FLOW ANALYSIS OF LAMBDA EXPRESSIONS
(Preliminary Version)

Neil D. Jones
Aarhus University, Denmark

0. INTRODUCTION

A method is described to extract from an untyped λ-expression information about the sequence of intermediate λ-expressions obtained during its evaluation. The information can be used to give "safe positive answers" to questions involving termination or nontermination of the evaluation, dependence of one subexpression on another and type errors encountered while applying δ rules, thus providing an alternative to techniques of Morris and Levy ([Mor68], [Lev75]). The method works by building a "safe description" of the set of states entered by a call-by-name interpreter and analyzing this description. A similar and more complete analysis of a call-by-value interpreter may be found in [Jon81].

From a flow analysis viewpoint these results extend existing interprocedural analysis methods to include call-by-name and the use of functions both as arguments to other functions and as the results returned by them. Further, the method naturally handles both local and global variables, extending [Cou77a] and [Sha80]. It seems clear that other traditional analyses such as available expressions, constant propagation, etc. can be carried out in this framework.

The main emphasis is on development of the framework and showing its relation to abstract interpretation, rather than on its efficient use in applications. A simplified and optimized version of the method would have applications in the efficient compilation of λ-calculus-based programming languages such as LISP, SCHEME and SASL ([McC63], [Ste75], [Tur76]).

The method provides a general way to find safe approximate descriptions of computations by algorithms which manipulate recursive data structures. It is thus not limited to the λ-calculus, but may be applied to analyze any programming language whose semantics can be implemented by an appropriate definitional interpreter.

Another application would be to extend the method to the flow analysis of denotational definitions of programming languages. This could be used in semantics-directed compiler generation as described in [JoS80], and provided the initial motivation for this study.

Related work

Lambda calculus evaluators have been studied in [Böh72], [Lan64], [McG70], [Plo75], [Rey72], [Sch80] and [Weg68]. Sufficient conditions for termination of reduction sequences have been developed by Morris and Levy, and Mycroft has investigated the replacement of call-by-name by call-by-value ([Mor68], [Lev75], [Myc80]).

Interprocedural flow analysis has been studied by Rosen, Cousot and Cousot, and Sharir and Pnueli ([Ros79], [Cou77a], [ShP81]). Sharir describes a general technique for flow analysis of applicative programs and develops a more efficient version for bitvectoring analyses ([Sha80]). The techniques do not handle call-by-name or the use of functions as arguments and results. Pleban describes a method to flow-analyze SCHEME in [Ple81], using a denotational semantics framework.

Outline of the paper

In Section 1 we introduce a call-by-name λ-expression evaluator CBN, and establish a useful property of its computation.

Section 2 develops analysis methods for a closed λ-expression M_0 without constants; this we call the control flow analysis of the call-by-value computation. The result is a safe description of

$$\text{States}(M_0) = \{\sigma \mid \text{CBV enters state } \sigma \text{ during its computation on } M_0\}$$

More specifically a finite lattice D of descriptions will be defined, each of whose elements d describes a set Desc(d) of machine states. An algorithm will be given to obtain from M_0 a "safe" description $d(M_0)$ such that $\text{States}(M_0) \subseteq \text{Desc}(d(M_0))$. This will be shown to imply that "safe positive answers" may be effectively obtained for a number of interesting questions about the computation. Note that precise answers cannot always be given, due to the undicidability, for instance, of the halting problem.

Section 3 briefly describes a similar development including constants and δ rules; a more complete development is found in [Jon81]. Section 4 ends with conclusions, future directions and acknowledgements.

Notational Conventions

The power set of X, written $\mathcal{P}(X)$, is the set of all subsets of X.

Given sets X and Y, $X \overset{P}{\rightarrow} Y$ is the set of all partial functions f from X to Y, and $X \overset{\sim}{\rightarrow} Y$ is the set of all f in $X \overset{P}{\rightarrow} Y$ such that

$$\text{Domain}(f) = \{x \mid f(x) \text{ is defined}\}$$

is a finite set. $\emptyset$ is the unique function in $X \overset{P}{\rightarrow} Y$ with empty domain. Two functions are equal iff they have the same domain and the same values on arguments in that domain. The notation $f[x \rightarrow y]$ (where $f \in X \overset{P}{\rightarrow} Y$, $x \in X$, $y \in Y$) denotes the function $f' \in X \overset{P}{\rightarrow} Y$ such that for all $z \in X$, $f'(z) =$ if $x = z$ then y else $f(z)$.

Given a relation $\rightarrow$ (always in infix notation), $\overset{n}{\rightarrow}$ is its n'th power ($n \geq 0$), $\overset{+}{\rightarrow}$ is its transitive closure and $\overset{*}{\rightarrow}$ is its reflexive transitive closure.

The Lambda Calculus

Given predefined disjoint sets Var = $\{x, y, z, \dots\}$ and Con = $\{a, b, c, \dots\}$ of variables and constants respectively, the set of λ-calculus terms Lam = $\{M, N, \dots\}$ is specified inductively by the abstract syntax

$$\text{Lam} ::= \text{Var} \mid \text{Con} \mid \text{Lam Lam} \mid \lambda \text{ Var Lam}$$

A combination is a term of form MN, and has operator M and operand N. An abstraction is a term λxM, and a value is a term which is not a combination.

