A Suite of Analysis Tools Based on a General Purpose Abstract Interpreter

Thomas Cheatham, Haiming Gao, Dan Stefanescu

[1] Harvard University***
[2] Software Options, Inc.
Cambridge, Mass. 02138

1 Introduction

This paper reports on one aspect of an ongoing project that is developing and experimenting with new compiling technology. The overall system is called the *Kernel Based Compiler System*, reflecting the fact that the representation of a program used in much of the processing is that of *kernel terms*, essentially the Lambda Calculus augmented with constants.

While the compiling technology is applicable to a wide spectrum of programming languages as well as a wide spectrum of target machine architectures, we expect High Performance FORTRAN (HPF) and its extensions to be programming languages of particular concern because of their anticipated importance to the High Performance Computing Community.

The compiler being developed is "unbundled" in the sense that it consists of several components, $C_1, \cdots, C_N$, and compilation consists of applying component C_1 to program text and applying C_j, for $j = 2, \cdots, N$ to the result computed by component C_{j-1}.

One of the issues that we are using the compiler system to study is the suitability of various new linguistic constructs for particular application areas and techniques for the efficient realization of these constructs on a variety of High Performance Computers. Thus, we want it to be as straightforward as possible to extend each C_j, to deal with the new constructs. Ideally, we want each component to have a firm mathematical basis so that, for example, we can prove appropriate correctness results.

No compiler for a non-trivial language or target is ever really *simple* in one sense because it is inevitably a large program. However, we have strived to make each component, C_j, simple in the sense that it offers general purpose mechanisms that can be specialized to a particular tasks, so that we can separate the concerns regarding the mechanisms and those regarding the particular specializations in order to achieve simplicity.

This paper is concerned with a suite of *Analysis* components of the system that annotate a program with static estimates of certain aspects of the behavior

*** Work supported by ARPA Contract Nr. F19628-92-C-0113. Authors current address: Aiken Computation Laboratory, Harvard University, 33 Oxford St, Cambrige, MA 02138. E-mail: cheatham, gao, dan@das.harvard.edu

of that program. The point of this annotation is to prepare for annotation-enabled transformations that produce a more efficient program. Our approach is to develop a general purpose *Abstract Interpreter* that does the control flow analysis that is required by all analysis tasks and constructs a data structure called the *behavior graph* that relates the behaviors of various terms – herein called the *flows* of those terms — in such a way that doing a wide variety of analysis tasks is quite straightforwardand inexpensive.

In the next section we discuss some related work. We then define the kernel terms that are used and present an overview of the abstract interpreter that is followed by several examples of particular analysis tasks. Following this, we discuss the details of the abstract interpreter and close with a discussion of the current status of the project, and some future plans.

2 Related Work

Our method falls into the category of data-flow analyses based on abstract interpretations (see [4]). A seminal work by Cousot in this area is [3] which sets a rigorous framework for such analyses for the case of flowchart languages. It is not straightforward to extend Cousot's work to languages that permit higher order functions where there is no explicit "flowchart" available, that is, to situations where one aspect of the abstract interpretation is to determine the control flow. Any solution to this problem has to deal with the issue of abstracting functions. Basically, there are two approaches in this area.

A function can be abstracted denotationally, i.e. its concrete graph has to be transformed into an abstract graph. In general, this approach may encounter problems due to the size of the abstract graph even if one considers only the program imposed inputs as in [4].

The other option is to use a syntactic based approximation which computes only an approximation of the abstract graph of a function, but it does it using some fixed resources. Typically, (see [11], [9], [6] and [5]) one uses resources proportional with the number of different entities in the program.

A more elaborate solution to this problem was given by Shivers (see [7] and [8]) who introduced a technique for computing approximations to the control-flow graph at compile time. His method defines abstract semantic functions for CPS (continuation passing style) Scheme which are then used to implement an instrumented interpreter which returns the result of the required analysis. Two analyses (0CFA and 1CFA) are used to illustrate the approach.

The function abstraction methodology is clarified and generalized in [10] which presents a theory for general flow analysis which approximates values (both basic and functional) computed within a functional program according to a syntactic notion of equivalence. The analysis is presented in an equational setting in the style of Cousots' original paper (see [3]).

While in the spirit of [10], the method presented in this paper uses a different, but equivalent, system of constraints chosen to expose optimization opportunities

which avoid the unnecessary fixpoint iterations inherent in Shivers approach and therefore its large running costs ([12]).

Finally, our work is similar to that reported in [13]. We differ by dealing with higher order functions and employing a call context mechanism that permits a finer grained analysis of behavioral differences that arise at different call sites.

3 Kernel Terms

The kernel terms that we consider are, essentially, terms in the Lambda Calculus that is augmented with constants (both data and function constants).

