ISSN 0105-8517

FLOW ANALYSIS OF LAMBDA EXPRESSIONS

by

Neil D. Jones

DAIMI PB-128
January 1981

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

FLOW ANALYSIS OF LAMBDA EXPRESSIONS

Neil D. Jones
Aarhus University, Denmark

0. INTRODUCTION

Overview

Program flow analysis determines properties of the computation(s) induced by a program without actually running it. The purpose is usually to extract information which may be used to optimize the program, compile code, certify the absence of certain runtime errors, etc. Excellent introductions to flow analysis may be found in [Hec77] and [Aho77].

We describe a method to analyze the data and control flow during mechanical evaluation of lambda expressions. The method produces a finite approximate description of the set of all states entered by a call-by-value λ-calculus interpreter; a similar approach can easily be seen to work for call-by-name. A proof is given that the approximation is "safe", i.e. that it includes descriptions of every intermediate λ-expression which occurs in the evaluation.

From a programming languages point of view the method extends previously developed interprocedural analysis methods to include both local and global variables, call-by-name or call-by-value parameter transmission and the use of procedures both as arguments to other procedures and as the results returned by them.

The main emphasis is on development of the flow analysis framework rather than on applications, although a few are given (termination, finiteness, dependence, constant propagation). Other familiar analyses easily fit into the same framework, e.g. available expressions and deciding whether a base function's arguments always have the right type.

The information gathered could be used to compile unusually efficient code for programming languages based on the λ-calculus such as LISP and SCHEME [Ste76]. Hopefully it will be possible to extend these methods to the flow analysis of denotational definitions. The use of such analyses in compiler generation was described in [JoS80] and provided the initial motivation for this study.

The methods developed here are not limited to the λ-calculus, but may be applied to any programming language whose semantics are specified by (or specifiable) by a definitional interpreter using recursively defined data structures. The λ-calculus was chosen because of the challenge of tracing control and data flow in a computation; it provides a "worst-case" example of many problems encountered in interprocedural analysis.

Related Work

Lambda calculus machines include the SECD and CUCH machines of Landin and Böhm ([Lan64], [Böh72]) and those due to Reynolds, Wegner and McGowan ([Rey72], [Weg68], [McG70]). McGowan and Plotkin ([McG70], [Plo75]) have proved correctness of their machines, and Plotkin further investigates a number of questions concerning call-by-name and call-by-value.

Interprocedural flow analysis has been investigated by (among others) Rosen, Cousot and Cousot, and Sharir and Pnueli ([Ros79], [Cou77], [Sha80]). Levy has developed sufficient conditions for termination of β-reduction sequences in [Lev75], and Mycroft ([Myc80]) developed sufficient conditions for the replacement of call-by-need by call-by-value in schemes of recursion equations using the flow analytic idea of "abstract interpretation". Pleban is currently doing a flow analysis of the SCHEME language, expressed using denotational semantics [Ple80].

Outline of the Paper

In the first section we introduce the OI interpreter, a nondeterministic machine which can perform an arbitrary sequence of outside-in β- and δ-reductions on a closed lambda expression. The initial state of the OI interpreter for input M_0 will be Load (M_0), and every OI state σ will represent a λ-expression Unload (σ). A computation Load $(M_0) = \sigma_0 \Rightarrow \sigma_1 \Rightarrow \sigma_2 \Rightarrow \dots$ will correspond to a series of β-reductions, δ-reductions or identity transformations.

There are two reasons for introducing yet another λ-calculus interpreter. First, the OI machine can naturally be restricted to yield deterministic call-by-name and call-by-value submachines CBN and CBV. While we analyze only CBV (because of its similarity with existing programming languages) it will be apparent that CBN can be analyzed by the same methods. Second, both CBV and CBN seem to be significantly simpler than other λ-calculus interpreters, which in turn simplifies our model-building process. The appendix* contains correctness proofs for these machines.

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 of this is a safe description of

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

---

* Appendix omitted in this preliminary version.

A lattice D will be defined whose elements δ will each describe a set of states. An effectively computable method will be described to obtain from M_0 a description $\delta(M_0) \in D$. It will be proven that this description is "safe" in the sense that every state $\sigma \in$ States (M_0) is represented in $\delta(M_0)$. Safeness implies that answers to questions about the computation which can be answered by examination of $\delta(M_0)$ can at worst err "on the safe side", since every state in States (M_0) is accounted for. However precise answers cannot be given to all such questions since some (such as the halting problem) are undecidable. Technically this occurs since D is finite, with size recursively bounded in the size of M_0, so some information must be lost.

Section 2 concludes by constructing a context-free grammar $G(M_0)$ which generates linear representations of all σ in States (M_0), followed by a proof that safe answers may be computably obtained to several questions about the computation.

Section 3 extends the method to handle λ-expressions with constants and δ-reduction, using an approximation lattice to describe effectively sets of constant values. An example is given which is suitable for constant propagation, i.e. to find out which variables only receive constant values, and what those values are. Applications to error-checking and type-correctness of base functions are considered.

Section 4 ends with conclusions, future directions and acknowledgments.

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 partial functions from X to Y. Given $f \in (X \overset{P}{\Rightarrow} Y)$, Domain(f) is its domain. Two functions in $X \overset{P}{\Rightarrow} Y$ are equal iff their domains are equal and they have the same values on arguments for which they are defined. If $x \in X$, $y \in Y$ and $f \in X \overset{P}{\Rightarrow} Y$ then $f\{y/x\}$ denotes the unique partial function f' such that $f'(z) = f(z)$ if $z \neq x$ and $f'(x) = y$.

A function $f \in X \overset{P}{\Rightarrow} Y$ with Domain(f) $= \{x_1, \ldots, x_n\}$ may be written as $\{x_1 \to f(x_1), \ldots, x_n \to f(x_n)\}$. The totally undefined function is written $\{\ \}$.

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

Inductive definitions will be written in the style of the abstract syntax of McCarthy [McC63]; for example binary lists can be defined by List ::= Atom | List List where a list in List may be thought of as an abstract syntax tree. This notation will be extended to encompass sets of partial functions, e.g. E ::= Var $\overset{P}{\to}$ Cl where Var is a countable set of variables. This may also be viewed as defining abstract syntax trees, with at most one subtree of type Cl for each variable. The notation is similar to VDM notation [Bjø78].

The Lambda Calculus

Given predefined disjoint sets Var = $\{x, y, z, \ldots\}$ and Con = $\{a, b, c, \ldots\}$ of variables and constants respectively, the set of λ-calculus terms Lam = $\{M, N, \ldots\}$ is the smallest set such that

(1) Any variable or constant is in Lam.

(2) If x is a variable and M is in Lam then the abstraction λxM is in Lam.

(3) If M and N are in Lam then the combination MN is also in Lam. M is its rator and N is its rand.

A term is a value if it is not a combination.

This inductive definition may also be written as follows, using abstract syntax:

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

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]. Plotkin calls a closed term a program.

The set of contexts $c[\ ]$ is defined by

$$\text{Ctx} ::= [\ ] \mid \text{Ctx Lam} \mid \text{Lam Ctx} \mid \lambda\ \text{Var Ctx}$$

A context may be viewed as a lambda expression with a "hole" $[\ ]$ in it. Noting that a context would be a λ-expression if $[\ ]$ were regarded as a variable, we define $c[M]$ to be the result of "filling the hole" in context $c[\ ]$ by term M.

Now supposing we are given a partial function

$$\text{Constapply} : \text{Con} \times \text{Con} \overset{P}{\rightarrow} \text{Closed Values}$$

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

Note that if M is closed and $M > N$ without α reduction then no renaming occurs. Define an <u>outside-in context</u> to be one formed without the rule Ctx ::= λ Var Ctx, so the "hole" is not in the scope of any λ. We write $M \underset{oi}{>} N$ if $M > N$ by β or δ reduction, possibly in an outside-in context.

A machine-independent definition (from [Plo75]) of <u>call-by-value evaluation</u> is given by the partial function eval_V: Programs $\overset{P}{\to}$ Programs defined recursively as follows:

$$\text{eval}_V(a) = a; \qquad \text{eval}_V(\lambda x M) = \lambda x M;$$

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

The call-by-name evaluation function eval_N: Lam $\overset{P}{\to}$ Lam is similarly defined:

$$\text{eval}_N(a) = a; \qquad \text{eval}_N(\lambda x M) = \lambda x M;$$

$$\text{eval}_N(MN) = \begin{cases} \text{eval}_N([N/x]M') & \text{if } \text{Eval}_N(M) = \lambda x M' \\ a' & \text{if } \text{eval}_N(M) = a,\ \text{eval}_N(N) = 6 \text{ and} \\ & \text{Constapply}(a, b) = a' \text{ is defined} \end{cases}$$

It is shown in [Plo75] that these are good definitions of partial functions. Note that both could have been restricted to Programs $\overset{P}{\to}$ Closed values. The following is easily shown.

<u>Lemma</u> 0.1 If M is closed and $\text{eval}_V(M)$ is defined then $M \overset{*}{\underset{oi}{>}} \text{eval}_V(M)$; and similarly for $\text{eval}_N(M)$.