The free and bound variables FV(M) and BV(M) of a term M are defined by

(1) $FV(a)=\emptyset$; $FV(x)=\{x\}$; $FV(MN)=FV(M) \cup FV(N)$; $FV(\lambda xM)=FV(M) \setminus \{x\}$

(2) $BV(a)=\emptyset$; $BV(x)=\emptyset$; $BV(MN)=BV(M) \cup BV(N)$; $BV(\lambda xM)=BV(m) \cup \{x\}$

A term M is closed if $FV(M)=\emptyset$. The substitution prefix $[M/x]$ defines the following operation on Lam: $[M/x]N$ is the result of substituting M for all free occurrences of x in N, renaming variables of N as necessary to avoid capturing bound variables as in [Cur58]. A closed term is called a program as in [Plo75].

Now supposing we are given a partial function (Cap = "Constant apply")

$$\text{Cap: Con} \times \text{Con} \overset{P}{\to} \text{Con}$$

we define the reduction relation > on terms by

1.	$\lambda xM > \lambda y[y/x]M$	(if $y \notin FV(M)$)	α reduction
2.	$(\lambda xM)N > [N/x]M$		β reduction
3.	$ab > \text{Constapply}(a, b)$	(if this is defined)	δ reduction
4.	$\dfrac{M > N}{c[M] > c[N]}$	for any context c[ ]	reduction in context

A machine independent call-by-name evaluation function eval: Program $\overset{P}{\to}$ Program is defined as follows; it comes from [Plo75], which also contains a call-by-value analogue:

$$\text{eval}(a) = a; \quad \text{eval}(\lambda xM) = \lambda xM;$$

$$\text{eval}(MN) = \begin{cases} \text{eval}([N/x]M') & \text{if Eval}(M) = \lambda xM' \\ a' & \text{if eval}(M) = a,\ \text{eval}(N) = b \text{ and} \\ & \text{Constapply}(a, b) = a' \text{ is defined} \end{cases}$$

Lemma If M is a program and eval(M) is defined then M $\overset{*}{>}$ eval(M) without renaming.

1. A CALL-BY-NAME λ-EXPRESSION EVALUATOR

We introduce a λ-calculus interpreter without constants and establish some useful properties. Due to space limitations we only consider a simple call-by-name interpreter CBN, very similar to one studied by Schmidt [Sch80]. The same flow analysis techniques are also applicable to call-by-value; [Jon81] presents a CBV interpreter, proves that it correctly performs call-by-value evaluation and that the flow analysis of CBV is "safe". Further, [Jon81] shows that Landin's original SECD machine [Lan64] is quivalent to the simpler CBV, using a characterization of SECD by Plotkin [Plo75]. The notations and ideas in this section owe much to [Plo75].

Closures and Environments

Following Landin we avoid explicit substitution into λ-expressions by representing a λ-expression by a closure (M, e), where M is a term and e is an environment giving the values of those free variables of M which have been bound as the result of β reductions.

Letting CL denote the set of all closures, the function Real: CL → Lam will take a closure cl ∈ CL into the λ-expression it represents. Real, CL and the set of environments E are defined as follows:

Closures	: cl ∈ CL = Lam x E
Environments	: e ∈ E = Var $\tilde{\to}$ CL

Real : CL → Lam

$$\text{Real}((M, e)) = [\text{Real}(e(x_1))/x_1]\ldots[\text{Real}(e(x_n))/x_n]M$$

where $\text{Domain}(e) = \{x_1, \ldots, x_n\}$

Figure 1. Environment Closures and Real

The equations above are to be taken as inductive definitions of certain sets, not as Scott-style domains (elements of E and CL may be thought of as finite trees due to the restriction to environments with finite domains). In this paper Real(cl) will always be closed. For example

a) if $cl_1 = (\lambda yy, \emptyset)$ then $\text{Real}(cl_1) = \lambda yy$

b) if $cl_2 = (xx, \emptyset[x \to cl_1])$ then $\text{Real}(cl_2) = (\lambda yy)(\lambda yy)$

Interpreter States and Transition Rules

The interpreter evaluates an expression M_0 by performing a series of state transitions $\sigma_0 \Rightarrow \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n$, where $\sigma_0 = \text{Load}(M_0)$ is the initial state corresponding to M_0, and σ_n is terminal (meaning that for no σ does $\sigma_n \Rightarrow \sigma$ hold). The result of the evaluation is $\text{Unload}(\sigma_n)$.

A state is a pair $\sigma = <cl, cl_1, \ldots, cl_n>$ where cl is the control closure and $cl_1 \ldots cl_n$ is a sequence of closures called the context stack. It represents the λ-expression $\text{Unload}(\sigma) = \text{Real}(cl)\text{Real}(cl_1)\ldots\text{Real}(cl_n)$. A state transition $\sigma_1 \Rightarrow \sigma_2$ is determined by the form of σ_1's control closure. The following table defines CBN, using ϵ to indicate the empty context stack.