We assume there are disjoint sets $\mathcal{C}$, $\mathcal{N}$, and $\mathcal{F}$, where $\mathcal{C}$ is a set of data constants (like integers and reals), $\mathcal{N}$ is a set of parameter names, and $\mathcal{F}$ is a set of function constants, or *primitive functions*, like those for arithmetic operations on integers and reals or those for communicating data between different processors.

A kernel term, t, is:

- $t \sim c$ — a constant, $c \in \mathcal{C}$, or
- $t \sim p$ — a primitive, $p \in \mathcal{F}$, or
- $t \sim x$ — a parameter, $x \in \mathcal{N}$, or
- $t \sim \lambda x_1 \cdots x_k.B$ — an abstraction, where $k \geq 0$, the x_j are distinct parameter names, $x_j \in \mathcal{N}$, for $j = 1, \cdots, k$, and B is a term, or
- $t \sim {}_i(t_1 \cdots t_p)$ — an application, where i is a label identifying that application, $p \geq 1$, and the t_j are terms, for $j = 1, \cdots, p$.

We assume that the parameters of each abstraction are distinct, that the parameters of distinct abstractions are distinct, and that the labels of distinct applications are distinct. We also assume that the set $\mathcal{F}$ contains the primitive *cond* that takes three arguments — a predicate and two values — and returns the first value if the predicate is true and the second if it is false.

While the use of kernel terms as an "intermediate language" is not unusual in compilers for languages like ML and other functional languages, it may seem somewhat surprising when considering as a major application a language such as FORTRAN. The reason for choosing kernel terms is fourfold. First they provide the functionality necessary for any non-trivial programming language such as binding names and applying functions. The second reason is that they are sufficient in the sense that given an appropriate set of primitives, any programming language construct can be modeled as kernel terms. Thirdly, they are very simple — there are only five basic constructs — and this simplicity induces a corresponding simplicity in many of the compiler components and makes the task of proving various properties of these components tractable. Finally, the ability to deal with higher order functions makes the compiler applicable to all programming languages and permits higher order extensions to HPF to be considered.

With each kernel term, t, we associate Φ_t, the *flow* of t, an estimate of some aspect of the behavior of t. If t is an application, $t \sim {}_i(t_1 \cdots t_p)$, we use ϕ_i

as shorthand for $\Phi_{i(t_1 \cdots t_p)}$ and that is the reason for introducing labels for applications. The purpose of the abstract interpreter is to determine, for each term t, its flow, Φ_t.

4 An Overview of the Abstract Interpreter

An abstract interpreter seeks to determine certain aspects of the behavior of the terms of a program. One aspect of the behavior that is always of interest is the *control flow* of the program, that is, the set of functions that can be called from the operator position of each application. We follow [8] and define **nCFA** analysis to produce estimates of the inputs to and results of each function that depend on the call site and up to $\max(\mathbf{n}-1, 0)$ previous call sites. In particular **0CFA** is the analysis determining a single estimate of the inputs to and results of each function independent of its call sites.

In this paper we actually use a slight variant of **0CFA** that might be denoted **0+1CFA** which combines **0CFA** estimates for λ-expressions with **1CFA** estimates for primitive functions.

Some of the analysis tasks, other than control flow analysis, that might be of interest are the following:

- What is the *type* of the value associated with each term, so that when the type can be determined statically we can eliminate certain run-time checks and/or dispatches.
- Which parameters are *useless*, in the sense that no result depends upon them, so that they can be removed from the program.
- What are the sets of *basic induction parameters*, that is, the sets $\pi_j = \{x_{j1}, \cdots, x_{jn_j}\}$, for $j = 1, \cdots, k$, such that x_{ji} is bound only to a constant or to x_{jq} plus or minus a constant for one or more q, $1 \leq q \leq n_j$, that may permit certain strength reduction transformations, like replacing array subscript computations with pointer incrementing operations.
- Which parameters of a function are *strict* and which are *lazy*, where a parameter that is strict is always used and one that is lazy may or may not be used, and thus its computation might be deferred until it is determined that it is actually used.

For each analysis task, we require an appropriate set of *abstract values* to describe the behaviors of terms for that analysis task. For control flow analysis, the appropriate set of abstract values is the set of all subsets of the primitives and abstractions that occur in a program term. For type analysis, it is the set of all subsets of the types of the terms in a program. We discuss several other sets when we introduce the examples later.

Whatever set of abstract values is appropriate for some analysis task, they form a lattice, $\mathcal{L}$. If we have two estimates, say φ_1 and φ_2, of the flow of some term, Φ_t, then the least upper bound, $\varphi_1 \sqcup \varphi_2$, where $\sqcup$ is the join defined on the lattice $\mathcal{L}$, is also an estimate of Φ_t. We sometimes denote "φ_1 is an estimate of Φ_t" by $\Phi_t \geq \varphi_1$, treating $\geq$ as the partial order operator in the lattice $\mathcal{L}$.

