On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications

ALAIN DEUTSCH*
ICSLA Team
Laboratoire d'Informatique de l'Ecole Polytechnique (LIX)
91128 Palaiseau Cedex - France.
deutsch@poly.polytechnique.fr

Abstract

We present a static analysis method for determining aliasing and lifetime of dynamically allocated data in lexically scoped, higher-order, strict and polymorphic languages with first class continuations. The goal is validate program transformations that introduce imperative constructs such as destructive updatings, stack allocations and explicit deallocations in order to reduce the run-time memory management overhead. Our method is based on an operational model of higher order functional programs from which we construct statically computable abstractions using the abstract interpretation framework. Our method provides a solution to a problem left open [Hudak 86] : determining isolation of data in the case of higher order languages with structured data.

1 Introduction

Functional specifications are a powerful description tool. They are used in denotational specifications, functional languages and specification languages. Our goal is to implement efficiently functional specifications on conventional sequential computers. However, such specifications lack control over memory management : there are no means of controlling assignment and deallocation of heap allocated data structures such as tuples, sums, partial applications, arrays, numbers and continuations. Because of this, functional programs tend to be much slower than their imperative equivalents. As destructive updating operations have constant time and space cost, it is desirable to transform applicative updatings into imperative updatings (as shown in [Aasa, Holmstrom & Nilsson 88]).

We have developed a method for detecting opportunities to automatically transform applicative constructs into imperative constructs such as destructive updatings of composite objects (such as partial applications, arrays, complex numbers or continuations ...), bounded-extent allocations (for instance stack allocations) and explicit deallocations (i.e compile-time garbage collection). In order to validate these program transformations, two classes of informations are computed. The *liveness* of data structures is used to control bounded-extent allocations. The *isolation* of data structures controls the introduction of destructive operations. These informations are themselves computed from safe approximations of the possible states of an abstract machine simulating the execution of programs from which we compute approximate informations about *reachability* of data. These informations can be used even in the case of imperative languages, as the introduction of explicit deallocation commands in a programming language renders the language unsafe, as a program may deallocate a valid reference. Compile-time determination of liveness of data may be used to check that deallocations are safe. This method is an application of the formal framework of abstract interpretation [Cousot & Cousot 79,Cousot 81].

1.1 Related work

Several methods have been proposed to reduce the run-time cost of heap management :

Lifetime analyses have been proposed in [Barth 77,Hughes 87,Ruggieri & Murtagh 88]. These analyses are used to validate the replacement of indefinite extent allocations by bounded-extent allocations. [Hughes 87] describes a method suitable for higher-order purely functional programs with structured data, based on a combination of a forward analysis and a backward analysis (to determine transmission properties of procedures). [Chase 88] discusses the safety of such transformations.

Appel has shown in [Appel 87] that garbage collection can be faster that stack allocation. An example of this is provided by the Standard ML [Appel 89] garbage collector which is sufficiently efficient to allow the heap allocation of the entire run-time stack. In such a context it is probably not worthwhile to transform heap allocations into stack allocations. Unfortunately such a technique is not always usable, as it requires assignments not to be frequent, which is not the case with imperative languages or lazy languages (because of the need to update delay closures).

Another approach consists in replacing dynamic alloca-

*This work has been partly funded by the Greco de Programmation du CNRS

tions by static allocations, for example by replacing local variables by global variables [Raoult & Sethi 84,Raoult & Sethi 85,Schmidt 85,Kastens & Schmidt 86,Sestoft 89].

An alternative approach to the elimination of temporary data is symbolic composition, as proposed in [Wadler 88]. Given an expression $f(g(x))$, this method computes at compile time a procedure f_g such that $f(g(x)) = f_g(x)$, but which is less space consuming whenever the value of $g(x)$ is temporary. Although limited to first order linear programs, this approach has the advantage of improving also the time complexity of programs by eliminating multiple traversal of data structures.

Sharing analyses for purely functional languages have been proposed in : [Schwartz 78,Inoue, Seki & Yagi 88,Bloss 89,Jones & Le Metayer 89]. The goal of these analyses is to validate program transformations such as introduction of destructive updatings and explicit deallocations. [Schwartz 78] describes a verification system for user-supplied sharing declarations in a first order language without side-effects. Sharing is directly described by abstract values, and this can cause information to be lost across procedure calls. [Inoue, Seki & Yagi 88] presents a method to perform compile-time garbage collection of temporary results in a functional, lexically-scoped, strict, first order language with dynamically allocated data. It is based on the combination of an analysis that detects newly allocated cells and a transmission analysis. These analyses compute informations relative to a prespecified regular pattern (for example linearly linked lists), but have a cost linear in the size of the program. [Bloss 89] describes a method for determining isolation of data in the case of a lazy, first order language with flat arrays using path analysis. [Jones & Le Metayer 89] extends [Schwartz 78]. Rather than relying on user-supplied declarations, this method computes sharing information and is based on the combination of two backward analyses (transmission and necessity of data) and a forward analysis. The language is a lexically-scoped, strict, first order language with dynamically allocated data. Abstract values are made finite by a depth-limiting technique similar to that of [Jones & Muchnick 81].

Several sharing analyses for non-purely functional languages have been described. The method reported in [Cousot & Cousot 77b] is an alias analysis that computes at each program point a partition of the program variables into disjoint collections such that if two variables belong to distinct collections, then they cannot refer to the same record (even indirectly). [Jones & Muchnick 81,Jones & Muchnick 82] describe several methods to perform data flow analysis of languages with dynamic allocation and structured data. These methods are forward data flow analyses that compute descriptions of the possible structure of the values of variables in a first order list-processing language with destructive updating and dynamic allocation. The analysis of [Jones & Muchnick 81] computes at each program point a set of abstract stores. Each abstract store is a graph which is k-limited : no path from the roots has length $> k$. The method reported in [Jones & Muchnick 82] computes a single data description per program point. Recursively defined structures are approximated by the set of program points that allocated them, plus a retrieval function that maps program points to structure components. [Jones 81] describes a method to perform data and flow analysis of λ-terms under call-by-value and call-by-name. This is done by constructing a function that simulates the states reachable during the interpretation of a λ-term using a SECD like machine. [Coutant 86] describes an alias analysis for first order imperative languages. [Hudak 86] describes a method to compute approximations of reference counts of dynamically allocated data in a lexically-scoped, strict, first-order applicative language with dynamically allocated flat arrays. The analysis computes at each program point a set of pairs of environments mapping variables to (abstract) locations and stores mapping locations to approximate reference counts. [Neirynck, Panangaden & Demers 87] presents an alias analysis for a strict, higher-order language with side-effects, scalar data and bounded-extent allocation of mutable cells containing scalars. [Stransky 88] describes a general method to perform abstract interpretation of dynamically scoped, strict, first order languages with dynamically allocated mutable data. It computes for each program point an abstract environment and an abstract store represented by a graph. [Larus & Hilfinger 88] presents a method to determine aliasing of structured data in the case of a strict, lexically-scoped, first order language with dynamically allocated mutable data. It computes at each program point a graph modeling the set of possible stores than can arise. [Horwitz, Pfeiffer & Reps 89] describes a method for determining data dependences between program statements in a language with dynamically allocated data. It is an extension of [Jones & Muchnick 81].

[Weihl 80,Shivers 88] describe control-flow and call-graph estimation methods for languages with procedure parameters or first class procedures. These methods could be used to extend a first order analysis. But the resulting call graph could be too conservative in the case of programs which make intensive use of higher-order procedures. Moreover, in order to flow analyse a procedure call, we need to known what procedure is involved, but also its environment, so that a call graph is not sufficient.

A static analysis for a higher-order language with first class continuations was described in [Jouvelot & Gifford 89]. It is not based on abstract interpretation but on effect checking. This method can be used to detect stack allocability of objects, and relies to some extent on user supplied declarations. It does not however achieve the effect of sharing analysis.

Our goal is to develop a semantically based sharing and lifetime analysis method applyable to lexically-scoped, strict, higher-order languages with dynamically allocated data.

1.2 Overview

Section 2 recalls the framework of abstract interpretation. Section 3.1 describes a typical functional language which will be the subject of the discussion. This language will be described by means of an operational, state-transition based semantics that captures store-level details such as sharings. Liveness and isolation of data structures will be formulated by means of predicates on the set of reachable states (section 3.3). In this framework we construct an abstract semantics (section 4). We then construct approximate isolation predicates defined on approximate states (section 5). A summary of the correctness proof is then presented (section 6). We then conclude by a presentation of some results and possible extensions.

2 Preliminaries

2.1 Notations

If $f \in D_1 \to D_2$ and $S \in \wp(D_1)$, then $f(S)$ denotes $\{f(x) \mid x \in S\}$. If $D = D_1 + D_2$, then the injection functions are (by abuse of overloading) $D_1 \in D_1 \to D$ and $D_2 \in D_2 \to D$. If $s \in D^*$ and $d \in D$, then $d::s$ denotes $\langle d \rangle \S s$; $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ is the function than maps x_i to y_i ... and in the context were a total function is required, any $v \notin \{x_1, \ldots, x_n\}$ to $\bot$. If $s \in D^*$, then $(x \in s) \Leftrightarrow \exists n \in [1, \|s\|] : x = s \downarrow n$. Variables denoting sets or sequences are often starred, such as s^*. If f is a partial function, then $x \to_f y \Leftrightarrow x \in Dom(f) \wedge f(x) = y$. If $\to$ is a relation, then $\to^*$ is its reflexive, transitive closure, $post(\to)(S) = \{y \mid x \in S \wedge x \to y\}$, $pre(\to)(S) = \{x \mid y \in S \wedge x \to y\}$. If $f \in A \to A$ is a continuous function, A a complete lattice, $x \in A$ and $x \sqsubseteq f(x)$, then $lub f x$ is the least fixed point of f greater than x [Cousot 78, 2.7.0.1]. If $X \in \wp(A)$, and $\sqsubseteq$ is a partial ordering of A, then $\downarrow X = \{x' \mid x \in X \wedge x' \sqsubseteq x\}$.

