Computation of Aliases and Support Sets

Anne Neirynck*
Prakash Panagaden
Alan J. Demers†

TR 86-763
July 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This research was supported 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.

† Xerox PARC, Palo Alto, CA.

Computation of Aliases and Support Sets

Anne Neirynck *
Prakash Panangaden
Computer Science Department, Cornell University

Alan J. Demers
Xerox Parc, Palo Alto

July 3, 1986

Abstract

When programs are intended for parallel execution it becomes critical to determine whether the evaluations of two expressions can be carried out independently. We provide a scheme for making such determinations 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

In recent years there has been much interest in the development of languages for the expression of parallel algorithms. It is fair to say that most of the proposals are still being debated and there is no clear consensus on the ideal programming constructs that one should use. The parallel execution of purely functional

*This research was supported 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.

languages has received much serious attention in the literature; see for example [2,8,9,11,12,13]. It is well known that the use of purely applicative programs has some advantages and some disadvantages from the viewpoint of parallel execution. Briefly these are that the absence of side effects allows a compiler to very easily detect potential parallelism but the inability to update data structures in place leads to excessive copying. Various solutions have been proposed [2,9] but the efficacy and generality of such methods is still under study. We propose an alternative compromise, which is to start with an "almost" functional language with higher–order functions and to add imperative features to it. This language retains the semantic elegance of a purely applicative language while allowing in–place updating of data structures. Of course we lose the side effect freeness of a purely functional language in the process. What we provide instead is a static analysis scheme which allows one to make conservative estimates about whether a program fragment affects the global store. We shall refer to this as *purity analysis*. The analysis is applicable to a language which permits lambda abstraction and thus has functions of arbitrary higher–order type.

In the present work we develop an estimation scheme for determining whether an expression in an imperative language either uses or updates the store. Our scheme also determines the aliasing relations that may hold between program variables. In general, we can also tell whether, given two expressions, the evaluation of one of them affects the evaluation of the other. This generalizes the notion of purity mentioned in the last paragraph and is clearly useful if one is interested in parallel execution of programs. These problems arose in the context of designing a suitable programming environment for parallel execution of numerical algorithms.

An expression is said to be ***pure*** if its evaluation is independent of the values of any variables – it neither has an observable effect nor produces a value that depends on the current state. We take the view that a pure expression can involve updating or referencing of local variables but not of global variables. Thus a pure expression may contain impure subexpressions. By this criteria, all examples of Figure 1 are pure expressions whereas Figure 2 contains only impure expressions. The first example is obviously a pure expression, as well as being an example of a function without side effects and would be recognized as being pure by a naive algorithm. The task becomes more complex for the other examples. The syntactical occurrences of assignments or dereferencing in an expression do not imply that the expression is not pure; see for instance example

1. $\lambda x : \boldsymbol{int}.\ \lambda y : \boldsymbol{int}.\ x + y$

2. $\mathbf{new}\ x : \boldsymbol{int}\ \mathbf{in}\ x \leftarrow 3$

3. $\lambda x : \boldsymbol{int}.\ y \ \leftarrow\ x$

4. $\lambda f : \boldsymbol{int} \rightarrow \boldsymbol{int}.\ \lambda g : \boldsymbol{int} \rightarrow \boldsymbol{int}.\ \lambda i : \boldsymbol{int}.\ \mathbf{if}\ odd(i)\ \mathbf{then}\ f\ \mathbf{else}\ g$

5. $\lambda x : \boldsymbol{ref\ int}.\ \lambda y : \boldsymbol{ref\ int}.\ \lambda i : \boldsymbol{int}.\ \mathbf{if}\ odd(i)\ \mathbf{then}\ x\ \mathbf{else}\ y$

6. $\lambda i : \boldsymbol{int}.\mathbf{if}\ odd(i)\ \mathbf{then}\ x\ \mathbf{else}\ y$

Figure 1: Pure Expressions

2. When designing a decision procedure for purity testing, the main difficulty is to determine if a function application produces side effects. Example 3 is a pure expression but denotes a function with side effects. This information may depend on the arguments: in example 4, the expression is side effect free if its two functional arguments are side effect free. Example 5 is a case of a side effect free function even though its arguments are variables. So is the function of example 6, despite global identifier references.

1. $x\uparrow$

2. $\mathbf{new}\ y : \boldsymbol{int}\ \mathbf{in}\ y \leftarrow z\uparrow$

3. $\mathbf{let}\ f = \lambda x : \boldsymbol{int}.\ k \leftarrow x\ \mathbf{in}\ f(5)$

Figure 2: Impure Expressions

All expressions in Figure 2 are impure. Examples 1 and 2 contain dereferencing of global variables. The last example contains an application of a function with side effects. Of course, 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

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

In the subsequent sections of this paper we introduce the language and its formal semantics, discuss the relevant background from abstract interpretation, discuss a first attempt at the solution, discuss the actual solution and finally prove the correctness of our analysis technique.

2 Abstract interpretation

In this section we briefly survey recent work on abstract interpretation and mention the relation of abstract interpretation to our static analysis framework. 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 developed by Cousot and Cousot [5,6,7] to aid in the analysis of imperative programs. 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 [16,17] 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 [14,15] 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 [18,21,20,19,22]. 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 constructs. 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 [17] and by Mycroft and Nielson [18]. Recently Hudak and Young [10], Clack and Peyton-Jones [4] and Burn, Hankin and Abramsky [3], have all studied this question in the context of higher-order languages. The idea again has been to view strictness analysis as abstract interpretation. Strictness analysis is of importance if one is interested in parallel execution of programs written in an applicative language with lazy evaluation.

The essential idea of abstract interpretation can be described as follows. Suppose that programs in some programming language $\boldsymbol{L}$ take input from some domain $\boldsymbol{C}$ and produce output from domain $\boldsymbol{D}$. Then a suitable domain for defining the semantics of $\boldsymbol{L}$ is the function space $[\boldsymbol{C} \rightarrow \boldsymbol{D}]$. Now in abstract interpretation one redefines the semantics of $\boldsymbol{L}$ so that the input domain is $\boldsymbol{A}$, the output domain is $\boldsymbol{B}$ and the programs of $\boldsymbol{L}$ take their meaning in $[\boldsymbol{A} \rightarrow \boldsymbol{B}]$. The domains $\boldsymbol{A}$ and $\boldsymbol{B}$ are usually taken to be "simplifications" of the original domains $\boldsymbol{C}$ and $\boldsymbol{D}$. This means that each element of $\boldsymbol{A}$ or $\boldsymbol{B}$ abstracts away some properties of elements of $\boldsymbol{C}$ and $\boldsymbol{D}$ respectively. The set of elements of $\boldsymbol{C}$ that is represented by a particular element of $\boldsymbol{A}$ is given by a function γ called a concretization function. These functions are used to express the correctness of the abstract interpretation with respect to the standard semantics.

The commuting diagram is shown in Figure 3. Let $\boldsymbol{P}$ be a program in $\boldsymbol{L}$. Suppose that the semantic function describing the standard semantics is $\boldsymbol{\mu}$ and the semantic function describing the abstracted semantics is $\boldsymbol{\alpha}$. To relate the standard semantics to the abstracted semantics we first need to lift the standard semantics to the collection of sets since this is what the concretization maps γ map elements of the abstract domains to. The semantic function for $\boldsymbol{L}$ extended to sets is $\boldsymbol{\pi}$ and it maps elements from the powerset of $\boldsymbol{C}$ (written $\mathcal{P}(\boldsymbol{C})$) to the powerset of $\boldsymbol{D}$. In general one needs an appropriate powerdomain construction but we shall avoid discussing these issues in the present paper and will refer the interested reader to Nielson [19]. In our case the purity analysis which we discuss

$$\begin{tikzcd}
C \arrow[r, "\mu\llbracket P\rrbracket"] \arrow[d] & D \arrow[d] \\
\mathcal{P}(C) \arrow[r, "\pi\llbracket P\rrbracket"] & \mathcal{P}(D) \\
A \arrow[r, "\alpha\llbracket P\rrbracket"] \arrow[u] & B \arrow[u]
\end{tikzcd}$$

Figure 3: Abstract Interpretation Diagram

is only partly correct so we need not worry about the appropriate powerdomain construction. In terms of this diagram the correctness condition can be stated as:

$$\forall x \in A \quad \pi\llbracket P\rrbracket(\gamma_{AC}(x)) \subseteq \gamma_{BD}(\alpha\llbracket P\rrbracket(x))$$

In our formulation we shall express the soundness of our inference scheme as a relation between two semantics.