One novel aspect of our treatment of the various lattices appropriate for various analyses is that we do not represent the lattice elements directly, instead introducing a data structure that we call the *behavior graph*. The nodes of a behavior graph, $\mathcal{G}$, are flows of program terms and *surrogates* for various abstract values. The (directed) arcs of the behavior graph define the relationships between the flows of terms and the surrogates for abstract values. Given some particular analysis task, finding the abstract values that comprise the estimate of the behavior of some term, t, that is, Φ_t, with respect to that analysis task is done by *tracing* all non-cyclic paths in $\mathcal{G}$ from the node for Φ_t, mapping from nodes that represent surrogates to the particular abstract value appropriate for the analysis task at hand. We presently explore several examples.

Our method of abstract interpretation of some program term, p, involves two phases:

Phase I: Construct the *behavior graph* for p, where the nodes of the behavior graph are flows of program terms and *surrogates* for abstract values and the arcs of the behavior graph define the relationships between the flows of terms and the surrogates for abstract values. Phase I is independent of the particular analysis task(s) to be considered.

Phase II: Given the behavior graph, the behavior of some term with respect to some particular analysis task is determined by *tracing* the behavior graph in a way that depends on the analysis task and interpreting the surrogates in a fashion that also depends on that task.

We now consider several examples (for more examples see [1]).

5 Some Examples of Abstract Interpretation For Various Analysis Tasks

We here consider several examples of abstract interpretation for various analysis tasks.

5.1 Type Annotation

Consider the program (where the term p is ignored):

$$\boxed{P_1 = {}_1(\lambda f.{}_2(f\ 1)\ {}_3(cond\ p\ \lambda x.{}_4(+\ x\ 1)\ \lambda y.{}_5(*\ y\ 2)))}$$

Here, P_1 binds f to one of two abstractions and then applies f to 1.

The first phase of abstract interpretation results in the behavior graph shown in Figure 1.

Some comments:

- The interpretation of the node $\beta(\Phi_f)$ is the set of flows of the bodies of those abstractions that f can be bound to. A β node has no arcs emanating and is dealt with in a manner discussed below.

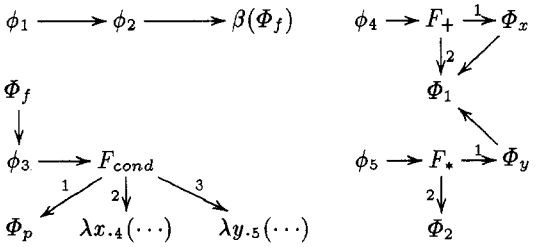
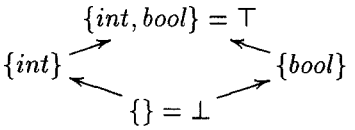


Fig. 1. Behavior Graph for P_1

- The interpretation of the fragment $\phi_1 \twoheadrightarrow \phi_2 \twoheadrightarrow \beta(\Phi_f)$ is that the flow of application $_1(\cdots)$ is at least the flow of application $_2(\cdots)$ and that is at least the flow of the body of whatever abstractions f can be bound to.
- The interpretation of the fragment $\Phi_x \twoheadrightarrow \Phi_1$ is that the flow Φ_x is at least Φ_1 and arises from the fact that x is a parameter of one of the abstractions bound to f and is thus bound to the constant 1.
- The interpretation of the fragment $\begin{array}{l}\phi_4 \twoheadrightarrow F_+ \xrightarrow{1} \Phi_x \\ \quad\quad\downarrow 2 \\ \quad\quad\Phi_1\end{array}$ is that the flow of $_4(\cdots)$ is at least the surrogate, $F_+(\Phi_x, \Phi_1)$, that represents the flow resulting from applying the primitive + to x and 1. Note that this surrogate is represented in the behavior graph using three nodes with ordered arcs (as indicated by the numbers on the arcs) from the node for F_+ to the flows of the two arguments that + is applied to.

Given the above behavior graph, consider the particular task of determining the types of the several terms of P_1. The abstract values appropriate for this task might be thought of as comprising the very simple lattice:



Here, *int* and *bool* are intended to suggest the types of integer and boolean values.

In order to do the analysis, we have to specify how to treat each of the surrogates, Φ_1, $F_+(\Phi_x, \Phi_1)$, and so on for this analysis task. For each primitive, p, we assume that there is a function F_p^{type} that, applied to the arguments of the surrogate node F_p, returns the type of that node. Additionally, we assume that for each constant surrogate, Φ_c, there is a map, $V^{type}(\Phi_c)$ that provides the type of that constant.