2.2 Definition and Construction of Abstract Interpretations

We briefly recall the framework of abstract interpretation as defined by [Cousot & Cousot 79,Nielson 85].

The *standard semantics* (operational or denotational) of a program $[\![P]\!]$ is typically defined by a mapping $\mathcal{M}$ from states to states :

$$\mathcal{M}[\![P]\!] \in State \to State$$

As we wish to express properties w.r.t the set of all reachable states, the $\mathcal{M}$ function is extended point to point to sets of states, thus providing the *static semantics* (or collecting semantics) :

$$\mathcal{M}_S[\![P]\!] \in \wp(State) \to \wp(State)$$

An *abstract semantics* is defined by a triple $\langle State^\#, \mathcal{M}^\#, \langle \alpha, \gamma \rangle \rangle$, where $\langle State^\#, \sqsubseteq \rangle$ is a complete lattice that abstracts sets of states and $\mathcal{M}^\#$ is an abstraction of $\mathcal{M}_S$. The relationship between $\langle \wp(State), \subseteq \rangle$ and $\langle State^\#, \sqsubseteq \rangle$ is defined by the *pair of adjoined functions* $\langle \alpha, \gamma \rangle$: both α and γ are required to be monotonic and to satisfy [Cousot & Cousot 79, 5.3.0.1,5.3.0.4] :

$$\begin{array}{rcl} \alpha & \in & \wp(State) \to State^\# \\ \gamma & \in & State^\# \to \wp(State) \\ id & \subseteq & \gamma \circ \alpha \\ \alpha \circ \gamma & \sqsubseteq & id \end{array}$$

$\mathcal{M}^\#$ is a *correct upper approximation* of $\mathcal{M}_S$ iff for all P [Cousot & Cousot 79, 7.1.0.2] :

$$\alpha \circ \mathcal{M}_S[\![P]\!] \circ \gamma \sqsubseteq \mathcal{M}^\#[\![P]\!] \qquad (1)$$

How is $State^\#$ constructed ? It is possible to invent $State^\#$, and then the pair of adjoined functions. Another approach consists in inducing $State^\#$ from the structure of $State$. Indeed $State$ is constructed from basic operators such as $\times, +, \to$ as well as basic sets such as $\mathbb{N}$ and $\mathbb{B}$. For each such operator, it is possible to define several abstraction (and concretization) functionals varying in cost and precision. These functionals synthesize new abstraction (concretization) functions from existing ones.

2.3 Constructing Abstraction Functions

In this section we describe useful abstraction functions and abstraction functionals that will be used to construct abstract domains form concrete ones. Most of these abstraction functions were given in [Cousot & Cousot 79] and [Nielson 85]. The concretization functions are not described explicitly, since they are determined by the abstraction functions provided these are surjective complete-$\sqcup$-morphisms [Cousot & Cousot 79, 5.3.0.5].

We begin with (almost) simplest abstraction function: α_2 maps the empty set on $\bot$, any non empty set on $\top$.

$$\begin{array}{rcl} \alpha_2(\emptyset) & = & \bot \\ \alpha_2(\{e_1, \ldots\}) & = & \top \end{array}$$

The less informative abstraction function α_1 maps any set onto $\bot$:

$$\alpha_1(S) = \bot$$

Another useful abstraction, α_c, maps any singleton set onto itself. It is used for constant propagation.

$$\begin{array}{rcl} \alpha_c(\emptyset) & = & \bot \\ \alpha_c(\{x\}) & = & x \\ \alpha_c(\{x_1, x_2, \ldots\}) & = & \top \end{array}$$

Given a lifted set $A_\bot$, we may want to abstract sets of elements. We have two orderings : an ordering on the elements, and the inclusion ordering. As the element ordering is simple, we can define an abstraction that preserves both orderings as follows :

$$\alpha_\bot(\alpha) = \lambda S.\alpha(S \backslash \{\bot\})$$

There are several methods to abstract a set of pairs $\wp(A \times B)$. First of all the independent attribute method that treats members of A and B separately. Given two abstraction functions $\alpha_A \in \wp(A) \to A^\#, \alpha_B \in \wp(B) \to B^\#$, $\alpha_{\times I}$ computes an abstraction function mapping sets of pairs to (strict) pairs of abstractions.

$$\begin{array}{l} \alpha_{\times I}(\alpha_A, \alpha_B) \in \wp(A \times B) \to (A^\# \times B^\#) \\ \alpha_{\times I}(\alpha_A, \alpha_B) = \lambda S.\langle \alpha_A\{a \mid (a,b) \in S\}, \alpha_B\{b \mid (a,b) \in S\} \rangle \end{array}$$

The abstract domain $A^\# \times B^\#$ can be constructed using the smash product. This identifies elements having the same meaning (through the induced concretization function γ) : for instance : $\gamma(\langle \bot, x \rangle) = \gamma(\langle x, \bot \rangle) = \gamma(\bot) = \emptyset$.

However this abstraction ignores the relations between members of A and B. To obtain better precision, the relational method can be used :

$$\begin{array}{rcl} \alpha_{\times R}(\alpha_A, \alpha_B) & \in & \wp(A \times B) \to \wp(A^\# \times B^\#) \\ \alpha_{\times R}(\alpha_A, \alpha_B) & = & \lambda S.\{\langle \alpha_A\{a\}, \alpha_B\{b\} \rangle \mid \langle a, b \rangle \in S\} \end{array}$$

An intermediate approach consists in recording for each value of $\alpha_A(a)$ the abstraction of the set of corresponding B values [Cousot & Cousot 79]. This uses the isomorphism between $\wp(A \times B)$ and $A \to \wp(B)$.

$$\alpha_{\times E}(\alpha_A, \alpha_B) \in \wp(A \times B) \to (A^\# \to B^\#)$$
$$\alpha_{\times E}(\alpha_A, \alpha_B) = \lambda S. \bigsqcup \{\alpha_A\{a\} \mapsto \alpha_B\{b\} \mid \langle a, b\rangle \in S\}$$

More generally, we may want to reduce the cardinality of a set of abstract values $\wp(A^\#)$. This can be done by means of a surjective function $f \in A \to B$ that extracts from an abstract value a distinctive information (the tokens of [Jones & Muchnick 82]) :

$$\alpha_=(f, \alpha) \in \wp(A) \to (B \to A^\#)$$
$$\alpha_=(f, \alpha)(S) \bigsqcup \{f(x) \mapsto \alpha\{x\} \mid x \in S\}$$

All these abstraction functions are useful, depending on the degree of precision needed[1]

F.Nielson has proposed to abstract $\wp(A+B)$ by $A^\# + B^\#$ [Nielson 85, p.181] :

$$\alpha_{+N}(\alpha_A, \alpha_B) \in \wp(A + B) \to A^\# + B^\#$$
$$\alpha_{+N}(\alpha_A, \alpha_B) =$$
$$\lambda S. A^\#(\alpha_A\{x \mid A(x) \in S\}) \sqcup B^\#(\alpha_B\{x \mid B(x) \in S\})$$

However, this abstraction can be insufficiently precise : consider a polymorphic language. Then the values of a polymorphic variable can be of several monomorphic types, and α_{+N} would abstract these values to $\top$. A more precise abstraction consists in abstracting $\wp(A + B)$ by $A^\# \times B^\#$ [Cousot & Cousot 79, 10.1.0.4], based on the isomorphism $\wp(A+B) \simeq (\wp(A) \times \wp(B))$, the abstraction function is then:

$$\alpha_{+C}(\alpha_A, \alpha_B) \in \wp(A + B) \to (A^\# \times B^\#)$$
$$\alpha_{+C}(\alpha_A, \alpha_B) =$$
$$\lambda S. \langle \alpha_A\{x \mid A(x) \in S\}, \alpha_B\{x \mid B(x) \in S\}\rangle$$

Because of the isomorphism $A^* \simeq (A^0 + A^1 + \cdots)$, it is possible to define an abstraction functional for $\wp(A^*)$ using the abstractions for sums and products :

$$\alpha_*(\alpha_+, \alpha_\times, \alpha_A) = \alpha_+(\alpha_2, \alpha_A, \alpha_\times(\alpha_A, \alpha_A), \ldots)$$

However because A^* is isomorphic to an unbounded sum of products its abstraction through α_* and $\alpha_{\times C}$ would result in an infinite product. Using the isomorphism $A^\infty \simeq (\mathbb{N} \to A)$ and specializing α_* w.r.t α_{+C} and α_{+N} yields :

$$\alpha_{*C}(\alpha_\times, \alpha_A) \in \wp(A^*) \to (\mathbb{N} \to B^\#)$$
$$\alpha_{*C}(\alpha_\times, \alpha_A)(S) =$$
$$\bigsqcup \{\|s\| \mapsto \alpha_\times(\alpha_A, \ldots, \alpha_A)\{s\} \mid s \in S\}$$

$$\alpha_{*N}(\alpha_\times, \alpha_A) \in \wp(A^*) \to B^\#$$
$$\alpha_{*N}(\alpha_\times, \alpha_A)(S) =$$
$$\begin{cases} \bot & S = \emptyset \\ x & \alpha_{*C}(\alpha_\times, \alpha_A)(S) = \{n \mapsto x\} \\ \top & \text{otherwise} \end{cases}$$

[1]Example. Let α_s be the abstraction function that associates to each set of naturals its sign.

$$\begin{aligned} S &= \{\langle -1, -1\rangle, \langle 0, 0\rangle, \langle 1, 1\rangle, \langle -1, 1\rangle\} \\ \alpha_{\times I}(\alpha_s, \alpha_s)(S) &= \langle \top, \top\rangle \\ \alpha_{\times R}(\alpha_s, \alpha_s)(S) &= \{\langle -, -\rangle, \langle 0, 0\rangle, \langle +, +\rangle, \langle -, +\rangle\} \\ \alpha_{\times E}(\alpha_s, \alpha_s)(S) &= \{- \mapsto \top, 0 \mapsto 0, + \mapsto +\} \end{aligned}$$