3 The Language and its Semantics

The language that we use resembles an ordinary typed functional programming language with **let** blocks, a **letrec** construct for recursive definitions and arbitrary typed lambda abstraction. Functions are almost "first-class objects": we do not permit variables with functional types. We add two explicitely imperative features: an ordinary assignment statement written $e_1 \leftarrow e_2$ and a dereferencing construct $e\uparrow$. A variable denotes a storage location; thus to denote the value of associated with the variable we need to use the dereferencing construct. Note that it is perfectly possible to have an expression evaluate to a variable rather than to the value associated with the variable as in **if** ***true*** **then** x **else** y.

In this section we shall specify the syntax and the standard semantics of the language.

The Abstract Syntax

This is the abstract syntax for the language:

$e \ = \ x$

$$\begin{array}{l}
\textbf{let } x = e_1 \textbf{ in } e_2 \\
\textbf{letrec } f_1 = \lambda x_1 : T_1.e_1 \text{ , } \ldots \text{ , } f_n = \lambda x_n : T_n.e_n \textbf{ in } e \\
\textbf{new } x : T \textbf{ in } e \\
\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \\
e_1 \ ; \ e_2 \\
e_1 \leftarrow e_2 \\
e\uparrow \\
\lambda x : T.\ e \\
e_1(e_2)
\end{array}$$

Only simple types are allowed. There are no constraints on the syntax except the ones required by typechecking: functions are not storable items, and only one level of reference or dereference is allowed. We are making this restriction for simplicity only; in future work we plan to investigate purity analysis in the presence of arbitrary indirection and with arrays. It is important to forbid l–values from being exported outside their scope. Consider, for example, the following expression $\lambda x : int\ .\ \textbf{new}\ y : int\ \textbf{in}\ y$. Here the address of y gets exported when the function is applied. This property can be detected statically by the methods developed here so we are justified in excluding it.

The Store

Before we give the actual semantic equations for the standard semantics of the language we need to discuss the way we model the store. This 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 we are able to prove the soundness of our inference scheme without being committed to any particular policy of storage management. We describe the relevant assumptions that we make about the store as axioms below. For us too the store management policy will affect the meanings of constructs, but 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.

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.

We will not give a fully specified model of the store. The soundness proofs for our abstract interpretations require only that the store model obey the axioms listed below. 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.

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

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).

$$\mathit{Allocate} : \mathit{Store} \rightarrow \mathit{Store} \times \mathit{Loc}$$

<u>Axiom 1</u>

if $\langle \mathit{newstore}, \mathit{address} \rangle = \mathit{Allocate}(\mathit{store})$
then

(a) $\mathit{newstore}\,|_{\mathit{Loc}-\{\mathit{address}\}} = \mathit{store}\,|_{\mathit{Loc}-\{\mathit{address}\}}$

(b) $\mathit{store}\,|_{\{\mathit{address}\}} = \perp$

$\mathit{newstore}\,|_{\{\mathit{address}\}} = \underline{\mathit{empty}}$

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

$$\mathit{DeAllocate} : \mathit{Store} \times \mathit{Loc} \rightarrow \mathit{Store}$$

Axiom 2

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

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

$$\mathit{Update} : \mathit{Store} \times \mathit{Loc} \times \mathit{Val} \rightarrow \mathit{Store}$$

$$\mathit{Eval} : \mathit{Store} \times \mathit{Loc} \rightarrow \mathit{Val}$$

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

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

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

$$\mathit{Eval}(\mathit{Update}(\mathit{store}, \mathit{address}, \mathit{value}), \mathit{address}) = \mathit{value}$$

By abuse of notation, we also allow $S \subseteq \mathcal{P}(Id)$, a subset of the variables already allocated in

$$\mathit{store}\ |_S$$

The Semantics

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

$$\begin{aligned} \mathit{Val} &= \mathit{Triv} + \mathit{Bool} + \mathit{Nat} + \mathit{FVal} + \mathit{Loc} \\ \mathit{Env} &= \mathit{Id} \rightarrow \mathit{Val} \\ \mathit{FVal} &= \mathit{Val} \rightarrow \mathit{Store} \rightarrow (\mathit{Val} \times \mathit{Store}) \end{aligned}$$

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:

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

and

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

A full definition of the semantics is given in Appendix A; the part included here is intended to illustrate the notation and style of semantic definition that we shall use.

$$\begin{aligned} \mathbf{M_e}[\![x]\!]\boldsymbol{env\ store} &= \boldsymbol{env(x)} \\ \mathbf{M_s}[\![x]\!]\boldsymbol{env\ store} &= \boldsymbol{store} \end{aligned}$$

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.

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

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

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

where

$$\begin{aligned} \boldsymbol{env'} &= \boldsymbol{env}[x \leftarrow \mathbf{M_e}[\![e_1]\!]\boldsymbol{env\ store}] \\ \boldsymbol{store'} &= \mathbf{M_s}[\![e_1]\!]\boldsymbol{env\ store} \end{aligned}$$

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.

$$\begin{aligned}\mathbf{M_e}[\![\mathbf{new}\ x : T\ \mathbf{in}\ e]\!] \mathit{env}\ \mathit{store} &= \mathbf{M_e}[\![e]\!]\ \mathit{env}'\ \mathit{store}' \\ \mathbf{M_s}[\![\mathbf{new}\ x : T\ \mathbf{in}\ e]\!] \mathit{env}\ \mathit{store} &= \mathit{store}'''\end{aligned}$$

where

$$\begin{aligned}&\langle \mathit{address}, \mathit{store}'\rangle = \mathit{Allocate}(\mathit{store}) \\ &\mathit{env}' = \mathit{env}[x \leftarrow \mathit{address}] \\ &\mathit{store}'' = \mathbf{M_s}[\![e]\!]\ \mathit{env}'\ \mathit{store}' \\ &\mathit{store}''' = \mathit{DeAllocate}(\mathit{store}'', \mathit{address})\end{aligned}$$

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.

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

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

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

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

where

$$\begin{aligned}&\mathit{address} = \mathbf{M_e}[\![e_1]\!] \mathit{env}\ \mathit{store} \\ &\mathit{store}' = \mathbf{M_s}[\![e_1]\!] \mathit{env}\ \mathit{store}\end{aligned}$$

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

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

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

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

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

4 Purity Analysis–A First Attempt

In this section we shall describe a simple approach to purity analysis based on an abstract interpretation. It will turn out that this is inadequate for our purposes but will serve as a good introduction to the more refined analyses presented in the next two sections.

This version of purity analysis uses two abstracted semantic functions, $\mathbf{P}_1$ and $\mathbf{P}_2$. The definition of these two functions is mutually recursive. The function $\mathbf{P}_1$ has a simple interpretation:

"$\mathbf{P}_1[e] penv\ =\ true$" means "expression e is pure "

The meaning of $\mathbf{P}_2$ is more subtle. Intuitively, it models side effects that arise when functions are applied. Recall that in the examples we saw a situation where a pure function had a side effect as soon as it was applied. To keep track of such possibilities we need the second semantic function. To continue with our present example, if e is not of function type, the value of $\mathbf{P}_2[e] penv$ is simply "$\underline{atom}$". If e denotes a function, then $\mathbf{P}_2$ keeps track of two facts: whether e itself has side effects, and how to evaluate $\mathbf{P}_1$ and $\mathbf{P}_2$ for the value that results after application of e. To simplify the notation, $\mathbf{P}_1$ and $\mathbf{P}_2$ are written as components of one semantic function $\mathbf{P}$. The semantic domains that we use are:

$$Val_P = \{\underline{atom}\} + Val_P \rightarrow D_P$$
$$Env_P = Id \rightarrow Val_P$$
$$D_P = Bool \times Val_P$$

while the combined semantic function $\mathbf{P}$ is

$$\mathbf{P} : Exp \rightarrow Env_P \rightarrow D_P$$

The domain D_P pairs together the information needed to define $\mathbf{P}_1$ and $\mathbf{P}_2$. Note that it has to be recursively defined in order to capture the relevant aspects of the types of higher–order functions. The element <u>*atom*</u> abstracts all nonfunctional values. In other words, if an expression is given the value <u>*atom*</u> by the semantic function then the expression is not a function (and there is no possibility that it could produce side effects when applied).

