of processes or higher-order entities. Two exceptions are work on CML (Concurrent ML) (Nielson *et al.* 1994, Colby 1995). Although not explicitly object-oriented, CML exhibits many of the challenges of concurrent object-oriented languages. Regarding formal analysis work that treats the combination of concurrency and object-oriented programming, (Kobayashi *et al.* 1995) presents a static analysis of communication which may be used to reduce the cost of implementing message passing, and (Plevyak *et al.* 1995) describes optimization techniques (*e.g.* inlining, state caching) based on type inference information.

Our objective is to carry-out an in-depth semantic study of a representative concurrent object-oriented language that is simple enough to manage but complete enough to exhibit the problems and challenges. The language we have chosen is an imperative and concurrent extension of the Fisher-Honsell-Mitchell functional object-oriented calculus described in (Fisher *et al.* 1993). This calculus belongs to the family of so-called prototype-based object-oriented languages, in which new objects are created from existing ones via the inheritance primitives of object extension and method override. Objects interact by message passing, which results in method invocation. Concurrency is achieved through a combination of asynchronous method invocation and the identification of objects and processes. Without asynchronous communication the amount of concurrency would be static, and it would be necessary to add some form of spawning primitive. We have also chosen to directly support synchronous (rpc) method invocation because it is a commonly used pattern of interaction and it is not conveniently encodable via asynchronous message passing, although translations of synchronous communication into asynchronous have been demonstrated (Amadio 1994, Mason *et al.* 1997). Synchronization constraints, which describe restrictions on the availability of methods, are specified through guards. We adopted guards because they provide one of the most natural ways to define synchronization code and require minimal additional syntax.

In (Di Blasio *et al.* 1996) we define an operational semantics for our language and develop a static analysis for it that includes a type inference system to detect *message-not-understood* errors and an effect system to guarantee that synchronization code is side-effect free. We prove the soundness of the type and effect system with respect to the operational semantics. In this paper, we describe a control-flow analysis technique for our language. In developing our analysis, we extend the approach outlined in (Flanagan *et al.* 1995) by Flanagan and Felleisen for a fragment of the higher-order language Scheme. We consider this work as a first step towards the application of formal techniques for program optimization to concurrent object-oriented languages. Along these lines, we show some examples of applying these techniques. Much of this material is based on the Ph.D. thesis of one of the authors (Di Blasio 1997), where proofs are presented in detail.

2 EXAMPLES

Eager invocation. To provide some intuition for programming in our calculus, we describe how to encode *eager invocation*, a communication mechanism that is a compromise between synchronous and asynchronous communication. Eager invocation increases concurrent activity with respect to synchronous communication because like asynchronous communication, sender objects do not block when they send messages. Unlike the asynchronous case, however, eager invocation allows the result of an invocation to be returned to the sender, by means of a *future variable*. When the sender needs a result, it accesses the relevant future variable. If the result has not yet been stored in the future variable, the sender blocks and waits for it. Future variables are encoded as objects as follows:

$$\begin{array}{lllll} \textit{future-var} = & \langle & \textit{get} & = & \textit{when}(\lambda \textit{self}.\, \textit{false})\ \ \lambda \textit{self}.\, \lambda \textit{arg}.\, c, \\ & & \textit{set} & = & \lambda \textit{self}.\, \lambda \textit{arg}. \langle \textit{self} \leftarrow \textit{get} = \lambda s.\, \lambda a.\, \textit{arg}\ \rangle; \\ & & & & \qquad\qquad\quad \langle \textit{self} \leftarrow \textit{set} = \textit{when}(\lambda s.\, \textit{false})\ \ \lambda s.\, \lambda a.\, \textit{nil}\ \rangle; \\ & & & & \qquad\qquad\quad \textit{nil}\quad \rangle \end{array}$$

The above code describes a concurrent object with two methods: *get* and *set*. Formally a method definition has two parts: a guard of the form $\textit{when}(\lambda s.\, e_g)$ and a body $\lambda s.\, e_b$, with e_b of function type. In both cases the variable s is called the *self* variable since it is bound to the object executing the method. We omit the guard if it is the constant *true*. When a method is invoked, first the guard is evaluated, then, if it returns true, the body is executed. The *get* method represents the storage of the future variable. Its guard ($\textit{when}(\lambda \textit{self}.\, \textit{false})$) insures that when the object is created the value contained in the future variable cannot be read. Constant c is an initial value of the proper type. The *set* method is immediately available when the future variable is created. In fact its guard (which is omitted for convenience) is constantly *true*. When invoked it stores the received value *arg* by modifying the *get* method and modifies itself so that it can no longer be invoked. The new *get* method is now available because its (omitted) guard is true, and simply returns the stored value *arg*. Expressions such as $\langle \textit{self} \leftarrow \textit{get} = \lambda s.\, \lambda a.\, \textit{arg}\ \rangle$ are method override expressions. An override executed by an object on itself (called self-inflicted) is a method update and is the mechanism for state change.

As an example of the use of eager invocation, let us consider a program fragment in which we read values from a buffer via a future variable.

$$\begin{array}{ll} \mathit{let} & f = \mathit{future\text{-}var},\ b = \mathit{buffer} \\ \mathit{in} & b \Leftarrow_a \mathit{read}(f); \ldots \mathit{code} \ldots \mathit{let}\ x = f \Leftarrow_s \mathit{get}\ \mathit{in}\ \ldots \end{array}$$

where

$$\begin{array}{lllll} \mathit{buffer} = & \langle & \mathit{read} & = & \mathit{when}(\lambda \mathit{self}. \ldots)\, \lambda \mathit{self}.\, \lambda a. \ldots a \Leftarrow_a \mathit{set}(v) \ldots, \\ & & \ldots & & \rangle \end{array}$$

The process which executes this program fragment first invokes the buffer's *read* method with the future variable f as an argument. This invocation is asynchronous ($\Leftarrow_a$), so the *code* part of the program can be executed concurrently with the *read* operation in the buffer object. When the program needs the value from the buffer, it accesses the future variable f by synchronously invoking ($\Leftarrow_s$) f's *get* method. If the buffer has not yet stored the result (v) in the future variable, then (and only then) the program blocks to wait for it.