1. LAMBDA CALCULUS INTERPRETERS

We first introduce a nondeterministic interpreter which can do arbitrary outside-in reduction sequences and some lemmas about its behaviour. This machine is then restricted to yield two deterministic interpreters CBV and CBN which are proved to perform correctly call-by-value and call-by-name reductions. The terminology and methods of this section owe much to [Plo75].

<u>The OI interpreter</u>

The OI interpreter is given by a set Σ of states and a binary transition relation $\underset{oi}{\Rightarrow}$ on Σ; its data structures and transition rules are summarized in Figure 1. Auxiliary functions Load: Closed terms $\to \Sigma$ and Unload: $\Sigma \to$ Lam are used to initialize the machine and to read out the λ-expression denoted by a state.

Data Structures

To avoid explicit substitutions into λ-expressions (in the interest of efficiency) an expression will be represented in an interpreter state by a closure of the form (M, e) where M is a term and $e \in E$ is an environment binding its free variables (if any) to other closures. Further, a closure may take the form $cl_1 cl_2$, i.e. a combination of two closures. The function Real: $Cl \to Lam$ mapping the set Cl of all closures into the λ-expressions they denote is given as follows.

In this paper every closure (M, e) will satisfy $FV(M) \subseteq Domain(e)$, so $Real(cl)$ will always be closed.

$$\begin{aligned} Real(cl_1 cl_2) &= Real(cl_1)\ Real(cl_2) \\ Real((M, e)) &= [Real(e(x_1))/x_1] \ldots [Real(ex_n))/x_n]M \\ &\quad \text{where } Domain(e) = \{x_1, \ldots, x_n\} \end{aligned}$$

To extend the idea of outside-in contexts to apply to closures we define the set of Cl-contexts by $C ::= [\ ] \mid C\ Cl \mid Cl\ C$. The notations $c[[\ ]cl]$ and $c[cl[\ ]]$ will have many uses; they denote the Cl-contexts obtained by replacing the single occurrence of $[\ ]$ in c by the Cl-contexts $[\ ]cl$ and $cl[\ ]$, respectively.

A state σ in Σ may be viewed intuitively as a closure in which a particular subclosure has been identified to be processed next by the interpreter. Formally σ is a pair consisting of a Cl-context c and a closure cl, written in the suggestive notation $c[cl]$ (which does not indicate substitution). If $c = c_1[cl_1[\ ]]$ then $c[cl]$ may also be written $c_1[cl_1[cl_2]]$, and similarly $c_1[[\ ]cl_2[cl_1]]$ may be written $c_1[[cl_1]cl_2]$.

Load: Programs $\to \Sigma$ and Unload: $\Sigma \to Lam$ are now defined, using $\{\ \}$ for the initial environment with empty domain. Note that $Unload(c[cl])$ may be seen as the result of substituting $Real(cl)$ into the context naturally obtained from c.

$$\begin{aligned} Load(M_0) &= [(M_0, \{\ \})] \\ Unload([cl]) &= Real(cl) \\ Unload(c[cl_1[cl_2]]) &= Unload(c[[cl_1]cl_2]) = Unload(c[cl_1 cl_2]) \end{aligned}$$

Transition Rules

Figure 1 contains the transition rules defining $\underset{oi}{\Rightarrow}$ and repeats the definitions of Cl, E etc. in more compact form.

Data Structures

cl	: Cl	::=	Lam E | Cl Cl	Closures
e	: E	::=	Var $\xrightarrow{P}$ Cl	Environments
c[ ]	: C	::=	[ ] | C Cl | Cl C	Cl-contexts
σ	: Σ	::=	C[Cl]	States

Transition Rules

1.	$c[[(\lambda x M, e)]cl]$	$\underset{oi}{\Rightarrow}$	$c[(M, e\{cl/x\})]$	β reduction
2.	$c[[(a, e)](b, e')]$	$\underset{oi}{\Rightarrow}$	$c[(a', \{\ \})]$	δ reduction (if a' = Constapply(a,b) is defined)
3.	$c[(x\ e)]$	$\underset{oi}{\Rightarrow}$	$c[e(x)]$	variable expansion (if e(x) is defined)
4.	$c[(MN, e)]$	$\underset{oi}{\Rightarrow}$	$c[(M, e)(N, e)]$	combination
5.	$c[cl_1 cl_2]$	$\underset{oi}{\Rightarrow}$	$c[[cl_1]cl_2]$	scan rator
6.	$c[[cl_1]cl_2]$	$\underset{oi}{\Rightarrow}$	$c[cl_1 cl_2]$	return from rator
7.	$c[cl_1 cl_2]$	$\underset{oi}{\Rightarrow}$	$c[cl_1[cl_2]]$	scan rand
8.	$c[cl_1[cl_2]]$	$\underset{oi}{\Rightarrow}$	$c[cl_1 cl_2]$	return from rand

Figure 1. OI Interpreter.

Example Computation

Following is an example OI computation on $M_0 = (\lambda\, ff7)$square:

Load(M_0) =	$[((\lambda ff7)\text{square}, \{\ \})]$	combination
$\underset{oi}{\Rightarrow}$	$[(\lambda ff7, \{\ \})(\text{square}, \{\ \})]$	scan rator
$\underset{oi}{\Rightarrow}$	$[[(\lambda ff7, \{\ \}](\text{square}, \{\ \})]$	β reduce
$\underset{oi}{\Rightarrow}$	$[(f7, \underbrace{\{f \rightarrow (\text{square}, \emptyset)\}}_{\text{call this } e})]$	scan rator
$\underset{oi}{\Rightarrow}$	$[[(f, e)](7, e)]$	expand "f"
$\underset{oi}{\Rightarrow}$	$[[(\text{square}, \{\ \})](7, e)]$	δ reduce
$\underset{oi}{\Rightarrow}$	$[(49, \{\ \})]$	

Mathematical Justification

The following provides the mathematical justification of the OI interpreter; its proof is straightforward but detailed and so appears in the appendix.

Theorem 1.1

a) If $\sigma_1 \overset{*}{\underset{oi}{\Rightarrow}} \sigma_2$ and Unload(σ_1) is closed then Unload(σ_1) $\overset{*}{\underset{oi}{>}}$ Unload(σ_2)

b) If M $\overset{*}{\underset{oi}{>}}$ N and M is closed then Load(M) $\overset{*}{\underset{oi}{\Rightarrow}}$ σ for some σ with Unload(σ) = N.

A similar interpreter has been constructed and proven correct which can do arbitrary reduction sequences (not just outside-in), at the expense of more complexity in handling environments. A deterministic restriction of it could provide an alternative to the CUCH machine [Böh72], but is omitted due to the difficulties encountered in approximating a renaming interpreter.

Call-by-Value and Call-by-Name Interpreters

The OI interpreter may be simplified and made deterministic by imposing a consistent ordering on operator and operand evaluation. If the operand is always left unevaluated we have the usual implementation of call-by-name. Figure 2 contains the CBN interpreter; it was obtained from OI by combining transition rules 4 and 5, dropping 2, 6, 7, 8 and simplifying closures by omitting Cl ::= Cl Cl. This machine has been studied by Schmidt [Sch81].

Define $\mathrm{Last}_N : \Sigma \overset{P}{\to} \Sigma$ and Eval_N : Lam $\overset{P}{\to}$ Lam by:

$$\mathrm{Last}_N(\sigma) = \begin{cases} \mathrm{Last}_N(\sigma') & \text{if } \sigma \underset{cbn}{\Rightarrow} \sigma' \text{ for some } \sigma'; \\ \sigma & \text{otherwise} \end{cases}$$

$$\mathrm{Eval}_N(M_0) = \mathrm{Unload}(\mathrm{Last}_N(\mathrm{Load}(M_0)))$$

Theorem 1.2

$\mathrm{Eval}_N(M_0) = \mathrm{eval}_N(M_0)$ for all closed constant-free terms M_0.

Thus CBN correctly performs call-by-name evaluation. Proof is omitted; a very similar proof for call-by-value is found in the appendix. Constants are omitted for simplicity, and because CBN will not be studied in detail. They could be handled by adding transition rules to evaluate the operand in case the operator value is a constant, plus a δ-reduction rule. A simpler alternative (from [Sch81]) is to require that constant functions be applied in postfix order instead of prefix.

Data Structures

$cl : Cl$	$::=$	$\mathrm{Lam}\ E$	Closures
$e : E$	$::=$	$\mathrm{Var} \xrightarrow{P} Cl$	Environments
$c[\ ] : C$	$::=$	$[\ ] \mid C\ Cl \mid Cl\ C$	Cl-contexts
$\sigma : \Sigma$	$::=$	$C[Cl]$	States

Transition Rules

1.	$c[[(\lambda x M, e)]cl]$	$\underset{cbn}{\Rightarrow}$	$c[(M, e\{cl/x\})]$	β reduction
2.	$c[(x, e)]$	$\underset{cbn}{\Rightarrow}$	$c[e(x)]$	variable expansion (if $e(x)$ is defined)
3.	$c[(MN, e)]$	$\underset{cbn}{\Rightarrow}$	$c[[(M, e)](N, e)]$	combination

Figure 2. CBN Interpreter.

