International Journal of Parallel Programming, Vol. 18, No. 1, 1989

Effect Analysis in Higher-Order Languages

Anne Neirynck,[1,2] Prakash Panangaden,[1,2] and Alan J. Demers[3]

Received July 1987; revised June 1989

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 typed language with higher-order constructs and imperative features. The heart of our scheme is a mechanism for estimating 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 in a compositional fashion reminiscent of abstract interpretation. We prove the soundness of our estimates with respect to the standard semantics of the language.

KEY WORDS: Effect analysis; alias analysis; abstract interpretation; parallel evaluation.

1. INTRODUCTION

In the last several years, there has been substantial interest in the development of programming languages appropriate for parallel architectures. The basic design conflict is that languages with simple semantics tend to use storage inefficiently, whereas programs written in languages that allow the programmer to use storage efficiently tend to be difficult to analyze. Thus, it is easy to detect potential parallelism in programs written in a pure functional language but, as data structures are modified incrementally, the entire data structure has to be recopied since there is no mechanism for expressing in-place updates. In a typical imperative language one can use

[1] Supported by National Science Foundation Grant DCR-8602072.
[2] Computer Science Department, Cornell University.
[3] Xerox Parc, Palo Alto.

0885-7458/89/0200-0001$06.00/0

828/18/1-1

assignment statements to perform in-place updating of data structures but this produces side-effects, which makes parallelism difficult to detect.

We develop a compile-time 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 tell whether, given two expressions, the evaluation of one of them affects the evaluation of the other. Thus, we allow programs to be written in a semantically elegant functional style but, avoid the overhead associated with replicating data unnecessarily. The present paper is an elaboration of work reported earlier.[1]

The language is a simply-typed higher-order functional language with a few imperative constructs. It has the essential control features of many modern typed languages such as ML.[2] Inferring runtime properties of *higher-order* languages involves confronting the problems associated with interprocedural flow analysis. Thus, the analyses we present subsume the analyses previously available in the literature.

Our analysis is relevant to detecting parallelism in a higher-order language that permits side-effects. We intend that our estimation scheme be used in a parallelizing compiler for a higher-order language. There are several efforts aimed at developing such parallelizing techniques for Fortran-like languages, for example the recent work of Burke and Cytron,[3] or Allen and Kennedy.[4] Alias analysis is an essential part of these schemes and has been the subject of study by itself.[5]

The *support* of an expression is the set of non local variables involved in the evaluation of that expression. Support sets turn out to be the key to determining whether an expression is pure, i.e., its evaluation is side-effect free, or in general, whether two expressions can be evaluated independently. The computation of support sets requires knowledge of potential aliases of some expressions.

Our approach is to define functions that assign to expressions estimates of their support and alias sets. This style of presentation for static analysis is often called *abstract interpretation* and was pioneered by the Cousots.[6,7] It allows one to describe a static analysis scheme in a fashion that makes clear the relation between the analysis scheme and the standard semantics. The phrase "abstract interpretation," however, has come to signify a particular format for presenting the relationship between the standard semantics and the abstract semantics, due to Mycroft and Nielson.[8,9] We do not use this particular format, though we clearly have the same view about how flow analysis relates to semantics.

Interference analysis is a critical aspect of compilation for parallel architectures and there has been much published on it.[10] In the last ten years, this work has focussed on the difficult aspects of such an analysis;

namely interprocedural interference analysis.[11–13] The work cited is for first-order languages only and relies on knowledge of the calling pattern of procedures. Our investigations apply to higher-order languages where one does not know the call-graph but are confined to the case where there are no pointers or arrays. A very thorough treatment of aliasing for arrays in first-order languages can be found in Ref. 3. There has also been recent work on interference in languages with dynamic data structures[14,15] that uses a different abstraction of the standard semantics.

The most important precursor of our work is a paper by Reynolds,[16] in which he defined syntactic restrictions to eliminate interference in a higher-order language with *call-by-name* parameter passing. The soundness of this scheme was proved by Tennent.[17] Our scheme for estimating interference is more precise, though more complicated. In the conclusion, we shall discuss this and other related work[18,19] more closely.

2. THE LANGUAGE AND ITS SEMANTICS

The language is a typed, functional language with a recursive definition facility. To this core we add three imperative features: static allocation performed with the **new** construct, an assignment statement written $e_1 \leftarrow e_2$, and a dereferencing construct $e\uparrow$. We use the word *variable* to stand for an identifier that denotes a storage location. To denote the value associated with the variable we use an explicit dereferencing construct, $x\uparrow$. Thus to increment x by 1 we write $x \leftarrow x\uparrow + 1$. An expression can evaluate to a variable rather than to the value associated with the variable as in **if** *true* **then** x **else** y, assuming that x and y are global variables. In such languages, one can easily write complex expressions that have nontrivial aliasing behavior. We are not advocating the use of our conventions in a practical programming language.