Control flow analysis. To provide some intuition about the program analysis we are interested in, let us consider the following fragment of code. For brevity, we omit guards.

$$\begin{array}{lrl} \mathit{let} & o_1 = & \langle m_1 = \lambda s_1.\, e_1,\ m_2 = \lambda s_2.\, \lambda a_2.\, s_2 \Leftarrow_a m_1(\_)\rangle, \\ & o_2 = & \langle o_1 \leftarrow m_2 = \lambda s_3.\, e_3\rangle, \\ & f = & \lambda x.\, x \Leftarrow_a m_2(\_) \\ \mathit{in} & f\, o_1; & \end{array}$$

Our goal is to approximate the values that variables may assume at run-time. With this aim, set-based analysis produces two finite tables: a *set environment* $\mathcal{H}$ and a *set object environment* $\mathcal{A}$.

A *set environment* maps each program variable x to a set of abstract values, $\mathcal{H}(x)$, which approximates the values that that program variable may assume. In the example, variable o_1 assumes as values the addresses of the objects created at site o_1. We statically approximate these object addresses by the variable o_1 itself, returning $\mathcal{H}(o_1) = \{o_1\}$. At the site denoted by o_2, we have a method override operation. The semantics of method override depends on whether the operation is *self-inflicted* or not. We say that an operation is self-inflicted if the target object is the same as the object that invokes the operation. If $\langle o_1 \leftarrow m_2 = \lambda s_3.\, e_3\rangle$ is self-inflicted, we replace o_1's m_2-method with the new one and return the modified object. If it is not self-inflicted, we return a new object obtained by overriding o_1's m_2-method. In general, we do not know statically if the operation is self-inflicted or not; consequently, the values that o_2 may assume are the addresses of objects created at sites o_1 and o_2. Thus our analysis returns $\mathcal{H}(o_2) = \{o_1, o_2\}$. Finally, variable x gets the values of f's actual parameter o_1; hence, $\mathcal{H}(x) = \mathcal{H}(o_1) = \{o_1\}$.

A *set object environment* maps each program variable that denotes a possible object creation site (*e.g.* o_1, o_2) to a set of abstract method tables (denoted by $\mathcal{A}(o_1)$, $\mathcal{A}(o_2)$). Each such set approximates the method tables that objects created at that site can have at run-time. In the example, objects created at site o_1 may have abstract method tables $\{m_1 = \lambda s_1.\, e_1, m_2 = \lambda s_2.\, \lambda a_2.\, s_2 \Leftarrow_a m_1(\_)\}$ and $\{m_1 = \lambda s_1.\, e_1, m_2 = \lambda s_3.\, e_3\}$ (the second one if the method override in line 2 is self-inflicted). Objects created at site o_2 may have abstract method table $\{m_1 = \lambda s_1.\, e_1, m_2 = \lambda s_3.\, e_3\}$ (if the method override is non-self-inflicted).

This analysis allows us to produce a control flow graph of the program. In fact we know the code possibly invoked at any function or method call site. For example, we know that the method tables of the objects that x may refer to are the ones associated with the objects created at site o_1. Thus the bodies possibly invoked by $x \Leftarrow_a m_2(\_)$ are $\lambda s_2.\, \lambda a_2.\, s_2 \Leftarrow_a m_1(\_)$ and $\lambda s_3.\, e_3$.

3 THE LANGUAGE AND ITS A-NORMAL FORM

3.1 Language Syntax

The syntax of our language is given by the following grammar.

Expressions:

$$
\begin{array}{rl}
e \in \mathit{Exp} \ ::= & x \mid c \mid \lambda x.\, e \mid e_1 e_2 \\
& \mid \langle\rangle \mid \langle e_1 \longleftarrow\!\!\!+ \ m = \mathit{when}(e_2)e_3\rangle \mid \langle e_1 \leftarrow m = \mathit{when}(e_2)e_3\rangle \\
& \mid e_1 \Leftarrow_a m(e_2) \mid e_1 \Leftarrow_s m(e_2)
\end{array}
$$

In the definition of *Exp*, x is a variable, c is a constant symbol, $\lambda x.\, e$ is a lambda abstraction, and $e_1 e_2$ is function application. The remaining syntactic forms are the object primitives. Expression $\langle\rangle$ creates an empty object while $\langle e_1 \longleftarrow\!\!\!+ \ m = \mathit{when}(e_2)e_3\rangle$ returns a new object obtained by extending e_1 with a new method m having guard e_2 and body e_3. If $\langle e_1 \leftarrow m = \mathit{when}(e_2)e_3\rangle$ is self-inflicted, we replace e_1's m-method with the new one. If it is not self-inflicted, we return a new object obtained by overriding e_1's m-method. Note that non-self-inflicted extension and override operations allow us to support width and depth inheritance respectively. Expression $e_1 \Leftarrow_a m(e_2)$ sends message m asynchronously with argument e_2 to e_1. The corresponding synchronous invocation is $e_1 \Leftarrow_s m(e_2)$. λ is a binding construct and *let* abbreviates lambda application as usual. An operational semantics for our language is given in (Di Blasio *et al.* 1996).

In the rest of the paper we use the following meta-notations: $\Leftarrow$ stands for both $\Leftarrow_a$ and $\Leftarrow_s$, and $\leftarrow\!\circ$ stands for both $\longleftarrow\!\!\!+$ and $\leftarrow$. We also write $\langle m_1 = \mathit{when}(e_1)e'_1, \ldots, m_k = \mathit{when}(e_k)e'_k\rangle$ for $\langle \ldots \langle\langle\rangle \longleftarrow\!\!\!+ \ m_1 = \mathit{when}(e_1)e'_1\rangle \ldots \longleftarrow\!\!\!+ \ m_k = \mathit{when}(e_k)e'_k\rangle$.

3.2 A-Normal Form

Performing program analysis on source code can be a complex task, particularly in a calculus like ours in which functions and objects are first-class data and expressions can be arbitrarily nested. Consider, for example, the following code fragment.

$$let\ f = h\,y\ in \quad let\ x = ((f\,g) \Leftarrow_s m(v)) \Leftarrow_s n(w)\ in\ e \qquad (*)$$

To gather information about the values that variable x may assume, we first have to locate the subexpressions in the expression defining x and give each of them a label: l_1 for $f\,g$ and l_2 for $l_1 \Leftarrow_s m(v)$. Then we associate the labels with intermediate values. In other words, we associate l_1 with the values resulting from the function application $(f\,g)$ and l_2 with the values resulting from the invocation of m. More generally, we need a mechanism for referring to program points to talk about the corresponding intermediate values. One approach is to modify the syntax by labelling each subexpression (cf. (Nielson *et al.* 1997)). The approach we take is to transform programs to equivalent programs in A-normal form (Flanagan *et al.* 1995, Flanagan *et al.* 1993). In this transformation, a variable is introduced for each subexpression while preserving the execution order. For example, an A-normal form of the code fragment $(*)$ is

$$\begin{array}{l} let\ f = h\,y\ in \\ \quad let\ l_1 = f\,g\ in \\ \qquad let\ l_2 = l_1 \Leftarrow_s m(v)\ in \\ \qquad\quad let\ x = l_2 \Leftarrow_s n(w)\ in\ \ M \qquad \text{where } M \text{ is an A-normal-form of } e \end{array}$$

The syntax of the A-normal-form (ANF) expressions for our language is given by the following grammar.