Data Structures

$cl : Cl$	$::=$	$\mathrm{Lam}\ E$	Closures
$e : E$	$::=$	$\mathrm{Var} \xrightarrow{P} Cl$	Environments
$c[\ ] : C$	$::=$	$[\ ] \mid C\ Cl \mid Cl\ C$	Cl-contexts
$\sigma : \Sigma$	$::=$	$C[Cl]$	States

Transition Rules

1.	$c[[(\lambda x M, e)]cl]$	$\Rightarrow$	$c[(M, e\{cl/x\})]$	β reduction
2.	$c[[(a, e)](b, e_1)]$	$\Rightarrow$	$c[(a', \{\ \})]$	δ reduction (if $a' = \mathrm{Constapply}(a, b)$ is defined)
3.	$c[(x, e)]$	$\Rightarrow$	$c[e(x)]$	variable expansion (if $e(x)$ is defined)
4.	$c[(MN, e)]$	$\Rightarrow$	$c[(M, e)[(N, e)]]$	combination
5.	$c[cl_1[cl_2]]$	$\Rightarrow$	$c[[cl_1]cl_2]$	scan operator

(if $cl_2 = (M, e)$ where M is a closed value)

Figure 3. CBV Interpreter.

For call-by-value we evaluate both operator and operand before β or δ reduction. The result is in Figure 3; it was obtained from OI by combining transition rules 4 and 7, and applying 8 followed by 5 if operand evaluation produced a value. Rule 6 is omitted since either β or δ reduction must occur after operator evaluation. Define $\mathrm{eval}_V(M_0) = \mathrm{Unload}(\mathrm{Last}(\mathrm{Load}(M_0)))$ where

$$\mathrm{Last}(\sigma) = \begin{cases} \mathrm{Last}(\sigma') \text{ if } \sigma \Rightarrow \sigma' \\ \sigma \qquad\qquad \text{if } \sigma \not\Rightarrow \sigma' \text{ for all } \sigma' \text{ and } \sigma = [\mathrm{cl}] \text{ for some cl} \end{cases}$$

Proof of the following is found in an appendix.

Theorem 1.3

$\mathrm{Eval}_V(M_0) = \mathrm{eval}_V(M_0)$ for all closed terms M_0.

Corollary 1.4

CBV is computationally equivalent to the SECD machine.

Proof Plotkin has shown that SECD computes eval_V in [Plo75].

A Useful Property

In every closure (M, e) which was obtained in the example computation, M was a subexpression of M_0 or a constant. This is in fact always true.

Lemma 1.5 Suppose M is closed and $\mathrm{Load}(M_0) \overset{*}{\underset{\mathrm{OI}}{\Rightarrow}} c[(M, e)]$. Then

a) $\mathrm{Domain}(e) \subseteq \mathrm{BV}(M_0)$ and

b) M is a subexpression either of M_0 or of Constapply(a, b) for some $a, b \in \mathrm{Con}$.

Proof Define "p appears in cl" for closures cl and λ-expressions or environments p as follows:

i) M and e appear in (M, e)

ii) if p appears in e(x) for some $x \in \mathrm{Domain}(e)$ then p appears in (M, e)

iii) if p appears in cl_1 or cl_2 then it appears in $\mathrm{cl}_1\mathrm{cl}_2$.

An easy induction on n now verifies that if $\mathrm{Load}(M_0) \overset{n}{\underset{\mathrm{OI}}{\Rightarrow}} c[\mathrm{cl}]$ and p appears in cl or any closure in c, then p satisfies a) or b) above.

□

Lemma 1.5 implies that for each fixed input M_0 we may regard OI as operating on occurrences of expressions 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 an OI λ-calculus machine can manipulate pointers instead of arbitrary λ-expressions. Incidentally, the SECD machine also has this property.

The approximations to be developed later will trace occurrences (so all x's are not treated alike, for instance), so we introduce some terminology.

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

$$\mathrm{Subcon} = \{M \mid M \text{ is an occurrence of a subexpression in } N = \mathrm{Constapply}(a,b), \text{ where } a,b \in \mathrm{Con} \text{ and } N \notin \mathrm{Con}\}$$

$$\mathrm{Lam}(M_0) = \mathrm{Sub}(M_0) \cup \mathrm{Subcon} \cup \mathrm{Con}$$

The specialization of the OI interpreter to M_0 is written $\mathrm{OI}(M_0)$ and defined in Figure 4. The same concept will also be applied to the call-by-value interpreter CBV yielding its specialized form $\mathrm{CBV}(M_o)$.

Clearly $\mathrm{OI}(M_0)$ has a computation $\sigma_1 \Rightarrow \sigma_2 \Rightarrow \ldots \Rightarrow \sigma_n$ with $\mathrm{Unload}(\sigma_n) = M$ if and only if OI has a corresponding computation $\mathrm{Load}(M_0) = \sigma_1' \Rightarrow \sigma_2' \Rightarrow \ldots \Rightarrow \sigma_n'$ with $\mathrm{Unload}(\sigma_n') = M$.

Data Structures

$cl : Cl$	$::=$	$\mathrm{Lam}(M_0)\ E \mid Cl\ Cl$	Closures
$e : E$	$::=$	$BV(M_0) \overset{P}{\to} Cl$	Environments
$c[\ ] : C$	$::=$	$[\ ] \mid C\ Cl \mid Cl\ C$	Cl-contexts
$\sigma : \Sigma$	$::=$	$C[Cl]$	States

Transition Rules

Identical to those of OI, but regarded as operating on elements of $\mathrm{Lam}(M_0)$.

Figure 4. $\mathrm{OI}(M_0)$ Interpreter.

2. ANALYSIS OF CONTROL FLOW

Let M_0 be a given closed λ-expression without constants, and let $\mathrm{CBV}(M_0)$ be the CBV interpreter specialized to M_0. Define

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

Note that $\mathrm{CBV}(M_0)$ can only halt by entering a state of the form $[(\lambda x M, e)]$, so no "error halts" are possible. It will be shown that a safe description $\delta(M_0)$ may be effectively obtained as follows.

1. A method will be developed to represent finitely the data structures of $CBV(M_0)$, yielding a finite lattice D containing computation descriptions δ.

2. For each $\delta \in D$ a representation relation $\overset{\delta}{\sim} \subseteq \Sigma \times \Sigma'$ will be defined, where Σ' models the states of $CBV(M_0)$. The relation $\sigma \overset{\delta}{\sim} \sigma'$ will mean that state σ is represented by $\sigma' \in \Sigma'$ in the computation description δ.

3. A continuous simulation function $f: D \to D$ will be defined satisfying

Lemma 2.1 If $\sigma_1 \Rightarrow \sigma_2$ and $\sigma_1 \overset{\delta}{\sim} \sigma_1'$ then $\sigma_2 \overset{f(\delta)}{\sim} \sigma_2'$ for some $\sigma_2' \in \Sigma'$.

4. Safeness will be shown by the following, where $\delta(M_0)$ is the least element of D which describes $Load(M_0)$ and is a fixpoint of f.

Theorem 2.2

If $\sigma \in States(M_0)$ then $\sigma \overset{\delta(M_0)}{\sim} \sigma'$ for some $\sigma' \in \Sigma'$.

5. A linear encoding le of states will be introduced, and a context-free grammar $G(M_0)$ will be constructed from $\delta(M_0)$ will be constructed from $\delta(M_0)$ such that

$$L(G(M_0)) \supseteq \{ le(\sigma) \mid \sigma \in States(M_0) \}$$

6. Effective methods will be developed to give "safe" positive answers to the following questions

- Is a subexpression of M never evaluated?
- Will the computation terminate?
- Is $States(M_0)$ finite?
- Is M independent of N, for subexpressions M, N of M_0?

The Description Lattice D

The sets of closures, contexts etc. of $CBV(M_0)$ are infinite, so for effective approximation it is desirable to represent them finitely. We first develop informal methods for finite representation and then give a mathematical definition of what a representation is. The data structure representations (and more) are summarized in Figure 5.

The sets E and Cl are infinite, since defined by mutual recursion. However, notice that every value $e(x) \in Cl$ comes from a Cl-context as the result of β-reduction; in fact CBV could easily be modified to work with $E ::= Var \overset{P}{\to} C$ instead of $E ::= Var \overset{P}{\to} Cl$. We thus approximate E by $E' ::= Var \to C'$ where C' is an as-yet-undefined representation of C.

Since M_0 is constant-free, the closures computed by CBV(M_0) must all lie in Sub(M_0) x E. We thus approximate closures by $Cl' ::= \mathrm{Sub}(M_0)E'$. State $\sigma = c[cl]$ will be represented by a pair $\sigma' = (c', cl')$, so we define $\Sigma' ::= C'\ Cl'$.

Cl-context is also infinite due to its recursive definition. During the CBV(M_0) computation, contexts are manipulated "inside out", i.e. from the vicinity of [ ]. The reduction rules remove the innermost closure, and rule 3 has no effect on the closure. Rule 4 "deepens" the Cl-context, changing c[ ] to c[(M, e)[ ]] when processing a combination (MN, e), and rule 5 changes $c[cl_1[\ ]]$ to $c[[\ ]cl_2]$.

Let Com(M_0) denote the set of occurrences of combinations within M_0; these will be used for local representations of C.