Interpreter States: $\sigma \in \Sigma = CL \times C$

Context Stacks : $c \in C = CL^*$

Readin Function(Load: Lam $\rightarrow \Sigma$):

$$\text{Load}(M_0) = <(M_0, \emptyset), \epsilon>$$

Readout Function(Unload: $\Sigma \rightarrow$ Lam):

$$\text{Unload}<cl_0, cl_1, \ldots, cl_n> \; = \; \text{Real}(cl_0)\ldots\text{Real}(cl_n)$$

<u>Transition Rules</u>

Scan Operator: $<(MN, e), cl_1 \ldots cl_n> \Rightarrow <(M, e), (N, e)cl_1 \ldots cl_n>$

Expand Variable: $<(x, e), cl_1 \ldots cl_n> \;\Rightarrow <e(x), cl_1 \ldots cl_n>$

β reduction: $<(\lambda xM, e), cl_1 cl_2 \ldots cl_n> \Rightarrow <(M, e[x \rightarrow cl_1]), cl_2 \ldots cl_n>$

Figure 2. CBN Interpreter Without Constants

Figure 3 shows how CBN carries out the reduction sequence $M_0 = (\lambda xxx)(\lambda yy)\lambda zz > (\lambda yy)(\lambda yy)\lambda zz > (\lambda yy)\lambda zz > \lambda zz$. The CBN computation is considerably longer than the reduction sequence due to the need for explicit steps to descend into syntactic substructures and to lookup variable bindings.

Interpreter States	Actions	Reductions
$\text{Load}(M_0) =$		
$= <((\lambda xxx)(\lambda yy)\lambda zz, \emptyset), \quad \epsilon \quad >$	scan operator	$(\lambda xxx)(\lambda yy)\lambda zz$
$\Rightarrow <((\lambda xxx)\lambda yy, \emptyset \quad , \quad (\lambda zz, \emptyset)>$	scan operator	
$\Rightarrow <(\lambda xxx, \emptyset) \quad , \quad (\lambda yy, \emptyset)(\lambda zz, \emptyset)>$	β reduce	$>$
$\Rightarrow <(xx, \underbrace{[x \rightarrow (\lambda yy, \emptyset)]}), \quad (\lambda zz, \emptyset)>$	–	$(\lambda yy)(\lambda yy)\lambda zz$
$= <(xx, \text{ call this } e_1) \quad , \quad (\lambda zz, \emptyset)>$	scan operator	
$\Rightarrow <(x, e_1) \quad , \quad (x, e_1)(\lambda zz, \emptyset)>$	expand x	$>$
$\Rightarrow <(\lambda yy, \emptyset) \quad , \quad (x, e_1)(\lambda zz, \emptyset)>$	β reduce	
$\Rightarrow <(y, \emptyset[y \rightarrow (x, e_1)]) \quad , \quad (\lambda zz, \emptyset >$	expand y	$(\lambda yy)\lambda zz$
$\Rightarrow <(x, e_1) \quad , \quad (\lambda zz, \emptyset)>$	expand x	
$\Rightarrow <(\lambda yy, \emptyset) \quad , \quad (\lambda zz, \emptyset)>$	β reduce	$>$
$\Rightarrow <(y, \emptyset[y \rightarrow (\lambda zz, \emptyset)]) \quad , \quad \epsilon \quad >$	expand y	
$\Rightarrow <(\lambda zz, \emptyset) \quad , \quad \epsilon \quad >$	halt	λzz

$<$Control Closure , Context Stack $>$

Figure 3. Example Computation by CBN

The following asserts the correctness of CBN, and is proven by induction on the definition of eval and the number of steps in a CBN computation. Note that the value of a λ-expression without constants can only be an abstraction.

Theorem 1. Let M be a program without constants.

a) if eval(M) = N then $\exists\sigma$ (Load(M) $\overset{*}{\Rightarrow}$ σ and N = Unload(σ))

b) if Load(M) $\overset{*}{\Rightarrow}$ σ and Unload(σ) is a value then eval(M) = Unload(σ)

A Useful Property

In every closure (M, e) which was obtained in the example computation, M was a subexpression of the starting expression. This is in fact always true.

Lemma 2. Suppose M is closed and Load(M_0) $\overset{*}{\Rightarrow}$ $<$(M, e), $cl_1 \ldots cl_n$$>$. Then

a) Domain(e) $\subseteq$ BV(M_0) and

b) M is a subexpression of M_0

Proof Define "p appears in cl" for λ-expressions or environments p and closures cl as follows: M and e appear in (M, e); if p appears in e(x) then p appears in (M, e). An easy induction on n now verifies that if Load(M_0) $\overset{n}{\Rightarrow}$ $<cl_0, cl_1 \ldots cl_n>$ and p appears in any closure cl_i, then p satisfies a) or b) above. □ □

Lemma 2 implies that for each fixed input M_0 we may regard CBN as operating on occurrences of expressions (in M_0) rather than on arbitrary expressions. This useful property follows from the fact that we only do outside-in reductions. It implies that a computer implementation of CBN can manipulate pointers instead of arbitrary λ-expressions. Incidentally, the SECD machine also has this property. Another way to view this is that λ calculus evaluation cannot create new "program text"; it can only reinterpret the original program text in new environments. A similar result also holds if δ reduction is included.

The CBN(M_0) Interpreter

The approximate description of M_0 to be developed shortly will trace occurrences, so that for example not all x's in M_0 are treated alike. Hence we define

Sub(M_0) = $\{$M | M is an occurrence of a subexpression in $M_0\}$

Define CBN(M_0) to be the result of specializing CBN to a specific input M_0 as follows:

1. Closures and environments are redefined to be

$$\boxed{\begin{array}{l} CL = Sub(M_0) \times E \\ E \;\;= BV(M_0) \overset{\sim}{\rightarrow} CL \end{array}}$$

2. Real, Load, Unload and the transition relation $\Rightarrow$ are defined exactly as they were for CBN; However they are interpreted over the new E and C .