Finally, the partial order of domain D_P is defined by :

$$\begin{aligned} &\bot_{D_P} = \langle \bot_{Bool}, \bot_{PVal} \rangle \\ &\lambda u.\langle b_1, v_1 \rangle \sqcap \lambda u.\langle b_2, v_2 \rangle \;=\; \lambda u.\langle b_1 \wedge b_2, v_1 \sqcap v_2 \rangle \end{aligned}$$

The element <u>*atom*</u> is related only to $\bot$. This is reasonable since once we definitely know that an expression is not a function it cannot become one as computation progresses. We begin with an intentionally simplistic definition of purity, namely that an expression is pure if and only if its computation doesn't require any assignments to or evaluations of variables. In the next section, a more exact definition is discussed.

$$\mathbf{P}[\![x]\!]penv \;=\; \langle\, true,\; penv(x) \rangle$$

This clause of the abstract interpretation says that an identifier by itself is pure and that its potential for producing side effects during applications depends on whether it is bound to a function. The bindings of identifiers are kept in an environment *penv*.

$$\begin{aligned} &\mathbf{P}[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!]penv \;= \\ &\quad \langle \mathbf{P}_1[\![e_1]\!]penv \wedge \mathbf{P}_1[\![e_2]\!]penv[x \leftarrow \mathbf{P}_2[\![e_1]\!]penv]\;,\; \mathbf{P}_2[\![e_2]\!]penv[x \leftarrow \mathbf{P}_2[\![e_1]\!]penv]\ \rangle \end{aligned}$$

$$\begin{aligned} &\mathbf{P}[\![\mathbf{letrec}\ f_1 = e_1,\ \ldots\ , f_n = e_n\ \mathbf{in}\ e]\!]penv \;=\; \mathbf{P}[\![e]\!]penv' \\ &\quad \text{where } penv' \;=\; lfp(\lambda penv.penv[\ldots, f_i \leftarrow \mathbf{P}_2[\![e_i]\!]penv, \ldots]) \end{aligned}$$

Informally we can understand the clause above as saying that a **let** block is pure when the evaluation of its binding is free of side effects and the evaluation of its body in the new environment is free of side effects. Note that we need to compute $\mathbf{P}_2[e_1]penv$ in order to set up the environment properly for the evaluation of $\mathbf{P}_1[e_2]penv$. The **letrec** case is essentially identical except that the environment has to be built using a least fixed point construction.

An expression allocating a new variable is pure if the subexpression is pure in the new environment. The new variable being declared is bound to $\underline{atom}$ since it is not a function.

$$\mathbf{P}[\![\mathbf{new}\ x : T\ \mathbf{in}\ e]\!]penv \;=\; \mathbf{P}[\![e]\!]penv[x \leftarrow \underline{atom}]$$

A conditional expression is pure if its three subexpressions are pure. We cannot decide which branch of the expression will be executed, therefore the second component of $\mathbf{P}$ is the join of the two branches. The partial order on D_P has been chosen in such a way that the join represents a conservative estimate of the purity.

$$\begin{aligned}\mathbf{P}[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]penv \;=\;& \\ \langle\, \mathbf{P}_1[\![e_1]\!]penv \wedge \mathbf{P}_1[\![e_2]\!]penv \wedge \mathbf{P}_1[\![e_3]\!]penv \;,\; & \mathbf{P}_2[\![e_2]\!]penv \sqcap \mathbf{P}_2[\![e_3]\!]penv \,\rangle\end{aligned}$$

$$\mathbf{P}[\![e_1 \,;\, e_2\,]\!]penv \;=\; \langle\, \mathbf{P}_1[\![e_1]\!]penv \wedge \mathbf{P}_1[\![e_2]\!]penv \;,\; \mathbf{P}_2[\![e_2]\!]penv \,\rangle$$

$$\mathbf{P}[\![e_1 \leftarrow e_2\,]\!]penv \;=\; \langle\, false \,,\, \underline{atom} \,\rangle$$

$$\mathbf{P}[\![e\!\uparrow]\!]penv \;=\; \langle\, false \,,\, \underline{atom} \,\rangle$$

The above two clauses assert that assignments and dereferencing are not pure.

$$\mathbf{P}[\![\lambda x : T.e]\!]penv \;=\; \langle\, true \;,\; \lambda u.\mathbf{P}[\![e]\!]penv[x \leftarrow u] \,\rangle$$

A lambda abstraction by itself is always pure because its evaluation does nothing as a glance at the standard semantics confirms. For this reason we cannot check purity by searching for textual occurrences of the imperative constructs. It is during applications that the side effects concealed by lambda abstractions become manifest and the following semantic clause gives the rule for determining whether an application is pure. Note how the information in $\mathbf{P}_2\llbracket e_1\rrbracket penv$ is needed to determine $\mathbf{P}_1\llbracket e_1(e_2)\rrbracket penv$. The subscripts 1 and 2 in the expressions below are used to denote the first or second component of the resulting tuple.

$$
\begin{aligned}
&\mathbf{P}\llbracket e_1(e_2)\rrbracket penv \; = \\
&\quad \langle\, \mathbf{P}_1\llbracket e_1\rrbracket penv \wedge \mathbf{P}_1\llbracket e_2\rrbracket penv \wedge (\mathbf{P}_2\llbracket e_1\rrbracket penv\mathbf{P}_2\llbracket e_2\rrbracket penv)_1 \;, \\
&\qquad (\mathbf{P}_2\llbracket e_1\rrbracket penv\mathbf{P}_2\llbracket e_2\rrbracket penv)_2 \,\rangle
\end{aligned}
$$

The above definition of purity declares an expression impure even if its evaluation affects only local variables. This is certainly too stringent to be useful in general. A more standard definition of purity states that an expression is pure if its computation doesn't require any assignments or evaluations of *global* variables. In contrast with the simplistic definition, assignments and evaluations of local variables are allowed. It is impossible to modify the definition of **P** to approximate the standard definition of purity. An example will make this clear. Consider the following two expressions:

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

and

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

The first expression would be impure by the simple definition but not by the standard definition. The second expression, however, is impure by both definitions. The two expressions are structurally very similar and there is no minor modification to the definition of **P** that distinguishes them. Notice that the information missing from **P** is the set of variables affected by or which affect the evaluation of an expression. Thus instead of using the boolean domain we will use a richer structure capable of expressing this information. 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. In the next two sections we describe a more elaborate semantic function, **S**, sufficient to compute the standard definition of purity.

5 An Abstract Interpretation for Computing Aliases

Before we can calculate the support of an expression, we need another semantic function to determine its aliasing behavior. Consider for instance a dereferencing expression $e\uparrow$. 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 computation of the aliases as an example of an abstract interpretation.

In the absence of data structures, it is straightforward to compute the aliasing behavior of an expression. The semantic function **A**, defined below, gives the possible aliases of an expression as a set of variable identifiers. An r-valued expression cannot be an alias, because the type discipline forbids more than one level of reference (no pointer to a pointer), and its value will be the empty set. The recursive structure of the semantic domains is similar to the one used to compute **P**. Only the base domain *Bool* is replaced by $\mathcal{P}(Id)$, the powerset of *Id*, the syntactical class of identifiers.