The Cl-contexts appearing in a computation will be represented by two data structures: a set C' of <u>local representations</u>, plus a single global <u>retrieval function</u> $cc: C' \to \wp(Cl'\ C')$ which is used to extract the structure of any Cl-context if given its local representation. A containment $(cl', c_1') \in cc(c')$ indicates that c' locally represents a context $c_1[cl[\ ]]$ or $c_1[[\ ]cl]$, where cl' and c_1' represent cl and c_1 respectively. More specifically,

- [ ] locally represents the empty Cl-context.
- MN[ ] locally represents any Cl-context created by a transition c[(MN, e)] $\Rightarrow$ c[(M, e)[(N, e)]]. The remaining structure is represented globally by the containment $((M, e'), c') \in cc(MN[\ ])$, where e', c' represent e, c. (M, e') represents the operator of the combination (MN, e). Further, if MN[ ] represents c[cl[ ]] locally, then
- [ ]MN similarly represents (in the local sense) the context created by $c[cl_1[cl_2]] \Rightarrow [[cl_1]cl_2]$. This is globally represented by $(cl_2', c') \in cc([\ ]MN)$ where c' locally represents c[ ] and cl_2' represents the reduced form of operand (N, e).

Consequently we set $C' ::= [\ ] \mid [\ ]\mathrm{Com}(M_0) \mid \mathrm{Com}(M_0)[\ ]$ and $CC ::= C' \to \wp(Cl'\ C')$.

An entire computation $\sigma_1 \Rightarrow \sigma_2 \Rightarrow \ldots$ will be represented by a pair $\delta = (S, cc)$ where $S \subseteq \Sigma'$ contains representations of the states (using local context representations) and cc globally represents the structures of all the Cl-contexts.

Let $D ::= \wp(\Sigma)\ CC$ with the ordering: $(S_1, cc_1) \sqsubseteq (S_2, cc_2)$ iff $S_1 \subseteq S_2$ and $\forall c' \in C'\ cc_1(c') \subseteq cc_2(c')$. D is clearly a complete finite lattice, and is called the <u>description lattice</u>. We write $\perp$, $\top$, $\sqcup$, $\sqcap$ for the least element, greatest element, least upper bound and greatest lower bound, respectively.

The way in which elements of D represent closures, states, etc. is now made precise:

Definition 2.3 Let $\delta = (S, cc) \in D$. The representation relation $\overset{\delta}{\sim} \subseteq (Cl \times Cl') \cup (E \times E') \cup (C \times C') \cup (\Sigma \times \Sigma')$ is defined inductively by:

a) $(M, e) \overset{\delta}{\sim} (M, e')$ if $e \overset{\delta}{\sim} e'$

b) $e \overset{\delta}{\sim} e'$ if $\forall x \in \text{Domain}(e)\ \exists (cl', c') \in cc(e'(x))$ such that $e(x) \overset{\delta}{\sim} cl'$

c) $[\ ] \overset{\delta}{\sim} [\ ]$

$c[cl[\ ]] \overset{\delta}{\sim} MN[\ ]$ if $\exists (cl', c') \in cc(MN[\ ])$ such that $cl \overset{\delta}{\sim} cl'$ and $c \overset{\delta}{\sim} c'$

$c[[\ ]cl] \overset{\delta}{\sim} [\ ]MN$ if $\exists (cl', c') \in cc([\ ]MN)$ such that $cl \overset{\delta}{\sim} cl'$ and $c \overset{\delta}{\sim} c'$

d) $c[cl] \overset{\delta}{\sim} (c', cl')$ if $c \overset{\delta}{\sim} c'$, $cl \overset{\delta}{\sim} cl'$ and $(c', cl') \in S$.

□

Note that $\overset{\delta}{\sim}$ is monotonic with respect to δ: $\delta_1 \sqsubseteq \delta_2$ and $\sigma \overset{\delta_1}{\sim} \sigma'$ implies $\sigma \overset{\delta_2}{\sim} \sigma'$.

This technique may be used to approximate the behavior of many algorithms using recursively defined data types, providing an alternative for example to the methods of Jones and Muchnick [JoM81] and Reynolds [Rey68] for simple LISP-like programs. The underlying idea is to represent a recursively-defined structure by the set of program points which contain constructor operations, plus a retrieval function on this set which can be used to retrieve the components put together by a construction. These program points are propagated as descriptions of variables' values along control paths during flow analysis, and selector operations are performed by consulting the retrieval function.

The $CBV(M_0)$ Simulation Function

It is well known that most forward flow analysis methods essentially carry out an abstract interpretation of the simulated algorithm over a lattice of approximations to states or sets of states. Descriptions of this approach may be found in [Sin72] and [Cou79].

This approach was used to construct the simulation function $f: D \to D$ which is defined in Figure 5; the clauses defining f in essence apply the $CBV(M_0)$ transition rules to representations of states. Clearly f is monotonic; D is finite, so f is also continuous and so $\delta(M_0)$ is well-defined. Lemma 2.1 will be proved after an example; we now prove that Theorem 2.2 follows from Lemma 2.1.

Proof of Theorem 2.2

Suppose $\text{Load}(M_0) = \sigma_0 \Rightarrow \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n$. For $n = 0$, $\text{Load}(M_0) = ([\ ], (M_0, \{\ \})) \overset{\delta_0}{\sim} ([\ ], (M_0, \{\ \})) = \sigma_0'$ where $\delta_0 = (\{\sigma_0'\}, \{\ \})$ and $\{\ \}$ is the empty Cl-context retrieval function. Now $\sigma_0 \overset{\delta(M_0)}{\sim} \sigma_0'$ follows since $\delta_0 \sqsubseteq \delta(M_0)$.

Now suppose inductively that $\sigma_i \overset{\delta(M_0)}{\sim} \sigma_i'$. By Lemma 2.1 there exists σ_{i+1}' such that $\sigma_{i+1} \overset{\delta}{\sim} \sigma_{i+1}'$ where $\delta = f(\delta(M_0)) \sqsubseteq \delta(M_0)$, so $\sigma_i \overset{\delta(M_0)}{\sim} \sigma_{i+1}'$.

□

Data Structure Descriptions

$cl' : Cl'$	::=	$\mathrm{Sub}(M_0)E'$	Closures
$e' : E'$	::=	$\mathrm{BV}(M_0) \overset{P}{\to} C'$	Environments
$c' : C'$	::=	$[\ ] \mid [\ ]\mathrm{Com}(M_0) \mid \mathrm{Com}(M_0)[\ ]$	Contexts – local descriptions
$cc : CC$	::=	$C' \to \mathcal{P}(Cl'\ C')$	Contexts – global descriptions
$S \subseteq \Sigma'$	::=	$C'\ Cl'$	States
$\delta : D$	::=	$\mathcal{P}(\Sigma')CC$	Descriptions of states and sequences

Simulation Function $f: D \to D$

Let $\delta = (S, cc) \in D$. Then $f(\delta)$ is the least pair $(S_1, cc_1) \sqsupseteq (S, cc)$ such that the following hold:

1. β reduction: if $([\ ]NP, (\lambda xM, e')) \in S$ and $(cl', c') \in cc([\ ]NP)$ then $(c', (M, e'\{[\ ]NP/x\} \in S_1$
2. Variable expansion: if $(c', (x, e')) \in S$ and $(cl', c_1') \in cc(e'(x))$ then $(c', cl') \in S_1$
3. Combination: if $(c', (MN, e')) \in S$ then $(MN[\ ], (N, e')) \in S_1$ and $((M, e'), c') \in cc_1(MN[\ ])$
4. Scan operator: if $(MN[\ ], cl_2') \in S$ where $cl_2' = (P, e')$ for some closed value P and $(cl_1', c') \in cc(MN[\ ])$ then $([\ ]MN, cl_1') \in S_1$ and $(cl_2', c') \in cc_1([\ ]MN)$

Control Flow Description $\delta(M_0) \in D$

This is the least solution to the equation $\delta = f(\delta) \sqcup \mathrm{Load}'(M_0)$, where

$\mathrm{Load}'(M_0) = (\{([\ ], (M_0, \{\ \}))\}, \{\ \})$ describes $\{\mathrm{Load}(M_0)\}$.

Figure 5. Simulation Function and Description Lattice.

An Example

Let $M_0 = AB = (\lambda xxx)(\lambda yy)$. The reduction sequence $M_0 > (\lambda yy)(\lambda yy) \geq \lambda yy$ as computed by $\mathrm{CBV}(M_0)$ and as represented by $\hat{\delta}(M_0) = (S, cc)$ are as in Figure 6.

CBV(M_0) Computation State	Action	Approximation S	cc
$[(AB,\{\})]$	scan operand	$([\ ],(AB,\{\}))$	
$\Rightarrow [(A,\{\})[(B,\{\})]]$	scan operator	$(AB[\ ],(B,\{\}))$	$((A,\{\}),[\ ]) \in cc(AB[\ ])$
$\Rightarrow [[(A,\{\})](B,\{\})]$	β reduce	$([\ ]AB,(A,\{\}))$	$((B,\{\}),[\ ]) \in cc([\ ]AB)$
$\Rightarrow [(xx,\underbrace{\{x \to (B,\{\})\}}_{\text{call this } e})]$	scan operand	$([\ ],(xx,\underbrace{\{x \to [\ ]AB\}}_{\text{call this } e'}))$	
$\Rightarrow [(a,e)[(x,e)]]$	expand x	$(xx[\ ],(x,e'))$	
$\Rightarrow [(x,e)[(B,\{\})]]$	scan operator	$(xx[\ ],(B,\{\}))$	$((x,e'),[\ ]) \in cc(xx[\ ])$
$\Rightarrow [[(x,e)](B,\{\})]$	expand x	$([\ ]xx,(x,e'))$	$((B,\{\}),[\ ]) \in cc([\ ]xx)$
$\Rightarrow [[(B,\emptyset)](B,\{\})]$	β reduce	$([\ ]xx,(B,\{\}))$	
$\Rightarrow [(y,\{y \to (B,\{\})\})]$	expand y	$([\ ],(y,\{y \to [\ ]xx\}))$	
$\Rightarrow [(B,\{\})]$		$([\ ],(B,\{\}))$	

Figure 6. CBV(M_0) Computation and Approximation.