ANF Expressions:

$$\begin{array}{rcl} M, N \in \Lambda_a & ::= & s \mid let\ x = r\ in\ M \mid let\ x = N\ in\ M \\ r & ::= & c \mid \lambda y.N \mid s\,s' \mid s \Leftarrow m(s') \mid \langle\rangle \mid \langle s \leftarrow\!\!\circ\ m = when(s')s'' \rangle \\ s \in Sval & ::= & x \mid l \end{array}$$

Intuitively, an ANF expression can be either a simple value s or a *let* expression. Simple values are either variables x or locations l. We introduce locations into the syntax so that we may express computation syntactically. There exists a simple procedure, that we omit here for space considerations, that translates an expression $e \in Exp$ to its corresponding ANF (Di Blasio 1997). Intuitively, an expression e, which can be uniquely decomposed as $C[e_{rdx}]$ for some context C and redex e_{rdx}, is equivalent to the expression $let\,x = e_{rdx}\ in\,M$, where M is the ANF of $C[x]$.

In the ANF syntax we introduce the more general form $let\ x = N\ in\ M$ to guarantee that terms resulting from the reduction of function applications ($let\ x = ss'\ in\ M$) will be in ANF (see Section 4.1). $FV[M]$ denotes the set of *free variables* in M. A term M is *closed* if it contains no free variables and *static* if it contains no locations. Finally, *programs*, which we will denote by meta-variable P, are closed static expressions.

Throughout the rest of the paper, we assume that all variables in an ANF term bound by either λ or *let* are distinct. We will use meta-variables L, M, N, and P to denote ANF terms arising via translation from programs typeable in the source language using the type inference system of (Di Blasio *et al.* 1996).

4 OPERATIONAL SEMANTICS

We formalize the operational semantics of ANF terms as a transition relation on configurations in the same style as for the original language (Di Blasio *et al.* 1996). A *configuration* $\langle\!\langle\ H \mid \alpha \mid \mu\ \rangle\!\rangle$ (meta-variable g ranges over configurations) consists of a *global heap* H in which the values associated with variables are allocated, an *object soup* α, containing all created objects, and a collection of *pending asynchronous messages* μ.

A heap H is a finite map from locations l to heap values h. Heap values include constants, c, functions, $\lambda x.\, M$, and object addresses, a. The set of locations *Loc* is partitioned into infinite subsets of subscripted locations Loc_x, one partition for each variable x. Every value assigned to x is allocated at a new location l_x taken from the set Loc_x.

An *object* is represented at run-time as a triple $(a, \eta, [S])$, where a is the object's *address*, η is its *method table*, and S its *state*. Similarly to the set of locations, we partition the set of object addresses *ObjAddr* into infinite subsets $ObjAddr_x$, one for each variable x. We will see the reason for this partitioning in Section 5. Method tables are finite functions from method names to pairs of locations. These locations store the guard and method body functions, respectively. The state S can be either idle ($[I]$) or busy ($[M]$), in which case the expression M represents the remaining computation. An object passes from the idle to a busy state in response to either a synchronous or an asynchronous method invocation. At the end of the resulting computation, it returns to the idle state.

Definition 1 (Initial Configurations) *The initial configuration corresponding to a program, P, is $g_P = \langle\!\langle\ H_0 \mid \alpha_0 \mid \mu_0\ \rangle\!\rangle$ where H_0 is empty, $\alpha_0 = (main, \eta_\emptyset, [P])$, where $\eta_\emptyset$ is the empty method table, and μ_0 is an empty collection of messages.*

The following grammar defines ANF redexes (M_{rdx}) and reduction contexts (E). We have the unique decomposition property – each ANF expression is either a simple value or decomposes uniquely in the form $E[M_{rdx}]$.

ANF Redexes (M_{rdx}) and **Reduction contexts** (E):

$$\begin{array}{rcl} M_{rdx} & ::= & let\ x = s\ in\ M \mid let\ x = r\ in\ M \\ E & ::= & [\,] \mid let\ x = E\ in\ M \end{array}$$

4.1 Reduction Rules

We describe in detail the transition rules for some expressions to illustrate the key concepts of the transition system. The complete set of rules can be found in (Di Blasio 1997).

Function application (*apply*). The heap value at location l_y is a function. The transition stores the value of the actual parameter $H(l_z)$ at a new location

l_w in the heap and replaces all occurrences of w within N by l_w. The function new_w takes a heap and returns a new location from Loc_w.

$$\left\langle\!\!\left\langle\; H \mid \alpha, (a, \eta, [E[let\ x = l_y l_z\ \ in\ M]]) \mid \mu \;\right\rangle\!\!\right\rangle \longmapsto$$
$$\left\langle\!\!\left\langle\; H \cup \{l_w = H(l_z)\} \mid \alpha, (a, \eta, [E[let\ x = N[w \leftarrow l_w]\ \ in\ M]]) \mid \mu \;\right\rangle\!\!\right\rangle$$

where $H(l_y) = \lambda w. N$, and $l_w{=}new_w(H)$.

Empty object creation ($\langle\rangle$). A new object with empty method table and idle state is created. Its address is subscripted by the name of the variable x that marks its creation site. The function $new\text{-}a_x$ takes an object soup and return a fresh object address from $ObjAddr_x$. The value a_x is stored at a new location l_x in the heap.

$$\left\langle\!\!\left\langle\; H \mid \alpha, (a, \eta, [E[let\ x = \langle\rangle\ \ in\ M]]) \mid \mu \;\right\rangle\!\!\right\rangle \longmapsto$$
$$\left\langle\!\!\left\langle\; H \cup \{l_x = a_x\} \mid \alpha, (a, \eta, [E[M[x \leftarrow l_x]]]), (a_x, \eta_\emptyset, [I]) \mid \mu \;\right\rangle\!\!\right\rangle$$

where $l_x{=}new_x(H)$, $a_x = new\text{-}a_x(\alpha, (a, \eta, \ldots))$.