For this application, we assume that F_+^{type} and F_*^{type} return *int* (independent of their arguments) and that the map V^{type} for Φ_c returns *int* for c an integer. We further assume that $F_{cond}^{type}(\Phi_{N_1}, \Phi_{N_2}, \Phi_{N_3}) = \Phi_{N_2} \cup \Phi_{N_3}$.

We define the iterator $Trace^{type}(\varphi)$ that traces all non-cyclic paths from some node, φ, and does the following:

- For each F_p node with k (ordered) argument arcs to nodes $\varphi_1, \cdots, \varphi_k$ it returns $F_p^{type}(\varphi_1, \cdots, \varphi_k)$, as discussed above.
- For Φ_c it returns $V^{type}(\Phi_c)$.
- For a $\lambda x_1 \cdots x_k.B$ node it returns

$$\Phi_{x_1} \times \cdots \times \Phi_{x_k} \rightarrow \Phi_B$$

- For $\beta(\varphi)$ it returns the flows of the bodies of all abstractions in the flow φ.

We then use $Trace^{type}$ on the nodes of the value graph to find:

$$\phi_1 = \phi_2 = \beta(\Phi_f) = \phi_4 = \phi_5 = \Phi_x = \Phi_y = int$$

$$\Phi_f = \phi_3 = int \rightarrow int$$

5.2 Basic Induction Parameter Detection

The sets of *basic induction parameters* are the sets $\pi_j = \{x_{j1}, \cdots, x_{jn_j}\}$, for $j = 1, \cdots, k$, such that x_{ji}, for $i = 1, \cdots, n_j$, is bound only to a constant or to x_{jq} plus or minus a constant for one or more q, $1 \leq q \leq n_j$.

Consider the following example:

$$\boxed{P_2 = {}_1(f\ 0)}$$

where f and g are bound(via some mechanism akin to LETREC), respectively, to the following lambdas:

$$\lambda n.{}_2(cond\ p\ {}_3(f\ {}_4(g\ n))\ {}_5(\lambda m.{}_6(f\ {}_7(+\ m\ 2))\ {}_8(+\ n\ 3)))$$

$$\lambda x.{}_9(+\ x\ 1))$$

and p is ignored.

Here f is a function of n that either calls itself with the result of applying g to n or introduces m bound to $n+3$ and calls itself with $m+2$. The function g simply adds one to its argument. The behavior graph for P_2 is shown in Figure 2.

The first question that must be addressed is that of the appropriate set of abstract values to employ for this analysis task. Since we are interested only in parameters that are bound to integers or other parameters plus or minus integers, a natural lattice might be $\bot \cup \top \cup 2^{\{\langle \Phi_x, \Phi_c \rangle\}} \cup 2^{\{\Phi_c\}}$, where Φ_x ranges over all parameter flows and Φ_c ranges over all the flows of integer constants that occur in P_2. Here, $\langle \Phi_x, \Phi_c \rangle$ is interpreted to mean the parameter x plus the constant c and Φ_c is interpreted to mean the constant c.

Now consider the following interpretation for F_+^{biv}, that is, F_+ for the task of basic induction variable detection. If F_+^{biv} has as arguments the flow of a parameter, say Φ_x, and the flow of an integer constant, say Φ_c, its flow is $\langle \Phi_x, \Phi_c \rangle$,

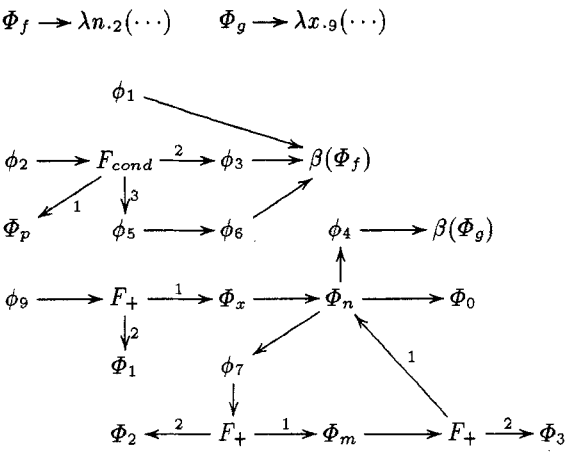


Fig. 2. Behavior Graph for P_2

and otherwise its flow is $\top$, and similarly for other primitives that can increment an argument.

The interpretation of F_{cond}^{biv} is that it returns the union of its second and third arguments, and for the remaining primitives, p, F_p^{biv} returns $\top$.