Proof of Lemma 2.1

Suppose $\sigma_1 \Rightarrow \sigma_2$ and $\sigma_1 \overset{\delta}{\sim} \sigma_1'$ where $\delta = (S, cc)$.

Case 1. $c[[\lambda xM,e)]cl] = \sigma_1 \Rightarrow \sigma_2 = c[(M,e\{cl/x\})]$.

Let $\sigma_1 \overset{\delta}{\sim} ([\ ]NP,(\lambda xM,e'))$. Then $\exists c', cl'$ such that $e \overset{\delta}{\sim} e'$, $c \overset{\delta}{\sim} c'$, $cl \overset{\delta}{\sim} cl'$ and $(cl',e') \in cc([\ ]NP)$. By definition of f, $\sigma_2' = (c',M,e'\{[\ ]NP/x\}) \in S_1$. To show $\sigma_2 \overset{f(\delta)}{\sim} \sigma_2'$ we need only establish $e\{cl/x\} \overset{f(\delta)}{\sim} e'\{[\ ]NP/x\}$. This follows from the definition of $\overset{\delta}{\sim}$ and the facts that $e \overset{\delta}{\sim} e'$ and $(cl',c') \in cc([\ ]NP)$.

Case 2. $c[(x,e)] = \sigma_1 \Rightarrow \sigma_2 = c[e(x)]$.

If $\sigma_1 \overset{\delta}{\sim} (c',(x,e'))$ then $\sigma_2 \overset{f(\delta)}{\sim} (c',e'(x))$ by the definitions of $e \overset{\delta}{\sim} e'$ and f.

Case 3. $c[(MN,e)] = \sigma_1 \Rightarrow \sigma_2 = c[(M,e)[(N,e)]]$.

Let $\sigma_1 \overset{\delta}{\sim} (c',(MN,e'))$. By definition of f, $\sigma_2' = (MN[\ ],(N,e')) \in S_1$ and $((M,e'),c') \in cc_1(MN[\ ])$. Now $(M,e) \overset{\delta}{\sim} (M,e')$ so $c[(M,e)[\ ]] \overset{\delta}{\sim} MN[\ ]$. This and $(N,e) \overset{\delta}{\sim} (N,e')$ implies $\sigma_2 \overset{f(\delta)}{\sim} \sigma_2'$.

Case 4. $c[cl_1[cl_2]] = \sigma_1 \Rightarrow \sigma_2 = c[[cl_1]cl_2]$ with cl_2 suitably restricted.

Let $\sigma_1 \overset{\delta}{\sim} (MN[\ ], cl_2')$. Then $\exists c', cl_1'$, such that $cl_2 \overset{\delta}{\sim} cl_2'$, $c \overset{\delta}{\sim} c'$, $cl_1 \overset{\delta}{\sim} cl_1'$ and $(cl_1', c') \in cc(MN[\ ])$. By definition of f, $\sigma_2' = ([\ ]MN, cl_1') \in S_1$ and $(cl_2', c') \in cc_1([\ ]MN)$. Clearly $c[[\ ]cl_2] \overset{\delta}{\sim} [\ ]MN$ so $\sigma_2 \overset{f(\delta)}{\sim} \sigma_2'$ is immediate.

□

A Context-free Approximation to States(M_0)

First we define a way to encode states, etc. as linear strings of symbols. A Cl-context will be written linearly as $(i_1, cl_1) \ldots (i_n, cl_n)$ where each i_j is 1 if cl_j is in operator position and 2 if in operand position. For example $c = (cl_1([\ ]cl_2))cl_3$ becomes $(2, cl_2)(1, cl_1)(2, cl_3)$. This would be suitable for computer implementation since the OI and CBV transition rules in effect treat such a string as a stack with the top at the left end. An environment e will be encoded as $\{x_1 \to e(x_1), \ldots, x_n \to e(x_n)\}$ where $\mathrm{Domain}(e) = \{x_1, \ldots, x_n\}$ and $x_1 < x_2 < \ldots < x_n$ relative to some arbitrary fixed order relation on Var.

The linear encoding function $le: Cl \cup E \cup C \cup \Sigma \to A^*$ is defined as follows, where A is the alphabet $A = \mathrm{Var} \cup \{(,),\{,\},[,],\to,1,2\} \cup \{,\}$.

$$
\begin{aligned}
le((M,e)) &= (M, le(e)) \\
le(e) &= \{x_1 \to le(e(x_1)), \ldots, x_n \to le(e(x_n))\} \\
&\quad \text{where } \mathrm{Domain}(e) = \{x_1, \ldots, x_n\} \text{ and } x_1 < \ldots < x_n \\
le([\ ]) &= \epsilon \ \text{(the empty string)} \\
le(cl\ c) &= le(c)(1, le(cl)) \\
le(c\ cl) &= le(c)(2, le(cl)) \\
le(c[cl]) &= le(c)[le(cl)]
\end{aligned}
$$

Definition 2.4 Let $\delta = (S, cc)$ be the control flow description of CBV(M_0). The context-free grammar $G(M_0)$ is defined in Figure 7.

An example derivation from Figure 5 is

$$
\begin{aligned}
S &\Rightarrow \overline{[\ ]}\,[(\overline{xx, e'})] \overset{*}{\Rightarrow} [(xx, \overline{e'})] \\
&\Rightarrow [(xx, \{x \to (\overline{B, \{\ \}})\})] \Rightarrow [(xx, \{x \to (\lambda yy, \{\ \})\})]
\end{aligned}
$$

Nonterminals	:	$\{S_0\} \cup \{\overline{X} \mid X \in Cl' \cup E' \cup C' \cup \Sigma'\}$		
Terminals	:	A ; Start symbol : S_0		
Productions	:			
Closures:	:	$\overline{(M, e')}$	$::= (M, \overline{e'})$	for each $M \in Sub(M_0), e' \in E'$
Environments:		$\overline{e'}$	$::= \{x_1 \to \overline{cl_1'}, \ldots, x_n \to \overline{cl_n'}\}$	
			where $Domain(e') = \{x_1, \ldots, x_n\}$, $x_1 < \ldots < x_n$	
			and $\forall i \in [1,n]\ \exists c_i'\ (cl_i', c_i') \in cc(e'(x_i)))$	
Cl-contexts	:	$\overline{[\ ]}$	$::= \epsilon$	
		$\overline{MN[\ ]}$	$::= (1, \overline{cl'})\ c'$	for each $(cl', c') \in cc(MN[\ ])$
		$\overline{[\ ]MN}$	$::= (2, \overline{cl'})\ c'$	for each $(cl', c') \in cc([\ ]MN)$
States	:	$\overline{(c', cl')}$	$::= \overline{c'}\ [\overline{cl'}]$	for each $(c', cl') \in S$
Initial	:	S_0	$::= \overline{\sigma'}$	for each $\sigma' \in S$

Figure 7. Context-free Grammar $G(M_0)$ Approximating $States(M_0)$.

For any nonterminal $\overline{X}$ let $L(\overline{X}) = \{x \in A^* \mid \overline{X} \overset{*}{\Rightarrow} x\}$.

Theorem 2.5

Let M_0 be a closed constant-free λ-expression and δ its control flow description. For each $x' \in X'$ where $X = Cl, E, C$ or Σ:

$$L(\overline{x'}) = \{ le(x) \mid x \overset{\delta}{\sim} x' \text{ and } x \in X \}$$

Proof is a straightforward induction on definition 2.3 of $x \overset{\delta}{\sim} x'$.

□

Corollary 2.6

$$L(G(M_0)) \supseteq \{ le(\sigma) \mid \sigma \in States(M_0) \}.$$

Applications

Let a "safe positive reply" P to a question whose answer is Q be one such that P logically implies Q. Given $N \in Sub(M_0)$ 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 $M, N \in Sub(M_0)$ if S contains a state $c[(M, e)]$ with (M, e) dependent on N.

Theorem 2.7

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 $c[(M, e)]$?).

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 $M, N \in Sub(M_0)$)?

Proof is by showing how to analyze the structure of $\delta(M) = \delta = (S, cc)$. Question 1 is simple: if S contains no pair $(c', (M, e'))$ then $States(M_0)$ contains no state $c[(M, e)]$ by Theorem 2.2.