Clearly CBN(M_0) has a computation Load(M_0) = $\sigma_1 \Rightarrow \sigma_2 \Rightarrow \dots \Rightarrow \sigma_n$ with Unload(σ_n) = M if and only if CBN has a corresponding computation Load(M_0) = $\sigma_1' \Rightarrow \sigma_2' \Rightarrow \dots \Rightarrow \sigma_n'$ with Unload(σ_n') = M.

2. ANALYSIS OF CONTROL FLOW

It is well known that most forward program flow analysis methods essentially carry out an abstract interpretation of the program over a lattice whose elements approximate sets of states. Descriptions of this approach may be found in [Sin72] and [Cou77b]. Given a closed λ-expression M_0 without constants, we will use abstract interpretation to construct effectively a "safe" description of the computation by CBN(M_0). More precisely we show how to find a description of a superset of

$$\text{States}(M_0) = \{\sigma \mid \text{Load}(M_0) \overset{*}{\Rightarrow} \sigma \text{ by CBN}(M_0)\}$$

Overview

1. A finite set D of descriptions will be defined, along with a function Desc: $D \to \mathcal{P}(\Sigma)$. Each description $d \in D$ will represent a set of states Desc(d).

2. Define d to be closed if $\sigma_1 \in$ Desc(d) and $\sigma_1 \Rightarrow \sigma_2$ implies $\sigma_2 \in$ Desc(d). D will be given a lattice structure, and a form of abstract interpretation will be used to prove:
   Lemma (Simulation Lemma) There is an effective way to obtain from M_0 a non-trivial closed description d(M_0) such that Load(M_0) $\subseteq$ Desc(d(M_0)).
   Corollary (Safeness Theorem) States(M_0) $\subseteq$ Desc(d(M_0)).

3. It will be shown that d(M_0) may be analyzed to give "safe positive answers" to the following questions
   - will evaluation terminate?
   - will evaluation not terminate?
   - is M independent of N, for given subexpressions M, N of M_0?

Representation of CBN(M_0) Data Structures

The sets of closures, states,etc. are infinite so for effective approximation it is desirable to represent them finitely. We first develop a method to do this.

The sets of E and CL are infinite since defined by mutual recursion. Notice that during a computation every binding $e[x \to cl]$ occurs as the result of popping cl off the context stack during a β reduction. In fact CBN(M_0) could be easily modified to work with E = BV(M_0) $\overset{\sim}{\to}$ C, that is CL could be replaced by C.

Consequently a finite representation C' for C provides finite representations for all the other data structures of CBN(M_0), to wit:

Closures	: $cl' \in CL'$	$= Sub(M_0) \times E'$
Environments:	$e' \in E'$	$= BV(M_0) \tilde{\to} C'$
Interpreter States	: $\sigma' \in \Sigma'$	$= CL' \times C'$

The exact nature of C' turns out to be inessential to our development of methods for data representation and abstract interpretation, so we defer its choice to later. It is only important now to know that each token $c' \in C'$ will represent one or more context stacks.

It seems difficult at first sight to see how to represent an arbitrary context stack $c = cl_1 \ldots cl_n$ with unbounded n by a fixed finite token set C' in such a way that the structure of c can be retrieved; however this must be done to simulate β reduction. As often happens the problem turns out to be simpler if embedded in a larger problem – that of developing a finite description d of a possibly infinite set of states.

To this end we use in addition to C' an auxiliary <u>retrieval function</u> $r: C' \to \mathcal{P}(CL' \times C')$. The intention is that if token c_1' represents context stack $c_1 = cl_1 cl_2 \ldots cl_n$, then $r(c_1')$ will contain a pair (cl_1', c_2') where cl_1' represents cl_1 and c_2' represents $cl_2 \ldots cl_n$. The representation relations $c' \overset{r}{\sim} c$ and $cl' \overset{r}{\sim} cl$ are defined recursively since C, E and CL are also defined by mutual recursion.

<u>Definition</u> Let $r: C' \to \mathcal{P}(CL' \times C')$ be a retrieval function and let $\epsilon' \in C'$ be a designated token. The <u>representation relation</u> $\overset{r}{\sim} \subseteq C' \times C \cup E' \times E \cup CL' \times CL$ is defined as follows

a) $\epsilon' \overset{r}{\sim} \epsilon$ (i.e. token ϵ' represents the empty stack)

b) $c_1' \overset{r}{\sim} cl_1 cl_2 \ldots cl_n$ if $cl_1' \overset{r}{\sim} cl_1$ and $c_2' \overset{r}{\sim} cl_1 \ldots cl_n$ for some pair $(cl_1', c_2') \in r(c_1')$

c) $(M, e') \overset{r}{\sim} (M, e)$ if $e' \overset{r}{\sim} e$

d) $e' \overset{r}{\sim} e$ if for all $x \in Domain(e)$ there exists $(cl', c') \in r(e'(x))$ such that $cl' \overset{r}{\sim} e(x)$. □

<u>Example</u> Suppose $r(c') = \{(cl_1', \epsilon'), (cl_2', c')\}$ where $cl_1' = (M_1, \emptyset)$, $cl_2' = (M_2, \emptyset)$. Let $cl_1 = (cl_1')$ and $cl_2 = cl_2' \in CL$. Then

1. $\emptyset \overset{r}{\sim} \emptyset$ by d), so $cl_1' \overset{r}{\sim} cl_1$ and $cl_2' \overset{r}{\sim} cl_2$ by c).
2. $\epsilon' \overset{r}{\sim} \epsilon$ by a), so $c' \overset{r}{\sim} cl_1'$ by b).
3. Applying b) repeatedly, $c' \overset{r}{\sim} cl_2 cl_1$, $c' \overset{r}{\sim} cl_2 cl_2 cl_1$, etc.
4. $\emptyset[x \to cl_1] \overset{r}{\sim} \emptyset[x \to c']$ by d) and 2.

