Computation of Aliases and Support Sets

Anne Neirynck, Prakash Panangaden
Computer Science Department, Cornell University

Alan J. Demers
Xerox Parc, Palo Alto

Received 10/16/86

Abstract

We provide a scheme for determining which global variables are involved when an expression is evaluated in a language with higher order constructs and imperative features. The heart of our scheme is a mechanism for computing the *support* of an expression, i.e. the set of global variables involved in its evaluation. This computation requires knowledge of all the aliases of an expression. The inference schemes are presented as abstract semantic interpretations. We prove the soundness of our estimates by establishing a correspondence between the abstract semantics and the standard semantics of the programming language.

1 Introduction

We present an abstract semantic interpretation for computing the support set and alias set for an expression. The language we use is a typed higher order functional language with a few imperative constructs.

Support sets turn out to be the key ingredient in determining whether an expression is side effect free (pure) or, in general, whether two expressions can be evaluated independently. We shall refer to this as *purity analysis*. We define semantic functions which assign to expressions their support and alias sets. This style of presentation for static analysis was pioneered by the Cousots [9,8]. It allows one to present a static analysis scheme in a fashion which makes clear the relation between the abstract semantics and the standard semantics.

Abstract interpretation provides a general semantic framework for justifying schemes for the static inference of properties of programs. Such inference is of use in program optimization, transformation and partial correctness. The main idea is that static analysis consists of a scheme for estimating the run-time properties of a program. Such estimations can be understood as arising from a semantics, for the underlying programming language, defined on a *non-standard* domain of interpretation. The non-standard domain is chosen in such a way as to reflect the features of interest from the actual (or standard) semantics but with suitable simplifications so that the semantic maps become computable. The inference mechanism is then viewed as a semantic calculation over the non-standard interpretation. The advantage of such a view is that the soundness of the inference scheme can be reduced to the problem of comparing two semantic definitions of the language.

Abstract interpretation was first introduced in the context of imperative programs [7,8,9]. In this setting programs are modelled as flowcharts. The standard semantics is defined in terms of sets of possible states that

a program could be in. This is a pointwise extension of the normal standard semantic schemes. The collection of sets of states forms a complete lattice under the usual inclusion ordering.

In developing the abstract interpretation of applicative programs, Mycroft [14,15] observed that the framework of Cousot and Cousot was useful only for inference schemes which were partially correct; termination cannot be expressed in their framework. In place of the powerset based formulation as used by the Cousots, a powerdomain based formulation is necessary. This view was further developed in by Mishra and Keller[12,13] where the abstract interpretation of non-flat "stream" domains was considered. This approach using powerdomains was put on a firm theoretical foundation by Mycroft and Nielson [17,19,20,21,22]. One of the problems with the traditional approach to abstract interpretation is that the framework is strictly first order.

Several authors have examined the extension to higher order languages. In particular, there has been a great deal of interest in the problem of *strictness inference*. Here the problem is to determine whether functions are strict in their arguments. The first order case was thoroughly studied by Mycroft [15] and by Mycroft and Nielson [17]. Recently Hudak and Young [11], Clack and Peyton-Jones [6] and Burn, Hankin and Abramsky [5], have all studied this question in the context of higher order languages. The idea again has been to view strictness analysis as abstract interpretation.

The theoretical basis of abstract interpretation was discussed by Mycroft and Nielson [17] and by Mycroft and Jones [16]. The extension to the higher order case has been discussed by several authors [1] [5] [11] all of whom examined the particular problem of strictness analysis. The soundness of static analysis schemes presented in this way boils down to showing that the abstract semantics and the standard semantics are appropriately related. We shall state a soundness theorem for our analysis scheme and give a partial presentation of the proof. A complete discussion is available in a technical report [18].

2 Purity of Expressions

An expression is said to be *pure* if its evaluation is independent of the values of any global variables. The notion of global variable is relative to a given expression so a pure expression may contain impure subexpressions. An example of this is **new** x : *int* **in** $x \leftarrow 3$. Unfortunately, the reverse situation can also arise; an impure expression may be built up out of pure subexpressions. A lambda abstraction is always pure because any potential effects occur only during application. Thus an expression like $\lambda x : int.\ y \leftarrow x$ would be pure. The main problem with purity determination in a higher order language is that we need to maintain enough information so that we can tell whether a particular application is pure. This may depend on the arguments, for example in $\lambda f : int \rightarrow int.\ \lambda g : int \rightarrow int.\ \lambda i : int.$ **if** $odd(i)$ **then** f **else** g we have a function taking functional arguments and returning a functional result. The result may or may not yield impure applications depending on the functional arguments f and g. Not all applications of a function with side effects produce impure expressions. As soon as the global variables involved in the side effect are captured by their enclosing declarations, the resulting expression is pure. This is the case with

$$\textbf{new } x : int \textbf{ in let } f = \lambda y : ref\ int.\ y \leftarrow 3 \textbf{ in } f(x)$$

Thus the data-flow analyses for purity which suffice for first order languages would be inadequate to the task at hand [3] [2] [23] [4].