This is the abstract syntax for the language:

$$\begin{aligned} e = {} & c \mid x \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \mid \lambda x\colon t.e \mid e_1(e_2) \\ & \textbf{letrec } f_1 = \lambda x_1\colon t_1.e_1, \ldots, f_n = \lambda x_n\colon t_n.e_n \textbf{ in } e \\ & \textbf{new } x\textbf{:ref } t \textbf{ in } e \mid e_1 \leftarrow e_2 \mid e\uparrow \mid e_1; e_2 \end{aligned}$$

The type system is defined inductively as follows:

$$t = \mathit{int} \mid \mathit{bool} \mid \textbf{ref } \mathit{int} \mid \textbf{ref } \mathit{bool} \mid t_1 \rightarrow t_2$$

We do not have pointers or "structured types" or storable functions, although the first two of these are considered in Neirynck's thesis.[20] Notice that our type system has only finite types, i.e., no recursively defined types. The arrow construct is used for functional types, and the **ref** con-

struct for 1-values. The type of an identifier declared by a **new** has to be of the form **ref** t. There are no constraints on the syntax except the ones required by typechecking: the two branches of a conditional should be of the same type, functions are not storable items, and only one level of reference or dereference is allowed. These restrictions were chosen to simplify the analysis. Furthermore, because we have lexical scoping, 1-values cannot be exported outside their scope. This property can be detected statically by the methods developed here. Parameters are passed by value. When the parameter is a variable, as in $e(x)$, we have what is called "call-by-reference"; to get call-by-value parameter passing, we write $e(x\uparrow)$.

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 usual to find that the denotational definition of a programming language includes a model of the store and of storage management. We are assuming that when the denotational semantics makes reference to the store, an implementation of the language actually does manipulate the store. In other words, we are taking the denotational semantics as giving some aspects of the operational behavior. This goes against the spirit in which denotational semantics is ofter understood but in fact all the denotational definitions one may see in standard textbooks[21,22] do have such operational significance.

The denotational definition of a programming language is typically given in two stages. The first stage is a translation of the programming language into a semantic meta-language, usually some minor variation of the lambda-calculus. The second stage is an interpretation of this meta-language, usually in terms of domains. This is often not done explicitly, instead reference is made to the literature on semantics of the lambda-calculus. The point of significance for us is that the semantic meta-language also comes with an operational semantics, though this is rarely made explicit. It is the operational semantics of the meta-language which we are using as the basis of our inference. Our semantic meta-language is the typed lambda-calculus with certain constants to represent the store and store-manipulation functions. We shall express the store manipulations in terms of a semantic function, written $\mathbf{M}_s$, so that we can follow the compositional style of denotational definitions.

The store is modeled as a partial function with finite domain from locations to values.

$$Store = Loc \rightarrow Val$$

We need a notation to express the restriction of that function to a subset of the locations.

$$store|_L \text{ where } L \subseteq Loc$$

Given a store, *Allocate* returns a store and the address of the newly allowated space. The new store must coincide with the old store over all addresses except the one just allocated (Aciom Ia). We also require that the new address is not already allocated (Axiom Ib).

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

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 I

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

then

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

(b) $Eval(store, address) = \perp$
$Eval(newstore, address) = \underline{empty}$

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

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

Axiom II

$$\text{if } newstore = DeAllocate(store, address)$$

then

$$newstore|_{Loc-\{address\}} = store|_{Loc-\{address\}}$$

Axiom III

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

Because *l*-values are not stored, the other usual axioms about *Update* and *Eval* are not required.

The Semantics

Our denotational semantics is given in terms of domains and semantic functions from the expressions to the appropriate semantic domains. We

use a different domain for each type and there is a corresponding semantic function for each type as well. We will almost always omit the type labels as most clauses of the semantic functions are either the same at all types or are part of a semantic function that has a type that is clear from context. The basic semantic domains that we use in our denotational are *Bool*, *Nat* and *Loc*; these are the usual "flat" domains of truthvalues, integers and locations respectively.

We also need domains to represent high-order constructs. Evaluating a function application will return a value and, in general, modify the store. We use a pair construction to package together both these effects. Traditionally one writes down recursive domain equations to define a domain of functions. It is now becoming more customary to write down the semantics of a typed language in terms of a family of inductively defined domains rather than a single recursively defined domain. We adopt the latter approach.