Remark For each $c' \in C'$ and r, the set $\{c \mid c' \overset{r}{\sim} c\}$ is essentially a regular set of trees (assuming C' is finite). In fact rules a)-d) may be regarded as productions in a regular tree grammar ([Eng75], [Tha73]) with nonterminal set C'.

Definition 1. A description is a pair $d = (S, r)$ where r is a retrieval function and $S \subseteq \Sigma' = CL' \times C'$. The set of states described by d is

$$Desc(d) = \{<cl, c> \mid cl' \overset{r}{\sim} cl \text{ and } c' \overset{r}{\sim} c \text{ for some } <cl', c'> \in S\}$$

2. Let $R = C' \to \mathcal{P}(CL' \times C')$ be the set of all retrieval functions and $D = \mathcal{P}(\Sigma') \times R$ the set of all descriptions. D may be made into a lattice (finite if C' is finite) by giving it the ordering $\sqsubseteq$ (where $\subseteq$ is subset inclusion):

$$(S_1, r_1) \sqsubseteq (S_2, r_2) \text{ iff } S_1 \subseteq S_2 \text{ and } \forall c' \in C' \ (r_1(c') \subseteq r_2(c')).$$

3. A description d is closed if $\sigma_1 \Rightarrow \sigma_2$ implies $\sigma_2 \in Desc(d)$ whenever $\sigma_1 \in Desc(d)$.

□

Remark Given a description $d = (S, r)$ it is not difficult to construct a context-free grammar which generates Desc(d) (environments are represented in a linear form). The technique is closely related to that of the previous remark; an example is found in [Jon81].

Abstract Interpretation of $CBN(M_0)$ over D

Our task is to find a closed description $d = (S, r)$ such that $Load(M_0) \in Desc(d)$. Suppose $\sigma_1 = <cl_1, c_1> \Rightarrow <cl_2, c_2> = \sigma_2$ where σ_1 is represented in d by $<cl_1', c_1'> \in S$ with $cl_1' \overset{r}{\sim} cl_1$ and $c_1' \overset{r}{\sim} c_1$.

If $\sigma_1 \Rightarrow \sigma_2$ by a variable expansion then $c_1 = c_2$, and if $\sigma_1 \Rightarrow \sigma_2$ by β reduction then c_1 is popped to give c_2. In either case the closure property can be maintained by adding appropriate pairs $<cl_2', c_2'>$ to S, where cl_2' and c_2' are easily determined from cl_1', c_1' and r.

If $cl_1 = (MN, e)$ then $\sigma_2 = <(M, e), (N, e) c_1>$, so a new token c_2' may be required to represent $c_2 = (N, e)c_1$; further the retrieval function r must be extended to retrieve representations of (N, e) and c_1 from c_2'. As before we defer discussion of the token set C' and of how c_2' is chosen, and simply write $c_2' = token(\sigma_1')$.

This informal description of the abstract interpretation is made precise in Figure 4. The lattice structure of D ensures that $d(M_0)$ is uniquely defined (the figure may be taken to define a function f: $D \to D$ whose least fixpoint is $d(M_0)$; f is clearly monotone and so continuous since D is finite, so its least fixpoint is well-defined).

Data structures

Closures	$cl' \in CL'$	$= Sub(M_0) \times E'$
Environments	$e' \in E'$	$= BV(M_0) \tilde{\to} C'$
Context Stacks	$c' \in C'$	= specified later
States	$\sigma \in \Sigma'$	$= CL' \times C'$
Descriptions	$d \in D$	$= \wp(\Sigma') \times R$
Retrieval function	$r \in R$	$= C' \tilde{\to} \wp(CL' \times C')$

The Description $d(M_0) = (S, r)$

$d(M_0)$ is the smallest element of D satisfying:

0. $<(M_0, \emptyset), \epsilon'> \in S$ (description of Load(M_0) in d(M_0))

1. "Scan operator" simulation:
   If $\sigma' = <(MN, e'), c'> \in S$
   then $<(M, e'), token(\sigma')> \in S$ and $((N, e'), c') \in r(token(\sigma'))$

2. "Variable expansion" simulation:
   If $<(x, e'), c'> \in S$ and $(cl', c_1') \in r(e'(x))$
   then $<cl', c'> \in S$

3. "β reduction" simulation:
   If $<(\lambda xM, e'), c'> \in S$ and $(cl', c_1') \in r(c')$
   then $<(M, e'[x \to c']), c_1'> \in S$

Figure 4. Abstract Interpretation of CBN(M_0)

Lemma 3 (Simulation Lemma). $d(M_0)$ is closed and Load(M_0) $\in$ Desc(d(M_0)).

Corollary 4 (Safeness Theorem). States(M_0) $\subseteq$ Desc(d(M_0)).

Proof of Lemma 3 is straightforward from the definition of $\overset{\Gamma}{\sim}$. Corollary 4 is immediate.