In order for our scheme to deem an expression impure only when global variables are affected we need to compute the set of global variables actually read or written into during evaluation of that expression. Consider the following two expressions:

$$\textbf{new } x : int \textbf{ in } x \leftarrow 1$$

$$\textbf{new } x : int \textbf{ in } y \leftarrow 1$$

The first expression is not impure whereas the second one is. The two expressions are structurally very similar. If we hope to distinguish between these two cases we must actually know which variables are being affected by subexpressions, not just that the subexpressions do affect a variable. We shall call the set of global variables which are affected by or whose values affect the evaluation of an expression the *support* of that expression. An expression is pure when its support set is empty.

3 The Language and its Semantics

The language that we use is an ordinary typed functional language. We add three imperative features: an assignment statement written $e_1 \leftarrow e_2$, a dereferencing construct $e\uparrow$ and static allocation performed with the **new** construct. A variable denotes a storage location; thus to denote the value associated with the variable we need to use the dereferencing construct. An expression can evaluate to a variable rather than to the value associated with the variable as in **if** *true* **then** x **else** y.

This is the abstract syntax for the language:

e = x
  let $x = e_1$ **in** e_2
  letrec $f_1 = \lambda x_1 : T_1.e_1$, ... ,
    $f_n = \lambda x_n : T_n.e_n$ **in** e
  if e_1 **then** e_2 **else** e_3
  $\lambda x : T.\ e$
  $e_1(e_2)$
  new $x : T$ **in** e
  $e_1\ ;\ e_2$
  $e_1 \leftarrow e_2$
  $e!$

There are some constraints on the language: we have only simple types, functions are not storable items, and at most one level of reference or dereference is allowed. These restrictions can enforced by typechecking, and were chosen to simplify the abstract interpretation. Furthermore, we insist that l-values cannot be exported outside their scope. This property can be detected statically by the methods developed here.

The Store

We will use the normal store mechanism and interpret an expression as being impure if its denotational semantics involves a reference to the store. This is a delicate issue. At first it may appear that we are taking the denotational semantics as indicative of the operational semantics. After all purity appears to be a very operational concept. Note, however, that we are claiming only that we can tell whether an expression *is* pure. If our approximate analysis says that an expression is not pure it only means that the expression may or may not be pure. Whether an expression does have a pure version is indeed a property that can be reflected in the denotational semantics. If we were claiming to make a precise analysis of purity and impurity then we would be forced to use an operational formulation.

The way we model the store is particularly important since the crux of our analysis is the determination of which expressions actually affect the store or depend on the store. It is in fact usual to find that the denotational definition of a programming language includes a model of the store and of storage management. In our case the abstract interpretation is insensitive to the storage allocation policy and the relationship between the standard semantics and the abstract interpretation will hold regardless of the storage management policy. Thus the proof of our soundness theorem requires minimal assumptions about the store.

Part of a store can be viewed as a finite function from locations to values. We will need a notation to express the restriction of that function to a subset of the locations.

$$store\,|_S \quad \text{where } S \subseteq Loc$$

We will also allow $S \subseteq P(Id)$, a subset of the variables already allocated when we use the restriction notation for stores.

Given a store, *Allocate* returns a store and the address of the newly allocated space. The new store must coincide with the old store over all addresses except the one just allocated, as stated in Axiom 1a, and the new address was not already allocated (Axiom 1b).

$$Allocate : Store \rightarrow Store \times Loc$$

Axiom 1

if $(newstore,\ address) = Allocate(store)$
then
(a) $\quad newstore\,|_{Loc-\{address\}} = store\,|_{Loc-\{address\}}$
(b) $\quad Eval(store, address) = \perp$
$\quad\quad Eval(newstore, address) = \underline{empty}$

Given a store and an address, *DeAllocate* returns the same store, with the address deallocated. The remainder of the store is unchanged.

$$DeAllocate : Store \times Loc \rightarrow Store$$

Axiom 2

$newstore\,|_{Loc-\{address\}} = store\,|_{Loc-\{address\}}$
  where $newstore = DeAllocate(store, address)$

The following two functions are used to perform reading and updating.

$$Update : Store \times Loc \times Val \rightarrow Store$$

$$Eval : Store \times Loc \rightarrow Val$$

Axiom 3 states that *Update* only affects the location indicated and leaves the rest of the store unaltered.
Axiom 3

$Update(store, address, value)\,|_{Loc-\{address\}}$
  $= store\,|_{Loc-\{address\}}$

The last axiom defines the effect of *Eval* and *Update*.
Axiom 4

$Eval(Update(store, address, value), address) = value$

The Semantics

The semantic domains which we use in our denotational semantics are as follows.

$$Val = Triv + Bool + Nat + FVal + Loc$$
$$Env = Id \rightarrow Val$$
$$FVal = Val \rightarrow Store \rightarrow (Val \times Store)$$

The domains *Triv*, *Bool*, *Nat* and *Loc* contain the atomic values used in the language. We shall assume that the domains are flat and the sum of domains being used is the coalesced sum. The domain *Loc* represents the domain of storage locations. The domain *FVal* is used to represent higher order constructs or "functional" values. An element of *FVal* is viewed as a pair. Evaluating a function will return a value and, in general, modify the store. The pair construction is used to "package" together functions which represent both effects.