Self-inflicted, synchronous method invocation ($\Leftarrow_s$ *-self*). Since method invocation involves a double application, we might naively reduce the expression $let\ x = l_y \Leftarrow_s m(l_z)\ \ in\ M$ to $let\ x = l_b\, l_y\, l_z\ \ in\ M$. Unfortunately, this expression is not in A-normal form. Thus we need to normalize it, a process which produces expression M_1. The normalization is achieved by introducing two fresh variables n_2 and n_3.

$$\left\langle\!\!\left\langle\; H \mid \alpha, (a, \eta, [E[let\ x = l_y \Leftarrow_s m(l_z)\ \ in\ M]]) \mid \mu \;\right\rangle\!\!\right\rangle \longmapsto$$
$$\left\langle\!\!\left\langle\; H \mid \alpha, (a, \eta, [E[M_1]]) \mid \mu \;\right\rangle\!\!\right\rangle$$

where $H(l_y) = a$, $\eta(m) = (l_g, l_b)$ and
$M_1 \equiv let\ n_2 = l_b l_y\ \ in \quad let\ n_3 = n_2\, l_z\ \ in \qquad let\ x = n_3\ \ in\ M$

5 SET-BASED ANALYSIS

Set-based analysis (Heintze 1992, Flanagan *et al.* 1995) is a program analysis technique which provides static information about the values that a program variable may assume at run-time. Since calculating this information exactly is undecidable, set-based analysis computes for each program variable x a finite set of 'abstract values', which collectively represent an upper bound on the set of values that x may assume at run-time.

In this section, we adapt and extend the set-based analysis of (Flanagan *et al.* 1995) for Scheme to our calculus. The fact that our calculus permits shared mutable objects requires an elaboration of their analysis. In particular, although Flanagan and Felleisen allow assignment to variables, they do not treat reference cells as first-class values. Hence, they do not address imperative features such as memory, object addresses, and the alias problem. To remedy this deficiency, our analysis produces two finite approximations instead of just

Run-time

$$
\mathrm{H}: \begin{array}{c} \cdots \\ \hline (l_x,\, h) \\ \hline (l'_x,\, h') \\ \hline \cdots \end{array}
\qquad
\alpha: \begin{array}{c} \cdots \\ \hline (a_y,\, \eta,\, S) \\ \hline (a'_y,\, \eta',\, S') \\ \hline \cdots \end{array}
\qquad
\eta: \begin{array}{c} \cdots \\ \hline (m,\, l_g,\, l_b) \\ \hline \cdots \end{array}
$$

$h \in \mathit{Hval} \; ::= \; c \mid \lambda x.\, M \mid a_y$ (Heap values)

Abstraction

$$
\mathcal{H}: \begin{array}{c} \cdots \\ \hline (x,\, \{\ldots, \hat{h}, \hat{h'}, \ldots\}) \\ \hline \cdots \end{array}
\qquad
\mathcal{A}: \begin{array}{c} \cdots \\ \hline (y,\, \{\ldots, \hat{\eta}, \hat{\eta}' \ldots\}) \\ \hline \cdots \end{array}
\qquad
\hat{\eta}: \begin{array}{c} \cdots \\ \hline (m,\, g,\, b) \\ \hline \cdots \end{array}
$$

$\hat{v}$	$\in$	$\widehat{\mathit{Val}}$	$::=$	$\hat{h} \mid \hat{\eta}$	(Ab. values)
$\hat{h}$	$\in$	$\widehat{\mathit{Hval}}$	$::=$	$c \mid \lambda x.\, M \mid y$	(Ab. heap values)
$\hat{\eta}$	$\in$	$\widehat{\mathit{MeTable}}$	$::=$	$\{m_1 = (g_1, b_1), \ldots, m_n = (g_n, b_n)\}$	(Ab. method tables)

Figure 1 Static approximations of run-time values

one: a *set environment* $\mathcal{H}$ and a *set object environment* $\mathcal{A}$ (see Figure 1). *Set environment* $\mathcal{H}$ maps each program variable to a set of abstract heap values. This set approximates the values that program variables may assume during an execution. $\mathcal{H}$ is an abstraction of the global heap H: locations subscripted by x (*e.g.* l_x, l'_x) are statically represented by variable x, while heap values (*e.g.* h, h') are represented by abstract heap values (*e.g.* $\hat{h}$, $\hat{h'}$), defined in Figure 1. Environment $\mathcal{H}$ is a valid approximation of heap H if every value h that H associates with a location l_x may be obtained from an abstract value ($\hat{h}$) in $\mathcal{H}(x)$ by substituting appropriate locations for free variables in $\hat{h}$.

Set object environment $\mathcal{A}$ maps each program variable that potentially denotes an object creation site (*i.e.*, variables that label empty object, method override, and extension expressions) to a set of abstract method tables. Intuitively, the set of abstract method tables for a given object creation site approximates the method tables that objects created at that site may have at run-time. An object environment $\mathcal{A}$ is an abstraction of an object soup α: addresses of objects created at site y (*e.g.* a_y, a'_y) are represented by variable y, while method tables (*e.g.* η, η') are represented by abstract method tables (*e.g.* $\hat{\eta}$, $\hat{\eta}'$), which are defined in Figure 1. $\mathcal{A}$ is a valid approximation of α if every method table η that objects created at site y assume at run-time is obtained from an abstract method table in $\mathcal{A}(y)$ by substituting appropriate locations for free variables in the abstract method table. Method tables are approximated by abstract method tables in which the locations containing the guard and method body functions are represented by the corresponding abstract locations, *e.g.*, g for l_g and b for l_b .

To formalize our analysis, we need to introduce a bit more notation. Given a program P, let Var_P be the set of variables occurring in P. Let Var_O be the set of variables labelling possible object creation points occurring in P. Let $\widehat{Hval}_P$ be the set of abstract heap values of P. Let $\mathcal{P}_{fin}$ be the finite power-set constructor, and let $\alpha_{mt}(a)$ denote the method table of object a in α.

We now define the environments $\mathcal{H}$, $\mathcal{A}$ for P and the relation $P \models \mathcal{H}, \mathcal{A}$, which establishes the conditions under which $\mathcal{H}, \mathcal{A}$ are valid for P. In particular, the relation $P \models \mathcal{H}, \mathcal{A}$ holds if $\mathcal{H}, \mathcal{A}$ are valid approximations of every heap, object soup combination H, α produced during any execution of P.