Representation of Context Stacks

Given the description $d(M_0) = (S, r)$, each token $c' \in C'$ represents a set of context stacks, namely $\{c \mid c' \overset{\Gamma}{\sim} c\}$. Intuitively it seems clear that the larger C' is, the more precise the abstract interpretation can be, since each token can represent a smaller set of context stacks. As one extreme we may get a precise simulation of CBN(M_0) at the expense of an infinite set of tokens:

Theorem 5 Suppose $C' = (CL')^*$, $\epsilon' = \epsilon$ and $\text{token}(\sigma') = (N, e')c'$ for all $\sigma' = \langle (MN, e'), c' \rangle \in \epsilon'$. Then $\text{States}(M_0) = \text{Desc}(d(M_0))$.

A more practical choice is the following:

1. $C' = \{\epsilon'\} \cup \{MN \mid MN \text{ is an occurrence of a combination in } M_0\}$
2. $\text{token}\langle (MN, e'), c' \rangle = MN$

With this approach flow information pertinent to all of the times the interpreter enters a state $\langle (MN, e), c \rangle$ is combined under the single token MN. This is analogous to conventional flow analysis, in which flow information pertinent to all times control passes through a given program point is associated with that point.

The following table illustrates the abstract interpretation of the $CBN(M_0)$ computation exemplified before where $M_0 = AB$ and $A = (\lambda xxx)(\lambda yy)$, $B = \lambda zz$.

Actual State Sequence		Simulated States		Retrieval Function
Control Closure	Context Stack	Control Closure	Token	
$\langle (AB, \emptyset),$	$\epsilon \rangle$	$\langle (AB, \emptyset),$	$\epsilon' \rangle$	
$\langle (A, \emptyset),$	$(B, \emptyset) \rangle$	$\langle (A, \emptyset),$	$AB \rangle$	$((B, \emptyset), \epsilon') \in r(AB)$
$\langle (\lambda xxx, \emptyset),$	$(\lambda yy, \emptyset)(B, \emptyset) \rangle$	$\langle (\lambda xxx, \emptyset),$	$A \rangle$	$((\lambda yyy, \emptyset), AB) \in r(A)$
$\langle (xx, \underbrace{\emptyset[x \to (\lambda yy, \emptyset)]}_{e_1}),$	$(B, \emptyset) \rangle$	$\langle (xx, \underbrace{\emptyset[x \to A]}_{e_1'}),$	$AB \rangle$	
$\langle (x, e_1),$	$(x, e_1)(B, \emptyset) \rangle$	$\langle (x, e_1'),$	$xx \rangle$	$((x, e_1'), AB) \in r(xx)$
$\langle (\lambda yy, \emptyset),$	$(x, e_1)(B, \emptyset) \rangle$	$\langle (\lambda yy, \emptyset),$	$xx \rangle$	
$\langle (y, \emptyset[y \to (x, e_1)]),$	$(B, \emptyset) \rangle$	$\langle (y, \emptyset[y \to xx]),$	$AB \rangle$	
$\langle (x, e_1),$	$(B, \emptyset) \rangle$	$\langle (x, e_1'),$	$AB \rangle$	
$\langle (\lambda yy, \emptyset),$	$(B, \emptyset) \rangle$	$\langle (\lambda yy, \emptyset),$	$AB \rangle$	
$\langle (y, \emptyset[y \to (B, \emptyset)]),$	$\epsilon \rangle$	$\langle (y, \emptyset[y \to AB]),$	$\epsilon' \rangle$	
$\langle (B, \emptyset),$	$\epsilon \rangle$	$\langle (B, \emptyset),$	$\epsilon' \rangle$	

Figure 5. Abstract Interpretation Example

Applications

Let a "safe positive reply" P to a questions whose answer is Q be one such that P logically implies Q. Given $N \in \text{Sub}(M_0)$ and $(M, e) \in CL$ define (M, e) to depend on N if either $M = N$ or for some $x \in FV(M)$, $e(x)$ depends on N. We say that M depends on N for $M, N \in \text{Sub}(M_0)$ if S contains a state $c[(M, e)]$ with (M, e) dependent on N.

Theorem 6. There is a decidable method to obtain nontrivial safe positive replies to the following questions about a closed constant-free-λ-expression M_0:

1. Is evaluation of $M \in Sub(M_0)$ never attempted? (Meaning: does $States(M_0)$ contain no state $\langle (M,e),c \rangle$?)
2. Will the computation terminate?
3. Will the computation fail to terminate?
4. Is $States(M_0)$ finite?
5. Is M independent of N (given $M, N \in Sub(M_0)$)?

<u>Proof</u> is by showing how to analyze the structure of $d(M_0) = (S,r)$. Question 1 is simple: if S contains no pair $\langle (M,e'),c' \rangle$, then $States(M_0)$ contains no state $\langle (M,e),c \rangle$ by Corollary 4 and the definition of $Desc(d(M_0))$.

Define the <u>flowchart</u> of M to be a directed graph with nodes in Σ' and edges $\sigma_1' \Rightarrow \sigma_2'$ just in case $\sigma_1' \in S$ implies $\sigma_2' \in S$ according to the rules of Figure 4. Clearly if we let the CBN(M_0) computation be $Load(M_0) = \sigma_0 \Rightarrow \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n$, there exists a corresponding path $\sigma_0' \rightarrow \sigma_1' \rightarrow \ldots \rightarrow \sigma_n'$ in the flow chart, such that if $\sigma_i = \langle cl, c \rangle$ then $\sigma_i' = \langle cl', c' \rangle$ for some cl', c' with $cl' \stackrel{\sqsubset}{\sim} cl$ and $c' \stackrel{\sqsubset}{\sim} c$.