A more approximate abstraction can be defined by identifying all elements of the sets of sequences :

$$\alpha_*(\alpha) \in A^* \to A^\#$$
$$\alpha_*(\alpha)(S) = \bigsqcup \{x \mid s \in S \wedge x \in s\}$$

Sets of partial functions can be abstracted by monotone maps. To ensure monotonicity we use the following function :

$$mon(f) = \bigsqcup \{x' \mapsto f(x') \mid x' \sqsubseteq x\}$$

Now a set of functions can be abstracted by :

$$\alpha_\to(\alpha_A, \alpha_B) \in \wp(A \to B) \to mon(A^\# \to B^\#)$$
$$\alpha_\to(\alpha_A, \alpha_B)(F) =$$
$$mon\left(\bigsqcup \{\alpha_A\{x\} \mapsto \alpha_B\{f(x)\} \mid f \in F \wedge x \in Dom(f)\}\right)$$

The abstract equivalent of application is application, the abstract equivalent

of updating is :

$$upd^\#(f, x, y) = f \sqcup \bigsqcup \{x' \mapsto y \mid x' \sqsubseteq x\}$$

Whenever the target of α_A (say $A^\#$) is such that any element is equal to the union of a finite number of atoms (an atom is a minimal, non $\bot$ element), and provided α_A is totally strict (α_A is strict and $\alpha_A(x) = \bot \Leftrightarrow x = \bot$), a more approximate version of $\alpha_\to$ can be given. Indeed we can restrict the domain of α_A to the atoms of $A^\#$. Let $\mathcal{D}_{A^\#}(a)$ be the atomic decomposition of $a \in A^\#$:

$$\alpha_{\to'}(\alpha_A, \alpha_B) \in \wp(A \to B) \to (A^\# \to B^\#)$$
$$\alpha_{\to'}(\alpha_A, \alpha_B)(F) =$$
$$\bigsqcup \{x' \mapsto \alpha_B\{f(x)\} \mid f \in F \wedge x \in Dom(f) \wedge$$
$$x' \in \mathcal{D}_{A^\#}(\alpha_A\{x\})\}$$

The abstract equivalent of application is no more application but the union of the images of the decomposition :

$$apply'^\#\langle f, x\rangle = \bigsqcup f(\mathcal{D}_{A^\#}(x))$$
$$upd'^\#(f, x, y) = f \sqcup \bigsqcup \{x' \mapsto y \mid x' \in \mathcal{D}_{A^\#}(x)\}$$

Generally $\alpha_{\to'}(\alpha_A, \alpha_B)(F)$ is less precise than $\alpha_\to(\alpha_A, \alpha_B)(F)$, unless α_A maps atoms to atoms, in which case they are equivalent in precision.

PROPOSITION 1 *If $\alpha_A \in \wp(A) \to A^\#$ and $\alpha_B \in \wp(B) \to B^\#$ are abstraction functions, α_A is totally strict, and every element of $A^\#$ is the union of a finite number of atoms, then $\alpha_{\to'}(\alpha_A, \alpha_B)$ is an abstraction function from $\wp(A \to B)$ to $mon(A^\# \to B^\#)$.*

3 Concrete Semantics

3.1 Operational Semantics

Rather than directly analyzing a high level language, we consider a language suited to the implementation of functional

$$
\begin{array}{lcl}
I & \in & \text{Cmd} \\
C & \in & \texttt{Cst} \\
P & \in & \texttt{Pgm} \\
L & \in & \texttt{Lab} \\
N & \in & \texttt{Num} \\
Pr & \in & \texttt{Prim} \quad = \quad \{+, =, \texttt{inject}, \texttt{tuple}, \texttt{cc}, \texttt{array} \ldots\} \\
P & \rightarrow & I_{L_1}, \ldots, I_{L_n} \\
I & \rightarrow & \texttt{Dup(N)} \mid \texttt{Cst(C)} \\
 & \mid & \texttt{Case}(L_1, \ldots, L_n) \mid \texttt{Jump(L)} \mid \texttt{Apply} \mid \texttt{Return} \\
 & \mid & \texttt{Closure(L,N)} \mid \texttt{Stop} \mid \texttt{Prim(Pr)}
\end{array}
$$

Figure 1: Syntax

languages (as in [Hecht 77,Nielson 85,Stransky 88] with other languages). This language is a variant of the SECD machine [Landin 64] not dissimilar to the FAM [Cardelli 84], to the the Ponder abstract machine [Fairbairn & Wray 86] and to the abstract machine of [Nielson & Nielson 86]. The syntax of the language is shown at figure 1. Note that it is possible to translate arbitrary programs into this language using for instance the two level semantics approach of [Nielson & Nielson 88]. Given a language L defined by its denotational semantics, we can analyse L programs by translating their TML denotations into our language.

Commands operate on states consisting of a value stack, a store, a reference to a continuation and a program counter. Stacks are represented by sequences of values, stores by finite mappings from locations (Loc) to stored values (Sv) and continuations by states not comprising stores. Expressible values (Ev) are either scalar objects (integers, ...) or reference to sharable objects such as sums, tuples, partial applications (closures) and continuations. Ev also contains a least element Ω which denotes undefined values. This induces a partial order on $State$.

$$
\begin{array}{rcl}
State & = & \text{Lab} \times Stk \times Store \times Cont \\
Stk & = & Ev^* \\
Store & = & Loc \rightarrow Sv \\
Cont & = & Loc_\Omega \\
Ev & = & (Int + Unit + Loc)_\Omega \\
Sv & = & Sum + Tup + Cls + Cnt + Vec \\
Sum & = & \mathbb{N} \times Ev \\
Tup & = & Ev^* \\
Cls & = & \text{Lab} \times Ev^* \times \mathbb{N} \\
Cnt & = & \text{Lab} \times Stk \times Cont \\
Vec & = & Ev^*
\end{array}
$$

A program P is a sequence of labeled commands. The $\texttt{Dup}(n)$ command pushes the nth stack value on top of stack, $\texttt{Cst}(C)$ pushes a constant, $\texttt{Case}(L_1, \ldots, L_n)$ branches to L_t, where t is the tag of the sum object on top of the stack, and pushes the untagged sum value, $\texttt{Jump}(L)$ transfers control (only forward, so that no loops can be constructed without **Apply**), **Apply** applies a procedure to an argument. If the procedure is a closure, then the application may result either in the construction of a new closure (a partial application), or in an effective application. If the procedure is a continuation, then the current local state is discarded. $\texttt{Closure}(L, N)$ constructs a closure object of order N of the procedure starting at label L, **Stop** halt the machine and $\texttt{Prim}(Pr)$ perform various data operations such as arithmetic ($+, -, \ldots$), tuple construction and component selection ($\texttt{tuple}_1, \texttt{tuple}_2, \ldots, \texttt{select}_1, \ldots$), sum injection ($\texttt{inject}_1, \texttt{inject}_2, \ldots$), array creation, selection, destructive and applicative updating (**array**,**sel**,**upd**,**fupd**). The cc primitive captures the current procedure continuation, which is sufficiently powerful to model the Scheme **call/cc** construct [Haynes & Friedman 87].

The meaning of a program will be defined by the partial state transition function τ mapping states to states (see figure 2). The meaning of constants is defined by the auxiliary function $\mathcal{K}$. Primitive operations are defined by $\mathcal{P}$.

$$
\begin{array}{lcl}
\mathcal{K} & \in & \text{Cst} \rightarrow Ev \\
\mathcal{P} & \in & (\text{Prim} \times Ev^* \times State) \rightarrow Ev \times Store
\end{array}
$$

New store locations are allocated by the *new* function. The exact structure of Loc is left unspecified yet, for instance whole states may be used as locations (although this would require domains rather than sets). Indeed the common usage of integers (or time stamps) as locations is related : to each location uniquely corresponds a state (not considering garbage collection).

$$
new \quad \in \quad State \rightarrow Loc
$$

We outline some typical primitive definitions :

$$
\begin{array}{l}
\mathcal{P}[\![\texttt{tuple}_n]\!]v'^*\langle L, v^*, \sigma, \kappa\rangle = \\
\quad \langle Loc(\ell), \sigma[\ell \mapsto Tup(v'^*)]\rangle \text{ where } \ell = new\langle L, v^*, \sigma, \kappa\rangle \\
\mathcal{P}[\![\texttt{inject}_n]\!]\langle v\rangle\langle L, v^*, \sigma, \kappa\rangle = \\
\quad \langle Loc(\ell), \sigma[\ell \mapsto Sum(n, v)]\rangle \text{ where } \ell = new\langle L, v^*, \sigma, \kappa\rangle \\
\mathcal{P}[\![+]\!]\langle Int(v_1), Int(v_2)\rangle\langle L, v^*, \sigma, \kappa\rangle = \langle Int(v_1 + v_2), \sigma\rangle \\
\mathcal{P}[\![+]\!]\langle \Omega, \_\rangle\langle L, v^*, \sigma, \kappa\rangle = \langle \Omega, \sigma\rangle \\
\mathcal{P}[\![+]\!]\langle \_, \Omega\rangle\langle L, v^*, \sigma, \kappa\rangle = \langle \Omega, \sigma\rangle \\
\mathcal{P}[\![\texttt{cc}]\!]\langle\rangle\langle L, v^*, \sigma, \kappa\rangle = \langle Cont(\kappa), \sigma\rangle
\end{array}
$$

Now the meaning of a program P in the initial configuration $c_0 = \langle L, v^*, \sigma, \kappa\rangle$ is defined as the (possibly infinite) sequence of states $\langle c_0, \tau(c_0), \tau(\tau(c_0)) \ldots\rangle$ whose last element (if the sequence is finite) is either an exit state or an error state.