In defining the denotational semantics we shall use a pair of semantic functions called $\mathbf{M_e}$ and $\mathbf{M_s}$. The first one defines the value of an expression while the second one describes how the store is modified. This is just a notational variation of the ordinary semantic definitions one sees in textbooks. This notation is particularly convenient for our discussion since our principal concern is how the store is affected by constructs in the language. The arities of these functions are:

$$\mathbf{M_e} : Exp \rightarrow Env \rightarrow Store \rightarrow Val$$

and

$$\mathbf{M_s} : Exp \rightarrow Env \rightarrow Store \rightarrow Store.$$

A full definition of the semantics is given in a technical report [18]; the part included here is intended to illustrate the notation and style of semantic definition that we shall use.

$$\mathbf{M_e}[\![x]\!]env\ store = env(x)$$
$$\mathbf{M_s}[\![x]\!]env\ store = store$$

This illustrates the simplest construct namely an identifier. The value of an identifier is the location to which it is bound and there is no effect on the store.

$$\mathbf{M_e}[\![\textbf{let } x = e_1 \textbf{ in } e_2\ ]\!]env\ store =$$

$$\begin{cases} \bot & \text{if } \mathbf{M_e}[\![e_1]\!]env\ store = \bot \\ \mathbf{M_e}[\![e_2]\!]\ env'\ store' & \text{otherwise} \end{cases}$$

$$\mathbf{M_s}[\![\textbf{let } x = e_1 \textbf{ in } e_2\ ]\!]env\ store$$
$$= \mathbf{M_s}[\![e_2]\!]\ env'\ store'$$

where

$$env' = env[x \leftarrow \mathbf{M_e}[\![e_1]\!]env\ store]$$
$$store' = \mathbf{M_s}[\![e_1]\!]env\ store$$

These illustrate how the scope rules for **let** blocks operate. Note that the above semantic clauses are standard and are meant only to illustrate our notation. If an identifier is already bound before entry into a **let** block it will get rebound by the declaration but the old environment is passed to expressions outside the scope of a let. Thus the usual scoping rules are enforced and we may reuse identifiers as we please when entering local scopes.

$$\mathbf{M_e}[\![\textbf{new } x : T \textbf{ in } e]\!]env\ store$$
$$= \mathbf{M_e}[\![e]\!]\ env'\ store'$$
$$\mathbf{M_s}[\![\textbf{new } x : T \textbf{ in } e]\!]env\ store = store'''$$

where

$$(address, store') = Allocate(store)$$
$$env' = env[x \leftarrow address]$$
$$store'' = \mathbf{M_s}[\![e]\!]\ env'\ store'$$
$$store''' = DeAllocate(store'', address)$$

This semantic clause illustrates how the store model is used to define the meaning of the **new** construct. Deallocation occurs when the scope of the new declaration is exited. Note that declaring a variable changes the store and not just the environment.

The following clauses illustrate the explicit imperative features namely assignment and dereferencing.

$$\mathbf{M_e}[\![e_1 \leftarrow e_2]\!]env\ store =$$

$$\begin{cases} \bot & \text{if } address = \bot \\ value & \text{otherwise} \end{cases}$$

$$\mathbf{M_s}[\![e_1 \leftarrow e_2]\!]env\ store =$$

$$\begin{cases} \bot & \text{if } address = \bot \\ Update(store'', address, value) & \text{otherwise} \end{cases}$$

where

$$address = \mathbf{M_e}[\![e_1]\!]env\ store$$
$$store' = \mathbf{M_s}[\![e_1]\!]env\ store$$
$$value = \mathbf{M_e}[\![e_2]\!]\ env\ store'$$
$$store'' = \mathbf{M_s}[\![e_2]\!]\ env\ store'$$

$$\mathbf{M_e}[\![e\uparrow]\!]env\ store =$$

$$\begin{cases} \bot & \text{if } address = \bot \\ Eval(store, address) & \text{otherwise} \end{cases}$$

where $address = \mathbf{M_e}[\![e]\!]env\ store$

$$\mathbf{M_s}[\![e\uparrow]\!]env\ store = \mathbf{M_s}[\![e]\!]env\ store$$

4 An Abstract Interpretation for Aliases

Before we can calculate the support of an expression, we may need another semantics function to determine the aliasing behavior of some of its subexpressions. Consider for instance a dereferencing expresssion $e\uparrow$. The support of this expression is the support of e together with all the variables possibly aliased by e. A similar phenomenon occurs in assignment expressions. This section describes the semantic function **A**, which gives the set of possible aliases of an expression.