Definition 2 *Let P be a program.*

- *A set environment for P is a mapping $\mathcal{H}: \; Var_P \rightarrow \mathcal{P}_{fin}(\widehat{Hval}_P)$.*
- *A set object environment for P is a mapping $\mathcal{A}: \; Var_O \rightarrow \mathcal{P}_{fin}(\widehat{MeTable})$.*
- *$P \models \mathcal{H}, \mathcal{A}$ holds if $g_P \longmapsto^* \left\langle\!\!\left\langle\, H \mid \alpha \mid \mu \,\right\rangle\!\!\right\rangle$ implies $H, \alpha \models \mathcal{H}, \mathcal{A}$.*
- *$H, \alpha \models \mathcal{H}, \mathcal{A}$ holds if for all bindings $(l_x = h) \in H$*

$$\begin{cases} z \in \mathcal{H}(x) \text{ and } \alpha_{mt}(a_z) \in Cl(\mathcal{A}(z)) & \text{if } h = a_z \\ h \in Cl(\mathcal{H}(x)) & \text{otherwise} \end{cases}$$

where

$$Cl(\hat{v}) = \{\hat{v}[x_1 \leftarrow l_{x_1}, \ldots, x_n \leftarrow l_{x_n}] \mid FV[\hat{v}] = \{x_1, \ldots, x_n\}, \text{ and } l_{x_i} \in Loc_{x_i}\}$$

5.1 Set Constraints

Having defined what we mean by valid environments for a program P, we need to be able to compute such environments. To that end, we generate from P a collection of set constraints such that any $\mathcal{H}$ and $\mathcal{A}$ satisfying these constraints are valid for P. Properties of the constraints for P guarantee they have a least solution. A proof of the existence of a least solution and an algorithm that computes it can be found in (Di Blasio 1997).

Given a program P, a set constraint is composed of a premise A_P which concerns the environments $\mathcal{H}, \mathcal{A}$ and the program P, and a conclusion B which concerns only the environments $\mathcal{H}, \mathcal{A}$. A pair of environments $\mathcal{H}, \mathcal{A}$ *satisfies* such a set constraint relative to P if whenever A_P holds for $\mathcal{H}, \mathcal{A}$ and P, then B also holds for $\mathcal{H}, \mathcal{A}$.

Before proceeding with the constraints, we first introduce some notation. We write $M \in P$, to indicate that M is an ANF term that occurs in P. Given an abstract method table $\hat{\eta}$, $\hat{\eta}' = \hat{\eta}[m = (w, z)]$ is equal to $\hat{\eta}$ except for the entry m; in this case, $\hat{\eta}'(m) = (w, z)$ (see C^P_{ov}, C^P_{ext} constraints in Figure 2). The empty abstract method table is denoted by $\hat{\eta}_\emptyset$. Function *FinalVar*, defined below, computes for any ANF term M the variable whose value will be the result of reducing M.

$$C^P_{const} \quad \frac{(\mathit{let}\ x = c\ \mathit{in}\ M) \in P}{c \in \mathcal{H}(x)} \qquad C^P_{lamb} \quad \frac{(\mathit{let}\ x = \lambda y.\, N\ \mathit{in}\ M) \in P}{\lambda y.\, N \in \mathcal{H}(x)}$$

$$C^P_{emp-obj} \quad \frac{(\mathit{let}\ x = \langle\rangle\ \mathit{in}\ M) \in P}{x \in \mathcal{H}(x) \qquad \hat{\eta}_\emptyset \in \mathcal{A}(x)} \qquad C^P_{var} \quad \frac{(\mathit{let}\ x = N\ \mathit{in}\ M) \in P}{\mathcal{H}(\mathit{FinalVar}[N]) \subseteq \mathcal{H}(x)}$$

$$C^P_{apply} \quad \frac{(\mathit{let}\ x = y\ z\ \mathit{in}\ M) \in P \qquad \lambda w.\, N \in \mathcal{H}(y)}{\mathcal{H}(z) \subseteq \mathcal{H}(w) \qquad \mathcal{H}(\mathit{FinalVar}[N]) \subseteq \mathcal{H}(x)}$$

$$C^P_{ext} \quad \frac{(\mathit{let}\ x = \langle y \longleftarrow\!\!\!\!+\ m = \mathit{when}(w)z\rangle\ \mathit{in}\ M) \in P \qquad y' \in \mathcal{H}(y) \qquad \hat{\eta} \in \mathcal{A}(y')}{x \in \mathcal{H}(x) \qquad \hat{\eta}' \in \mathcal{A}(x)}$$

$$C^P_{ov} \quad \frac{(\mathit{let}\ x = \langle y \leftarrow m = \mathit{when}(w)z\rangle\ \mathit{in}\ M) \in P \qquad y' \in \mathcal{H}(y) \qquad \hat{\eta} \in \mathcal{A}(y')}{x \in \mathcal{H}(x) \qquad \hat{\eta}' \in \mathcal{A}(x) \qquad y' \in \mathcal{H}(x) \qquad \hat{\eta}' \in \mathcal{A}(y')}$$

$$C^P_{s-inv} \quad \frac{\begin{array}{c}(\mathit{let}\ x = y \Leftarrow_s m(z)\ \mathit{in}\ M) \in P \\ y' \in \mathcal{H}(y) \qquad \hat{\eta} \in \mathcal{A}(y') \qquad \hat{\eta}(m) = (g, b) \\ \lambda s_1.\, N_1 \in \mathcal{H}(g) \qquad \lambda s_2.\, N_2 \in \mathcal{H}(b) \qquad \lambda w.\, N_3 \in \mathcal{H}(\mathit{FinalVar}[N_2])\end{array}}{\mathcal{H}(y) \subseteq \mathcal{H}(s_1) \quad \mathcal{H}(y) \subseteq \mathcal{H}(s_2) \quad \mathcal{H}(z) \subseteq \mathcal{H}(w) \quad \mathcal{H}(\mathit{FinalVar}[N_3]) \subseteq \mathcal{H}(x)}$$

$$C^P_{as-inv} \quad \frac{\begin{array}{c}(\mathit{let}\ x = y \Leftarrow_a m(z)\ \mathit{in}\ M) \in P \\ y' \in \mathcal{H}(y) \qquad \hat{\eta} \in \mathcal{A}(y') \qquad \hat{\eta}(m) = (g, b) \\ \lambda s_1.\, N_1 \in \mathcal{H}(g) \qquad \lambda s_2.\, N_2 \in \mathcal{H}(b) \qquad \lambda w.\, N_3 \in \mathcal{H}(\mathit{FinalVar}[N_2])\end{array}}{\mathcal{H}(y) \subseteq \mathcal{H}(s_1) \qquad \mathcal{H}(y) \subseteq \mathcal{H}(s_2) \qquad \mathcal{H}(z) \subseteq \mathcal{H}(w) \qquad \mathit{nil} \in \mathcal{H}(x)}$$

Figure 2 Set constraints for A-normal-form expressions

Definition 3 *The function FinalVar on ANF terms is defined as follows:*

$$\begin{array}{rcl} \mathit{FinalVar}[x] = \mathit{FinalVar}[l_x] & = & x \\ \mathit{FinalVar}[\mathit{let}\ x = \ldots\ \mathit{in}\ M] & = & \mathit{FinalVar}[M] \end{array}$$

Figure 2 shows the set constraints for our language. In the following, we give the intuition behind the soundness of some of the constraints.