3.2 Static Semantics

We now define the static semantics as the point to point extension of τ to sets of states [Cousot & Cousot 77a] :

$$
\begin{array}{rcl}
State_S & = & \wp(State) \\
\tau_S(c_S) & = & post(\rightarrow_\tau)(c_S)
\end{array}
$$

Given a set of initial states Φ, the set of its immediate successors is defined by $post(\rightarrow_\tau^*)(\Phi)$ which by [Cousot 81, 10-4] can be computed as $M_S(\Phi)$:

$$
M_S(c_S) = luis\,(\tau_S \cup id)\,c_S \qquad (2)
$$

3.3 Exact Isolation Predicates

The M_S function computes the (possibly infinite) set of successors of the initial states. From it, we can now provide a

$\tau \in State \to State$
$\tau(s) =$
case s **of**
$\langle [\![\mathbf{Dup}(n)_L]\!], v^*, \sigma, \kappa\rangle \to \langle L+1, (v^* \downarrow n)::v^*, \sigma, \kappa\rangle$
$\langle [\![\mathbf{Cst}(C)_L]\!], v^*, \sigma, \kappa\rangle \to \langle L+1, (\mathcal{K}[\![C]\!])::v^*, \sigma, \kappa\rangle$
$\langle [\![\mathbf{Case}(L_1,\ldots,L_n)_L]\!], Loc(\ell)::v^*, \sigma[\ell \mapsto Sum(i,v)], \kappa\rangle \to \langle L_i, v::v^*, \sigma[\ell \mapsto Sum(i,v)], \kappa\rangle$
$\langle [\![\mathbf{Jump}(L')_L]\!], v^*, \sigma, \kappa\rangle \to \langle L', v^*, \sigma, \kappa\rangle$
$\langle [\![\mathbf{Apply}_L]\!], v::Loc(\ell)::v^*, \sigma[\ell \mapsto Cls\langle L', v'^*, 1\rangle], \kappa\rangle \to$
$\quad \langle L', v::v'^*, \sigma[\ell \mapsto Cls\langle L', v'^*, 1\rangle, \ell' \mapsto Cnt\langle L+1, v^*, \kappa\rangle], \ell'\rangle$ **where** $\ell' = new(s)$
$\langle [\![\mathbf{Apply}_L]\!], v::Loc(\ell)::v^*, \sigma[\ell \mapsto Cls\langle L', v'^*, n+1\rangle], \kappa\rangle \to$
$\quad \langle L+1, Loc(\ell')::v^*, \sigma[\ell \mapsto Cls\langle L', v'^*, n+1\rangle, \ell' \mapsto Cls\langle L', v::v'^*, n\rangle], \kappa\rangle$ **where** $\ell' = new(s)$
$\langle [\![\mathbf{Apply}_L]\!], v::Loc(\ell)::v^*, \sigma[\ell \mapsto Cnt\langle L', v'^*, \kappa'\rangle], \kappa\rangle \to \langle L', v::v'^*, \sigma[\ell \mapsto Cnt\langle L', v'^*, \kappa'\rangle], \kappa'\rangle$
$\langle [\![\mathbf{Return}_L]\!], v::v^*, \sigma[\kappa \mapsto Cnt\langle L', v'^*, \kappa'\rangle], \kappa\rangle \to \langle L', v::v'^*, \sigma[\kappa \mapsto Cnt\langle L', v'^*, \kappa'\rangle], \kappa'\rangle$
$\langle [\![\mathbf{Closure}\langle L', n\rangle_L]\!], v^*, \sigma, \kappa\rangle \to \langle L+1, Loc(\ell)::v^*, \sigma[\ell \mapsto Cls\langle L', \langle\rangle, n\rangle], \kappa\rangle$ **where** $\ell = new(s)$
$\langle [\![\mathbf{Prim}(P)_L]\!], v_1::\cdots v_{arity[\![P]\!]}::v^*, \sigma, \kappa\rangle \to \langle L+1, v::v^*, \sigma, \kappa\rangle$ **where** $\langle v, \sigma'\rangle = \mathcal{P}[\![P]\!]\langle v_1, .., v_{arity[\![P]\!]}\rangle s$

Figure 2: The state transition function τ

semantic characterization of isolation : an expressible value v is isolated if it is not accessible : a value is accessible if there are no valid paths from active areas (stack and continuation) to the location referred to by v. This provides a definition of isolation as a property which we now make more precise :

DEFINITION 1 (ACCESSIBILITY) *A location $\ell \in Loc$ is accessible from a value v through the store σ iff $\pi_{Ev}(v, \ell, \sigma)$, where :*

$$\begin{aligned}
\pi_{Ev}(v,\ell,\sigma) &= (v \in Loc) \vee \pi_{Loc}((v|Loc),\ell,\sigma)\\
\pi_{Loc}(\ell_1,\ell_2,\sigma) &= (\ell_1 = \ell_2) \vee \bigvee_{v \in S(\sigma(\ell_1))} \pi_{Ev}(v,\ell_2,\sigma)\\
\pi_{Cont}(\kappa,\ell,\sigma) &= (\kappa \neq \Omega) \wedge \pi_{Loc}(\kappa,\ell,\sigma)
\end{aligned}$$

and $S(sv)$ are the directly accessible sons of sv :

$$\begin{aligned}
S &\in Sv \to Ev^*\\
S(Sum(t,v)) &= \langle v\rangle\\
S(Tup(v^*)) &= v^*\\
S(Cls\langle L, v^*, n\rangle) &= v^*\\
S(Cnt\langle L, v^*, \ell\rangle) &= \langle Loc(\ell)::v^*\rangle\\
S(Vec(v^*)) &= v^*
\end{aligned}$$

The active parts of a state are those that will be used in a future computation. We will define the restriction of a state c to its active parts as the least state equivalent to c. Two states can be defined as equivalent either if both fail to terminate, or if both produce identical outputs. We assume the existence of a function $output \in State \to Out$ that selects from a state the output computed so far (we could be more precise by adjoining an output store to *State*).

DEFINITION 2 (STATE EQUIVALENCE) *Two states $c_1, c_2 \in State^2$ are* equivalent *iff $c_1 \approx c_2$, where :*
$c_1 \approx c_2 \Leftrightarrow ((c_1 \to_\tau^* c_1' \not\to_\tau) \wedge (c_2 \to_\tau^* c_2' \not\to_\tau) \wedge (output(c_1') = output(c_2')))$

In the case of programs with partial output, this definition would not be suitable, as non terminating programs with different outputs would be considered as equivalent. A revised definition would be : given two computation traces $\langle c_1, \ldots\rangle$ and $\langle c_2, \ldots\rangle$, for each c_i there must exists a state c_j such that $output(c_i) = output(c_j)$ (if τ preserves the order of output). In this way, all non-terminating programs would not be considered as equivalent.

Now we can define the restriction of a state c_1, as the least defined state c_2 that is still equivalent to c_1 :

DEFINITION 3 (STATE RESTRICTION) *Given a state $c \in State$, the smallest state equivalent to c is $\mathcal{R}(c)$, where :*

$$\begin{aligned}
\mathcal{R} &\in State \to State\\
\mathcal{R}(c) &= \sqcap\{c' \mid c' \sqsubseteq c \wedge c' \approx c\}
\end{aligned}$$

The $\mathcal{R}$ function is a lower closure operator (i.e a reductive projection). Another use of projections in semantic analysis was reported in [Launchbury 87] in the context of binding time analysis.

DEFINITION 4 (ISOLATION) *The n-th stack value is always isolated in the context L if given an initial description $\Phi \in State_S$, $(I\ (\mathcal{R}(\mathcal{M}_S(\Phi)))\ L\ n)$ holds, where :*

$$\begin{aligned}
&I \in State_S \to \mathrm{Lab} \to \mathbb{N} \to \mathbb{B}\\
&I\ c_S\ L\ n \Leftrightarrow\\
&\quad (\forall\langle L, v^*, \sigma, \kappa\rangle \in c_S(L)), v = (v^* \downarrow n) \wedge v \in Loc \wedge\\
&\quad \neg\pi_{Cont}(\kappa, (v|Loc), \sigma) \wedge\\
&\quad \bigwedge_{1 \le i \le \|v^*\| \wedge i \neq n} \neg\pi_{Ev}((v^* \downarrow i), (v|Loc), \sigma)
\end{aligned}$$

4 Abstract Semantics

4.1 Partitioning the States

The first step consists in partioning sets of states by program point [Cousot & Cousot 77a]. To each program point is associated the set of all corresponding states. The correspondence between $State_S$ and $State_P$ is immediate and is based on the isomorphism $\wp(A \times B) \simeq A \to \wp(B)$:

$$\begin{aligned}
&State_P = \mathrm{Lab} \to \wp(Stk \times Store \times Dump)\\
&\tau_P \in State_P \to State_P\\
&\tau_P(c_P) = \bigsqcup\{\tau_{P1}\langle L, v^*, \kappa, \sigma\rangle \mid\\
&\qquad L \in Dom(c_P) \wedge \langle v^*, \sigma, \kappa\rangle \in c_P(L)\}\\
&\tau_{P1}(c) = \{L' \mapsto \{\langle v'^*, \sigma', \kappa'\rangle\} \mid c \to_\tau \langle L', v'^*, \sigma', \kappa'\rangle\}
\end{aligned}$$

The meaning of a whole program, given a description of the initial states $\Phi \in State_P$ is $M_P(\Phi)$ where :

$$M_P(c_P) = luis\ (\tau_P \cup id)\ c_P \qquad (3)$$

The next step is to construct appropriate abstractions of $State_P$. More precisely, we need to construct abstraction functions for each component of $State_P$: (sets of) value stacks, stores and continuations.

4.2 Abstracting Local Stacks

As the language does not allow loops (jump commands can only skip forward), local stacks have finite heights. Furthermore we assume that the set of all stacks obtainable at a given point have the same height. Thus the α_{*N} is sufficient :

$$\begin{array}{rcl} \alpha_{Stk} & \in & \wp(Stk) \to Stk^\# \\ \alpha_{Stk} & = & \alpha_{*N}(\alpha_{\times I}, \alpha_{Ev}) \\ Stk^\# & = & \alpha_{Stk}(\wp(Stk)) = (Ev^{\#^*})^\top_\bot \end{array}$$

Assuming that abstract operations are doubly strict, the abstractions of concatenation and projection are concatenation and projection.