The recursive structure of the semantic domains is similar to the ones used by Hudak and Young [11] and reflects the fact that the simplified domain being used for the abstract interpretation must describe the aliasing behavior of expressions containing arbitrarily higher order functions.

$$\begin{aligned} Val_A &= \{\underline{atom}\} + D_A \to D_A \\ Env_A &= Id \to D_A \\ D_A &= \mathcal{P}(Id) \times Val_A \\ \mathbf{A} &: Exp \to Env_A \to D_A \end{aligned}$$

The lattice structure over domain D_A is defined by :

$$\begin{aligned} &\perp_{D_A} = \langle \emptyset, \perp_{Val_A} \rangle \\ &\lambda u.\langle b_1, v_1 \rangle \sqcup \lambda u.\langle b_2, v_2 \rangle = \lambda u.\langle b_1 \cup b_2, v_1 \sqcup v_2 \rangle \end{aligned}$$

where $\sqcup$ is the least upper bound operator. The element $\underline{atom}$ is related only to $\perp$. The intuition behind this choice of domains is as follows. A particular expression is going to denote an element of D_A. The first component of D_A is just a set of identifiers, just what one would expect, while the second contains the information needed to determine the aliasing behavior of other expressions which contain applications of the first expression. Note how the operation of least upper bound is obtained by taking unions of possible alias sets. This reflects the conservative nature of the estimates we are making.

As in the standard semantics, we need an environment which we shall call *aenv*; this is a map from identifiers to D_A. The following are the semantic clauses. We shall use subscripts if we wish to use only the first or second components of a pair in D_A.

$\mathbf{A}[\![x]\!]aenv = aenv(x)$

$\mathbf{A}[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]aenv$
$= \mathbf{A}[\![e_2]\!]aenv[x \leftarrow \mathbf{A}[\![e_1]\!]aenv]$

$\mathbf{A}[\![\textbf{letrec } f_1 = e_1 \ldots f_n = e_n \textbf{ in } e]\!]aenv$
$= \mathbf{A}[\![e]\!]aenv'$
where $aenv' = lfp(\lambda env.\, env[\ldots, f_i \leftarrow \mathbf{A}[\![e_i]\!]env, \ldots])$

$\mathbf{A}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]aenv$
$= \mathbf{A}[\![e_2]\!]aenv \sqcup \mathbf{A}[\![e_3]\!]aenv$
For a conditional, the least upper bound operation will compute the possible aliases as the union of the sets of possible aliases for each arm of the conditional.

$\mathbf{A}[\![\textbf{new } x : T \textbf{ in } e]\!]aenv$
$= \mathbf{A}[\![e]\!]aenv[x \leftarrow \langle \{"x"\}, \underline{atom} \rangle]$
This clause tells us that a new variable starts out being aliased only to itself.

$\mathbf{A}[\![e_1 \,;\, e_2]\!]aenv = \mathbf{A}[\![e_2]\!]aenv$

$\mathbf{A}[\![e_1 \leftarrow e_2]\!]aenv = \langle \emptyset, \underline{atom} \rangle$
The result of an assignment is the (r-value) e_2. Because we are only allowing one level of indirection this cannot be an l-value thus we are assured that the set of possible aliases for this expression is the empty set. It is possible to extend this clause to handle arbitrary levels of indirection. Similar remarks apply to the semantic clause below for the dereferencing operator.

$\mathbf{A}[\![e\uparrow]\!]aenv = \langle \emptyset, \underline{atom} \rangle$

$\mathbf{A}[\![\lambda x : T.e]\!]aenv = \langle \emptyset, \lambda u.\, \mathbf{A}[\![e]\!]aenv[x \leftarrow u] \rangle$
A lambda abstraction cannot possibly be an identifier so we compute its alias set as the empty set, but we need the second component to compute the alias set of expressions involving applications of this function.

$\mathbf{A}[\![e_1(e_2)]\!]aenv = (\mathbf{A}[\![e_1]\!]aenv)_2\, \mathbf{A}[\![e_2]\!]aenv$

The intuitive justification behind our scheme for estimating alias sets is clear and the resemblance to the standard semantics is manifest.

5 Support Sets as an Abstract Interpretation

The support set of an expression is estimated by the semantic function **S**. The domains are very similar to those introduced earlier for aliases.

$$\begin{aligned} Val_S &= \{\underline{atom}\} + Val_S \to D_S \\ Env_S &= Id \to Val_S \\ D_S &= \mathcal{P}(Id) \times Val_S \\ \mathbf{S} &: Exp \to Env_S \to D_S \end{aligned}$$

When we use the semantic function **A** in the definition of **S** we assume that the environment *aenv* results from a computation of **A** for the same expression as the one for which the action of **S** is being defined.

$\mathbf{S}[\![x]\!]senv = \langle \emptyset, senv(x) \rangle$

$$\mathbf{S}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]senv =$$
$$\langle\, \mathbf{S}_1[\![e_1]\!]senv \cup \mathbf{S}_1[\![e_2]\!]senv \cup \mathbf{S}_1[\![e_3]\!]senv\,,$$
$$\mathbf{S}_2[\![e_2]\!]senv \sqcap \mathbf{S}_2[\![e_3]\!]senv\,\rangle$$

$$\mathbf{S}[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]senv =$$
$$(\mathbf{S}_1[\![e_1]\!]senv \cup \mathbf{S}_1[\![e_2]\!]senv[x \leftarrow \mathbf{S}_2[\![e_1]\!]senv]\,,$$
$$\mathbf{S}_2[\![e_2]\!]senv[x \leftarrow \mathbf{S}_2[\![e_1]\!]senv]\,\rangle$$

$$\mathbf{S}[\![\textbf{letrec } f_1 = e_1 \ldots f_n = e_n \textbf{ in } e_2]\!]senv = \mathbf{S}[\![e]\!]senv'$$
where $senv' = lfp(\lambda env.env[\ldots, f_i \leftarrow \mathbf{S}_2[\![e_i]\!]env, \ldots])$

$$\mathbf{S}[\![\textbf{new } x : T \textbf{ in } e]\!]senv =$$
$$(\mathbf{S}_1[\![e]\!]senv[x \leftarrow \underline{atom}] - \{"x"\}\,,$$
$$\mathbf{S}_2[\![e]\!]senv[x \leftarrow \underline{atom}]\,\rangle$$