- ($C^P_{emp-obj}$) When we execute *let* $x = \langle\rangle$ *in* M, the value associated with x is a new object address a_x, and the method table of the created object is empty. Thus we add x to $\mathcal{H}(x)$ and $\hat{\eta}_\emptyset$ to $\mathcal{A}(x)$.

- (C^P_{var}) When we execute $let\ x = N\ in\ M$, the value associated with x is the value of the return variable of N. Thus we add the constraint $\mathcal{H}(\mathit{FinalVar}[N]) \subseteq \mathcal{H}(x)$.
- (C^P_{apply}) The two constraints on the value sets in the conclusion represent respectively the flow of values from the actual to the formal parameter of the function and from the variable containing the result of the function, $\mathit{FinalVar}[N]$, to the variable x.
- $(C^P_{ext,ov})$ When we execute $let\ x = \langle l_y \leftarrow\!\!\circ\ m = when(l_w)l_z\rangle\ in\ M$, where $\leftarrow\!\!\circ$ is either $\longleftarrow\!\!+$ or the non-self-inflicted version of $\leftarrow$, the value associated with x is a new address a_x; thus we add x to $\mathcal{H}(x)$. The method table of a_x is obtained by adding $m = (l_w, l_z)$ to the method table η of object $a_{y'}$, where $a_{y'} = H(l_y)$. Thus we add $\hat{\eta}' = \hat{\eta}[m = (w, z)]$ to $\mathcal{A}(x)$. In C^P_{ov}, we must further consider the case that the operation is self-inflicted. Here, the value associated with x is $H(l_y) = a_{y'}$. Thus we add y' to $\mathcal{H}(x)$. Since $a_{y'}$ has a new method table obtained by overriding method m in η with the new pair (l_w, l_z), we add $\hat{\eta}' = \hat{\eta}[m = (w, z)]$ to $\mathcal{A}(y')$.
- $(C^P_{s,as-inv})$ The execution of a synchronous method invocation may result in the application of a guard and a body. The constraint $\mathcal{H}(y) \subseteq \mathcal{H}(s_1)$ represents the flow of values from the actual to the formal parameter in the guard function. The method body application is a generalization of the function application case. It consists of a double application. Thus the second and third constraint in the conclusion represent the flow of values during input, while the fourth constraint represents the return flow. For the asynchronous method invocation case the value *nil* is returned immediately. Thus we add *nil* to $\mathcal{H}(x)$.

5.2 Examples

Inlining. To show how set-based analysis works and its possible applications, we consider the following ANF of the program described in Section 2.

$$
\begin{array}{l}
let\ o_3 = \langle\rangle\ in \quad let\ l_1 = \lambda s_1.\, N_1\ in \\
\quad let\ o_4 = \langle o_3 \longleftarrow\!\!+\ m_1 = l_1\rangle\ in \quad let\ l_2 = \lambda s_2.\, N_2\ in \\
\quad\quad let\ o_1 = \langle o_4 \longleftarrow\!\!+\ m_2 = l_2\rangle\ in \quad let\ l_3 = \lambda s_3.\, N_3\ in \\
\quad\quad\quad let\ o_2 = \langle o_1 \leftarrow m_2 = l_3\rangle\ in \quad let\ f = \lambda x.\, N_4\ in \\
\quad\quad\quad\quad let\ x_1 = f\ o_1\ in\ x_1
\end{array}
$$

where $N_2 \equiv let\ f_2 = \lambda a_2.\, N_2'\ in\ f_2$, $N_2' \equiv let\ i_1 = s_2 \Leftarrow_a m_1(\_)\ in\ i_1$, and $N_4 \equiv let\ i_2 = x \Leftarrow_a m_2(\_)\ in\ i_2$ and N_1, N_3 are ANFs of e_1, e_3 respectively.

We apply the set constraints in Section 5.1 to the program in ANF. Some of the resulting constraints, one for each type of expression in the example, are showed in Figure 3. We annotate each constraint with the name of the set constraint that produced it and with the variable denoting the expression to which the constraint was applied. We assume that the unspecified code

Set Constraints

$C^P_{emp-obj}$ on o_3:	$o_3 \in \mathcal{H}(o_3)$, $\eta_\emptyset \in \mathcal{A}(o_3)$
C^P_{lamb} on l_1:	$\lambda s_1.\, N_1 \in \mathcal{H}(l_1)$
C^P_{ext} on o_4:	$o_4 \in \mathcal{H}(o_4)$, and for all $y' \in \mathcal{H}(o_3)$, $\hat{\eta} \in \mathcal{A}(y')$, $\hat{\eta}[m_1 = (\_, l_1)] \in \mathcal{A}(o_4)$
C^P_{as-inv} on i_1:	for all $y' \in \mathcal{H}(s_2)$ and $\hat{\eta} \in \mathcal{A}(y')$, with $\hat{\eta}(m_1) = (\_, b)$ and $\lambda s.\, N \in \mathcal{H}(b)$, we have $\mathcal{H}(s_2) \subseteq \mathcal{H}(s)$ and $nil \in \mathcal{H}(i_1)$
C^P_{ov} on o_2:	$o_2 \in \mathcal{H}(o_2)$, and for all $y' \in \mathcal{H}(o_1)$ and $\hat{\eta} \in \mathcal{A}(y')$, we have $y' \in \mathcal{H}(o_2)$, $\hat{\eta}[m_2 = (\_, l_3)] \in \mathcal{A}(o_2)$, and $\hat{\eta}[m_2 = (\_, l_3)] \in \mathcal{A}(y')$
C^P_{apply} on x_1:	for all $\lambda w.\, N \in \mathcal{H}(f)$, $\mathcal{H}(o_1) \subseteq \mathcal{H}(w)$, $\mathcal{H}(FinalVar[N]) \subseteq \mathcal{H}(x_1)$