4.3 Abstracting Stored Values

Stored values are abstracted using the standard abstraction functionals :

$$\begin{array}{rcl} \alpha_{Sum} & = & \alpha_{\times E}(id, \alpha_{Ev}) \\ Sum^\# & = & \mathbb{N} \to Ev^\# \\ \alpha_{Tup} & = & \alpha_{*G}(\alpha_{\times I}, \alpha_{Ev}) \\ Tup^\# & = & \mathbb{N} \to Ev^{\#^*} \\ \alpha_{Cls} & = & \alpha_{\times E}(id, \alpha_{\times I}(\alpha_{*N}(\alpha_{\times I}, \alpha_{Ev}), \alpha_c)) \\ Cls^\# & \simeq & \mathrm{Lab} \to (\mathbb{N} \times Ev^{\#^*}) \\ \alpha_{Cnt} & = & \alpha_{\times E}(id, \alpha_{\times I}(\alpha_{Stk}, \alpha_{Cont})) \\ Cnt^\# & \simeq & \mathrm{Lab} \to (Stk^\# \times Cont^\#) \\ \alpha_{Vec} & = & \alpha_*(\alpha_{Ev}) \\ Vec^\# & = & Ev^\# \\ \alpha_{Sv} & = & \alpha_{+G}(\alpha_{Sum}, \alpha_{Tup}, \alpha_{Cls}, \alpha_{Vec}) \\ Sv^\# & = & Sum^\# \times Tup^\# \times Cls^\# \times Cnt^\# \end{array}$$

The corresponding abstract injection operations are defined as follows :

$$\begin{array}{rcl} Sum^\#\langle t, v\rangle & = & \langle \{t \mapsto v\}, \bot, \bot, \bot, \bot\rangle \\ Tup^\#(v^*) & = & \langle \bot, \{\|v^*\| \mapsto v^*\}, \bot, \bot, \bot\rangle \\ Cls^\#\langle L, n, v^*\rangle & = & \langle \bot, \bot, \{L \mapsto \langle n, v^*\rangle\}, \bot, \bot\rangle \\ Cnt^\#\langle L, v^*, \kappa\rangle & = & \langle \bot, \bot, \bot, \{L \mapsto \langle v^*, \kappa\rangle\}, \bot\rangle \\ Vec^\#(v) & = & \langle \bot, \bot, \bot, \bot, v\rangle \end{array}$$

The projection operations are :

$$\begin{array}{rcl} Sum^{\#^{-1}}(S) & = & \{\langle t, v\rangle \mid v = (S \downarrow 1)(t)\} \\ Tup^{\#^{-1}}(S) & = & \{v^* \mid v^* = (S \downarrow 2)(n)\} \\ Cls^{\#^{-1}}(S) & = & \{\langle L, n, v^*\rangle \mid \langle n, v^*\rangle = (S \downarrow 3)(L)\} \\ Cnt^{\#^{-1}}(S) & = & \{\langle L, v^*, \kappa\rangle \mid \langle v^*, \kappa\rangle = (S \downarrow 4)(L)\} \\ Vec^{\#^{-1}}(S) & = & \{S \downarrow 5\} \end{array}$$

4.4 Abstracting Locations

Until now, the structure of Loc has not been specified. Let us suppose that each concrete $\ell \in Loc$ is a tuple representing the state in which ℓ has been allocated. That is, a location is composed of a label, a stack, a store and a dump. The new function is then : $new(s) = s$.

The lattice of abstract locations can be constructed using one of the abstraction functionals for products. As we wish precise approximation of locations, we use the relational abstraction function :

$$\alpha_{Loc} = \alpha_{\times R}(\alpha_{\mathrm{Lab}}, \alpha'_{Stk}, \alpha'_{Store}, \alpha'_{Cont})$$

Now several definitions of $\langle \alpha_{\mathrm{Lab}}, \alpha'_{Stk}, \alpha'_{Store}, \alpha'_{Cont}\rangle$ are suitable.

For instance using $\langle id, \alpha_1, \alpha_1, \alpha_1\rangle$ would distinguish locations by birth point : all objects allocated at a given program point are referenced by the same abstract location. In this case we have (using simplification isomorphisms) :

$$Loc^\# \simeq \wp(\mathrm{Lab})$$

This precisely models the approximation method of [Jones 81, p.389] and [Jones & Muchnick 82] later used by [Ruggieri & Murtagh 88,Mogensen 87]. Extensions of this method [Hudak 86,Stransky 88,Larus & Hilfinger 88] use more precise abstractions taking into account other state components such as continuations or initial cell values. They can be described by a suitable choice of the abstraction functions. For example the family of approximations proposed in [Hudak 86] can be modeled by $\alpha^{!n}_{Cont}$, where n is the order of approximation and :

$$\begin{array}{l} \alpha^{!n}_{Cont}(S) = \\ \quad \{\langle L_i \mid i \in [1,n] \wedge Cnt^\#\langle L_i, \_, \ell_{i+1}\rangle \in \sigma(\ell_i)\rangle \mid \\ \quad\ \langle \sigma, \ell_1\rangle \in S\} \end{array}$$

In that case :

$$Loc^\# \simeq \wp(\mathrm{Lab}^{n+1})$$

In the case of programs with recursive data structures such as trees, these abstractions may fail to detect unsharings. This is because the abstractions chosen for locations identify data that have different structures. For instance a non convergent binary tree and a convergent binary tree can be approximated in the same way. Storeless methods representing directly sharing information [Schwartz 78,Inoue, Seki & Yagi 88,Jones & Le Metayer 89] are more precise in these cases. But unfortunately they are less precise across procedure calls, and are not appropriate for languages with side-effects.

4.5 Abstracting Stores

A set of stores can be abstracted using the $\alpha_{\to}$ abstraction function. However, as abstract locations are sets, we can use the $\alpha_{\to'}$ abstraction, since each element of a power set is the union of atoms (in this case singleton sets), and that

$$
\begin{array}{l}
\tau''^{\#} \in Lstate^{\#} \to \wp(Lstate^{\#}) \\
\tau''^{\#}(s) = \\
\textbf{case } s \textbf{ of} \\
\quad \langle [\![\texttt{Dup}(n)_L]\!], v^*, \sigma, \kappa\rangle \to \{\langle L+1, (v^* \downarrow n)::v^*, \sigma, \kappa\rangle\} \\
\quad \langle [\![\texttt{Cst}(C)_L]\!], v^*, \sigma, \kappa\rangle \to \{\langle L+1, (K^{\#}[\![C]\!])::v^*, \sigma, \kappa\rangle\} \\
\quad \langle [\![\texttt{Case}(L_1, \ldots, L_n)_L]\!], v::v^*, \sigma, \kappa\rangle \to \{\langle L_i, v'::v^*, \sigma, \kappa\rangle \mid \langle i, v'\rangle \in Sum^{\#^{-1}}(apply'^{\#}\langle \sigma, Loc^{\#^{-1}}(v)\rangle)\} \\
\quad \langle [\![\texttt{Jump}(L')_L]\!], v^*, \sigma, \kappa\rangle \to \{\langle L', v^*, \sigma, \kappa\rangle\} \\
\quad \langle [\![\texttt{Apply}_L]\!], v::v'::v^*, \sigma, \kappa\rangle \to \\
\qquad \{\langle L', v::v'^*, upd'^{\#}\langle \sigma, \ell', Cnt\langle L+1, v^*, \kappa\rangle\rangle, \ell'\rangle \mid \\
\qquad \langle L', n, v'^*\rangle \in Cls^{\#^{-1}}(apply'^{\#}\langle \sigma, Loc^{\#^{-1}}(v')\rangle) \wedge n = 1 \wedge \ell' = new^{\#}(s)\} \cup \\
\qquad \{\langle L+1, Loc^{\#}(\ell)::v^*, upd'^{\#}\langle \sigma, \ell, Cls^{\#}\langle L', n-1, v::v'^*\rangle\rangle, \kappa\rangle \mid \\
\qquad \langle L', n, v'^*\rangle \in Cls^{\#^{-1}}(apply'^{\#}\langle \sigma, Loc^{\#^{-1}}(v')\rangle) \wedge n > 1 \wedge \ell' = new^{\#}(s)\} \cup \\
\qquad \{\langle L', v::v'^*, \sigma, \kappa'\rangle \mid \langle L', v'^*, \kappa'\rangle \in Cnt^{\#^{-1}}(apply'^{\#}\langle \sigma, Loc^{\#^{-1}}(v')\rangle)\} \\
\quad \langle [\![\texttt{Return}_L]\!], v::v^*, \sigma, \kappa\rangle \to \{\langle L', v::v'^*, \sigma, \kappa'\rangle \mid \langle L', v'^*, \kappa'\rangle = Cnt^{\#^{-1}}(apply'^{\#}\langle \sigma, Loc^{\#^{-1}}(\kappa)\rangle)\} \\
\quad \langle [\![\texttt{Closure}(L', N)_L]\!], v^*, \sigma, \kappa\rangle \to \{\langle L+1, Loc^{\#}(\ell), upd'^{\#}\langle \sigma, \ell, Cls^{\#}\langle L', N, \langle\rangle\rangle\rangle, \kappa\rangle \mid \ell = new^{\#}(s)\} \\
\quad \langle [\![\texttt{Stop}_L]\!], v^*, \sigma, \kappa\rangle \to \{\langle L, v^*, \sigma, \kappa\rangle\} \\
\quad \langle [\![\texttt{Prim}(Pr)_L]\!], v_1::\cdots v_{arity[\![Pr]\!]}::v^*, \sigma, \kappa\rangle \to \{\langle L+1, v::v^*, \sigma', \kappa\rangle \mid \langle v, \sigma'\rangle = P^{\#}[\![Pr]\!]\langle\langle v_1, \cdots, v_{arity[\![Pr]\!]}\rangle, s\rangle\} \\
\quad \langle L, v^*, \sigma, \kappa\rangle \to \{\langle L, v^*, \sigma, \kappa\rangle\}
\end{array}
$$

$$
\begin{array}{l}
\tau'^{\#} \in Lstate^{\#} \to State^{\#} \\
\tau'^{\#}(s) = \{L \mapsto \{\rho^{\#}\langle v^*, \sigma, \kappa\rangle \mapsto \langle v^*, \sigma, \kappa\rangle\} \mid \langle L, v^*, \sigma, \kappa\rangle \in \tau''^{\#}(s)\}
\end{array}
$$

$$
\begin{array}{l}
\tau^{\#} \in State^{\#} \to State^{\#} \\
\tau^{\#}(c) = \bigsqcup\{\tau'^{\#}\langle L, v^*, \sigma, \kappa\rangle \mid L \in Dom(c) \wedge \langle t, \langle v^*, \sigma, \kappa\rangle\rangle \in c(L)\}
\end{array}
$$