In the above clause we need to enforce the scoping rules. When a new variable is declared we note that inside its scope its support is just itself. On the other hand the support of the entire block must not include the new variable since the scope of the new variable ends when the block is exited; so we explicitly remove the new variable from the support which we compute for the body of the block.

$$\mathbf{S}[\![e_1\ ;\ e_2\ ]\!]senv =$$
$$\langle\, \mathbf{S}_1[\![e_1]\!]senv \cup \mathbf{S}_1[\![e_2]\!]senv\,,\ \mathbf{S}_2[\![e_2]\!]senv\,\rangle$$

$$\mathbf{S}[\![e_1 \leftarrow e_2\ ]\!]senv =$$
$$\langle\, \mathbf{S}_1[\![e_1]\!]senv \cup \mathbf{S}_1[\![e_2]\!]senv \cup \mathbf{A}_1[\![e_1]\!]aenv\,,$$
$$\underline{atom}\,\rangle$$

$$\mathbf{S}[\![e\uparrow]\!]senv = \langle\, \mathbf{S}_1[\![e]\!]senv \cup \mathbf{A}_1[\![e]\!]aenv\,,\ \underline{atom}\,\rangle$$

In determining the support of the explicitly imperative constructs we need to know the aliases of some subexpressions. Thus in the assignment statement above we union together the supports of the two sides of the assignment and also all possible aliases of the left hand side .

$$\mathbf{S}[\![\lambda x : T.e]\!]senv = \langle\, \emptyset\,,\ \lambda u.\mathbf{S}[\![e]\!]senv[x \leftarrow u]\,\rangle$$

$$\mathbf{S}[\![e_1(e_2)]\!]senv =$$
$$\langle\, \mathbf{S}_1[\![e_1]\!]senv \cup \mathbf{S}_1[\![e_2]\!]senv$$
$$\cup(\mathbf{S}_2[\![e_1]\!]senv\ \mathbf{S}_2[\![e_2]\!]senv)_1\,,$$
$$(\mathbf{S}_2[\![e_1]\!]senv\ \mathbf{S}_2[\![e_2]\!]senv)_2\,\rangle$$

The correctness is already intuitively plausible because of the correspondence between the standard semantics and the abstract interpretations used to define support and alias.

6 Soundness Theorem for Support Sets

In this section we shall state a soundness theorem for our abstract interpretations. The heart of the proof of the soundness theorem is a joint induction on the structure of terms in the programming language as well as on their types. Hudak and Young prove a soundness theorem for strictness analysis which they claim holds for the untyped lambda calculus. In fact, their proof crucially uses induction on the type of lambda terms and, as they observe, they need to enforce a "weak type discipline" to guarantee termination.

The soundness theorem for the function S says that the evaluation of an expression depends solely on the values of variables in its support set and does not affect the store outside that set.

Theorem(support sets)

If the evaluation of e terminates,
$\forall\, env, senv$ corresponding environments,
$\mathbf{S}_1[\![e]\!]senv \subseteq S \Rightarrow$
$\forall\, store\ (\mathbf{M_s}[\![e]\!]env\ store)\,|_{\overline{S}} = store\,|_{\overline{S}}$
$\mathbf{S}_1[\![e]\!]senv \subseteq S \Rightarrow$
$\forall\, store, store'$ such that $store\,|_S = store'\,|_S$
$\mathbf{M_e}[\![e]\!]env\ store = \mathbf{M_e}[\![e]\!]\ env\ store'$
$\mathbf{M_s}[\![e]\!]env\ store\,|_S = \mathbf{M_s}[\![e]\!]\ env\ store'\,|_S$

The requirement that a pure expression always evaluates to the same value may be too stringent for l-values and could be replaced by a weaker one. Note also that the S semantics cannot distinguish non-terminating computations.

We need a precise definition of corresponding environments *senv* and *env*. As in Hudak and Young [11], we define partial application operators in both semantic domains, which will facilitate the proof by providing a mechanism for carrying out induction on the type structure. The partial applicators define the reduction of a sequence of nested applications. The notation PAP_n is used for the abstract domain while AP_n is used in the standard domain.
$PAP_n : (D_S)^{n+1} \rightarrow \mathcal{P}(Id)$
$PAP_n(s, s_1, \ldots, s_n) =$

$$\begin{cases} (s)_1 & n = 0 \\ (s)_1 \cup (s_1)_1 \cup PAP_{n-1}((s)_2 s_1, \ldots, s_n) & n > 0 \end{cases}$$

The partial applicator AP_n is defined in such a way as to incorporate the effect on the store when a cascaded sequence of applications is partially reduced. The definition is chosen so that as each application is performed a value, store pair is produced and the new store is used in the next application.