The interpretation of $V^{biv}(\Phi_c)$ is Φ_c for integers and $\top$ for anything else. Let $Trace^{biv}$ be an iterator that returns the result of tracing some node and returning all the abstract values it encounters, evaluating F_p^{biv} on its arguments if F_p nodes are encountered and V_c^{biv} if constants are encountered. Then for each parameter flow, Φ_x, we do the following:

1. Let Φ_x^{biv} be $\{\}$, the empty set.
2. For each φ in $Trace^{biv}(\Phi_x)$, if $\varphi = \top$ then set $\Phi_x^{biv} = \top$ and otherwise set $\Phi_x^{biv} = \Phi_x^{biv} \cup \{\varphi\}$.

For the above example we obtain:

$$\Phi_n^{biv} = \{\langle \Phi_x, \Phi_1 \rangle, \Phi_0, \langle \Phi_m, \Phi_2 \rangle\}$$

$$\Phi_x^{biv} = \{\langle \Phi_x, \Phi_1 \rangle, \Phi_0, \langle \Phi_m, \Phi_2 \rangle\}$$

$$\Phi_m^{biv} = \{\langle \Phi_n, \Phi_3 \rangle\}$$

Thus, n, x, and m constitute a set of basic induction variables.

6 Phase I of Abstract Interpretation — Establishing the Behavior Graph

The construction of the behavior graph for some program term takes place in two phases:

Phase IA: Let p be some program term and $\mathcal{G}$ the behavior graph for p, initially the empty graph. The first phase in constructing $\mathcal{G}$ is carried out by the function γ that, starting with p, recursively considers the terms in p, constructing pieces of $\mathcal{G}$ and determining a set of constraints that are used in Phase IB to complete the behavior graph.

Phase IB: In this phase, we "discharge" the constraints accumulated in Phase IA to complete the behavior graph.

6.1 The Function $\gamma(t)$ for Phase IA of Constructing the Behavior Graph $\mathcal{G}$

$\gamma(t)$ for term t is as follows:

$\boxed{t = c}$ Install Φ_c in $\mathcal{G}$

$\boxed{t = \lambda x_1 \cdots x_k.B}$ Call $\gamma(x_1), \cdots, \gamma(x_k), \gamma(B)$ and install $\lambda x_1 \cdots x_k.B$ in $\mathcal{G}$

$\boxed{t = x}$ Install Φ_x in $\mathcal{G}$

$\boxed{t = {}_i(M\ N_1 \cdots\ N_k)}$ Call $\gamma(M), \gamma(N_1), \cdots, \gamma(N_k)$, and dispatch on M as follows:

$\boxed{M = \lambda x_1 \cdots x_k.B:}$ Install in $\mathcal{G}$:

$$\Phi_{x_j} \longrightarrow \Phi_{N_j}, \text{ for } j = 1, \cdots, k \text{ and}$$

$$\phi_i \longrightarrow \Phi_B$$

$\boxed{M = p, p \in \mathcal{F}:}$ Install in $\mathcal{G}$:

$$\begin{array}{l} \phi_i \longrightarrow F_p \xrightarrow{1} \Phi_{N_1} \\ \qquad\quad \downarrow k \\ \qquad\quad \Phi_{N_k} \end{array}$$

$\boxed{M = \phi_j \text{ or } M = \Phi_f:}$ Call

$$\mathit{Unknown\text{-}operator}(i, \Phi_M, \langle \Phi_{N_1}, \cdots \Phi_{N_k} \rangle)$$

The non-trivial case for γ is that for dealing with an operator that is an application or a parameter and this is the case that establishes the constraints alluded to above. Consider the application term $t \sim {}_i(f\ t_1\ \cdots\ t_k)$, where f is a parameter and the t_j are arbitrary terms. When we know what set of abstractions f can be bound to we can, for the j-th parameter, Φ_{y_j} of each of these abstractions, establish the arc $\Phi_{y_j} \longrightarrow \Phi_{N_j}$. However, as we traverse the program term using γ, we may encounter an application of f before we have determined the set of all the abstractions and primitives that f can be bound to and thus we need some sort of mechanism to defer construction of parts of the behavior graph until these sets can be determined. Such a mechanism is discussed in the next section.

6.2 α Sets, α Constraints, and β Flows

As noted above, we require some sort of mechanism to defer dealing with the flows of operators that are not explicit abstractions or primitives until such time as we can determine which abstractions might be contained in those flows. For this purpose, we introduce the set $\alpha_j(\Phi_M)$ that is a set of parameter flows that will, ultimately, contain the flow of the j-th explicit parameter of each abstraction that can be in the operator flow, Φ_M.

Additionally, we introduce the constraints $\alpha_j(\Phi_M) \succeq \varphi$ that specify, for each parameter flow, $\Phi_x \in \alpha_j(\Phi_M)$, that the fragment $\Phi_x \rightarrow \varphi$ belongs in $\mathcal{G}$. Thus, α constraints *package* fragments of $\mathcal{G}$ that have yet to be installed and will be installed when we determine the contents of the corresponding α sets.