The higher types look like $t \to t'$. We define

$$Val^{(t \to t')} = [(Val^t \times Store) \to (Val^{t'} \times Store)]$$

as the semantic domains associated with the higher types. The constructor $X \to Y$ is the domain of continuous functions from X to Y ordered pointwise. Whenever X and Y are Scott domains (consistently-complete ω-algebraic complete partial orders), so is $X \to Y$. Since our base domain is a Scott domain it follows that all the semantic domains are as well. We need the domain structure in order to give meaning to recursive constructs as fixed-points.

The denotational semantics is defined in terms of a pair of semantic functions called $\mathbf{M}_e$ and $\mathbf{M}_s$. [Strictly speaking, we really have a type-indexed family of pairs of semantic functions, however, the definitions are very similar at each type so we will not make this explicit and speak as if we had a single pair of semantic functions and a single semantic domain.] The first one gives the value of an expression while the second describes how the store is modified. This is just a notational variation on 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. We define environments as mappings of identifiers to elements of the appropriate domain. We use superscripts whenever we wish to make the types explicit. The arities of the semantic functions are:

$$\mathbf{M}_e{}^t : Exp^t \to Env \to Store \to Val^t$$

and

$$\mathbf{M}_s{}^t : Exp^t \to Env \to Store \to Store$$

The domain *Env* consists of functions from identifiers to values of the appropriate type. In order to be completely rigourous about the type of *Env* there needs to be a single domain of semantic values constructed by forming a disjoint sum of all the semantic domains; we shall, however, ignore this from now on. A full definition of the semantics is given in the appendix; we comment on a few clauses here to illustrate the notation and style of semantic definition.

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

where

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

This semantic clause shows 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 define the two other imperative features, assignment and dereferencing.

$$\mathbf{M_e}[\![e_1 \leftarrow e_2]\!]env\ store = value$$
$$\mathbf{M_s}[\![e_1 \leftarrow e_2]\!]env\ store = Update(store'', address, value)$$

where

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

$$\mathbf{M_e}[\![e\uparrow]\!]env\ store = Eval(store', address)$$
$$\mathbf{M_s}[\![e\uparrow]\!]env\ store = store'$$

where

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

3. THE PROBLEM OF ESTIMATING SIDE-EFFECTS

In this section, we discuss the problems associated with estimating side-effects in a language with higher-order functions; 1-valued expressions,

and local variables. We will argue that, to solve this problem, one needs to determine support sets and aliases. In other words, a simple approach, based on using information about whether the subexpressions are side-effect free, will not work.

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, and a pure expression may contain impure subexpressions. An example of this is **new** x:**ref** int **in** $x \leftarrow 3$. Unfortunately, the reverse situation can also arise: an impure expression may be built up out of pure subexpressions. Thus a syntactic scan for occurrence of imperative constructs is not enough to make an accurate estimate of purity.

A lambda-abstraction is always pure because any potential effects occur only during application. For instance, an expression like $\lambda x{:}int.\ y \leftarrow x$ is pure, but its application will cause a side-effect; if this expression is applied, even to a pure expression, the result expression is impure. The main problem with side-effect determination in a higher-order language is that we need to maintain enough information so that we can tell whether a particular application causes side-effects. To complicate matters further, this may depend on the arguments. For example in

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

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 potential side-effects actually do produce a side-effect. For example, if the global variables involved in the side-effect are all captured by their enclosing declarations, the resulting expression is pure. This is the case with

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

Thus the data-flow analyses for purity which suffice for first-order languages would be inadequate to the task at hand.[3,11–13] These analyses rely on knowing the call-graph of procedures and this is of course impossible with a higher-order language. Weihl[13] addresses the issue of functional parameters, but uses a language that is otherwise first-order.

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

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

Although the expressions are structurally very similar, the first one is pure whereas the second one is not. 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 some variable. We call the set of global variables that are affected by or whose values affect the evaluation of an expression the *support* of that expression. An expression is then pure when its support set is empty, and two expressions interfere if their support sets are not disjoint.

In more formal terms, we define a function to estimate the support set of an expression with respect to an environment. The support set is expressed as a set of "abstract" locations, a representation that is intended to avoid the problems associated with name conflicts in different scopes. In the case of recursive procedures, the environment is necessary to keep track of which incarnation of the procedure is intended when a variable name is used. We use the same device that is used in the standard semantics, i.e., an abstraction of the actual memory.

Before calculating the support of an expression, we may need the aliasing behavior of some of its subexpressions. 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. In order to simplify the presentation, we present these two estimates as if they were separate. It is essential, however, to realize that the two are being defined together.