Define the flowchart of M to have nodes in Σ' and an edge $\sigma_1' \Rightarrow \sigma_2'$ just in case $\sigma_2' \in S_1$ where $f(\{\sigma_1'\}, cc) = (S_1, cc)$. It is easy to see that if $\sigma_1 \overset{\delta}{\sim} \sigma_1'$ then $\sigma_1 \Rightarrow \sigma_2$ by $CBV(M_0)$ implies $\sigma_1' \Rightarrow \sigma_2'$ in the flowchart for some σ_2' with $\sigma_2 \overset{\delta}{\sim} \sigma_2'$. Letting the $CBV(M_0)$ computation be $Load(M_0) = \sigma_0 \Rightarrow \sigma_1 \Rightarrow \ldots$, there exists a path $\sigma_0' \Rightarrow \sigma_1' \Rightarrow \ldots$ where each $\sigma_i \overset{\delta}{\sim} \sigma_i'$ for $n \geq 0$ and $\sigma_0' = ([\ ], (M_0, \{\ \}))$.

If the computation is infinite there must exist σ' such that $\sigma_0' \overset{*}{\Rightarrow} \sigma' \overset{+}{\Rightarrow} \sigma'$, 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 $\sigma_0' \overset{*}{\Rightarrow} \sigma'$ where $\sigma' \not\Rightarrow \sigma_1'$ for all σ_1'. Note that safe answers to questions 2 and 3 can both be "no".

Question 4 can be answered simply by constructing $G(M_0)$ and testing $L(G(M_0))$ for finiteness; by Corollary 2.6 a positive answer implies $States(M_0)$ is finite. It is well known that finiteness of context-free languages is decidable.

For question 5, define "cl' depends on N" for $cl' \in Cl'$ by

a) (M, e') depends on M for any $e' \in E'$

b) (M, e') depends on N if there exist $x \in FV(M)$ and $(cl', c') \in cc(e'(x))$ such that cl' depends on N.

It is easily seen that if (M, e) depends on N and $(M, e) \overset{\delta}{\sim} (M, e')$ then (M, e') depends on N. The set of all pairs (M, cl') such that cl' depends on M is clearly effectively computable. Finally, M is independent of N if there exists no $(c, (M, e')) \in S$ with (M, e') dependent on N.

□

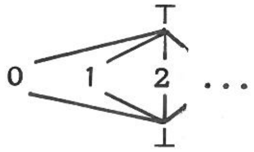
3. ANALYSIS OF DATA AND CONTROL FLOW

A method will now be given to obtain a safe description of States(M_0) where M_0 is an arbitrary λ-expression. The method involves the use of an auxiliary lattice to approximate sets of constants. The development is otherwise quite parallel to the previous one, so only the essential details are given. For the method to succeed finitely, however, we need the following reasonable

<u>Assumption</u> Subcon is finite.

<u>Approximation of Constants</u>

Since the set of constants is infinite and our descriptions of States(M_0) are finite, it is necessary to find a way to finitely represent unbounded sets of constants. This is traditionally done in flow analysis by use of a complete <u>approximation lattice</u> L whose elements represent sets of constants via an <u>abstraction function</u> abs: $\mathcal{P}(\mathrm{Con}) \to L$. For example a suitable L for "constant propagation" would be the one in Figure 8. Constant propagation allows the recognition of subexpressions which always evaluate to the same value, and determination of that value.



abs: $\mathcal{P}(\{0, 1, 2, \ldots\}) \to L$

abs(A) = <u>if</u> $A = \emptyset$ <u>then</u> $\bot$ <u>else</u> <u>if</u> $A = \{a\}$ <u>then</u> a <u>else</u> T

<u>Figure 8.</u> A Simple Approximation Lattice L.

Not all abstraction functions and lattices are suitable for flow analysis. An appropriate and useful restriction due to Cousot is that there exist a concretization function conc: $L \to \wp(Con)$ such that (abs, conc) are a pair of adjoined functions. Letting $\sqsubseteq$ be the ordering on L this means that

$$\forall A \subseteq Con \quad \forall l \in L \qquad A \subseteq conc(l) \quad \text{iff} \quad abs(A) \sqsubseteq l$$

Motivation of this definition and useful mathematical properties may be found in [Cou79]. A suitable concretization function for constant propagation is: $conc(\perp) = \emptyset$, $conc(a) = \{a\}$, $conc(\top) = Con$.

In order to effectively obtain finite descriptions we also require that L have the finite chain property, i.e. that there exist no infinite properly increasing chains. Note that the example is infinite but has the finite chain property.

The Description Lattice Dcon

This extends the D lattice used before. Figure 9 contains Dcon and the simulation function f_{con}. Note that E', C' and Σ' are as in Figure 5 but that Cl', CC and Dcon are more elaborate.

By Lemma 1.5 an achievable closure (M, e) must have $M \in Lam(M_0) = Sub(M_0) \cup Subcon \cup Con$. In Dcon we represent a closure by either $l \in L$ describing a set of constant values, or a pair (M, e') with $M \in Sub(M_0) \cup Subcon$.

We cannot use Dcon ::= $\wp(\Sigma')CC$ since $\wp(\Sigma') = \wp(C'\ Cl')$ may not possess the finite chain property (as in Figure 8). This problem is resolved by merging a set $\{(c', l_1), \ldots, (c', l_n)\} \subseteq C' \times C_i'$ with $l_1, \ldots, l_n' \in L$ into the single pair $(c', \bigsqcup_{i=1}^{n} l_i)$. $\wp(\Sigma')$ is replaced by

$$C' \otimes Cl' = \{X \subseteq C' \times Cl' \mid (c', l_1), (c', l_2) \in C' \times L \text{ implies } l_1 = l_2\}$$

Clearly $C' \otimes Cl'$ has no infinite ascending chains. We now define Dcon ::= $(C' \otimes Cl')CC$. An order relation $\sqsubseteq$ can now be imposed on $C' \otimes Cl'$ to make Dcon a complete lattice, as follows. The following function is easily seen to be an isomorphism:

$$g : C' \otimes Cl' \to \wp(C' \times ((Sub(M_0) \cup Subcon) \times E) \times (C' \xrightarrow{P} L)$$

$$g(X) = (X \cap (C' \times ((Sub(M_0) \cup Subcon) \times E)), \lambda c.\ \underline{\text{if}}\ (c, l) \in X\ \underline{\text{then}}\ l\ \underline{\text{else}}\ \text{undefined})$$

Informally, if $g(X) = (Y, h)$ then Y consists of those pairs (c', cl') where cl' is not in L, and h maps c' to l iff $(c', l) \in X$ and $l \in L$. The range of g is certainly a complete lattice with $(Y, h) \leq (Y', h')$ iff $Y \subseteq Y'$ and $\forall c \in C'\ (h(c) \subseteq h(c'))$. Now define $X_1 \sqsubseteq X_2$ iff $g(X_1) \leq g(X_2)$, making $C' \otimes Cl'$ into a complete lattice.

Note that $(c', cl') \in X$ and $\{(c', cl')\} \sqsubseteq X$ are both meaningful. Both are used in Figure 9.

Similarly the set of global context descriptions $CC ::= C' \to Cl' \otimes C'$ is a complete lattice, with $Cl' \otimes C'$ defined symmetrically to $C' \otimes Cl'$. It should now be evident that Dcon is a complete lattice with the finite chain property.

The Simulation Function

This function f_{con} is constructed in the same way as f, to simulate the effect of a CBV(M_0) transition $\sigma_1 \Rightarrow \sigma_2$ in terms of the data structure δ representing σ_1. Constants can either be computed or original subexpressions of M_0; thus we add a rule to convert closure (con, e) into its representation abs{con} in L. For δ reduction we must map backwards from L to Con and then forward into L again. This is done with the aid of Conap: $L \times L \to \wp(\text{Con})$ and Lamap : $L \times L \to \wp(\text{Lam})$ defined by:

$$\text{Conap}(l_1, l_2) = \text{Con} \cap \text{Back}, \quad \text{Lamap}(l_1, l_2) = \text{Subcon} \cap \text{Back}$$

$$\text{Back}(l_1, l_2) = \{a' \mid \exists\, a \in \text{conc}(l_1),\ b \in \text{conc}(l_2) \text{ such that } a' = \text{Constapply}(a, b) \text{ is defined}\}$$

Further Development

It seems clear that the steps taken earlier for control flow analysis can now be paralleled: a representation relation $\sigma \overset{\delta}{\sim} \sigma'$ could be defined, safeness proved, and a context-free grammar generating a superset of $\{ \text{le}(\sigma) \mid \sigma \in \text{States}(M_0)\}$ could be constructed. We do not do this for brevity and because no new ideas are involved.

The CBV(M_0) interpreter may perform an "error halt" if M_0 contains constants; a state $c[[(a, e)](b, e_1)]$ with $a \in$ Con causes nonstandard termination unless $b \in$ Con and Constapply(a, b) is defined. Taking this into account it appears straightforward to extend the methods of Theorem 2.7 to obtain effectively safe positive replies 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.)