Figure 3: The abstract function $\tau^{\#}$

α_{Loc} is totally strict :

$$
\begin{array}{rcl}
\alpha_{Store} & = & \alpha_{\to'}(\alpha_{Loc}, \alpha_{Sv}) \\
Store^{\#} & = & Loc^{\#} \to Sv^{\#}
\end{array}
$$

4.6 Abstracting Values

How can a set of values $\{v_1, \ldots\} \in \wp(Ev)$ be approximated ? $\wp(Ev)$ is a set of sum objects. As the language we are analyzing is polymorphic, several instances of a variable may be bound to values of different types. Thus we use the α_{+C} abstractor. Furthermore as a value can be undefined (the Ω value), we use the α_{Ω} abstraction :

$$
\begin{array}{rcl}
\alpha_{Ev} & = & \alpha_{\Omega}(\alpha_{+C}(\alpha_{Int}, \alpha_{Unit}, \alpha_{Loc})) \\
\alpha_{Int} & = & \alpha_2 \\
\alpha_{Unit} & = & \alpha_2 \\
Ev^{\#} & = & 2 \times 2 \times Loc^{\#}
\end{array}
$$

Note that the choice of α_{Int} is arbitrary. For instance we could have chosen an abstraction suitable for constant propagation or range estimation.

The abstract injection and projection functions are :

$$
\begin{array}{rclrcl}
Int^{\#}(n) & = & \langle n, \bot, \bot\rangle & Int^{\#^{-1}}(v) & = & v \downarrow 1 \\
Unit^{\#}(u) & = & \langle \bot, u, \bot\rangle & Unit^{\#^{-1}}(v) & = & v \downarrow 2 \\
Loc^{\#}(\ell) & = & \langle \bot, \bot, \ell\rangle & Loc^{\#^{-1}}(v) & = & v \downarrow 3
\end{array}
$$

4.7 Abstracting continuations

As procedure continuations are represented by locations, we can use the same abstraction as for locations, lifted by α_{Ω} :

$$
\begin{array}{rcl}
\alpha_{Cont} & = & \alpha_{\Omega}(\alpha_{Loc}) \\
Cont^{\#} & = & Loc^{\#}
\end{array}
$$

4.8 Abstracting States

Abstract states can be constructed following two approaches. It is first possible to consider the set of possible triples of abstract stacks, abstract stores and abstract continuations using $\alpha_{\times R}$. This corresponds to the relational method: the analysis would determine relations between stack components (see for instance [Jones & Muchnick 81,Hudak 86,Horwitz, Pfeiffer & Reps 89]). However a fully relational analysis can have a cost exponential in the size of the program, as each n-way conditional with m predecessors can yield nm successors. We can use the independent attribute method (as in [Jones & Muchnick 82,Stransky 88,Larus & Hilfinger 88]), by ignoring relations between stacks, stores and continuations using $\alpha_{\times I}$.

$$
\begin{array}{rcll}
\alpha_{State} & = & \lambda S.(\alpha_{\times I}(\alpha_{Stk}, \alpha_{Store}, \alpha_{Cont}) \circ S) & \\
State^{\#} & = & \alpha_{State}(State_P) & (4) \\
 & = & \text{Lab} \to Stk^{\#} \times Store^{\#} \times Cont^{\#} & (5)
\end{array}
$$

Alternatively, the analysis can be made more precise by using a limited form of relational analysis [Sharir & Pnueli 81,Jones & Muchnick 82,Stransky 88,Mogensen 89]. In this case we have :

$$
\alpha_{State} = \lambda S.(\alpha_{=}(\rho, \alpha_{\times R}(\alpha_{Stk}, \alpha_{Store}, \alpha_{Cont})) \circ S)
$$

$$State^{\#} = \text{Lab} \to (T \to (Stk^{\#} \times Store^{\#} \times Cont^{\#})) \quad (6)$$

where $\rho \in (Stk \times Store \times Cont) \to T$. The precision of the analysis will now depend on the choice of the ρ function. Setting $\rho = \lambda\langle v^*, \sigma, \kappa\rangle.\top$ yields the non relational analysis (5). Setting $T = \mathbb{N}_\perp{}^*$ and ρ as a function that extracts from the stack the tags of the sum objects directly accessible would yield a semi relational scheme ([Cousot & Cousot 79, 10.2.0.2]). We have found in practice that choosing a ρ that extracts from the current continuation the set of successive return points yields quite precise results ($T = \wp(\text{Lab})$) :

$$\begin{array}{l}\rho \in (Stk \times Store \times Cont) \to \wp(\text{Lab}) \\ \rho\langle v^*, \sigma, \kappa_1\rangle = \\ \quad \{L_{i+1} \mid \kappa_i \in Dom(\sigma) \wedge \langle L_{i+1}, v^*{}_{i+1}, \kappa_{i+1}\rangle = \sigma(\kappa_i)\}\end{array}$$

The abstraction $\rho^{\#}$ of ρ can be derived directly.

A further abstraction can be defined in order to get a smaller domain. We can ignore relations between stores and program points, thus yielding a single global store (the retrieval function of [Jones & Muchnick 82]) :

$$State^{\#} = Store^{\#} \times (\text{Lab} \to Stk^{\#} \times Cont^{\#}) \quad (7)$$

In any case the order on $State^{\#}$ is consistent both with the subset ordering on $State$ (because abstraction functions preserve order) and to the element ordering on $State$ (by definition of α_{Ev}) :

PROPOSITION 2 $\forall(C_1, C_2) \in State_P{}^2, (\forall L \in \text{Lab}, \forall c_1 \in C_1(L), \exists c_2 \in C_2(L) : c_1 \sqsubseteq_{State} c_2) \Rightarrow \alpha_{State}(C_1) \sqsubseteq_{State^{\#}} \alpha_{State}(C_2)$

4.9 Abstracting the Transition Function

The abstract equivalent of the transition function τ is shown at figure 3. The $\tau'^{\#}$ function computes the successors of an abstract state. The following auxiliary functions are used :

$$\begin{array}{rcl} Lstate^{\#} & = & \text{Lab} \times Stk^{\#} \times Store^{\#} \times Cont^{\#} \\ new^{\#} & \in & Lstate^{\#} \to Loc^{\#} \\ \mathcal{K}^{\#} & \in & \text{Cst} \to Ev^{\#} \end{array}$$

The exact definition of primitives depends on the definition of the corresponding abstraction functions. For instance :

$$\begin{array}{l} new^{\#}\langle L, v^*, \sigma, \kappa\rangle = \{L\} \\ p^{\#}[\![\text{tuple}_n]\!]v'^*\langle L, v^*, \sigma, \kappa\rangle = \\ \quad \langle Loc^{\#}(\ell), upd'^{\#}\langle \sigma, \ell, Tup^{\#}(v'^*)\rangle\rangle \\ \quad \textbf{where } \ell = new^{\#}\langle L, v^*, \sigma, \kappa\rangle \\ p^{\#}[\![+]\!]\langle v_1, v_2\rangle\langle L, v^*, \sigma, \kappa\rangle = \\ \quad \langle Int^{\#}(Int^{\#-1}(v_1)) +^{\#} (Int^{\#-1}(v_2)), \sigma\rangle \\ p^{\#}[\![\text{cc}]\!]v'^*\langle L, v^*, \sigma, \kappa\rangle = \langle Loc^{\#}(\kappa), \sigma\rangle \end{array}$$

The approximate analysis of a whole program given the abstract initial conditions Φ is $M^{\#}(\Phi)$ where :

$$M^{\#}(c) = luis\ (\tau^{\#} \sqcup id)\ c \quad (8)$$

4.10 Abstracting the restriction function $\mathcal{R}$

The restriction of a state to its necessary components is in essence a backward problem. Thus a precise solution requires a backward analysis that computes from an abstract state the smallest necessary predecessors (w.r.t $\sqsubseteq_{State^{\#}}$) compatible with the descendants of the initial states. More precisely, we look for a monotonic function $\tau_B{}^{\#}$ such that given a set of initial states Φ, a correct upper approximation of the initial states $\Phi^{\#}$ and a concrete state $s \in State$ (with $\wp(State) \simeq State_P$) :

$$\begin{array}{l} \alpha_{State}(\mathcal{R}[pre(\to_\tau)(s) \cap \downarrow post(\to_\tau^*)(\Phi)]) \subseteq \quad (9) \\ \tau_B{}^{\#}(\Phi^{\#})(\alpha_{State}\{s\}) \end{array}$$

The $\tau_B{}^{\#}$ function itself is constructed with the help of an auxiliary function $B^{\#} \in Lstate^{\#} \times Lstate^{\#} \to Lstate^{\#}$, which given an abstract state c and one of its abstract successors c', computes a restriction of c sufficient to generate c'. The $B^{\#}$ function is directly derived from the $\tau'^{\#}$ function and is not shown here.