In addition to the α sets and α constraints on flows, we introduce the β flows, where, if Φ_M is the flow of an operator, $\beta(\Phi_M)$ denotes the set of all the flows of the bodies of the abstractions that can be in Φ_M.

Now we can describe the effects of calling

$$\mathit{Unknown\text{-}operator}(i, \Phi_M, \langle \Phi_{N_1}, \cdots \Phi_{N_k} \rangle)$$

a call which is made when the flow variable Φ_M is in the operator position of application i and the flows of the arguments to Φ_M are $\Phi_{N_1}, \cdots, \Phi_{N_k}$. The result of such a call is to establish α constraints for Φ_M and to establish the flow ϕ_i as at least the β flow of Φ_M, that is, we establish the constraint $\alpha_j(\Phi_M) \succeq \Phi_{N_j}$ for $j = 1, \cdots, k$ and add the arc $\phi_i \longrightarrow \beta(\Phi_M)$ to $\mathcal{G}$.

7 Phase IB – Completing the Behavior Graph

When an explicit abstraction appears in the operator position of an application, the inequalities on the flows of its parameters are established directly by γ. However, when a flow variable (a parameter flow or application flow) appears in the operator position, constraints on the flows of its parameters are established indirectly via the constraints $\alpha_j(\Phi_M) \succeq \Phi_{N_j}$ that "package" what will eventually lead to arcs in the behavior graph, effectively deferring installing them until the members of the $\alpha_j(\Phi_M)$ sets are known. These members are discovered whenever there is some flow variable, Φ_M, that can be the flow of an operator and there is a abstraction, $\lambda x_1 \cdots x_k.B$ in the estimate of Φ_M, which implies that the j-th explicit parameter of that abstraction, Φ_{x_j}, is in $\alpha_j(\Phi_M)$, for $j = 1, \cdots, k$. The constraints on some parameter flow, Φ_y, are *revealed* when we realize that Φ_y is a member of some α_j set, $\Phi_y \in \alpha_j(\Phi_M)$, and thus for each constraint $\alpha_j(\Phi_M) \succeq \Phi_{N_j}$, we "unpackage" that $\succeq$ constraint by adding $\Phi_y \rightarrow \Phi_{N_j}$ to $\mathcal{G}$.

An algorithm for revealing constraints and completing the behavior graph $\mathcal{G}$ is:

1. For each flow Φ_M that has α constraints and each φ in $\mathit{Trace}^{cfa}(\Phi_M)$, if $\varphi = \lambda y_1 \cdots y_k.B$ then add Φ_{y_j} to the set $\alpha_j(\Phi_M)$, for $j = 1, \cdots, k$.

2. For each j, Φ_M, Φ_y, and φ such that $\Phi_y \in \alpha_j(\Phi_M)$ and $\alpha_j(\Phi_M) \succeq \varphi$, install $\Phi_y \rightarrow \varphi$ in $\mathcal{G}$ to *unpackage* a constraint.
3. If step 2 resulted in an increase in the estimate of any parameter flow, return to step 1 and otherwise terminate.

The iterator $Trace^{cfa}(\Phi_M)$ has the job of delivering all the abstractions in the (current) estimate of Φ_M and is discussed in the following section.

7.1 The $Trace^{cfa}$ Iterator

The iterator, $Trace^{cfa}(\Phi_M)$, has the task of delivering all the abstractions in the current estimate of some flow, Φ_M, that is the flow of an operator of some application that has α constraints. There are a couple of issues that must be dealt with. First, the behavior graph can have cycles that arise when $\Phi_x \geq \Phi_y$ and $\Phi_y \geq \Phi_x$ for two parameter flows, Φ_x and Φ_y, or when there are mutually recursive functions. The second issue is that of efficiency — we would like the cost of constructing the graph to be peoportional to the number of edges it contains. The "environment" global parameter, ν, is used to deal with breaking cycles in the behavior graph and is initially the empty environment, ϵ. In general, ν is a stack of frames where a frame is a set of flows that are being traced and it permits the detection of cycles.

The definition of $Trace^{cfa}(\Phi_M)$ is to initialize $\nu = \epsilon$ and then for each node φ on each path from Φ_M, to dispatch per case as follows:

$\boxed{\varphi \sim \lambda x_1, \cdots, x_k.B}$ $Deliver(\lambda x_1, \cdots, x_k.B)$

$\boxed{\varphi \sim \Phi_x \textbf{ for a parameter, } x}$ Proceed as follows:

1. If $\Phi_x \in Top(\nu)$, where $Top(\nu)$ is the topmost frame in ν then this parameter flow has already been traced and nothing is done.
2. Otherwise, add Φ_x to $Top(\nu)$ and trace the parameter flow.