4. AN ESTIMATION SCHEME FOR ALIASES

This section describes a function **A**, that estimates the set of possible aliases of an expression. It is defined in a fashion similar to the semantic functions and, thus, yields a compositional inference scheme. This sort of definition is often called abstract interpretation, but our presentation does not follow the "orthodox' format of Mycroft and Nielson.[8] The next section contains a similar definition for the computation of support sets.

The function **A** is similar to the semantic function $\mathbf{M}_e$ restricted to the computation of l-values. We define abstract alias domains associated with each member of the type hierarchy. Associated with each type t of we have a domain, Val^t_A. The types *Bool* and *Nat* are associated with the two-point domain consisting of $\perp$ and $\underline{rvalue}$. The alias domain for the type *Loc* is $\mathscr{P}(Loc_A)$, the powerset of the set of *abstract* locations ordered by inclusion. The set Loc_A is exactly like *Loc* but we view them as distinct in order to prevent one from confusing the bindings performed by **A** from the bindings performed by the semantic functions. Since the abstract domain is not finite

we need special care to ensure that **A** terminates on recursively defined procedures. The domain associated with $t \to t'$ is given by:

$$Val_A^{(t \to t')} = [Val_A^t \to (Store_A \to (Stack_A \to Val_A^{t'}))]$$

$Store_A$ is the set of abstract stores used in the computation of **A**. The role of $Stack_A$ is to prevent repeated allocation of variables by recursively defined functions. This is explained later. The partial orders on the domains Val_A^t are defined as described earlier for the base types and pointwise in the higher types.

Given an expression, an environment, a stack and a store, the function **A** maps the expression to an alias value. As with the standard semantics, the function **A** is really shorthand for a type-indexed family of functions.

$$\mathbf{A}: Exp \to Env_A \to Store_A \to Stack_A \to Val_A$$

The alias environment is a mapping from identifiers to alias values.

The store domain, $Store_A$, is greatly simplified compared to the standard semantic store: since the language does not allow the construction of dynamic data structures with pointers, the only function of the store required by the alias estimation is the ability to perform dynamic allocation of local variables. Thus the alias store is just a free list counter:

$$Store_A = Int$$

Our simplified allocation scheme allocates *at most* one location per instantiation of a variable, independently of its type. Furthermore, since variables cannot be exported outside their scope, deallocation occurs at exit of a block, and there is no need for the equivalent of $\mathbf{M}_s$ in the definition of **A**.

The alias stack, $Stack_A$, is used to detect whether the expression is evaluated in the context of a recursive call. If this is the case, the alias semantics will alias together all instantiations of a variable inside the recursive call.

$$Stack_A = Bool$$

The alias stack is represented as a Boolean value; *true* means that the context may include a recursive call. When a function is applied, its closure will contain an abstract alias stack as well. The value of $Stack_A$ is set to the disjunction of the two abstract stacks. Thus $Stack_A$ will be true if the expression being evaluated *may* be in a recursive context. This device assures us that the evaluation of **A** terminates. We could have used a finite domain for Loc_A, in which case all fixed-point computations are guaranteed to terminate. Unfortunately, if we did this, any recursive function that

performed local allocation would saturate Loc_A and the resulting information, though correct, would not be very useful. Thus **A**, evaluated on any given expression, essentially works on a finite domain but we cannot write down an a priori finite bound on the size of this domain that would suffice for all expressions.

Since we do not know at compile-time which branch of the conditional will be executed, we collect alias information from the two possible executions. Our estimate therefore associates sets of possible aliases with an expression. The domain *Loc* of the standard semantics becomes $\mathscr{P}(Loc_A)$. The alias domains are lattices so they have join operations defined on them. These are used in estimating the aliases of conditional expressions.

The clauses for the computation of aliases are given next. The aliasing behavior of global identifiers is provided by an alias environment called α, ζ is the alias store and θ is the alias stack.

Constants and identifiers.

$$\mathbf{A}[\![c]\!]\alpha\zeta\theta = \underline{rvalue}$$
$$\mathbf{A}[\![x]\!]\alpha\zeta\theta = \alpha(x)$$

We assume that there are no 1-valued or functional constants. The aliasing behavior of an identifier is determined by its binding in the alias environment.

Conditional and sequential evaluation.

$$\mathbf{A}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\alpha\zeta\theta = \mathbf{A}[\![e_2]\!]\alpha\zeta\theta \sqcup \mathbf{A}[\![e_3]\!]\alpha\zeta\theta$$
$$\mathbf{A}[\![e_1; e_2]\!]\alpha\zeta\theta = \mathbf{A}[\![e_2]\!]\alpha\zeta\theta$$

The least upper bound operation computes the possible alias as the union of the sets of possible aliases for each arm of the conditional. This clause introduces imprecision in the estimate. The value of $e_1; e_2$ is e_2, so its aliases are those of e_2.