$AP_n : (Store \to Val \times Store)^{n+1}$
$\to (Store \to Val \times Store)$
$AP_n(e, e_1, \ldots, e_n) =$

$$\begin{cases} e & n = 0 \\ AP_{n-1}(\lambda s.(e\ s)_1\ (e_1(e\ s)_2)_1\ (e_1(e\ s)_2)_2, & \\ e_2, \ldots, e_n) & n > 0 \end{cases}$$

These operators allow one to move down the type structure by applying the function component of a member of Val_S or Val. The fact that we have finite types only means that one can "reach" all types of interest by an inductive argument. Dually, it also means that computations in the approximating semantic domain must terminate. It is straightforward to check that the following identity holds:
$AP_{m+1}(\mathbf{M}[\![e_1]\!]env, \mathbf{M}[\![e_2]\!]env, v_1, \ldots, v_m)$
$= AP_m(\mathbf{M}[\![e_1(e_2)]\!]env, v_1, \ldots, v_m)$

Following Hudak and Young we shall use the term *safe* to characterize the correctness property that we need to prove.

$s \in D_S$ is safe at level n for value $e \in D$ if:
$\forall\, m \le n$, $s_i \in D_S$, $e_i \in D$, s_i safe at level $n-1$ for e_i
$PAP_m(s, s_1, \ldots, s_m) \subseteq S$
$\Rightarrow$
$\forall$ *store*, *store'* such that $store|_S = store'|_S$
$[AP_m(e, e_1, \ldots, e_m)store]_1 =$
$[AP_m(e, e_1, \ldots, e_m)store']_1$ (1a)
$[AP_m(e, e_1, \ldots, e_m)store]_2 |_S =$
$[AP_m(e, e_1, \ldots, e_m)store']_2 |_S$ (1b)

and
$\forall$ *store*
$[AP_m(e, e_1, \ldots, e_m, store)]_2 |_{\overline{S}} = store|_{\overline{S}}$ (2)

$s \in D_S$ is safe for value $e \in D$ if
it is safe at all levels

senv and *env* are corresponding environments if
$senv(x)$ is safe for $\lambda s.(env(x), s)$ $\forall\, x \in dom(senv)$

Proof :

We will show that $\mathbf{S}[\![e]\!]\rho$, where ρ is the support environment, is safe for $\mathbf{M}[\![e]\!]env$, where $\mathbf{M}$ is defined by

$$\mathbf{M}[\![e]\!]env = \lambda s.(\mathbf{M_e}[\![e]\!]\ env\ s, \mathbf{M_s}[\![e]\!]\ env\ s)$$

The proof proceeds by structural induction (SI) on e and by induction on the type of e. For the **letrec** construct we will need fixpoint induction as well. Assume, for simplicity, that all variable names are distinct. The proof of (1*a*) and (1*b*) are virtually identical, therefore we shall only describe the proof of (1*a*).

1. $e \equiv$ **new** $x : T$ **in** e_1

To show (2) for $n > 0$ and (1), it is sufficient to show that *env'* and *senv'* are corresponding environments, and then use SI on e_1.
By definition,
$env' = env[x \leftarrow address]$
$senv' = senv[x \leftarrow \underline{atom}]$
Since *env* and *senv* are corresponding environments, and x is not of functional type, it is sufficient to show that $\langle \emptyset, \underline{atom} \rangle$ is safe at level 0 for $\lambda s.\langle address, s \rangle$, which is true.

There is one equality left to show independently, that is (2) for $n = 0$, or $\mathbf{M_s}[\![e]\!]env\ store\ |_S = store\ |_S$.
Assume $\mathbf{S}_1[\![\textbf{new}\ x : T\ \textbf{in}\ e_1]\!]\rho \subseteq S$
(a) $\mathbf{S}_1[\![e_1]\!]\rho \subseteq S \cup \{"x"\}$ by definition of $\mathbf{S}$
(b) $store'|_S = store|_S$ by definition of *Allocate*
(c) $store''|_{S \cup \{"x"\}} = store'|_{S \cup \{"x"\}}$ by SI on e_1 and by using (a)
(d) $store''|_S = store'|_S$ by restriction on (c)
(e) Since S is disjoint from $\{x\}$, when deallocating the space for x,
the S-restriction of the store is not affected:
$store'''|_S = DeAllocate(store'', address)|_S$
$= store''|_S$ by axiom I
and we have our equality by transitivity of (b), (d), and (e)

2. $e \equiv e_1(e_2)$

Fix $n \ge 0$
We must show $\mathbf{S}[\![e_1(e_2)]\!]\rho$ is safe at level n for $\mathbf{M}[\![e_1(e_2)]\!]env$
Choose $m \le n$,
and s_i safe at level $n-1$ for v_i, $i = 1, \ldots, m$