Set environment

$\mathcal{H}(o_1) = \{o_1\}$	$\mathcal{H}(o_2) = \{o_1, o_2\}$	$\mathcal{H}(o_3) = \{o_3\}$	$\mathcal{H}(o_4) = \{o_4\}$	$\mathcal{H}(x) = \{o_1\}$
$\mathcal{H}(l_1) = \{\lambda s_1.\, N_1\}$	$\mathcal{H}(f) = \{\lambda x.\, N_4\}$	$\mathcal{H}(s_1) = \{o_1\}$	$\mathcal{H}(s_2) = \{o_1\}$	$\mathcal{H}(s_3) = \{o_1\}$
$\mathcal{H}(l_2) = \{\lambda s_2.\, N_2\}$	$\mathcal{H}(f_2) = \{\lambda a_2.\, N'_2\}$	$\mathcal{H}(x_1) = \{nil\}$	$\mathcal{H}(i_1) = \{nil\}$	$\mathcal{H}(i_2) = \{nil\}$
$\mathcal{H}(l_3) = \{\lambda s_3.\, N_3\}$	$\mathcal{H}(a_2) = \{\_\}$			

Set object environment

$\mathcal{A}(o_1) = \{\hat{\eta}_2, \hat{\eta}_3\}$	$\mathcal{A}(o_2) = \{\hat{\eta}_3\}$	$\hat{\eta}_1(m_1) = (\_, l_1)$	$\hat{\eta}_2(m_1) = (\_, l_1)$	$\hat{\eta}_3(m_1) = (\_, l_1)$
$\mathcal{A}(o_3) = \{\hat{\eta}_\emptyset\}$	$\mathcal{A}(o_4) = \{\hat{\eta}_1\}$		$\hat{\eta}_2(m_2) = (\_, l_2)$	$\hat{\eta}_3(m_2) = (\_, l_3)$

Figure 3 Inlining Example

N_1, N_3 does not affect the analysis. Figure 3 contains also the least solution $\mathcal{H}$, $\mathcal{A}$ of the constraints.

This analysis reveals the code that may be invoked at any function or method call site. We might take advantage of this control flow information to inline code. Inlining is an optimization technique which replaces a function or a method call with the called body (Jagannathan *et al.* 1996). The main benefit of inlining is that it eliminates the cost caused by call overhead. Inlining is particularly important for languages which provide objects, as run-time method lookup is a significant source of program inefficiency.

Inlining code at a call site is sound if there is a unique function or method applicable at that site. For example, we cannot inline $x \Leftarrow_a m_2(\_)$ because there is no unique applicable body (see Section 2). Note however, that even if the body were unique, we could inline it only if the language were sequential; in a concurrent setting we could not preserve the semantics of the original code because we would then be inlining a non-self-inflicted method invocation, thus changing the process that executes the code. In contrast, inlining $s_2 \Leftarrow_a m_1(\_)$ in the body of method m_2 is safe because the body is unique and the operation is self-inflicted. This example shows that the benefit from set-based analysis can be substantially enhanced by an in-depth study of self and non-self-inflicted operations. (See (Di Blasio 1997) for a first step.)

Colocation of objects in a distributed setting. The following fragment of code describes an object o_1 which executes a particular task using a helper object o_2. The task starts invoking method *do* on o_1. Method *do* creates the helper object o_2, sets the value of the *helper* method, and starts a computation in o_2 in parallel with the one in o_1. When o_2 completes the execution of its *continue* method, it acknowledges o_1. Depending on some conditions (omitted in the code) method *ack* of o_1 can either invoke another round of the *continue* method on itself and o_2 or alternatively return a result to some object y.

$$
\begin{array}{llcl}
\mathit{let}\ o_1 = \langle & \mathit{do} & = & \lambda s_1.\,\lambda a.\,\mathit{let}\ o_2 = \mathit{helper\_ob} \\
 & & & \quad \mathit{in}\ \langle s_1 \leftarrow \mathit{helper} = \lambda s.\,\lambda a.\,o_2\rangle; \\
 & & & \quad\quad o_2 \Leftarrow_a \mathit{start}(s_1);\ s_1 \Leftarrow_s \mathit{continue}, \\
 & \mathit{helper} & = & \lambda s.\,\lambda a.\,c, \\
 & \mathit{ack} & = & \lambda s_2.\,\lambda a.\ldots \mathit{let}\ x = (s_2 \Leftarrow_s \mathit{helper})\ \mathit{in}\ x \Leftarrow_a \mathit{continue}\ldots \\
 & & & \quad \ldots s_2 \Leftarrow_s \mathit{continue}\ldots y \Leftarrow_a \mathit{return}(\_), \\
 & \mathit{continue} & = & \lambda s.\,\lambda a.\ldots \mathit{do\ the\ task}\ldots\ \rangle \\
\multicolumn{4}{l}{\mathit{in}\ o_1 \Leftarrow_a \mathit{do}(\_)}
\end{array}
$$

where

$$
\begin{array}{llcl}
\mathit{helper\_ob} = \langle & \mathit{start} & = & \lambda s_3.\,\lambda a.\langle s_3 \leftarrow \mathit{helped\_ob} = \lambda s.\,\lambda x.\,a\rangle; \\
 & & & \quad s_3 \Leftarrow_s \mathit{continue}, \\
 & \mathit{helped\_ob} & = & \lambda s.\,\lambda a.\,c', \\
 & \mathit{continue} & = & \lambda s_4.\,\lambda a.\ldots \mathit{do\ the\ task}\ldots \\
 & & & \quad \mathit{let}\ x' = (s_4 \Leftarrow_s \mathit{helped\_ob})\ \mathit{in}\ x' \Leftarrow_a \mathit{ack}(\_)\ \rangle
\end{array}
$$

with c, c' constants of the proper type.

If we analyze the code above using the technique described in the previous example, we get for the variables on which methods are invoked the following set environment:

$$
\begin{array}{llll}
\mathcal{H}(o_2) = \{o_2\} & \mathcal{H}(s_1) = \{o_1\} & \mathcal{H}(s_2) = \{o_1\} & \mathcal{H}(x) = \{c, o_2\} \\
\mathcal{H}(s_3) = \{o_2\} & \mathcal{H}(s_4) = \{c, o_2\} & \mathcal{H}(x') = \{c', o_1\} &
\end{array}
$$

The first line shows that all method invocations by objects created at the program site denoted by o_1, except $y \Leftarrow_a \mathit{return}(\_)$, refer either to themselves or to objects created at site o_2. We have the analogous situation for the methods invoked by o_2. In a distributed setting, we can take advantage of this information by locating objects created at program sites o_1 and o_2 in the same network site.