Abstraction and application.

$$\mathbf{A}[\![\lambda x{:}t.e]\!]\alpha\zeta\theta = \lambda u.\lambda s.\lambda q.\mathbf{A}[\![e]\!]\alpha[x \leftarrow u]s(q \text{ or } \theta)$$
$$\mathbf{A}[\![e_1(e_2)]\!]\alpha\zeta\theta = (\mathbf{A}[\![e_1]\!]\alpha\zeta\theta)(\mathbf{A}[\![e_2]\!]\alpha\zeta\theta)\zeta\theta$$

The alias value of a lambda-term is a function of three arguments: the first one gives the alias value of the parameter, the second one is the alias store at the time of the call, the last one tells whether the expression is evaluated within a call to a recursive function.

Recursive functions.

$$\mathbf{A}[\![\textbf{letrec } f_1 = e_1 \cdots f_n = e_n \textbf{ in } e]\!]\alpha\zeta\theta = \mathbf{A}[\![e]\!]\alpha'\zeta\theta$$
where
$$\alpha' = lfp(\lambda\beta.\alpha[\ldots, f_i \leftarrow \mathbf{A}[\![e_i]\!]\beta\zeta\ true, \ldots])$$

In a **letrec** construct, the bindings are computed first and then the body is computed in the resulting environment, which is defined as a least fixed point. Recall that allocation is performed only once when we have a recursive content, thus fixed-point computations converge.

Imperative constructs.

$$\mathbf{A}[\![\textbf{new } x : \textbf{ref } t \textbf{ in } e]\!]\alpha\zeta\theta =$$
$$\begin{cases} \mathbf{A}[\![e]\!]\alpha\zeta\theta & \text{if } \theta = true \text{ and } x \in dom(\alpha) \\ \mathbf{A}[\![e]\!]\alpha[x \leftarrow \zeta](\zeta + 1)\theta & \text{otherwise} \end{cases}$$
$$\mathbf{A}[\![e_1 \leftarrow e_2]\!]\alpha\zeta\theta = \underline{rvalue}$$
$$\mathbf{A}[\![e\uparrow]\!]\alpha\zeta\theta = \underline{rvalue}$$

In a nonrecursive context, a new variable is aliased only to its location, and the free list counter is incremented by one for the scope of the declaration. In a recursive context, a variable is allocated once, and all further instantiations of that variable will be aliased to that same location. The result of an assignment or of a dereferencing is a *r*-value, and thus their values in the alias semantics are trivial.

The intuitive justification behind our scheme for estimating alias sets should be clear and the resemblance to the standard semantics is manifest. We prove a soundness theorem in a later section.

5. ESTIMATING THE SUPPORT SET

In this section we define a function **S** that provides an estimate of the support set of an expression. This estimate uses the function **A**. The two functions really are defined simultaneously; we are presenting them as separate only because we wish to discuss them separately. In the next section, we work out an example of the computation of **A** and **S** in which the computations are carried out simultaneously.

The definition of the support domains Val^t_S is similar to that of Val^t_A. As with aliases, we have a simple hierarchy of types starting with three ground types. Unlike with aliases, an expression at any higher type can have a nontrivial support set as having a potential additional side-effect when it is applied. Thus support set values need to be pairs. Specifically, we associate a domain Val^t_S with each type t. The domain associated with the

ground type is $\mathscr{P}(Loc_A) \times \underline{ground}$. For a type $t \to t'$ the associated domain is

$$\mathscr{P}(Loc_A) \times [Val_S^t \to Val_S^{t'}]$$

where the product of domains is not strict.

The arity of **S** at type t is

$$\mathbf{S}^t : Exp \to Env_S \to Val_S^t$$

where Env_S is the appropriate type of environments mapping identifiers to support values of the appropriate type.

Here, the definition of the function **S** is given. We use subscripts if we wish to denote only the first or second components of a pair in Val_S^t. Note, that when we use the function **A** we need the environment α; we assume that this environment results from a simultaneous computation of **A** for the same expression as the one for which the action of **S** is being defined. Thus **A** and **S** are really computed in parallel.

In order to simplify the notation, we use the infix symbol ':' to stand for application of elements of $Val_S^{(t \to t')}$ to elements of Val_S^t. Thus $s{:}s'$ means

$$\langle s_1 \cup s_1' \cup (s_2(s'))_1, (s_2(s'))_2 \rangle$$

To determine the effect of an application, we collect together all the locations accessed during the evaluation of the function (s_1), of its argument (s_1'), as well as of the additional locations that may be accessed during the application itself $(s_2(s'))_1$. The operator ':' associates to the left.