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

The lattice structure over domain D_A is defined by :

$$\perp_{D_A} = \langle \emptyset, \perp_{Val_A} \rangle$$
$$\lambda u.\langle b_1, v_1\rangle \sqcap \lambda u.\langle b_2, v_2\rangle = \lambda u.\langle b_1 \cup b_2, v_1 \sqcap v_2\rangle$$

where $\sqcap$ is the least upper bound operator. The element $\underline{atom}$ is related only to $\perp$. 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.

It is interesting to observe that the definition of **A** bears some resemblance to the definition of $\mathbf{M}_e$. In contrast, the semantics of purity and support have to do with side effects of evaluating an expression, and are similar to the definition of $\mathbf{M}_s$. The semantic clauses for the computation of aliases is given below. As before we need an environment which we shall call *aenv*. The environment tells us the set of identifiers that may alias each identifier.

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

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

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

$$\mathbf{A}[\mathbf{new}\ x : T\ \mathbf{in}\ e]aenv = \mathbf{A}[e]aenv[x \leftarrow \langle\ \{"x"\}, \underline{atom}\rangle]$$

This clause tells us that a new variable starts out being aliased only to itself.

$$\mathbf{A}[\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]aenv = \mathbf{A}[e_2]aenv \sqcap \mathbf{A}[e_3]aenv$$

This clause illustrates the conservative nature of our estimate in the presence of conditionals. 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.

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

$$\mathbf{A}[e_1 \leftarrow e_2\ ]aenv = \langle\ \emptyset\ ,\ \underline{atom}\ \rangle$$

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.

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

We shall not prove the soundness of this abstract interpretation yet but will defer the proof until after we have proven the soundness of the rest of our analysis. However, the intuitive justification behind our scheme for estimating alias sets is clear and the resemblance to the standard semantics is manifest.

6 Support Sets as an Abstract Interpretation

In this section we show how the support set can computed by using an appropriate abstract interpretation. The support of an expression is defined as the set of global variables whose values may affect or be affected by the evaluation of that expression. The support is approximated by the semantic function **S**. An expression is pure if its support is empty. This is sufficient to distinguish between the two examples of page 4, and to correctly infer that the first one is pure, but the second is not. The semantic domains and semantic function **S** are given below. The domains are very similar to those introduced earlier for aliases and for the first attempt at determining purity.

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

The definition of the semantic function **S** is given below. As with the definition of purity we shall use subscripts if we wish to use only the first or second components of a pair in D_S. The semantic function **A** is used in the definition of **S**. Note also that when we use the semantic function **A** we need the environment $aenv$; we assume that this environment results from a computation of **A** for the same expression as the one for which the action of **S** is being defined. Strictly speaking the functions **A** and **S** should be defined together but we chose to present them separately so that the intuitive content of the two computations is clearer.

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

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

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

The definitions of **S** in the above three cases are fairly straightforward. An identifier by itself has no support. In a **let** block the bindings are computed first and then the body is computed in the resulting environment. The second component is needed in case a non–trivial support results from a later function application. The **letrec** construct is once again exactly like the **let** construct except for the presence of the environment defined as a least fixed point.

$$\mathbf{S}[\![\mathbf{new}\ x : T\ \mathbf{in}\ e]\!]senv \ = \\ \langle \mathbf{S}_1[\![e]\!]senv[x \leftarrow \underline{atom}] \ - \ \{''x''\}\ ,\ \mathbf{S}_2[\![e]\!]senv[x \leftarrow \underline{atom}]\ \rangle$$

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 need to explicitly remove the new variable from the support which we compute for the body of the block.

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

The support is computed conservatively so that the supports of all three expressions in a conditional are unioned together.

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

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

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

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 need to union together the supports of the two sides of the assignment and also all possible aliases of the left hand side of the assignment. A similar observation applies to the dereferencing construct.

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

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

If one wishes to distinguish between variables read and variables assigned to, it is very easy to decompose the definition of **S** into two semantic functions, **R** and **W**, which have definitions identical to **S**, except for assignments and dereferences.

$$\mathbf{R}[\![e_1 \leftarrow e_2]\!]\boldsymbol{renv} \;=\; \langle\, \mathbf{R}_1[\![e_1]\!]\boldsymbol{renv} \cup \mathbf{R}_1[\![e_2]\!]\boldsymbol{renv}\,,\ \underline{atom}\,\rangle$$

$$\mathbf{R}[\![e\uparrow]\!]\boldsymbol{renv} \;=\; \langle\, \mathbf{R}_1[\![e]\!]\boldsymbol{renv} \cup \mathbf{A}_1[\![e]\!]\boldsymbol{aenv}\,,\ \underline{atom}\,\rangle$$

$$\mathbf{W}[\![e_1 \leftarrow e_2]\!]\boldsymbol{wenv} \;=\; \langle\, \mathbf{W}_1[\![e_1]\!]\boldsymbol{wenv} \cup \mathbf{W}_1[\![e_2]\!]\boldsymbol{wenv} \cup \mathbf{A}_1[\![e_1]\!]\boldsymbol{aenv}\,,\ \underline{atom}\,\rangle$$

$$\mathbf{W}[\![e\uparrow]\!]\boldsymbol{wenv} \;=\; \langle\, \mathbf{W}_1[\![e]\!]\boldsymbol{wenv} \cup \mathbf{A}_1[\![e]\!]\boldsymbol{aenv}\,,\ \underline{atom}\,\rangle$$

The support of an expression is the union of the set of variables read and the set of variables written:

$$\mathbf{S}[\![e]\!]\boldsymbol{senv} = \mathbf{R}[\![e]\!]\boldsymbol{renv} \cup \mathbf{W}[\![e]\!]\boldsymbol{wenv}$$

The environments ***senv***, ***wenv***, and ***renv*** must result from similar computations just as ***aenv*** and ***senv*** must in the definition of **S**.

In the next section we shall prove the correctness of our inference scheme. The correctness is already intuitively plausible because of the correspondence between the standard semantics and the abstract interpretations used to define support and alias. In our view, this is a major advantage of presenting static analysis schemes as abstract interpretations.

7 Soundness Theorem for Support Sets

In this section we shall prove a soundness theorem for our analysis scheme. The theorem that we prove is very similar to the theorem proved by Hudak and Young [10] for the soundness of their higher–order strictness analysis scheme. 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. Such a double induction is used in, for example, the proof of the strong normalization of the typed lambda calculus. Because we have a type system with only finite types we can use induction on the type system to prove interesting facts. It is for exactly the same reason that the inference mechanism converges. Thus it is no accident that we need to use aspects of the type system to prove the soundness theorem. 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 framework of Burn, Hankin and Abramsky [3] also uses a typed language. They point out that their scheme for strictness analysis depends critically on the presence of a type system.