$\boxed{\varphi \sim F_{cond}(\Phi_{N_1}, \Phi_{N_2}, \Phi_{N_3})}$ Proceed as follows:

1. For each φ in $Trace^{cfa}(\Phi_{N_2})$, $Deliver(\varphi)$.
2. For each φ in $Trace^{cfa}(\Phi_{N_3})$, $Deliver(\varphi)$.
3. Continue the outer iteration.

$\boxed{\varphi = \beta(\varphi')}$ Proceed as follows:

1. If $\beta(\varphi') \in \nu$, then this node has already been traced and we simply continue the iteration.
2. Otherwise, proceed as follows:
   (a) Push a new frame containing $\beta(\varphi')$ to ν.
   (b) For each φ in $Trace^{cfa}(\varphi')$ such that $\varphi = \lambda x_1 \cdots x_k.B$, deliver $Trace^{cfa}(B)$.
   (c) Pop ν and continue the outer iteration.

$\boxed{Otherwise}$ Continue the iteration.

Here, "$Deliver(\varphi)$" means that φ is one of the values delivered by the iterator. Some comments:

1. If an abstraction is encountered, it is delivered.
2. If an application of the primitive *cond* is encountered, we recursively call $Trace^{cfa}$ on its second and third arguments and then continue the original iteration. This is, of course, paradigmatic, and each other primitive that might return an abstraction must also be traced in the appropriate fashion.
3. The node $\varphi = \beta(\varphi')$ has no arcs emanating from it and denotes the set of bodies of all the abstractions that are in the estimate of φ'.
4. If the node encountered is anything else (like a surrogate for a constant other than an abstraction, an application, and so on), we deliver nothing and continue following the path we are on.

Efficiency Considerations The issue of the efficiency of $Trace^{cfa}$ is to avoid tracing parts of the behavior graph that have already been traced. To deal with this we introduce, for each flow that has α constraints, a set of flows containing each flow that has already been traced. We then consult this set and stop tracing whenever we encounter a node already traced. As a result, the construction of the behavior graph involves traversing its edges only once. Once the behavior graph is constructed, computing the flows takes at most the time taken by the local transitive closure algorithm.

7.2 Correctness

The minimal solution to the system of immediate, α and β constraints described in section 6.2 corresponds to the least fixed of the following system of equations[4]:

- For each constant term $t = k$, include the equation

$$\phi_t = \{\overline{k}\};$$

  where $\overline{k}$ is the abstraction of constant $k \in \mathcal{C}$.
- For each lambda expression $t = \lambda x.B$ include the equation

$$\phi_t = \{t\};$$

- For each function parameter x of $f = \lambda x.B$ include the equation

$$\phi_x = \bigcup\{\phi_b \mid f \in \phi_a\};$$

  where $_i(a\ b)$ is some application.
- For each application $_i(a\ b)$ include the equation:

$$\phi_i = \bigcup\{\phi_q \mid \lambda y._q B \in \phi_a\}$$

[4] For simplicity, we consider only one argument lambda expressions

The first two equation sets implement the immediate constraints. The last two equation sets have to do with the flow across function boundary. The third equation set corresponds to the α and shows that the flow of a parameter x is the collection of the arguments at all call sites of its function f. Finally, the last equation corresponds to the *beta* constraints and computes the flow of an application as the collection of all results returned by all functions applied there. The soundness of this system with respect to an operational semantics is proven in [10].

8 Status

We have developed a prototype implementation of the Abstract Interpreter described in this paper in Common Lisp and are in the process of experimenting with this prototype by applying it to various analyses, including approximative type recovery, strength reduction, basic induction parameters, and strictness analysis. Preliminary results are encouraging in that they show the efficiency of our approach to be much superior to that reported in [12]. With most practical programs the cost of constructing the behavior graph is linear in the number of terms in a program, becoming slightly non-linear only when some abstraction can be returned from many applications.

The abstract interpreter actually implemented deals with a number of complications that were omitted from this paper in the interests of readability and employs algorithms that are very incremental, doing, we believe, the minimum work required. The full details are presented in [1]. One difference is that the actual implementation deals with multiple values in the sense that it assumes that there is a primitive called *block* that returns whatever values are fed to it as arguments. Thus, (*block* 1 2) returns the two values 1 and 2 and the interpretation of $(f\ (g\ x)\ y)$ is that f is fed whatever values g returns plus that bound to y, and so on. The complication that this introduces is that when the γ function is traversing program terms and encounters the application ${}_i(t_1\ \cdots\ t_p)$ it must determine the operator and the operands of the application. That is, it is possible that t_1 and t_2 produce no values and that the operator is the first value produced by t_3. To deal with this case the lattice used for control flow analysis is augmented with the two abstract values *value* and *void* indicating, respectively, a single value and no value. If, when γ is traversing the program terms, it cannot determine the operator and/or the operands of an application, that application is deferred until Phase IA when sufficient information has been developed that the operator and operands can be determined.