If the computation is infinite there must exist σ' such that $\sigma_0' \stackrel{*}{\Rightarrow} \sigma' \stackrel{+}{\Rightarrow} \sigma'$, since Σ' is finite. This condition is certainly decidable, and its falsity implies the computation is finite. Question 3 may be answered "yes" if there is no path $\sigma_0' \stackrel{*}{\Rightarrow} \sigma'$ where $\sigma' \not\Rightarrow \sigma_1'$ for all σ_1'. Note that safe answers to questions 2 and 3 can both be "no".

Questions 4 and 5 may be answered by analysis of $d(M_0)$ – see [Jon81] for details. □

<u>Remarks on the Method</u>

The token set C' may be "tuned" to give varying degrees of faithfulness in the approximation, even including exact execution by Theorem 5. This desirable property of a flow analysis method is unfortunately not shared by most other interprocedural methods (but [Cou77a] is an exception).

The method with the finite C' mentioned above could be inefficient on some λ-expressions with complex reduction sequences due to the size of the set E' of simulated environments. A more practical approach could involve merging together all the environments associated with a single "control point" in $Sub(M_0)$. This could be accomplished by replacing the subset $S \subseteq \Sigma' = Sub(M_0) \times E' \times C'$ by the function $S: Sub(M_0) \rightarrow PE \times \mathcal{P}(C')$ where $PE = BV(M_0) \rightarrow \mathcal{P}(C')$, and defining abstract interpretation rules accordingly. This would reduce the worst-case storage requirement for S from an exponential function (of the size of M_0) to a polynomial, at the expense of some precision in simulation. This version would not, however, allow Theorem 5 to be proved.

3. ANALYSIS OF DATA FLOW

The method of Section 2 can be extended to include λ-expressions with constants.

Due to space limitations we only give an overview here; [Jon81] contains a more complete treatment for call-by-value evaluation.

I The CBN interpreter is extended to perform δ reductions as follows

a) Suppose the control closure (MN, e) is reached and that (M, e) evaluates to a constant (a, e_1). Then (N, e) must also be evaluated; meanwhile a will be put on the context stack for safekeeping (so C = (CL + Con)*). Thus we add to Figure 1 the transition rule:

$$<(a, e_1), (N, e) z_1 \ldots z_n> \Rightarrow <(N, e), a z_1 \ldots z_n>$$

b) Perform δ reduction if (N, e) evaluates to a constant (b, e_2). Add the transition rule

$$<(b, e_2), a z_1 \ldots z_n> \Rightarrow <(Cap(a,b), \emptyset), z_1 \ldots z_n> \text{ (if Cap(a,b) is defined)}$$

c) If (N, e) evaluates to a non-constant value or Cap(a, b) is undefined, transit to state $<(ERROR, \emptyset), \epsilon>$ (ERROR is assumed to be a special element of Con).

II An analog of Lemma 2 holds: if $Load(M_0) \overset{*}{\Rightarrow} <(M, e), c>$ then M is either a constant or a subexpression of M_0 (or both).

III A constant approximation method is needed since Con will usually be infinite. This can be done along the lines of [Cou79]:

a) Con' will be a lattice of finite height whose elements represent sets of constants via an <u>abstraction function</u> abs: $\mathcal{P}(Con) \to Con'$ and a <u>concretization function</u> conc: $Con' \to \mathcal{P}(Con)$. Desirable properties of abs and conc are found in [Cou79].

b) The constant application function Cap is approximated by a function Cap': $Con' \times Con' \to Con'$ satisfying for all $a', b' \in Con'$ ($\sqsubseteq$ is the lattice order on Con'):

$$Cap'(a', b') \sqsupseteq abs\{Cap(a,b) \mid a \in conc(a') \text{ and } b \in conc(b')\}$$

IV A description lattice D approximating sets of states may be defined using Con'.

V A set of rules similar to those of Figure 4 can be defined; their effect is to abstractly simulate the transition rules over D.

Finally, it appears straightforward to extend the methods of Theorem 6 to obtain effectively safe positive answers to the five questions stated there, plus:

6. Is the computation free of error halts?

7. Is a given variable occurrence bound only to a single constant value? (If so, its value can be obtained.)

4. CONCLUSIONS AND ACKNOWLEDGEMENTS

It has been shown that safe answers may be effectively obtained to a variety of questions about call-by-name reduction sequences including finiteness, termination, freedom of errors, and independence of subexpressions. The methods used are clearly applicable to call-by-value; further since abstract interpretation does not depend on determinism it seems likely that similar methods could be used to give information about arbitrary reduction sequences. One application would be to determine from the flow analytic information a combination of call-by-value and call-by-need which have the same termination properties as call-by-name but allow a more efficient implementation, extending results of Mycroft [Myc80].

The analysis method applied the classical flow-analytic idea of abstract interpretation to a call-by-name interpreter CBN. This application required a new description technique involving both local and global data representations due to the recursiveness of CB's data structures. The technique is applicable to many programs which manipulate tree-like data structures; it is anticipated that it can be used to develop practical interprocedural flow analysis methods for more conventional imperative programming languages. Another application would be the development of compiling methods for applicative languages capable of producing highly efficient object code.