If the support of an expression is empty, this expression is pure, that is its computation doesn't modify the store and its value doesn't depend on the store.

$$\mathbf{S}_1[\![e]\!]\mathit{senv} \;=\; \emptyset \;\Rightarrow\; \forall\, \mathit{store}\;\; \mathbf{M_s}[\![e]\!]\mathit{env}\;\mathit{store} \;=\; \mathit{store}$$

$$\mathbf{S}_1[\![e]\!]\mathit{senv} \;=\; \emptyset \;\Rightarrow\; \forall\, \mathit{store}, \mathit{store}'\;\; \mathbf{M_e}[\![e]\!]\mathit{env}\;\mathit{store} \;=\; \mathbf{M_e}[\![e]\!]\;\mathit{env}\;\mathit{store}'$$

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. For instance, the following function converges only if its functional argument is a strictly decreasing function.

$$
\begin{array}{l}
\textbf{letrec } f = \lambda g : \mathit{int} \rightarrow \mathit{int}.\lambda n : \mathit{int}. \\
\qquad\qquad \textbf{if } n = 0 \textbf{ then } g(0) \\
\qquad\qquad \textbf{else } f(g, g(n)) \\
\textbf{in } \ldots
\end{array}
$$

In addition, to take advantage of the structural definition of **S**, we shall prove a more general theorem: the computation of an expression doesn't modify the store outside of its support, and its value is independent of the store values outside of its support. We can now give a formal statement of the soundness theorem:

Theorem

If the evaluation of e terminates,
$\forall$ ***env***, ***senv*** corresponding environments,

$$\mathbf{S}_1[\![e]\!]\boldsymbol{senv} \subseteq S \;\Rightarrow\; \forall\, \boldsymbol{store}\; \mathbf{M_s}[\![e]\!]\boldsymbol{env}\; \boldsymbol{store} \mid_{\overline{S}} = \boldsymbol{store} \mid_{\overline{S}}$$

$$\mathbf{S}_1[\![e]\!]\boldsymbol{senv} \subseteq S \;\Rightarrow\; \forall\, \boldsymbol{store}, \boldsymbol{store'} \text{ such that } \boldsymbol{store} \mid_S = \boldsymbol{store'} \mid_S$$

$$\mathbf{M_e}[\![e]\!]\boldsymbol{env}\; \boldsymbol{store} = \mathbf{M_e}[\![e]\!]\; \boldsymbol{env}\; \boldsymbol{store'}$$

$$\mathbf{M_s}[\![e]\!]\boldsymbol{env}\; \boldsymbol{store} \mid_S = \mathbf{M_s}[\![e]\!]\; \boldsymbol{env}\; \boldsymbol{store'} \mid_S$$

We need a precise definition of corresponding environments ***senv*** and ***env***. As in Hudak and Young [10], 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.

$$PAP_n : (D_S)^n \rightarrow \mathcal{P}(Id)$$

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

The notation s_v represents the first (or "value") component of a pair from the domain Val_S while s_f represents the second (or "function") component of such a pair.

$$AP_n : Val^n \rightarrow Store \rightarrow Val \times Store$$

$$AP_n(e, e_1, \ldots, e_n, store) = \begin{cases} e\ store & n = 0 \\ AP_{n-1}((e\ e_1\ store)_1, \ldots, e_n, store) & n > 0 \end{cases}$$

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 semantics domain must terminate. 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:
$\forall m \leq n,\ s_i \in D_S,\ e_i \in D,\ s_i$ safe at level $n-1$ for e_i

$$
\begin{array}{ll}
PAP_m(s, s_1, \ldots, s_m) \subseteq S & \\
\quad \Rightarrow & \\
\forall\ store, store' \text{ such that } store \mid_S = store' \mid_S & \\
\quad [AP_m(e, e_1, \ldots, e_m, store)]_1 = [AP_m(e, e_1, \ldots, e_m, store')]_1 & \text{(1a)} \\
\quad [AP_m(e, e_1, \ldots, e_m, store)]_2 \mid_S = [AP_m(e, e_1, \ldots, e_m, store')]_2 \mid_S & \text{(1b)} \\
 & \\
\quad \wedge & \\
\forall\ store\ \ [AP_m(e, e_1, \ldots, e_m, store)]_2 \mid_{\overline{S}} = store \mid_{\overline{S}} & \text{(2)}
\end{array}
$$

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

$senv$ and env are <u>corresponding environments</u> if
$senv(x)$ is safe for $\lambda s.\langle env(x), s\rangle\ \ \forall\, x \in dom(senv)$

<u>Proof</u> :
We will show that $\mathbf{S}[\![e]\!]senv$ is safe for $\lambda s.\langle \mathbf{M_e}[\![e]\!]\ env\ s, \mathbf{M_s}[\![e]\!]\ env\ s\rangle$. 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. The reader should refer to Appendix A for the definitions and the notation used in the proof. The proof of $(1a)$ and $(1b)$ are virtually identical, therefore we shall only describe the proof of $(1a)$.

1. $e \equiv \textbf{new}\ x : T\ \textbf{in}\ e_1$

   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 $\mathbf{S}[\![\mathbf{new}\ x : T\ \mathbf{in}\ e_1]\!]senv \subseteq S$

(a) $\mathbf{S}[\![e_1]\!]senv \subseteq S \cup \{"x"\}$ 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 (a)

(d) $store''\ |_S = store'\ |_S$ by restriction on (c)

(e) Since S is disjoint of $\{x\}$, 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 $n \geq 0$
Must show $\mathbf{S}[\![e_1(e_2)]\!]senv$ is safe at level n for $\lambda s.\langle \mathbf{M_e}[\![e_1(e_2)]\!]\ env\ s, \mathbf{M_s}[\![e_1(e_2)]\!]\ env\ s\rangle$
Choose $m \leq n$, 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.

$$(1)\quad PAP_m(\mathbf{S}[\![e]\!]senv, s_1, \ldots, s_m) \subseteq S \quad \Rightarrow$$
$$AP_m(\mathbf{M_e}[\![e_1(e_2)]\!]env\ store, v_1, \ldots, v_m, store)$$
$$= AP_m(\mathbf{M_e}[\![e_1(e_2)]\!]\ env\ store', v_1, \ldots, v_m, store')$$
$$\text{if } store\ |_S = store'\ |_S$$

$$(2)\quad PAP_m(\mathbf{S}[\![e]\!]senv, s_1, \ldots, s_m) \subseteq S \quad \Rightarrow$$
$$(AP_m(\mathbf{M_e}[\![e_1(e_2)]\!]env\ store, v_1, \ldots, v_m, store))\ |_{\overline{S}} = store\ |_{\overline{S}}$$

To prove the first we proceed as follows:
By definition of PAP

$$PAP_m(\mathbf{S}[\![e_1(e_2)]\!]senv, s_1, \ldots, s_m)$$
$$= (\mathbf{S}[\![e_1(e_2)]\!]senv)_v \cup (s_1)_v \cup PAP_{m-1}((\mathbf{S}[\![e_1(e_2)]\!]senv)_f\ s_1, s_2, \ldots, s_m)$$

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.

$$= (\mathbf{S}[\![e_1]\!]senv)_v \cup (\mathbf{S}[\![e_2]\!]senv)_v \cup ((\mathbf{S}[\![e_1]\!]senv)_f \mathbf{S}[\![e_2]\!]senv)_v \cup$$

$$(s_1)_v \cup PAP_{m-1}(((\mathbf{S}[\![e_1]\!]senv)_f \mathbf{S}[\![e_2]\!]senv)_f s_1, \ldots, s_m)$$

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.

$$= (\mathbf{S}[\![e_1]\!]senv)_v \cup (\mathbf{S}[\![e_2]\!]senv)_v \cup PAP_m((\mathbf{S}[\![e_1]\!]senv)_f \mathbf{S}[\![e_2]\!]senv, s_1, \ldots, s_m)$$

$$= PAP_{m+1}(\mathbf{S}[\![e_1]\!]senv, \mathbf{S}[\![e_2]\!]senv, s_1, \ldots, s_m)$$

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:

$$(AP_{m+1}(\mathbf{M_e}[\![e_1]\!]env\ store, \mathbf{M_e}[\![e_2]\!]env\ store, v_1, \ldots, v_m, store))\ |_{\overline{S}} = store\ |_{\overline{S}}$$

But

$$(AP_{m+1}(\mathbf{M_e}[\![e_1]\!]env\ store, \mathbf{M_e}[\![e_2]\!]env\ store, v_1, \ldots, v_m, store))$$

$=AP_m((\mathbf{M_e}[\![e_1]\!]env\ store\ \mathbf{M_e}[\![e_2]\!]env\ store)_f, v_1, \ldots, v_m, store)$ by definition of AP

$=(AP_m(\mathbf{M_e}[\![e_1(e_2)]\!]env\ store, v_1, \ldots, v_m, store))$ by definition of $\mathbf{M_e}$

and therefore (2) holds.
For (1) we proceed as follows:

$$PAP_{m+1}(\mathbf{S}[\![e_1]\!]senv, \mathbf{S}[\![e_2]\!]senv, s_1, \ldots, s_m)$$

$$\Rightarrow$$

$$AP_{m+1}(\mathbf{M_e}[\![e_1]\!]env\ store, \mathbf{M_e}[\![e_2]\!]env\ store, , v_1, \ldots, v_m, store)$$

$$= AP_{m+1}(\mathbf{M_e}[\![e_1]\!]\ env\ store', \mathbf{M_e}[\![e_2]\!]\ env\ store', v_1, \ldots, v_m, store')$$

$$\Rightarrow$$

$$AP_m(\mathbf{M_e}[\![e_1(e_2)]\!]env\ store, v_1, \ldots, v_m, store)$$

$$= AP_m(\mathbf{M_e}[\![e_1(e_2)]\!]\ env\ store', v_1, \ldots, v_m, store')$$

The rest of the proof is in Appendix B.