Data Structure Descriptions

$cl' : Cl'$	$::=$	$(Sub(M_0) \mid Subcon)E \mid L$	Closures
$e' : E'$	$::=$	$BV(M_0) \overset{P}{\to} C'$	Environments
$c' : C'$	$::=$	$[\ ] \mid [\ ]Com(M_0) \mid Com(M_0)[\ ]$	Contexts - local
$cc : CC$	$::=$	$C' \to (Cl' \otimes C')$	Contexts - global
$S \subseteq \Sigma'$	$::=$	$C'\,Cl'$	State sets
$\delta : Dcon$	$::=$	$(C' \otimes Cl')CC$	States and sequences

Simulation Function f_{con}: Dcon $\to$ Dcon

Let $\delta = (S, cc) \in Dcon$. Then $f_{con}(\delta)$ is the least pair (S_1, cc) satisfying:

1. β reduction: if $([\ ]NP, (\lambda xM, e')) \in S$ and $(cl', c') \in cc([\ ]NP)$
   then $(c', (M, e\{[\ ]NP/x\})) \in S_1$
2. δ reduction: if $([\ ]NP, l_1) \in S_1$ and $(l_2, c') \in cc_1([\ ]NP)$
   then a) $\{(c', abs(Conap(l_1, l_2)))\} \sqsubseteq S_1$
   b) $(c', (M, \{\ \})) \in S_1$ for each $M \in Lamap(l_1, l_2)$
3. Constant representation: if $(c', (con, e)) \in S$
   then $(c', abs\{con\}) \sqsubseteq S_1$
4. Variable expansion: if $(c', (x, e')) \in S$ and $(cl', c_1') \in cc(e'(x))$
   then $(c', cl') \in S_1$
5. Combination: if $(c', (MN, e')) \in S$
   then $(MN[\ ], (N, e')) \in S_1$ and $((M, e'), c') \in cc_1(MN[\ ])$
6. Scan operator: if $(MN[\ ], cl_2') \in S$ where $cl_2' = (P, e')$ for some closed value P, and $(cl_1', c') \in cc(MN[\ ])$
   then $([\ ]MN, cl_1') \in S_1$ and $(cl_2', c') \in cc_1([\ ]MN)$

Data and Control Flow Description

$\delta(M_0)$ is the least solution to the equation $\delta = f(\delta) \sqcup Load'(M_0)$

Figure 9. Approximate Description of Data and Control Flow.

4. CONCLUSIONS AND ACKNOWLEDGEMENTS

It has been shown that safe answers may be effectively obtained to a variety of questions about call-by-value reduction sequences including finiteness, termination, freedom from errors, and independence of subexpressions. The methods used are clearly applicable to call-by-name; further since abstract interpretation does not depend on determinism it seems likely that the OI interpreter could be similarly analyzed, giving information about the set of all outside-in 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. This would extend the results of Mycroft [Myc80].

The analysis method applied the classical flow-analytic idea of abstract interpretation to a new call-by-value interpreter CBV which appears to be somewhat simpler than the SECD machine. This application required a new description technique involving both local and global data representations due to the recursiveness of CBV'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 Muchnick on various aspects of this work have been very helpful.

REFERENCES

Aho77 Aho, Alfred V., and Jefferey 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.

Cou77 Cousot, Patrick, and Radhia 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 Radhia 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).

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

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 Schmidt, David A., "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. Symposium on Programming, LNCS 83 (1980), 269-281.

Ple80 Pleban, Uwe, "A Denotational Semantics Approach to Program Optimization", Ph.D. Dissertation, Univ. of Kansas, Lawrence, KS (1980).

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

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

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

Sha80 Sharir, M. and A. Pnueli, "Two Approaches to Interprocedural Data Flow Analysis", <u>Program Flow Analysis</u>, S.S. Muchnick and N.D. Jones eds. Prentice Hall (1980).

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

Ste76 Steele, Guy Lewis Jr., "LAMBDA: The Ultimate Declarative", AI Memo 379 (November 1976), Artificial Intelligence Laboratory, MIT.

Weg68 Wegner, Peter, <u>Programming Languages, Information Structures and Machine Organization</u>, McGraw-Hill, New York (1968).

APPENDIX I

PROOF OF THEOREM 1.1

Lemma 1 $\quad$ $\mathrm{Real}(cl_1) \overset{*}{\underset{oi}{\geq}} \mathrm{Real}(cl_2)$ implies $\mathrm{Unload}(c[cl_1]) \overset{*}{\underset{oi}{\geq}} \mathrm{Unload}(c[cl_2])$.

Proof is by a simple induction on the size of c[ ].

Lemma 2 $\quad$ Suppose $\mathrm{Real}((\lambda x M, e)) = \lambda x M'$, $\mathrm{Real}(cl) = N$ and N is closed. Then $\mathrm{Real}((M, e\{cl/x\})) = [N/x]M'$.

Proof is straightforward and so omitted.

Lemma 3 $\quad$ $\sigma_1 \underset{oi}{\Rightarrow} \sigma_2$ and $\mathrm{Unload}(\tau_1)$ closed implies $\mathrm{Unload}(\sigma_1) \overset{*}{\underset{oi}{\geq}} \mathrm{Unload}(\sigma_2)$.

Proof $\quad$ If $\sigma_1 \underset{oi}{\Rightarrow} \sigma_2$ by transition rule 5, 6, 7 on 8 then $\mathrm{Unload}(\sigma_1) = \mathrm{Unload}(\sigma_2)$ by definition of Unload. If $\sigma_1 = c[cl_1] \Rightarrow c[cl_2] = \sigma_2$ by rule 3 or 4 then $\mathrm{Real}(cl_1) = \mathrm{Real}(cl_2)$ so the result holds by Lemma 1.

If $\sigma_1 = c[[(a,e)](b,e')] \Rightarrow c[(a',\{\ \})] = \sigma_2$ by rule 2 then $\mathrm{Real}((a,e)(b,e')) = ab \underset{oi}{\geq} a' = \mathrm{Real}((a',\{\ \}))$. By Lemma 1, $\mathrm{Unload}(\sigma_1) = \mathrm{Unload}(c[(a,e)(b,e')]) \overset{*}{\underset{oi}{\geq}} \mathrm{Unload}(\sigma_2)$. Finally suppose $\sigma_1 \Rightarrow \sigma_2$ as in rule 1, and let $\mathrm{Real}((\lambda x M, e)) = \lambda x M'$ and $\mathrm{Real}(cl) = N$. $\mathrm{Unload}(\sigma_1)$ is closed so N must also be closed. By β-reduction and Lemma 2,

$$\mathrm{Real}((\lambda x M, e)cl) = (\lambda x M')N \underset{oi}{\geq} [N/x]M' = \mathrm{Real}((M, e\{cl/x\}))$$

Thus $\mathrm{Unload}(\sigma_1) = \mathrm{Unload}(c[(\lambda x M, e)cl]) \overset{*}{\underset{oi}{\geq}} \mathrm{Unload}(\sigma_2)$ by Lemma 1.

□

This completes the proof of part a) of Theorem 1.1 For part b), first define a closure (M, e) to be <u>simple</u> if M is not a variable. Note that any state $\sigma = c[cl]$ yields a state $\sigma' = c[cl']$ with simple cl' by a finite number of applications of transition rule 3 (rule 3 cannot be applied infinitely often since environments are defined inductively). A consequence is that if Real(cl) is a combination M_1M_2 then there exist cl_1, cl_2 such that $c[cl] \overset{*}{\underset{oi}{\Rightarrow}} c[cl_1cl_2]$ and $\mathrm{Real}(cl_i) = M_i$ for $i = 1, 2$ (since the cl' mentioned above must be a combination by the definition of Real). Similar results hold if Real(cl) is an abstraction or a constant.

Part b) of Theorem 1.1 is an immediate consequence of the following.

Lemma 4 $\quad$ If $\mathrm{Unload}(\sigma_1) \underset{oi}{\geq} N$ and $\mathrm{Unload}(\sigma_1)$ is closed then $\sigma_1 \overset{*}{\underset{oi}{\Rightarrow}} \sigma_2$ for some σ_2 satisfying $N = \mathrm{Unload}(\sigma_2)$.

<u>Proof</u> It is easily seen that the inference rule for reduction in context can be replaced by the two simpler rules

$$4'.\quad \frac{M_1 > M_2}{M_1 N > M_2 N} \qquad\qquad 4''.\quad \frac{M_1 > M_2}{NM_1 > NM_2}$$

Proof of the lemma is by induction on the number k of times inference rules 4' or 4'' are used to establish $\text{Unload}(\sigma_1) \underset{oi}{>} N$.

<u>Basis</u> k = 0: Suppose $\text{Unload}(\sigma_1) = (\lambda x M)N \underset{oi}{>} [N/x]M$. The following machine computation occurs:

$$\begin{array}{lll}
\sigma_1 \overset{*}{\underset{oi}{\Rightarrow}} [cl] & & \text{where Unload}(\sigma_1) = \text{Real}(cl) \text{ by OI transition rules 6, 8} \\
\overset{*}{\underset{oi}{\Rightarrow}} [cl_1 cl_2] & & \text{where Real}(cl_1) = \lambda x M, \text{Real}(cl_2) = N \text{ by rule 3} \\
\underset{oi}{\Rightarrow} [[cl_1]cl_2] & & \text{OI rule 5} \\
\overset{*}{\underset{oi}{\Rightarrow}} [[\lambda x M', e)]cl_2] & & \text{where Real}((\lambda x M', e)) = \lambda x M \text{ by rule 3} \\
\underset{oi}{\Rightarrow} [(M', e\{cl_2/x\})] & & \text{OI rule 1} \\
= \sigma_2 & & \text{by definition.}
\end{array}$$

Now N must be closed so by Lemma 2 $\text{Unload}(\sigma_2) = \text{Real}((M', e\{cl_2/x\})) = [N/x]M$ as required. Delta reduction is similar but a bit simpler.