Then the approximate analysis of a whole program given the abstract entry states $\Phi \in State^{\#}$ and the abstract exit states $\Psi \in State^{\#}$ is $M_B{}^{\#}(\Phi, \Psi)$:

$$M_B{}^{\#}(\Phi, \Psi) = luis\ (\tau_B{}^{\#}(M^{\#}(\Phi)) \sqcup id)\ \Psi \quad (10)$$

Rather than providing explicitly an abstraction of $\mathcal{R}$, we have provided a backward analysis, which restricts the result of the forward analysis by computing backward the minimal states necessary to meet an output specification.

4.11 Correctness

In this section we provide a summary of the correctness proof of our abstract interpretation. We refer to [Deutsch 89] for details. The proof consists in showing that the abstract functions are correct approximations of their static counterparts.

First, we show that α_{State} is a correct abstraction function.

PROPOSITION 3 *α_{State} is a complete-$\sqcup$-morphism.*

DEFINITION 5 *An abstract state $c \in State^{\#}$ is a correct upper approximation of $c_P \in State_P$ by α_{State} iff $\alpha_{State}(c_P) \sqsubseteq c$*

We next show that $\tau^{\#}$ is a correct upper approximation of τ_P: given a correct upper approximation of a state c_P, $\tau^{\#}$ computes a correct upper approximation of $\tau_P(c_P)$. This will allow us to prove that $M^{\#}$ is a correct upper approximation of M_S. We give a lemma stating the correctness of $\tau'^{\#}$ w.r.t τ_{P1}.

PROPOSITION 4 *$\tau'^{\#}$ is safe : for all $c_P \in State_P$, for all $c \in State^{\#}$ such that c is a correct upper approximation of c_P:*

$$\begin{array}{l} \forall L \in Dom(c_P), \\ \forall\langle v^*, \sigma, \kappa\rangle \in c_P(L), \\ \forall\langle v^{\#*}, \sigma^{\#}, \kappa^{\#}\rangle = c(L), \\ \quad \alpha_{State}(\tau_{P1}\langle L, v^*, \sigma, \kappa\rangle) \sqsubseteq \tau'^{\#}\langle L, v^{\#*}, \sigma^{\#}, \kappa^{\#}\rangle \end{array}$$

PROOF : *By enumeration of the possible commands.* □

PROPOSITION 5 $\mathcal{M}^\#$ *is a correct upper approximation of* $\mathcal{M}_P$.

PROOF : *First show that* $\tau^\#$ *is a correct upper approximation of* τ_P. *Then let* $f_1(c_P) = c_P \cup \tau_P(c_P)$ *and* $f_2(c) = c \sqcup \tau^\#(c)$. *Let* $Q(c_P, c) \Leftrightarrow \alpha_{State}(c_P) \sqsubseteq c$, *then we show that* $Q(lub f_1 c_P, lub f_2 c)$ *by fixed point induction (Q is an inclusive predicate). From this we deduce that* $\alpha_{State} \circ \mathcal{M}_P \sqsubseteq \mathcal{M}^\# \circ \alpha_{State}$ *[Cousot 81, Theorem 10-25].* □

PROPOSITION 6 $\mathcal{M}_B{}^\#$ *is safe : for all* $\Phi \in State_P$, *let* $\Psi \in State_P$ *be the exit states reachable from* Φ, *then :* $\alpha_{State}(\mathcal{R}(\mathcal{M}_P(\Phi))) \sqsubseteq \mathcal{M}_B{}^\#(\alpha_{State}(\Phi), \alpha_{State}(\Psi))$

5 Approximate Isolation Predicates

Isolation of data is detected by performing a post analysis on the abstract states. We define a monotone (w.r.t implication ordering on $\mathbb{B}$) approximate isolation predicate $I^\# \in State^\# \to \mathrm{Lab} \to \mathbb{N} \to \mathbb{B}$. It must be related to I by the following property :

$$I^\#(\alpha_{State}(c_P))\, L\, n \Rightarrow I(c_P)\, L\, n \qquad (11)$$

DEFINITION 6 (ACCESSIBILITY ESTIMATES)

$$\pi^\#{}_{Ev}(v, \ell_1, \sigma, L) = \bigvee_{\ell_2 \in Loc^{\#-1}(v)} \pi^\#{}_{Loc}(\ell_2, \ell_1, \sigma, L)$$

$$\pi^\#{}_{Loc}(\ell, \ell, \sigma, L) = tt$$

$$\pi^\#{}_{Loc}(\ell_1, \ell_2, \sigma, \{\ell_1\} \cup L) = ff$$

$$\pi^\#{}_{Loc}(\ell_1, \ell_2, \sigma, L) = \bigvee_{v^* \in S^\#(\sigma(\ell))} \bigvee_{v \in v^*} \pi^\#{}_{Ev}(v, \ell, \sigma, \{\ell\} \cup L)$$

$$\pi^\#{}_{Cont}(\kappa, \ell, \sigma) = \pi^\#{}_{Ev}(Loc^\#(\kappa), \ell, \sigma, \emptyset)$$

The $S^\#$ *function extracts from an abstract storable value the set of sequences of abstract expressible values simultaneously accessible :*

$$\begin{aligned} S^\# &\in Sv^\# \to \wp(Ev^{\#*}) \\ S^\#(sv) &= \{\langle v \rangle \mid \langle \_, v \rangle \in Sum^{\#-1}(sv)\} \cup \\ &\quad \{v^* \mid v^* \in Tup^{\#-1}(sv)\} \cup \\ &\quad \{v^* \mid \langle \_, \_, v^* \rangle \in Cls^{\#-1}(sv)\} \cup \\ &\quad \{Loc^\#(\ell) :: v^* \mid \langle \_, v^*, \ell \rangle \in Cnt^{\#-1}(sv)\} \\ &\quad \{v \mid v \in Vec^{\#-1}(sv)\} \end{aligned}$$

DEFINITION 7 (ISOLATION ESTIMATES) *The* $I^\#$ *function is defined as follows:*

$$\begin{aligned} &I^\# \in State^\# \to \mathrm{Lab} \to \mathbb{N} \to \mathbb{B} \\ &I^\#\, c\, L\, n \Leftrightarrow \\ &\quad \forall \langle t \rangle (v^*, \sigma, \kappa) = c(L) \wedge \ell \in Loc^{\#-1}(v^* \downarrow n) \wedge \\ &\quad \neg \pi^\#{}_{Cont}(\kappa, \ell, \sigma) \wedge \bigwedge_{1 \le i \le \|v^*\| \wedge i \ne n} \neg \pi^\#{}_{Ev}(v^* \downarrow i, \ell, \sigma, \emptyset) \end{aligned}$$

PROPOSITION 7 $I^\#$ *provides safe isolation estimates : the property (11) is satisfied by* $I^\#$.

6 Results

6.1 Implementation

A preliminary prototype has been implemented in ML. This includes the analyser as well as a compiler for a subset of ML. The analyser itself implements the $\mathcal{M}^\#$ and $\mathcal{M}_B{}^\#$ functions. The crucial efficiency point is the order in which the program points are processed during the iteration. Classical data flow analysers perform a pre-analysis in order to determine a node listing from the dependencies between program points [Kennedy 76]. But we can not do so, because the control flow is not a priori available, as our language comprises higher-order procedures. Several evaluation orders have been tried. The most efficient we have experimented consists in locally iterating each procedure in turn until global stabilization. But more work is required to find a formal solution.

6.2 Examples

As an example, we show the analysis of a continuation-based denotational-like specification of a call by value λ-calculus with constants and a `call/cc`-like construct (figure 4). After a straightforward translation into our language (syntactic domains replaced by disjoint sum types, lambda-lifting,...), the analysis correctly recognizes that the continuations are non-isolated (possibly shared) with the base semantics. But if the equation corresponding to *cwcc* is suppressed, then the continuations are shown to be isolated just before application, and may thus be discarded during invocation. Indeed we have tried our method on a specification of a full core Scheme similar to that of [Rees & Clinger 86]. As with the last example, our analysis correctly recognizes that the continuations can be deallocated before invocation if the `call/cc` construct is suppressed. Moreover, the store component is shown to be isolated before updates (i.e : single-threaded). The example arrays programs of [Aasa, Holmstrom & Nilsson 88] have been successfully analysed.

6.3 Extensions

The work reported here is a first attempt to solve the problem of sharing determination for higher-order languages ; more work is needed to provide more precise abstractions for structured data.

Although we have considered the case of a low-level language, we could probably reformulate our work without much change in the case of higher level, expression oriented languages.

We have considered the case of a language with lexically scoped names. However, many languages support constructs which introduce dynamically scoped names, for instance the exception mechanisms of ML or ADA. The method proposed in [Stransky 88] could be used to handle such constructs.

We have not discussed how the isolation informations can be used to transform programs. Work this area is reported in [Ruggieri & Murtagh 88,Inoue, Seki & Yagi 88,Jones & Le Metayer 89].