8 Soundness Theorem for Aliases

In this section, S will denote a subset of the identifier set. 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:

$$
\begin{array}{l}
\forall\, S \subseteq \mathcal{P}(Id), \\
\quad \mathbf{A}_1[\![e]\!]aenv \subseteq S \;\Rightarrow \\
\quad\quad \forall\, y \in \overline{S},\ \forall\, store \;\; \mathbf{M_e}[\![e]\!]env\ store \neq \mathbf{M_e}[\![y]\!]env\ store
\end{array}
$$

Again, we need a precise definition of corresponding environments *aenv* and *env*. As in the previous section, we define partial application operators in both semantics domains.

$$PAP_n \;:\; (D_A)^n \rightarrow \mathcal{P}(Id)$$

$$PAP_n(a, a_1, \ldots, a_n) = \begin{cases} (a)_1 & n = 0 \\ PAP_{n-1}((a\, a_1)_2, \ldots, a_n) & n > 0 \end{cases}$$

$$AP_n \;:\; Val^n \rightarrow Store \rightarrow Val$$

$$AP_n(e, e_1, \ldots, e_n, store) = \begin{cases} e\ store & n = 0 \\ AP_{n-1}((e\, e_1)_1\ store, \ldots, e_n, store) & n > 0 \end{cases}$$

$a \in D_A$ is <u>safe at level n for value $e \in D$</u> if:

$$
\begin{array}{l}
\forall\, m \leq n,\ a_i \in D_A,\ e_i \in D,\ a_i \text{ safe at level } n-1 \text{ for } e_i \\
\quad PAP_m(a, a_1, \ldots, a_m) \subseteq S \\
\quad\quad \Rightarrow \\
\quad \forall\, y \in \overline{S}, \forall\, store \\
\quad\quad AP_m(e, e_1, \ldots, e_m, store) \neq \mathbf{M_e}[\![y]\!]env\ store
\end{array} \tag{3}
$$

$a \in D_A$ is <u>safe for value $e \in D$</u> if
it is safe at all levels

aenv and *env* are <u>corresponding environments</u> if
$aenv(x)$ is safe for $env(x)$ $\forall\, x \in dom(aenv)$

The proof is a simplified version of the one for support sets and offers no new insight. The complete text can be found in Appendix C.

9 Conclusion

In this paper we have described a method for determining the purity of expressions. In the process we have also produced a method for answering a more general question namely whether the evaluation of two expressions is independant of each other. The presentation of our method uses the technique of abstract interpretation. The question that needs to be settled is to what extent these abstract interpretations define decision procedures. Clearly if the semantic domains used are all finite then we have a decision procedure because the semantic functions are all computable and equality of functions is decidable. The abstract interpretations used by Mishra and Keller [15] and by Mycroft [17] used finite domains and came up with decidable conservative estimates of properties of interest.

In the presence of higher–order constructs it is not clear how to define interesting abstract interpretations which have underlying finite domains. Hudak and Young [10] describes a method for performing higher–order abstract interpretation but they use domains which have infinitely long ascending chains and hence the computation of least fixed points may not terminate. The domains which we use for our analysis are also infinite and thus in general we cannot produce an algorithm which computes our semantic functions **A** and **S**. One may argue that the analysis that we have provided is not useful for this reason, and indeed it would be much more satisfying to have a decision procedure for our analysis and for strictness analysis. Usually, however, we do not need to know a function completely, in other words it suffices to have an approximation to the function because higher–order functions are usually applied finitely many times. Hudak and Young use this as a justification for presenting an inference scheme for strictness which is not effective. However, as they observe in their paper, their scheme does indeed become effective when a weak type discipline is imposed. In our view, the higher–order abstract interpretations which have been discussed [10], [3] should be used with typed languages because it is in exactly that case that the abstract interpretation does define a decision procedure. In our approach we do have a typed language, this means that a particular higher–order function can only be applied finitely many times and the computation of fixed points will indeed terminate in the abstract domain. In the approach discussed by Burn, Hankin and Abramsky [3] they show that strictness analysis can be done effectively in a language with simple types and Abramsky [1] shows that the same holds for a language with polymorphic types.

We are considering two directions for future work in this area. We intend to introduce data structures into the language and study the viability of purity analysis in the presence of this extension. Preliminary investigations with arrays indicate that the abstract interpretation can be extended smoothly to handle this case. The second direction is to work out the theoretical foundations of general higher–order abstract interpretation. The work of Burn, Hankin and Abramsky [3] shows how to do this for strictness analysis.

A Standard Semantics

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

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

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

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

where

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

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

where

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

$$\mathbf{M_e}[\![\mathbf{letrec}\ f_1 = \lambda x_1 : T_1.e_1, \ldots, f_n = \ldots\ \mathbf{in}\ e]\!] env\ store \ = \ \mathbf{M_e}[\![e_2]\!]\ env'\ store$$

$\mathbf{M_s}[\textbf{letrec } f_1 = \lambda x_1 : T_1.e_1, \ldots, f_n = \ldots \textbf{ in } e]\mathit{env\ store} \ = \ \mathbf{M_s}[\![e_2]\!]\ \mathit{env'\ store}$
where

$\mathit{env'} \ = \ \mathit{lfp}(F\ \mathit{env})$

$F \ : \ \mathit{Env} \to \mathit{Env} \ =$

$\lambda \mathit{env}.\mathit{env}[\ldots,\ f_i \leftarrow \mathbf{M_e}[\![\lambda x_i : T_i.e_i]\!]\mathit{env\ store}, \ldots]$

$\mathbf{M_e}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\mathit{env\ store} \ =$

$$\begin{cases} \bot & \text{if } \mathit{val} = \bot \\ \mathbf{M_e}[e_2]\ \mathit{env\ store'} & \text{if } \mathit{val} = \mathit{true} \\ \mathbf{M_e}[e_3]\ \mathit{env\ store'} & \text{if } \mathit{val} = \mathit{false} \end{cases}$$

$\mathbf{M_s}[\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\mathit{env\ store} \ = \ \mathit{store''}$

where

$\mathit{val} \ = \ \mathbf{M_e}[e_1]\mathit{env\ store}$

$\mathit{store'} \ = \ \mathbf{M_s}[\![e_1]\!]\mathit{env\ store}$

$\mathit{store''} \ =$

$\mathbf{M_s}[e_2]\ \mathit{env\ store'}\ \text{ if }\ \mathit{val} = \mathit{true}$

$\mathbf{M_s}[e_3]\ \mathit{env\ store'}\ \text{ if }\ \mathit{val} = \mathit{false}$

$\mathit{val'} \ =$

$\bot\ \text{ if } \mathit{val} = \bot$

$\mathbf{M_e}[\![e_2]\!]\ \mathit{env\ store'}\ \text{ if }\ \mathit{val} = \mathit{true}$

$\mathbf{M_e}[\![e_3]\!]\ \mathit{env\ store'}\ \text{ if }\ \mathit{val} = \mathit{false}$

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

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

$\mathbf{M_s}[e_1\ ;\ e_2]\!]\mathit{env\ store} \ = \ \mathbf{M_s}[e_2]\ \mathit{env\ store'}$
where $\mathit{store'} \ = \ \mathbf{M_s}[e_1]\mathit{env\ store}$