<u>Inductive step</u> Suppose the result holds for fewer than k uses of 4' and 4'' where $k > 0$, and $\text{Unload}(\sigma_1) = M_1 M_2 \underset{oi}{>} NM_2$ by k uses. As in the basis case $\sigma_1 \overset{*}{\underset{oi}{\Rightarrow}} [[cl_1]cl_2]$ where $\text{Real}(cl_i) = M_i$ $(i = 1, 2)$, by a computation involving OI transition rules 6, 8, 3 and 5. By induction $[cl_1] \overset{*}{\underset{oi}{\Rightarrow}} \sigma_1'$ for some σ_1' with $\text{Unload}(\sigma_1') = N$. By OI rules 6, 8 $\sigma_1' \overset{*}{\underset{oi}{\Rightarrow}} [cl_1']$ for some cl_1' with $\text{Real}(cl_1') = N$.

Finally $[cl_1] \overset{*}{\underset{oi}{\Rightarrow}} [cl_1']$ implies $[[cl_1]cl_2] \overset{*}{\underset{oi}{\Rightarrow}} [[cl_1']cl_2]$. Let $\sigma_2 = [[cl_1']cl_2]$. Clearly $\text{Unload}(\sigma_2) = \text{Real}(cl_1')\text{Real}(cl_2) = NM_2$ as required. A symmetric argument applied if $\text{Unload}(\sigma_1) = M_1 M_2 \underset{oi}{>} M_1 N$.

□

APPENDIX II

PROOF OF THEOREM 1.3

We first define $eval_V$ more formally just as in [Plo75]. Let "M has value N at time t" mean $T(M, t) = N$ where T: Programs $\times \{0, 1, \ldots\} \overset{P}{\Rightarrow}$ Programs is defined as follows.

1. $T(M, 0) = M$ if M is a constant or an abstraction

2. $$\frac{T(M_1, t_1) = a,\ T(M_2, t_2) = b \text{ and } Constapply(a, b) \text{ exists}}{T(M_1M_2, t_1+t_2+1) = Constapply(a, b)}$$

3. $$\frac{T(M_1, t_1) = \lambda x M_1',\ T(M_2, t_2) = M_2',\ T([M_2'/x]M_1', t_3) = N}{T(M_1M_2, t_1+t_2+t_3+1) = N}$$

Now let $eval_V(M) = N$ iff $T(M, t) = N$ for some $t \geq 0$, i.e. iff M has value N at some time. Note that N cannot be a combination.

Lemma 1 If Real(cl) has closed value N at some time then CBV has a computation $[cl] \overset{*}{\Rightarrow} [cl']$ such that $N = Real(cl')$.

Corollary 2

$eval_V(M) = Eval_V(M)$ if $eval_V(M)$ exists.

Proof Let $T(M, t) = N$ where $Real(cl) = M$. Proof is by induction on t. If $t = 0$ it follows immediately with $cl = cl'$. Now suppose $t > 0$ and the result holds for all times less than t. By definition of T $Real(cl) = M_1M_2$ for some M_1, M_2. It is easy to see that $[cl] \overset{*}{\Rightarrow} [cl_1[cl_2]]$ by CBV transition rules 2, 4 where $Real(cl_i) = M_i$ $(i = 1, 2)$.

Case 1 $T(M_1, t_1) = a$, $T(M_2, t_2) = b$, $N = Constapply(a, b)$ exists and $t = t_1+t_2+1$. Then $t_1, t_2 < t$ so by induction $[cl_1] \overset{*}{\Rightarrow} [(a, e_1)]$ and $[cl_2] \overset{*}{\Rightarrow} [(b, e_2)]$ for some e_1, e_2. Consequently CBV has the computation

$$\begin{aligned}[cl] &\overset{*}{\Rightarrow} [cl_1[cl_2]] \overset{*}{\Rightarrow} [cl_1[(b, e_2)]] \Rightarrow [[cl_1](b, e_2)] \\ &\overset{*}{\Rightarrow} [[(a, e_1)](b, e_2)] \Rightarrow [(Constapply(a, b), \{\,\})]\end{aligned}$$

Now let $cl' = (Constapply(a, b), \{\})$.

Case 2 $\quad T(M_1, t_1) = \lambda x M_1'$, $T(M_2, t_2) = M_2'$, $T([M_2'/x]M_1', t_3) = N$ and $t = t_1+t_2+t_3+1$.

Then $t_1, t_2 < t$ so by induction $[cl_1] \overset{*}{\Rightarrow} [cl_1']$ and $[cl_2] \overset{*}{\Rightarrow} [cl_2']$ where $\mathrm{Real}(cl_1') = \lambda x M_1'$ and $\mathrm{Real}(cl_2') = M_2'$. Without loss of generality $cl_1' = (\lambda x M_1'', e)$ for some e, M_1''. Let $cl_3 = (M_1'', e\{cl_2'/x\})$. By Lemma 2 of Appendix I, $\mathrm{Real}(cl_3) = [M_2'/x]M_1'$. Using induction again, $[cl_3] \overset{*}{\Rightarrow} [cl']$ where $N = \mathrm{Real}(cl')$. Putting these together the required CBV computation is

$$[cl] \overset{*}{\Rightarrow} [cl_1[cl_2]] \overset{*}{\Rightarrow} [cl_1[cl_2']] \Rightarrow [[cl_1]cl_2']$$

$$\overset{*}{\Rightarrow} [[(\lambda x M_1'', e)]cl_2'] \Rightarrow [(M_1'', e\{cl_2'/x\})] = [cl_3] \overset{*}{\Rightarrow} [cl']$$

Lemma 3 $\quad$ If $[cl] \overset{*}{\Rightarrow} \sigma$, $\mathrm{Last}(\sigma) = \sigma$ and $\mathrm{Real}(\sigma)$ is closed then $T(\mathrm{Real}(cl), t) = \mathrm{Unload}(\sigma)$ for some $t \geq 0$.

This result and Corollary 2 yield Theorem 1.3.

Proof $\quad$ Let $[cl] \overset{n}{\Rightarrow} \sigma$ where $\mathrm{Last}(\sigma) = \sigma$, $\mathrm{Real}(cl)$ is closed and $cl = (M, e)$. The result is immediate if M is a constant or an abstraction, which must hold if $n = 0$.

Assume inductively that the result holds for all computations of length less than n, where $n > 0$. If M is a variable then $[cl] \Rightarrow [e(M)] \overset{n-1}{\Rightarrow} \sigma$ and $\mathrm{Real}(cl) = \mathrm{Real}(e(M))$ so the result holds by induction.

The remaining case if $M = M_1M_2$, so $[cl] \Rightarrow [cl_1[cl_2]] \overset{*}{\Rightarrow} \sigma$ where $cl_i = (M_i, e)$ for $i = 1, 2$. Consider the computation $[cl_2] = \sigma_1 \Rightarrow \sigma_2 \Rightarrow \ldots$. This must terminate in some σ_m without an "error stop", else $[cl] \overset{*}{\Rightarrow} \sigma$ is violated. It is easy to see that $\sigma_m = [cl_2']$ where $cl_2' = (M_2', e_2)$ for some value M_2'. Thus the computation may be refined further:

$$[cl] \overset{*}{\Rightarrow} [cl_1[cl_2]] \overset{n_2}{\Rightarrow} [cl_1[cl_2']] \Rightarrow [[cl_1]cl_2'] \overset{*}{\Rightarrow} \sigma$$

Since $n_2 < n$ and $\mathrm{Last}([cl_2']) = [cl_2']$ we may use induction to show that $T(\mathrm{Real}(cl_2), t_2) = \mathrm{Real}(cl_2')$ for some t_2.

By exactly the same reasoning applied to cl_1 there must exist $cl_1' = (M_1', e_1), t_1$ such that $[cl_1] \overset{*}{\Rightarrow} [cl_1']$ and $T(\mathrm{Real}(cl_1), t_1) = \mathrm{Real}(cl_1')$. The computation may be re-expressed:

$$[cl] \overset{*}{\Rightarrow} [[cl_1']cl_2'] \overset{*}{\Rightarrow} \sigma$$

Case 1 M_1' is a constant a.

Since the computation terminates M_2' must be a constant b such that Constapply(a,b) exists. By inference rule 2 of the definition of T

$$T(Real(cl), t_1+t_2+1) = T(Real(cl_1)Real(cl_2), t_1+t_2+1) = Constapply(a,b)$$

Case 2 M_1' is an abstraction $\lambda x M_1''$.

By CBV transition rule 1 $[cl] \overset{*}{\Rightarrow} [[cl_1']cl_2'] \Rightarrow [(M_1'', e\{cl_2'/x\})] = [cl_3] \overset{n_3}{\Rightarrow} \sigma$ where $n_3 < n$. By induction $T(Real(cl_3), t_3) = Unload(\sigma)$ for some t_3. Let $Real(cl_1') = \lambda x M_1'''$. By Lemma 2 of Appendix I

$$Real(cl_3) = [Real(cl_2')/x]M_1'''$$

By inference rule 3 of the definition of T, $T(Real(cl), t_1+t_2+t_3+1) = Unload(\sigma)$.

$\square$