We comment only on the clauses corresponding to the imperative constructs, which differ significantly from those for the case of the alias computation. In all clauses, it is worth noticing how the computation of subexpressions contributes to the first component of the support. In all the definitions here, we suppress the type annotations.

Constants and identifiers.

$$\mathbf{S}[\![c]\!]\rho = \langle \emptyset, \underline{ground} \rangle$$
$$\mathbf{S}[\![x]\!]\rho = \rho(x)$$

Conditional and sequential evaluation.

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

Abstraction and application.

$$\mathbf{S}[\![\lambda x{:}t.e]\!]\rho = \langle \emptyset, \lambda u.\mathbf{S}[\![e]\!]\rho[x \leftarrow u] \rangle$$
$$\mathbf{S}[\![e_1(e_2)]\!]\rho = (\mathbf{S}[\![e_1]\!]\rho){:}(\mathbf{S}[\![e_2]\!]\rho)$$

This could also be written:

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

Recursion.

$$\mathbf{S}[\![\textbf{letrec } f_1 = e_1 \cdots f_n = e_n \textbf{ in } e]\!]\rho = \mathbf{S}[\![e]\!]\rho'$$
$$\text{where } \rho' = \mathit{lfp}(\lambda\gamma.\rho[\cdots, f_i \leftarrow \mathbf{S}[\![e_i]\!]\gamma \cdots])$$

Here we know that the iteration terminates because **A** does and the support cannot be larger than set of variables computed by **A**. Once again termination is assured even though the abstract domain is infinite.

Imperative Clauses.

If we are not in a recursive context

$$\mathbf{S}[\![\textbf{new } x{:}\textbf{ref } t \textbf{ in } e]\!]\rho =$$
$$\langle \mathbf{S}_1[\![e]\!]\rho[x \leftarrow \langle \emptyset, \underline{\mathit{ground}} \rangle] - \alpha(x), \mathbf{S}_2[\![e]\!]\rho[x \leftarrow \langle \emptyset, \underline{\mathit{ground}} \rangle] \rangle$$

If we are in a recursive context then,

$$\mathbf{S}[\![\textbf{new } x{:}\textbf{ref } t \textbf{ in } e]\!]\rho = \mathbf{S}_1[\![e]\!]\rho[x \leftarrow \langle \emptyset, \underline{\mathit{ground}} \rangle]$$

Inside the scope of its declaration, the support of a variable is just itself, or, more precisely, the location it denotes, which is given by α. On the other hand, the support of the entire block must not include the new variable since the scope of the variable ends when the block is exited, so we explicitly remove variables from the support computed for the body of the block within which they are declared. In removing it, we use the location computed by **A**. Since variables are allocated afresh when a block is entered there is no aliasing to worry about in using α to remove the location of x from the support. If we are in a recursive context then we do *not* remove $\alpha(x)$ because we cannot be sure that x refers only to itself since we do not know that a fresh allocation occurred. This is again a conservative approximation.

$$\mathbf{S}[\![e_1 \leftarrow e_2]\!]\rho = \langle \mathbf{S}_1[\![e_1]\!]\rho \cup \mathbf{S}_1[\![e_2]\!]\rho \cup \mathbf{A}[\![e_1]\!]\alpha\zeta\theta, \underline{\mathit{ground}} \rangle$$
$$\mathbf{S}[\![e\uparrow]\!]\rho = \langle \mathbf{S}_1[\![e]\!]\rho \cup \mathbf{A}[\![e]\!]\alpha\zeta\theta, \underline{\mathit{ground}} \rangle$$

The support of the explicitly imperative constructs is partly a function of the aliases of some subexpressions. In particular, the support of an assignment expression is the union of the supports of the two sides of the assignment and of all possible aliases of the left-hand side of the assignment. Similarly, the support of a dereferencing construct includes the aliases of the dereferenced expression.

If one wishes to distinguish between variables read and variables updated, the definition of **S** can be decomposed into two functions. The definitions of these functions are straightforward.

The correctness of our inference scheme is already plausible, because of the correspondence between the standard semantics and the compositional scheme used to estimate support sets and aliases.

6. EXAMPLES

Consider the following expression:

$$\begin{array}{l}\textbf{new } a\textbf{:ref } int \textbf{ in}\\ \quad \textbf{let } id = \lambda x\textbf{:ref } int.x \textbf{ in}\\ \quad\quad \textbf{let } eval = \lambda x\textbf{:ref } int.x\uparrow \textbf{ in } id(a); eval(a)\end{array}$$

The alias set of $id(a)$ should be the location of a, and its support set should be empty. On the other hand, the alias set of $eval(a)$ should be empty, since it is an r-value, but its support set should be the location of a.