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

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

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

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

where

$\mathit{address} = \mathbf{M_e}\llbracket e_1 \rrbracket \mathit{env\ store}$

$\mathit{value} = \mathbf{M_e}[e_2]\ \mathit{env\ store}'$

$\mathit{store}' = \mathbf{M_s}\llbracket e_1]\mathit{env\ store}$

$\mathit{store}'' = \mathbf{M_s}\llbracket e_2 \rrbracket\ \mathit{env\ store}'$

$$\mathbf{M_e}\llbracket e\uparrow \rrbracket \mathit{env\ store}\ =$$

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

where $\mathit{address}\ =\ \mathbf{M_e}\llbracket e \rrbracket \mathit{env\ store}$

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

$\mathbf{M_e}\llbracket \lambda x : T.e]\mathit{env\ store}\ =\ \lambda v.\lambda s.\langle \mathbf{M_e}\llbracket e]\ \mathit{env}[x \leftarrow v]\ s, \mathbf{M_s}\llbracket e \rrbracket\ \mathit{env}[x \leftarrow v]\ s\rangle$

$\mathbf{M_s}[\lambda x : T.e]\mathit{env\ store}\ =\ \mathit{store}$

$$\mathbf{M_e}\llbracket e_1(e_2) \rrbracket \mathit{env\ store}\ =$$

$$\begin{cases} \perp & \text{if } \mathbf{M_e}\llbracket e_1 \rrbracket \mathit{env\ store} = \perp \\ \perp & \text{if } \mathbf{M_e}\llbracket e_2 \rrbracket\ \mathit{env\ store}' = \perp \\ \mathit{fe\ e\ store}'' & \text{otherwise} \end{cases}$$

$\mathbf{M_s}[e_1(e_2)]\mathit{env\ store}\ =\ \mathit{fs\ e\ store}''$

where

$\langle \mathit{fe}, \mathit{fs} \rangle\ =\ \mathbf{M_e}\llbracket e_1]\mathit{env\ store}$

$e\ =\ \mathbf{M_e}[e_2]\ \mathit{env\ store}'$

$\mathit{store}'\ =\ \mathbf{M_s}\llbracket e_1 \rrbracket \mathit{env\ store}$

$\mathit{store}''\ =\ \mathbf{M_s}[e_2]\ \mathit{env\ store}'$

B Proof of Soundness Theorem for Support Sets

The proof proceeds by structural induction on e and by induction on the type of e. For the **letrec** construct we will need fixpoint induction as well.

1. $e \equiv x$
   $\mathbf{M_s}[\![x]\!] env\ store \ = \ store$
   $\mathbf{M_e}[\![x]\!] env\ store \ = \ env(x)$
   $\mathbf{S}[\![x]\!] senv \ = \ senv(x)$
   Since $senv$ and env are corresponding environments,
   $senv(x)$ is safe for $\lambda s.\langle env(x), s\rangle$.

2. $e \equiv$ **let** $x = e_1$ **in** e_2
   This case is equivalent to $(\lambda x.e_2)(e_1)$

3. $e \equiv$ **letrec** $f_1 = e_1\ \ldots\ f_n = e_n$ **in** e'

   By fixpoint induction.
   - base case:
     $env_0 = env[\ldots, f_i \leftarrow \perp_D, \ldots]$ and $senv_0 = senv[\ldots, f_i \leftarrow \perp_{D_S}, \ldots]$ are corresponding environments.
     By structural induction on e', (1) and (2) are true.
   - induction step:
     Assume $senv_{n-1}$ and env_{n-1} are corresponding environments.
     $env_n = env_{n-1}[\ldots, f_i \leftarrow \mathbf{M_e}[\![e_i]\!]\ env_{n-1}\ store, \ldots]$
     $senv_n = senv_{n-1}[\ldots, f_i \leftarrow \mathbf{S}[\![e_i]\!] senv_{n-1}, \ldots]$
     Then by SI on e_i, $senv_n$ and env_n are corresponding environments.
     By fixpoint induction, $senv'$ and env' are corresponding environments,
     By structural induction on e', (1) and (2) are true.

4. $e \equiv$ **new** $x : T$ **in** e_1
   See Section 7

5. $e \equiv$ **if** e_1 **then** e_2 **else** e_3
   By structural induction on e_2 and e_3.

6. $e \equiv e_1 \,;\, e_2$
cf previous case.

7. $e \equiv e_1 \leftarrow e_2$

We need only to consider the case $n = 0$.
Show (1) for $n = 0$:
Assume $store_1 \mid_S = store_2 \mid_S$.
By structural induction on e_1, we have:

$$address = \mathbf{M_e}[\![e_1]\!]\ env\ store_1 = \mathbf{M_e}[\![e_1]\!]\ env\ store_2$$

and the resulting stores still agree on S.
Similarly, the right hand side evaluate to identical values:

$$value = \mathbf{M_e}[\![e_2]\!]\ env\ store_1' = \mathbf{M_e}[\![e_2]\!]\ env\ store_2'$$

The two stores are modified at the same address with the same values.

Show (2) for $n = 0$:

$$\mathbf{S}_1[\![e_1 \leftarrow e_2]\!]senv = \mathbf{S}_1[\![e_1]\!]senv \cup \mathbf{S}_1[\![e_2]\!]senv \cup \mathbf{A}_1[\![e_1]\!]aenv$$

$$\mathbf{S}_1[\![e_1 \leftarrow e_2]\!]senv \subseteq S \Rightarrow$$

$$\mathbf{S}_1[\![e_1]\!]senv \subseteq S \text{ (a)}$$

$$\mathbf{S}_1[\![e_2]\!]senv \subseteq S \text{ (b)}$$

$$\mathbf{A}_1[\![e_1]\!]aenv \subseteq S \text{ (c)}$$

By structural induction on e_1 and (a),

$$store' \mid_{\overline{S}} = store \mid_{\overline{S}}$$

By structural induction on e_2 and (b),

$$store'' \mid_{\overline{S}} = store' \mid_{\overline{S}}$$

By Axiom 3 and the soundness theorem for aliases,

$$\mathbf{A}[\![e_1]\!]aenv \subseteq S \Rightarrow store''' \mid_{\overline{S}} = store'' \mid_{\overline{S}}$$

and therefore (2)

8. $e \equiv e'\uparrow$

$\mathbf{M_e}[\![e'\uparrow]\!]env\ store = Eval(store, address)$ cannot be applied, since its type is not functional.
Thus, it is sufficient to prove (1) and (2) for $n = 0$.
If two stores agree on S, by structural induction, e' will evaluate to the

same value *address* in both.
$\mathbf{S}_1[e]senv = \mathbf{S}_1[e']senv \cup \mathbf{A}_1[e']aenv \subseteq S$
By the soundness theorem on aliases, $\{address\} \subseteq S$.
The equation $Eval(store'_1, address) = Eval(store'_2, address)$
follows from $(1b)$ on e'.

$\mathbf{M_s}[\![e\uparrow]\!]env\ store = store'$ by definition,
by structural induction on e',
$store' \mid_{\overline{S}} = store \mid_{\overline{S}}$
and we have (2).

9. $e \equiv e_1(e_2)$
   See section 7

10. $e \equiv \lambda x : T.e'$
    (2) for $n = 0$ is trivial, since:
    $\mathbf{M_s}[\![e \;=\; \lambda x : T.e']\!]env\ store = store$ by definition

    (2) for $n > 0$ and (1), we use structural induction on e':
    $\mathbf{M_e}[\![e \;=\; \lambda x : T.e']\!]env\ store = \lambda v.\lambda s.\langle \mathbf{M_e}[e']\ env[x \leftarrow v]\ s, \mathbf{M_s}[\![e']\!]\ env[x \leftarrow v]\ s\rangle$

    $\mathbf{S}[e \;=\; \lambda x : T.e']senv = \lambda v.\langle \mathbf{S}[\![e']senv[x \leftarrow v], \mathbf{S}[\![e']senv[x \leftarrow v]\rangle$

C Proof of Soundness Theorem for Aliases

We will prove that
$\forall\ aenv, env$ corresponding environments
$\mathbf{A}[\![e]aenv$ is safe for $\mathbf{M_e}[\![e]env\ store$

Proof by structural induction on e.
Choose $y \in \overline{S}$, and fix the store.