Discussions with Flemming Nielson, David Schmidt, Peter Mosses, Mogens Nielsen and Steven Mùchnick on various aspects of this work have been very helpful.

References

Aho77 Aho, Alfred V., and Jeffrey D. Ullman, Principles of Compiler Design, Reading, MA: Addison-Wesley, 1977.

Bjø78 Bjørner, Dines and Cliff B. Jones, The Vienna Development Method: The Meta-Language, Lecture Notes in Computer Science 61 (1978).

Böh72 Böhm, Corrado and Mariangiola Dezani, "A CUCH-Machine: The automatic Treatment of Bound variables", Int. J. Comp. Info. Sci. vol. 1, no. 2 (1972), 171-291.

Cou77a Cousot, Patrick and Radhia Cousot, "Static Determination of Dynamic Properties of Recursive Procedures", IFIP Working Conf. on Prog. Concepts, St. Andrews, Canada, North-Holland (1978), 237-277.

Cou77b Cousot, Patrick and Radhis Cousot, "Abstract Interpretation. A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints", Conf. Rec. of 4th ACM Symp. on Principles of Programming Languages, Los Angeles, CA (January 1977).

Cou79 Cousot, Patrick and Radhis Cousot, "Systematic Design of Program Analysis Frameworks", Conf. Rec. 6th ACM Symp. on Principles of Programming Languages, San Antonio, TX (January 1979), 269-282.

Cur58 Curry, Haskell B. and R. Feys, Combinatory Logic vol. 1, North-Holland, Amsterdam (1958).

Eng75 Engelfriet, Joost, "Tree Automata and Tree Grammars", DAIMI Report FN-10, Computer Science Department, Aarhus University (1975).

Hec77 Hecht, Matthew S., Flow Analysis of Computer Programs. New York: Elsevier North-Holland, 1977.

Jon81 Jones, Neil D., "Flow Analysis of Lambda Expressions", DAIMI PB-128, Technical Report, Aarhus University, Denmark (1981), 31 pp.

JoM81 Jones, Neil D. and Steven S. Muchnick, "Flow Analysis and Optimization of LISP-like Structures", in Program Flow Analysis, S.S. Muchnick and N.D. Jones (eds.), Prentice-Hall (1981).

JoS80 Jones, Neil D. and David A. Schmidt, "Compiler Generation from Denotational Semantics", in Semantics-Directed Compiler Generation, Lecture Notes in Computer Science 94 (1980), 70-93.

Lan64 Landin, P.J., "The Mechanical Evaluation of Expressions", Computer Journal vol. 6, no. 4 (1964).

Lev76 Levy, J.J., "An Algebraic Interpretation of the $\lambda\beta$k-calculus and an Application of a labelled λ-calculus", Theor. Comp. Sci. vol. 2 no. 1 (1976), 97-114.

McC63 McCarthy, J., "Towards a Mathematical Science of Computation" in Information Processing, North-Holland (1963).

McG70 McGowan, C., "The Correctness of a modified SECD Machine", Second ACM Symposium on Theory of Computation (1970).

Myc80 Mycroft, Alan, "The Theory and Practice of Transforming Call-by-need into Call-by-value", Internl. Symp. on Programming, LNCS 83 (1980), 269-281.

Ple81 Pleban, Uwe, "A Denotational Semantics Approach to Program Optimization", Ph.D. Dissertation, Univ. of Kansas, Lawrence, KS (1981).

Plo75 Plotkin, Gordon D., "Call-by-Name, Call-by-Value and the Lambda Calculus", Theor. Comp. Sci. 1 (1975), 125-159.

Rey72 Reynolds, John, "Definitional Interpreters for Higher-Order Programming Languages", Proc. ACM National Meeting (1972).

Ros79 Rosen, Barry K., "Data Flow Analysis for Procedural Languages", J. ACM, 26, no. 2 (April 1979), 322-344.

Sch80 Schmidt, D.A., "State Transition Machines for Lambda Calculus Machines", in Semantics-Directed Compiler Generation, Lecture Notes In Computer Science 94 (1980), 415-440.

Sha805 Sharir, M., "Data Flow Analysis of Applicative Programs", Courant Inst. Tech. Rep. 80-42, Columbia Univ., New York (1980). Also these proc's.

ShP81 Sharir, M. and A. Pnueli, "Two Approaches to Interprocedural Data Flow Analysis", Program Flow Analysis, S.S. Muchnick and N.D. Jones eds., Prentice-Hall (1981).

Sin72 Sintzoff, M., "Calculating Properties of Programs by Valuation on Specific Models", Proc. ACM Conf. on Proving Assertions About Programs, New Mexico (1972), 203-207.

Ste75 Steele, G.L. Jr., "SCHEME: An Interpreter for the Extended Lambda Calculus", AI Memo 349 (Dec. 1975), Artificial Intel. Lab., MIT.

Tha73 Thatcher, J.W., "Tree Automata: An Informal Survey", in Currents in the Theory of Computing, ed. A. Aho. Prentice-Hall, (1973), 143-172.

Tur76 Turner, D.A., SASL Language Manual, U. of St. Andrews, Fife, Scotland (1976).

Weg68 Wegner, P., Programming Languages, Information Structures and Machine Organization, McGraw-Hill, New York (1968).