As mentioned previously, the computation of aliases and support sets is performed simultaneously, so that the environments are updated in a consistent fashion. Here are the combined alias and support domains:

$$\begin{aligned}\mathbf{AS}^{t}&: Exp \to Env_{AS} \to Store_A \to Stack_A \to Val^{t}_{AS}\\ Val^{t}_{AS} &= Val^{0}_{A} \times Val^{0}_{S} \times FVal^{t}_{AS}\\ Val^{0}_{A} &= \{\underline{rvalue}\} + \mathscr{P}(Loc_A) + \{\underline{func}\}\\ Val^{0}_{S} &= \mathscr{P}(Loc_A)\\ FVal^{0}_{AS} &= \{\underline{ground}\}\\ FVal^{(t \to t')}_{AS} &= Val^{t}_{AS} \to Store_A \to Stack_A \to Val^{t'}_{AS}\end{aligned}$$

An alias&support value $(\mathbf{AS}[\![e]\!]\rho\zeta\theta)$ is now a triple, the first two components are the alias and support sets of the expression; $\underline{func}$ is the alias constant for all functional expressions. The third component is $\underline{ground}$ if e is a nonfunctional expression, and a function otherwise. The clause defining the alias and support value of an application is:

$$\mathbf{AS}[\![e_1(e_2)]\!]\rho'\zeta\theta = \langle alias, \mathbf{AS}_2[\![e_1]\!]\rho'\zeta\theta \cup \mathbf{AS}_2[\![e_2]\!]\rho'\zeta\theta \cup support, result\rangle$$

where

$$\langle alias, support, result \rangle = (\mathbf{AS}_3[\![e_1]\!]\rho'\zeta\theta)\langle \mathbf{AS}_1[\![e_2]\!]\rho'\zeta\theta, \emptyset, \mathbf{AS}_3[\![e_2]\!]\rho'\zeta\theta\rangle\zeta\theta$$

The other clauses needed for the examples are:

$$\mathbf{AS}[\![x]\!]\rho\zeta\theta = \rho(x)$$
$$\mathbf{AS}[\![\lambda x{:}t.e]\!]\rho\zeta\theta = \langle \underline{func}, \emptyset, \lambda u.\lambda s.\lambda q.\mathbf{AS}[\![e]\!]\rho[x \leftarrow u]s(q \text{ or } \theta)\rangle$$
$$\mathbf{AS}[\![\textbf{new } x{:}\textbf{ref } t \textbf{ in } e]\!]\rho\zeta\theta = \langle \mathbf{AS}_1[\![e]\!]\rho'\zeta'\theta, \mathbf{AS}_2[\![e]\!]\rho'\zeta'\theta - \{\zeta\}, \mathbf{AS}_3[\![e]\!]\rho'\zeta'\theta\rangle$$

where

$$\rho' = \rho[x \leftarrow \langle \{\zeta\}, \emptyset, \underline{ground}\rangle]$$
$$\zeta' = \zeta + 1$$

$$\mathbf{AS}[\![e_1 \leftarrow e_2]\!]\rho\zeta\theta =$$
$$\langle \underline{rvalue}, \mathbf{AS}_2[\![e_1]\!]\rho\zeta\theta \cup \mathbf{AS}_2[\![e_2]\!]\rho\zeta\theta \cup \mathbf{AS}_1[\![e_1]\!]\rho\zeta\theta, \underline{ground}\rangle$$

$$\mathbf{AS}[\![e\uparrow]\!]\rho\zeta\theta =$$
$$\langle \underline{rvalue}, \mathbf{AS}_2[\![e]\!]\rho\zeta\theta \cup \mathbf{AS}_1[\![e]\!]\rho\zeta\theta, \underline{ground}\rangle$$

$$\mathbf{AS}[\![e_1; e_2]\!]\rho\zeta\theta =$$
$$\langle \mathbf{AS}_1[\![e_2]\!]\rho\zeta\theta, \mathbf{AS}_2[\![e_1]\!]\rho\zeta\theta \cup \mathbf{AS}_2[\![e_2]\!]\rho\zeta\theta, \mathbf{AS}_3[\![e_2]\!]\rho\zeta\theta\rangle$$

Going back to our example, we now evaluate it in a nonrecursive content ($\theta = \mathit{false}$). The initial alias and support environment is empty. After the first line is processed, the environment contains the alias and support information associated with variable a, and the store contains one allocated address for a:

$$\rho' = [a \leftarrow \langle \{0\}, \emptyset, \underline{ground}\rangle]$$
$$\zeta' = \zeta + 1 = 1$$