1. $e \equiv x$
   Assume $\mathbf{A}_1[\![e]\!]aenv \subseteq S$
   $\mathbf{A}[e]aenv = aenv(x)$ by definition of $\mathbf{A}$

$\mathbf{M_e}[\![e]\!]$ *env store* $=$ *env*(x) by definition of $\mathbf{M_e}$
Since ***aenv*** and ***env*** are corresponding environments by hypothesis,
aenv(x) is safe for ***env***(x)

2. $e \equiv$ **let** $x = e_1$ **in** e_2
   This case is equivalent to $(\lambda x.e_2)(e_1)$

3. $e \equiv$ **letrec** $f_1 = e_1 \ \ldots \ f_n = e_n$ **in** e'

   By fixpoint induction.
   - base case:
     $env_0 = env[\ldots, f_i \leftarrow \bot_D, \ldots]$ and $aenv_0 = aenv[\ldots, f_i \leftarrow \bot_{D_A}, \ldots]$ are corresponding environments.
     By structural induction on e', (3) is true.
   - induction step:
     Assume $aenv_{n-1}$ and env_{n-1} are corresponding environments.
     $env_n = env_{n-1}[\ldots, f_i \leftarrow \mathbf{M_e}[\![e_i]\!]\ env_{n-1}\ store, \ldots]$
     $aenv_n = aenv_{n-1}[\ldots, f_i \leftarrow \mathbf{A}[e_i]aenv_{n-1}, \ldots]$
     Then by SI on e_i, $aenv_n$ and env_n are corresponding environments.
     By fixpoint induction, $aenv'$ and env' are corresponding environments,
     By structural induction on e', (3) is true.

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

   It is sufficient to show
   $aenv[x \leftarrow \langle\ \{"x"\}, \underline{atom}\rangle]$ is safe for $env[x \leftarrow \langle\ \{"x"\}, \underline{atom}\rangle]$.
   Since ***aenv*** and ***env*** are corresponding environments,
   all that is left to prove is that
   $aenv(x) = \langle\ \{"x"\}, \underline{atom}\rangle$ is safe at all levels for $env(x) = address$
   Since x is not a function type, only the case $n = 0$ is considered. That is:
   $address \neq \mathbf{M_e}[\![y]\!]env\ store$
   which is true by Axiom 1b.

5. $e \equiv$ **if** e_1 **then** e_2 **else** e_3
   By structural induction hypothesis on e_2 and e_3.

6. $e \equiv e_1 \,;\, e_2$
   cf previous case.

7. $e \equiv e_1 \leftarrow e_2$
   In both the standard semantics and the aliasing semantics, the meaning of an assignment is that of its right hand side.
   So, by structural induction hypothesis applied to e_2, (3) is true.

8. $e \equiv e'\uparrow$
   $\mathbf{A}_1[\![e'\uparrow]\!]aenv \;=\; \emptyset$
   Recall that the typing discipline enforces that e be an r–value. Therefore, in the standard semantics, e cannot be equal to any address.

9. $e \equiv e_1(e_2)$

fix $n \geq 0$
Must show $\mathbf{A}[\![e_1(e_2)]\!]aenv$ is safe at level n for $\mathbf{M_e}[\![e_1(e_2)]\!]env\ store$
For $m \leq n$, a_i safe at level $n-1$ for v_i, $i = 1, \ldots, m$
Show

$$PAP_m(\mathbf{A}[\![e]\!]aenv, a_1, \ldots, a_m) \subseteq S \;\;\Rightarrow$$
$$AP_m(\mathbf{M_e}[\![e_1(e_2)]\!]env\ store, v_1, \ldots, v_m, store) \neq \mathbf{M_e}[\![y]\!]env\ store$$

By definition

$$\begin{aligned}
&PAP_m(\mathbf{A}[\![e_1(e_2)]\!]aenv, a_1, \ldots, a_m)\\
&= PAP_{m-1}((\mathbf{A}[\![e_1(e_2)]\!]aenv)_f\ a_1, a_2, \ldots, a_m)\\
&= PAP_{m-1}(((\mathbf{A}[\![e_1]\!]aenv)_f \mathbf{A}[\![e_2]\!]aenv)_f a_1, \ldots, a_m)\\
&= PAP_m((\mathbf{A}[\![e_1]\!]aenv)_f \mathbf{A}[\![e_2]\!]aenv, a_1, \ldots, a_m)\\
&= PAP_{m+1}(\mathbf{A}[\![e_1]\!]aenv, \mathbf{A}[\![e_2]\!]aenv, a_1, \ldots, a_m)
\end{aligned}$$

By structural induction on e_1 and e_2:

$$AP_{m+1}(\mathbf{M_e}[\![e_1]\!]env\ store, \mathbf{M_e}[\![e_2]\!]env\ store, v_1, \ldots, v_m, store) \neq \mathbf{M_e}[\![y]\!]env\ store$$

But

$$\begin{aligned}
&AP_{m+1}(\mathbf{M_e}[\![e_1]\!]env\ store, \mathbf{M_e}[\![e_2]\!]env\ store, v_1, \ldots, v_m, store)\\
&= AP_m((\mathbf{M_e}[\![e_1]\!]env\ store\ \mathbf{M_e}[\![e_2]\!]env\ store)_f, v_1, \ldots, v_m, store)\\
&\qquad \text{by definition of } AP\\
&= AP_m(\mathbf{M_e}[\![e_1(e_2)]\!]env\ store, v_1, \ldots, v_m, store) \text{ by definition of } \mathbf{M_e}
\end{aligned}$$

and therefore (3)

10. $e \equiv \lambda x : T.e'$

$\mathbf{A}_1[\![\lambda x : T.e']\!]aenv \;=\; \emptyset$
The value of e is an r–value, and the same argument holds.
This proves safety at level 0.
For safety at level n, note that
$AP_m(\mathbf{M_e}[\![e]\!]env\ store, e_1 \ldots, e_m, store)$
$\quad = AP_m((\mathbf{M_e}[\![e]\!]env\ store\ e_1)_2\ store, e_2, \ldots, e_m, store)$
$\quad = AP_{m-1}((\mathbf{M_e}[\![e']\!]\ env[u \leftarrow\ e_1]\ store, e_2, \ldots, e_m, store)$
And similarly for the alias expression:
$PAP_m(\mathbf{A}[\![e]\!]aenv, a_1 \ldots, a_m, store)$
$\quad = PAP_{m-1}((\mathbf{A}[\![e']\!]env[u \leftarrow\ a_1]store, a_2, \ldots, a_m, store)$
By structural induction on e', and because a_1 is safe for e_1, we have (3).

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] Arvind and R. E. Thomas. *I-Structures: An Efficient Data Type for Functional Languages.* Technical Report, MIT Laboratory for Computer Science, 1980.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] John Darlington and Mike Reeve. Alice: a multi–processor reduction machine for the parallel evaluation of functional languages. In *Symposium on Functional Languages and Computer Architecture*, 1981.

[9] Jack Dennis. An operational semantics for a language with early completion data structures. In *Symposium on Functional Languages and Computer Architecture*, 1981.

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

[11] R. M. Keller and M. R. Sleep. Applicative caching. In *Conference on Functional programming and Computer Architecture*, 1981.

[12] R.M. Keller. Divide and concer:data structuring for applicative multi–processing. In *Proceedings of the Lisp Conference*, 1980.

[13] R.M. Keller, S. Patil, and G. Lindstrom. A loosely–coupled applicative multi–processing system. In *AFIPS Proceedings*, 1979.

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

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

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

[17] 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.

[18] 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.

[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.