Another feature of the implementation is that it deals with "exit fuctions". We introduce another parameter name space, $\hat{\mathcal{N}}$ and generalize abstractions to have the form $\lambda\hat{f}x_1 \cdots x_k.B$ where the interpretation of $\hat{f} \in \hat{\mathcal{N}}$ is, in the body B when that abstraction is being applied, an *exit function* that will, when called, exit that application of the abstraction with whatever values are fed to $\hat{f}$ as results. Thus exit functions are similar to `catch` and `throw` in most LISPs. The complication introduced are minor and discussed in detail in [1].

Our implementation has also been extended to produce finer estimates in order to cater to what we believe is an important type of optimization, particularly when dealing with a variety of high performance computers as targets. Suppose that there is a library of functions for dealing with, for example, linear algebra computations. Then one sort of optimization we might want is to specialize the library functions when certain arguments are known at compile time by doing partial evaluation in order save run time costs. We believe that this application requires call site specific analysis. Using the theory described in [10] we have extended the abstract interpreter described here to handle flows of the form Φ_t^σ, where t is a term and $\sigma \in \Sigma$ is an abstract *call conctext* with the result that tracing in the behavior graph will become sensitive to the call context component of nodes.

9 Conclusions and Future Work

This paper presented an approach for constructing a suite of optimizers based on a general purpose abstract interpreter coupled with an optimization task specific analyzer followed by transformations enabled by annotations.

We believe that the work reported here has great potential for providing a sound basis for and an efficient implementation of a number of optimizers, ranging from such standards as strength reduction and common sub-expression elimination to such novel applications as static determination of data mapping for languages designed for massively parallel computing.

However, much remains to be done. One important concern is to prove the soundness of the proposed approach and the correctness of the algorithms presented and to develop complexity measures of those algorithms and [10] provides partial answers.

Another is the development of tools that support the customization of the general framework to particular optimization problems, that is the implementation of the functions F_p^τ and operations on the lattice over which they operate.

Finally, we are concerned with implementing a large inventory of optimizers for HPF and other parallel languages like BSP-L([2]), an intermediate parallel language for the BSP model of computation([14]) and gathering experimental data that will let us judge the efficacy of our approach.

References

1. Thomas Cheatham, Haiming Gao, and Dan Stefanescu *The Harvard Abstract Interpreter* Technical Report, Harvard University, April 1993.
2. Thomas Cheatham, Jr., Amr Fahmy and Dan Stefanescu *BSP-L — A Programming Language for the Bulk Synchronous Processing Model*, Center for Research in Computing Technology, Harvard University, December 1993
3. Patrick Cousot and Radhia Cousot *Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximate fixpoints*, Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, pages 238-252, 1977.

4. Neil Jones and Alan Mycroft *Data Flow Analysis of Applicative Programs Using Minimal Functions Graphs: Abridged Version*, Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages, pages 296-306, 1986.
5. Mike Karr and Steve Rosen *Dynamic Crossreference Analysis*, lecture notes, 1992.
6. Jens Palsberg and Michael I. Schwartzbach *Safety Analysis versus Type Inference*, Information and Computation, to appear.
7. Olin Shivers *Control Flow Analysis in Scheme* ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta GA, June 22-24, 1988.
8. Olin Shivers *Control-Flow Analysis of Higher Order Languages or Taming Lambda*, Technical Report CMU-CS-91-145, Carnegie Mellon University, May 1991.
9. Peter Sestoft *Replacing Function Parameters by Global Variables*, FPCA'89, pages 39-53, 1989.
10. Dan Stefanescu and Yuli Zhou *An Equational Framework for the Abstract Analysis of Functional Programs*, Proceedings of ACM Conference on Lisp and Functional Programming, Orlando, 1994.
11. Mitchell Wand and Paul Steckler *Selective and Lightweight Closure Conversion*, ACM Symposium on Principles of Programming Languages, Portland, 1994.
12. Atty Kanamori and Daniel Weise *An Empirical Study of an Abstract Interpretation of Scheme Programs* Technical Report CSL-TR-92-521, Computer Systems Laboratory, Stanford University, April 1992.
13. Mary Hall and Ken Kennedy, *Efficient Call Graph Analysis*, ACM Letters on Programming Languages and Systems, Vol. 1, No. 3, Sept. 1992, pp 227-242.
14. L. G. Valiant *A Bridging Model for Parallel Computation* Communications of the ACM, 33(8):103-111, 1990