Here we need induction on the term structure as well as on the type structure. The latter is represented by the level n. We need to prove the following implications.
(1) $PAP_m(\mathbf{S}[\![e]\!]\rho, s_1, \ldots, s_m) \subseteq S \Rightarrow$
$[AP_m(\mathbf{M}[\![e_1(e_2)]\!]env, v_1, \ldots, v_m)\ store]_1$
$= [AP_m(\mathbf{M}[\![e_1(e_2)]\!]env, v_1, \ldots, v_m,)\ store']_1$
if $store|_S = store'|_S$
(2) $PAP_m(\mathbf{S}[\![e]\!]\rho, s_1, \ldots, s_m) \subseteq S$
$\Rightarrow$
$[AP_m(\mathbf{M}[\![e_1(e_2)]\!]env, v_1, \ldots, v_m)\ store]_2 |_{\overline{S}} = store|_{\overline{S}}$

To prove the first we proceed as follows:
By definition of *PAP*
$PAP_m(\mathbf{S}[\![e_1(e_2)]\!]\rho, s_1, \ldots, s_m)$

$= (\mathbf{S}[\![e_1(e_2)]\!]\rho)_1 \cup (s_1)_1 \cup$
$PAP_{m-1}((\mathbf{S}[\![e_1(e_2)]\!]\rho)_2\ s_1, s_2, \ldots, s_m)$

Note how the type of the term has been decreased by using the definition of PAP. The next equality follows by using the definition of S twice.
$= (\mathbf{S}[\![e_1]\!]\rho)_1 \cup (\mathbf{S}[\![e_2]\!]\rho)_1 \cup ((\mathbf{S}[\![e_1]\!]\rho)_2 \mathbf{S}[\![e_2]\!]\rho)_1$
$\cup (s_1)_1 \cup PAP_{m-1}(((\mathbf{S}[\![e_1]\!]\rho)_2 \mathbf{S}[\![e_2]\!]\rho)_2 s_1, \ldots, s_m)$

The next equality follows from the definition of PAP used in the reverse direction to go to a higher safety level but with the original application term broken down.
$= (\mathbf{S}[\![e_1]\!]\rho)_1 \cup (\mathbf{S}[\![e_2]\!]\rho)_1$
$\cup PAP_m((\mathbf{S}[\![e_1]\!]\rho)_2 \mathbf{S}[\![e_2]\!]\rho, s_1, \ldots, s_m)$
$= PAP_{m+1}(\mathbf{S}[\![e_1]\!]\rho, \mathbf{S}[\![e_2]\!]\rho, s_1, \ldots, s_m)$

The last step takes us to a higher safety level but with simpler terms. We now perform the analogous calculation in the standard semantics and establish the inductive step for this case of the proof.
Since e_1 and e_2 are structurally simpler than e, we get from the inductive hypothesis:
$[AP_{m+1}(\mathbf{M}[\![e_1]\!]env, \mathbf{M}[\![e_2]\!]env, v_1, \ldots, v_m)\ store]_2) \mid_{\overline{S}}$
$= store \mid_{\overline{S}}$
But :
$AP_{m+1}(\mathbf{M}[\![e_1]\!]env, \mathbf{M}[\![e_2]\!]env, v_1, \ldots, v_m)$
$= AP_m(\mathbf{M}[\![e_1(e_2)]\!]env, v_1, \ldots, v_m)$
and therefore (2) holds.
For (1) we proceed as follows:
$PAP_{m+1}(\mathbf{S}[\![e_1]\!], \mathbf{S}[\![e_2]\!], s_1, \ldots, s_m)$
$\Rightarrow$
$[AP_{m+1}(\mathbf{M}[\![e_1]\!]env, \mathbf{M}[\![e_2]\!]env, v_1, \ldots, v_m)\ store]_2$
$= [AP_{m+1}(\mathbf{M}[\![e_1]\!]env, \mathbf{M}[\![e_2]\!]env, v_1, \ldots, v_m)\ store']_2$
$\Rightarrow$
$[AP_m(\mathbf{M}[\![e_1(e_2)]\!]env, v_1, \ldots, v_m)\ store]_2$
$= [AP_m(\mathbf{M}[\![e_1(e_2)]\!]env, v_1, \ldots, v_m)\ store']_2$

The two cases we have shown illustrate the higher order situations and an imperative construct. The remaining cases are much simpler, the details are available in our technical report [18]. It is interesting to observe that this proof has the same sort of reasoning that one sees in proofs of strong normalization of the typed lambda-calculus. There again one needs to go down in the type structure which is done by performing an application and producing in the process a structurally more complicated term.

The corresponding soundness theorem for the abstract interpretation A for computing aliases is:
Theorem(alias sets)

If S is the set of variables possibly aliased by an expression e, then e can never evaluate to a variable not in S. The soundness of our abstract interpretation for aliases is defined formally in the following theorem:

If the evaluation of e terminates,
$\forall\ env$, $aenv$ corresponding environments,
$\forall\ S \subseteq P(Id)$,
$\mathbf{A}_1[\![e]\!]aenv \subseteq S \Rightarrow$
$\forall\ y \in \overline{S}$, $\forall\ store$
$\mathbf{M_e}[\![e]\!]env\ store \neq \mathbf{M_e}[\![y]\!]env\ store$

The proof is similar to the one for support sets, full details are provided in our technical report [18].