Introducing parallelism. Consider the following object:

$$o\ =\ \langle \ldots m = \lambda s.\,\lambda a.\ldots \mathit{let}\ x = o' \Leftarrow_s n(\_)\ \mathit{in}\ \ldots f(x)\ldots\ \rangle.$$

We can take advantage of the parallelism of our language by transforming the synchronous method invocation into an eager invocation following the example in Section 2. However, the transformation makes sense only if the invocation is always non-self-inflicted, *i.e.* the value of o' is always different from the value of the self parameter s. This condition is guaranteed if $\mathcal{H}(s) \cap \mathcal{H}(o') = \emptyset$.

5.3 Soundness of the Set Constraints

Given a program P and environments $\mathcal{H}, \mathcal{A}$ which satisfy the set constraints relative to P, proving the soundness of the set constraints means showing that for any configuration $g = \left\langle\!\!\left\langle\ H \mid \alpha \mid \mu\ \right\rangle\!\!\right\rangle$ to which the initial state for

P (g_P) may reduce, $\mathcal{H}, \mathcal{A}$ are sound approximations of H and α, in the sense of Definition 2. Following the outline of the soundness proof in (Flanagan *et al.* 1995), the proof is done by induction on the length of the reduction. To prove the inductive step we must first define an invariant, $g \models_P \mathcal{H}, \mathcal{A}$, for our configurations which contains the relation $H, \alpha \models \mathcal{H}, \mathcal{A}$ and is preserved by the reduction.

Definition 4 *Given a configuration* $g = \left\langle\!\!\left\langle\ H \mid \alpha \mid \mu\ \right\rangle\!\!\right\rangle$, $g \models_P \mathcal{H}, \mathcal{A}$ *holds if*

$$H, \alpha \models \mathcal{H}, \mathcal{A} \quad \textit{and} \quad \alpha \models_P^H \mathcal{H} \quad \textit{and} \quad \mu \models_P \mathcal{H}.$$

The above definition says that g is valid with respect to $\mathcal{H}, \mathcal{A}$, and P if each of its component is valid as well. An object soup α is valid for P and $\mathcal{H}$ ($\alpha \models_P^H \mathcal{H}$) if for every object in α with state $[M]$, M obeys the constraints on the set environments. A pending queue μ is valid for P ($\mu \models_P \mathcal{H}$) if every message in μ comes from an asynchronous method invocation in P. The formal specifications of these relations can be found in (Di Blasio 1997).

The following is the main Lemma we use to prove the induction step in the soundness theorem.

Lemma 1 *If* $g \models_P \mathcal{H}, \mathcal{A}$, $g \longmapsto g'$, *and* $\mathcal{H}, \mathcal{A}$ *satisfy the constraints relative to* P, *then* $g' \models_P \mathcal{H}, \mathcal{A}$.

Theorem 1 (Soundness of Constraints) *If* $\mathcal{H}, \mathcal{A}$ *satisfies the set constraints relative to* P, *then* $P \models \mathcal{H}, \mathcal{A}$.

6 CONCLUSIONS

In this paper, we have presented a set-based analysis technique for an imperative, concurrent object calculus and shown it to be sound. This analysis provides static information about the values that variables may assume; this information may be used to define a program control flow graph, which is the starting point for most other analysis. We showed some examples that demonstrate how this analysis may be applied to problems that arise in a concurrent setting.

This work is intended as a first step towards the development of formal methods for program optimization for concurrent object-oriented languages. However, more work needs to be done to establish the applicability and limitations of this technique. Our analysis corresponds to what is called 0-CFA (Shivers 1991), the least precise and least complex of a family of control flow analyses. One direction for future research is to improve the precision and efficiency of this analysis, for example, along the lines developed in (Nielson *et al.* 1997). Another potential focus for research is the problem of statically detecting (non-)self-inflicted operations. Such an analysis would attempt to determine when the target object of an operation is the same as the object executing the request. As the inlining example illustrates, such information may be used to avoid the cost of complex remote invocation protocols.

REFERENCES

Amadio, R. M. (1994) Translating core facile. *Tech. Rep.* ECRC-1994-3, European Computer-Industry Research Centre.

Colby, C. (1995) Analyzing the communication topology of concurrent programs. In Proc. of ACM *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 202–213.

Di Blasio, P. (1997) A calculus for concurrent objects: design and control flow analysis. *PhD thesis*, Università di Roma "La Sapienza".

Di Blasio, P. and Fisher, K. (1996) A calculus for concurrent objects. In Proc. of *CONCUR'96: Concurrency Theory*, vol. LNCS 1119, Springer, pp. 655–670.

Fisher, K., Honsell, F. and Mitchell, J. C. (1993) A lambda calculus of objects and method specialization. *Nordic J. Computing (*formerly *BIT)*.

Flanagan, C., Sabry, A., Dubra, B. and Felleisen, M. (1993) The essence of compiling with continuation. In Proc. of *PLDI*, pp. 237–247.

Flanagan, C. and Felleisen, M. (1995) Set-based analysis for full scheme and its use in soft-typing. *Tech. Rep.* TR95-253, Department of Computer Science, Rice University.

Heintze, N. (1992) Set based program analysis. *PhD thesis*, Carnegie-Mellon University.

Holzle, U. and Ungar, D. (1995) Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transaction on Programming Languages and Systems*, 18, 4, 355–400.

Jagannathan, S. and Wright, A. (1996) Flow-directed inlining. In Proc. of *PLDI'96*, pp. 193–205.

Kobayashi, N., Nakade, M. and Yonezawa, A. (1995) Static analysis of communication for asynchronous concurrent programming languages. In Proc. of *SAS'95*.

Magic, G. (1996) Telescript technology: Mobile agents. *White paper.*

Mason, I. A. and Talcott, C. L. (1997) A semantically sound actor translation. To appear in *ICALP'97*.

McGraw, G. and Felten, E. W. (1996) *Java Security.* Wiley Computer Publishing.

Nielson, H. and Nielson, F. (1994) Higher-order concurrent programs with finite communication topology. In Proc. of *POPL'94*.

Nielson, H. and Nielson, F. (1997) Infinitary control flow analysis: a collecting semantics for closure analysis. In Proc. of *POPL'97*.

Palsberg, J. and Swartzbach, M. (1991) Object-oriented type inference. In Proc. of *OOPSLA'91*, pp. 146–161.

Plevyak, J., Zhang, X. and Chien, A. (1995) Obtaining sequential efficiency for concurrent object-oriented languages. In Proc. of *POPL'95*.

Shivers, O. (1991) Control-flow analysis of higher-order languages or taming lambda. *PhD thesis*, Carnegie-Mellon University.