Although we have discussed a specific problem, several other problems can be solved in an uniform manner using our analysis. For instance the problem of detecting sharing of partial applications [Goldberg 87] can be solved by examining the abstract states corresponding to applications. We

Syntax

$V \in Var$
$C \in Cst$
$E \in Exp$
$E \rightarrow V|C|\lambda V.E|(E_1\ E_2)|(cwcc\ E)$

Domains

$e \in Ev = D + P + K$ *values*
$D = Cst$ *constants*
$P = K \rightarrow K$ *procedures*
$\kappa \quad K = Ev \rightarrow Ev$ *continuations*
$\rho \in U = Var \rightarrow Ev$ *environments*

Valuations

$apply \in Ev \rightarrow K \rightarrow K$
$apply(Dc)\kappa = \lambda v.\bot$
$apply(Pf)\kappa = f(\kappa)$
$apply(K\kappa')\kappa = \kappa'$

$\mathcal{E} \in \text{Exp} \rightarrow U \rightarrow K \rightarrow Ev$
$\mathcal{E}[\![V]\!]\rho\kappa = \kappa(\rho[\![V]\!])$
$\mathcal{E}[\![C]\!]\rho\kappa = \kappa[\![C]\!]$
$\mathcal{E}[\![\lambda V.E]\!]\rho\kappa = \kappa(\lambda\kappa' v.\mathcal{E}[\![E]\!](\rho[[\![V]\!] \mapsto v])\ \kappa')$
$\mathcal{E}[\![(E_1\ E_2)]\!]\rho\kappa = \mathcal{E}[\![E_1]\!]\rho(\lambda v.\mathcal{E}[\![E_2]\!]\ \rho\ (apply\ v\ \kappa))$
$\mathcal{E}[\![(cwcc\ E)]\!]\rho\kappa = \mathcal{E}[\![E]\!]\rho(\lambda v.apply\ v\ \kappa\ (K\kappa))$

Figure 4: Example specification of a λ-language with call/cc

can then derive propositions such as : every partial application of procedure f to k arguments is not shared. In order to extend our analysis to languages with call by need, we could formulate the semantics of a lazy language using explicit representations of delayed expressions as self-modifying procedural thunks. Our analysis method may prove sufficiently powerful to handle such cases.

7 Conclusion

We have presented a formal method for statically estimating sharings and lifetimes of dynamically allocated data in a higher order language with first class continuations. Our method is based on the abstract interpretation of a suitable language defined by an operational semantics that explicits details such as storage allocation and sharing. An exact formulation of the problem was given, then a simulation method that computes a superset of the reachable states has been constructed. The correctness of this simulation was established following the abstract interpretation proof method.

References

[Aasa, Holmstrom & Nilsson 88] A. Aasa, S. Holmström, and C. Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28:490–503, 1988.

[Appel 87] A. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, (25):275–279, Jun. 1987.

[Appel 89] A. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, Feb. 1989.

[Barth 77] J.M. Barth. Shifting garbage collection overhead to compile time. *CACM*, 20(7):513–518, Jul. 1977.

[Bloss 89] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *Conference on Functional Programming Languages and Computer Architecture*, pages 26–38, ACM Press, London, Sep. 1989.

[Cardelli 84] L. Cardelli. Compiling a functional language. In *Symposium on LISP and Functional Programming*, pages 208–209, ACM, 1984.

[Chase 88] D.R. Chase. Safety considerations for storage allocation optimizations. In *SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 1–9, Atlanta, Jun. 1988.

[Cousot & Cousot 77a] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, Jan. 1977.

[Cousot & Cousot 77b] P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. *SIGPLAN Notices*, 12(3):77–94, Mar. 1977.

[Cousot & Cousot 79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[Cousot 78] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* Thèse d'état, Mar. 1978. Université scientifique et médicale de Grenoble.

[Cousot 81] P. Cousot. *Program Flow Analysis: Theory and Applications*, chapter Semantic foundations of program analysis, pages 303–342. Prentice-Hall, 1981.

[Coutant 86] D. Coutant. Retargetable high-level alias analysis. In *13th Annual ACM Symposium on Principles of Programming Languages*, pages 110–118, Jan. 1986.

[Deutsch 89] A. Deutsch. *On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications (extended version).* Research Report LIX/RR/89/(to appear), Ecole Polytechnique, 91128 Palaiseau, France, 1989.

[Fairbairn & Wray 86] J. Fairbairn and S.C. Wray. Code generation techniques for functional languages. In *Conference Record of the 1986 ACM symposium on LISP and Functional Programming*, pages 94–104, Aug. 1986.

[Goldberg 87] B. Goldberg. Detecting sharing of partial applications in functional programs. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 408–425, Springer Verlag, Sep. 1987. Volume 274 of *Lecture Notes on Computer Science*.

[Haynes & Friedman 87] C.T. Haynes and D.P. Friedman. Embeding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, Oct. 1987.

[Hecht 77] M.S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, New York, 1977.

[Horwitz, Pfeiffer & Reps 89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Conference on Programming Language Design and Implementation*, pages 28–40, Jun. 1989. Volume 24 of *SIGPLAN Notices*.

[Hudak 86] P. Hudak. A semantic model of reference counting and its abstraction. In *Conference Record of the 1986 ACM symposium on LISP and Functional Programming*, pages 351–363, Aug. 1986.

[Hughes 87] J. Hughes. Backward analysis of functional programs. In D. Bjorner, A.P. Ershov, and N.D Jones, editors, *Proc. Workshop on Partial Evaluation and Mixed Computation*, pages 155–169, North-Holland, Denmark, Oct. 1987.

[Inoue, Seki & Yagi 88] K. Inoue, H. Seki, and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Transactions on Programming Languages and Systems*, 10(4):555–578, Oct. 1988.

[Jones & Le Metayer 89] S.B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Conference on Functional Programming Languages and Computer Architecture*, pages 54–74, ACM Press, London, Sep. 1989.

[Jones & Muchnick 81] N.D. Jones and S. Muchnick. *Program Flow Analysis: Theory and Applications*, chapter Flow Analysis and Optimisation of Lisp-like structures, pages 102–131. Prentice-Hall, New Jersey, 1981.

[Jones & Muchnick 82] N.D. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *9th Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, ACM Press, 1982.

[Jones 81] N.D. Jones. Flow analysis of lambda expressions. In *Symposium on Functional Languages and Computer Architecture*, pages 376–401, Chalmers University of Technology, Goteborg,Sweden, Jun. 1981.

[Jouvelot & Gifford 89] P. Jouvelot and D.K. Gifford. Reasoning about continuations with control effects. In *Conference on Programming Language Design and Implementation*, pages 218–226, ACM Press, Jun. 1989.

[Kastens & Schmidt 86] U. Kastens and M. Schmidt. Lifetime analysis for procedure parameters. In G. Goos and J. Hartmanis, editors, *European Symposium on Programming*, pages 53–69, Springer Verlag, Mar. 1986. Volume 213 of *Lecture Notes on Computer Science*.

[Kennedy 76] K.W. Kennedy. Node listings applied to data flow analysis. In *5th Annual ACM Symposium on Principles of Programming Languages*, pages 10–21, Jan. 1976.

[Landin 64] J. Landin. *The Mechanical Evaluation of Expressions*. Volume 6, Computer Journal, Jan. 1964.

[Larus & Hilfinger 88] J.R. Larus and P.N. Hilfinger. Detecting conflicts between structure accesses. In *SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 21–34, ACM, Jun. 1988.

[Launchbury 87] J. Launchbury. Projections for specialisation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Workshop on Partial Evaluation and Mixed Computation*, pages 299–315, North Holland, Oct. 1987.

[Mogensen 87] T.Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Workshop on Partial Evaluation and Mixed Computation*, pages 325–347, North Holland, Oct. 1987.

[Mogensen 89] T.Æ. Mogensen. Binding time analysis for polymorphically typed higher-order languages. In *Proc. TAPSOFT*, pages 298–312, Springer Verlag, 1989. Volume 352 of *Lecture Notes on Computer Science*.

[Neirynck, Panangaden & Demers 87] A. Neirynck, P. Panangaden, and A.J. Demers. Computation of aliases and support sets. In *14th Annual ACM Symposium on Principles of Programming Languages*, pages 274–283, 1987.

[Nielson & Nielson 86] H.R. Nielson and F. Nielson. Semantics directed compiling for functional languages. In *Annual ACM Conference on Lisp and Functional Programming*, pages 249–257, Aug. 1986.

[Nielson & Nielson 88] F. Nielson and H.R. Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, Jan. 1988.

[Nielson 85] F. Nielson. Expected forms of data flow analyses. In H. Ganziger and N.D. Jones, editors, *Programs as Data Objects*, pages 172–191, Springer Verlag, 1985. Volume 217 of *Lecture Notes on Computer Science*.

[Raoult & Sethi 84] J.C. Raoult and R. Sethi. The global storage needs of a subcomputation. In *11th Annual ACM Symposium on Principles of Programming Languages*, pages 149–157, ACM Press, 1984.

[Raoult & Sethi 85] J.C. Raoult and R. Sethi. *On Finding Stacked Attributes*. Technical Report 206, LRI, Université Paris-Sud, 91405 Orsay, France, Feb. 1985.

[Rees & Clinger 86] J. Rees and W Clinger. Revised^3 report on the algorithmic language scheme. *SIGPLAN Notices*, 21(12):37–79, Dec. 1986.

[Ruggieri & Murtagh 88] C. Ruggieri and T. Murtagh. Lifetime analysis of dynamically allocated objects. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, 1988.

[Schmidt 85] D. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, Apr. 1985.

[Schwartz 78] J. Schwartz. Verifying the safe use of destructive operations in applicative programs. In B. Robinet, editor, *Transformations de programmes : 3^e colloque international sur la programmation*, pages 394–410, Dunod, Paris, mar. 1978.

[Sestoft 89] P. Sestoft. Replacing function parameters by global variables. In *Conference on Functional Programming Languages and Computer Architecture*, pages 39–53, ACM Press, London, Sep. 1989.

[Sharir & Pnueli 81] M. Sharir and A. Pnueli. *Program Flow Analysis: Theory and Applications*, chapter Two approaches to interprocedural data flow analysis, pages 189–234. Prentice-Hall, 1981.

[Shivers 88] O. Shivers. Control flow analysis in scheme. In *Conference on Programming Language Design and Implementation*, pages 164–174, Jun. 1988.

[Stransky 88] I. Stransky. *Analyse sémantique de structures de données dynamiques avec application au cas particulier de langages LISPiens*. PhD thesis, Université de Paris-Sud, Orsay, France, Jun. 1988.

[Wadler 88] P. Wadler. Deforestation: transforming programs to eliminate trees. In *European Symposium On Programming*, Springer Verlag, 1988. Volume 300 of *Lecture Notes on Computer Science*.

[Weihl 80] W.E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *7th Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, 1980.