7 Conclusion

In this paper we have produced a method for determining (approximately) whether the evaluations of two expressions are independent of each other. The presentation of our method uses the technique of abstract interpretation. We feel that using abstract interpretation allows one to present static analysis schemes in a semantically appealing fashion. The resulting algorithm is easy to code and has been implemented in ML. Furthermore, the static analysis scheme which we describe brings an interesting and pragmatically relevant new problem within the scope of abstract interpretation techniques.

The problem of determining side effects has been studied for a fairly long time by workers developing flow analysis techniques [3] [2] [23]. The work of Barth addressed the problem of purity analysis in the presence of recursive procedures and with aliasing but he did not consider higher order functions or the presence of pointer variables. Banning also considers first order block structured languages and also does not allow l-valued expressions. Weihl actually considers variables of function type but the language he uses is otherwise first order. He does allow l-valued expressions. All three papers just cited use various ad hoc flow analysis algorithms rather than a semantic scheme. Accordingly the correctness of their schemes is hidden in algorithmic details. Recently, Gifford and Lucassen [10] have written a very interesting paper in which they relate effect checking (as they call it) to polymorphic type checking. Their scheme incorporates the relevant information into the type system and their inference mechanism becomes part of the of the type inference mechanism. Though their scheme is semantically motivated we feel their analysis lacks the precision of our scheme. Their language is indeed higher order but their

abstract domain for discussing side effects has only four elements. Their scheme needs programmer supplied annotations in order to obtain useful purity analysis. Furthermore they impose constraints on their language which prohibit, for example, a pure function from performing assignments to local variables. Thus they treat the store as a single unit.

We are considering two directions for future work in this area. First, we are going to remove the restrictions on pointers and on variables of functional type. We anticipate no new problems in carrying out this extension. Second, we intend to introduce data structures into the language and study the viability of purity analysis in the presence of this extension. A recent paper by Burke and Cytron [4] has given a thorough discussion of dependency analysis for a first order language. We are working on expressing their data flow analysis algorithms as part of our abstract interpretation thereby extending their work to the higher order case.

Acknowledgments

We thank Jim Hook, Dieter Maurer, and Keshav Pingali for useful comments. The first author thanks John Lucassen for helpful discussions and for pointers to the literature. The second author has benefited from discussions with Prateek Mishra and Uday Reddy. This research was supported by the NSF under grant DCR-860272 to Cornell University and also by the Cornell Center for Theory and Simulation in Science and Engineering, which is funded in part by the National Science Foundation, New York State and IBM Corporation.

References

[1] S. Abramsky. Strictness analysis and polymorphic invariance. In *Proceedings of Programs as Data Objects, Springer Lecture Notes in Computer Science 217*, 1986.

[2] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *6th Symposium of Principles of Programming Languages*, 1979.

[3] J. Barth. A practical interprocedural data flow analysis algorithm. *CACM*, 21:724–736, 1978.

[4] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *ACM Sigplan Notices, Vol 21,7*, 1986.

[5] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory and practice of strictness analysis for higher order functions. In *Proceedings of Programs as Data Objects, Springer Lecture Notes in Computer Science 217*, 1986.

[6] C.D. Clack and S.L. Peyton-Jones. Strictness analysis - a practical approach. In *Proceedings of IFIP Conference on Functional Programming and Computer Architecture, Springer Lecture Notes in Computer Science 201*, 1985.

[7] P. Cousot. Semantic fondations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, Prentice-Hall, 1981.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Record of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[9] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*, 1976.

[10] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *Proceedings of 1986 ACM Conference on Lisp and Functional Programming*, 1986.

[11] P. Hudak and J. Young. Higher–order strictness analysis in untyped lambda calculus. In *Proceedings 13th POPL*, 1986.

[12] P. Mishra. *Static Inference in Applicative Languages.* PhD thesis, University of Utah, 1985.

[13] P. Mishra and R. M. Keller. Static inference of properties of applicative programs. In *Proceedings of 11th POPL*, 1984.

[14] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, Scotland, 1981.

[15] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of 4th Int. Symposium on Programming, Lecture Notes in Computer Science 83*, pages 269–281, Springer–Verlag, 1980.

[16] A. Mycroft and N. D. Jones. A new framework for abstract interpretation. In *Proceedings of Programs as Data Objects, Springer Lecture Notes in Computer Science 217*, 1986.

[17] A. Mycroft and F. Nielson. Strong abstract interpretation using power domains. In *Proceedings ICALP 1983, Lecture Notes in Computer Science 154*, pages 536–547, Springer–Verlag, 1983.

[18] A. Neirynck, P. Panangaden, and A.J. Demers. *Computation of Aliases and Support Sets.* Technical Report TR86-763, Cornell University, 1986.

[19] F. Nielson. *Abstract Interpretation Using Domain Theory.* PhD thesis, University of Edinburgh, Scotland, 1984.

[20] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18, 1982.

[21] F. Nielson. Program transformations in a denotational setting. In *ACM Transactions on Programming Languages and Systems*, 1985.

[22] F. Nielson. Towards viewing nondeterminism as abstract interpretation. In *FST & TCS3*, 1983.

[23] W. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *7th ACM Symposium on Principles of Programming Languages*, 1980.