The alias and support information for the identity function *id* is:

$$\mathbf{AS}[\![id]\!]\rho'\zeta'\theta = \langle \underline{func}, \emptyset, \lambda u\lambda s\lambda q.u\rangle$$

For the evaluation function *eval*, we have:

$$\mathbf{AS}[\![eval]\!]\rho'\zeta'\theta = \langle \underline{func}, \emptyset, \lambda u\lambda s\lambda q.\langle \underline{rvalue}, u_1, \underline{ground}\rangle\rangle$$

Let us call ρ' the alias and support environment updated with the values for *id* and *eval*. We are ready to compute the effect of each application:

$$\begin{aligned} \mathit{effect}(id(a)) &= (\mathbf{AS}_3[\![id]\!]\rho'\zeta'\theta)\langle \mathbf{AS}_1[\![a]\!]\rho'\zeta'\theta, \emptyset, \mathbf{AS}_3[\![a]\!]\rho'\zeta'\theta\rangle\zeta'\theta \\ &= (\lambda u, \lambda s\lambda q.u)(\langle \{0\}, \emptyset, \underline{ground}\rangle)\zeta'\theta \\ &= \langle \{0\}, \emptyset, \underline{ground}\rangle \end{aligned}$$

$$\begin{aligned} \mathit{effect}(\mathit{eval}(a)) &= (\mathbf{AS}_3[\![\mathit{eval}]\!]\,\rho'\zeta'\theta)\langle \mathbf{AS}_1[\![a]\!]\,\rho'\zeta'\theta, \emptyset, \mathbf{AS}_3[\![a]\!]\,\rho'\zeta'\theta\rangle\zeta'\theta \\ &= (\lambda u.\lambda s\lambda q.\langle \underline{\mathit{rvalue}}, u_1, \underline{\mathit{ground}}\rangle)\langle\{0\}, \emptyset, \underline{\mathit{ground}}\rangle\zeta'\theta \\ &= \langle \underline{\mathit{rvalue}}, \{0\}, \underline{\mathit{ground}}\rangle \end{aligned}$$

Since the support of an identifier is the empty set, we immediately obtain:

$$\begin{aligned} \mathbf{AS}[\![\mathit{id}(a)]\!]\,\rho'\theta &= \mathit{effect}(\mathit{id}(a)) = \langle\{0\}, \emptyset, \underline{\mathit{ground}}\rangle \\ \mathbf{AS}[\![\mathit{eval}(a)]\!]\,\rho'\theta &= \mathit{effect}(\mathit{eval}(a)) = \langle \underline{\mathit{rvalue}}, \{0\}, \underline{\mathit{ground}}\rangle \end{aligned}$$

For an example illustrating allocation of local variables, consider the function *mod_apply*, a modified version of *apply* with a local variable. It is used to define *mod_eval*, an abstraction of the dereferencing construct. The expression in the scope of variable a causes two calls to *mod_apply*, with two instantiations of local variable b, since we evaluate the expression in a nonrecursive context ($\theta = \mathit{false}$).

```
new a:ref int in
    let mod_apply = λg:ref int → int.λx:ref int.
            new b:ref int in b ← x↑; g(b)
    in let mod_eval = λy:ref int.mod_apply(λz:ref int.z↑)(y)
    in
       mod_apply(mod_eval)(a)
```

The effect of an application of *mod_apply* is summarized in the third line of the equality shown here. It says that the result is an *r*-value, that the support had included s_2, a local variable (the set difference is not simplified for illustration purposes), and now includes $(u_2)_1$, the alias of the second parameter, and the side-effect of applying the first argument to the local variable:

$$\begin{aligned} &\mathbf{AS}[\![\mathit{mod\_apply}]\!]\,\rho\zeta\theta = \\ &\quad\langle \underline{\mathit{func}}, \emptyset, \lambda u_1.\lambda s_1.\lambda q_1.\langle \underline{\mathit{func}}, \emptyset, \lambda u_2.\lambda s_2.\lambda q_2. \\ &\quad\quad\langle \underline{\mathit{rvalue}}, \{s_2\} - \{s_2\} \cup (u_2)_1 \cup ((u_1)_3\langle\{s_2\}, \emptyset, \underline{\mathit{ground}}\rangle(s_2+1))_2, \underline{\mathit{ground}}\rangle\rangle\rangle \end{aligned}$$

Applying *mod_eval* also results in an *r*-value, but the support includes the aliases of the argument (and had included s_3, the local variable created by the call to *mod_apply*).

$$\begin{aligned} &\mathbf{AS}[\![\mathit{mod\_eval}]\!]\,\rho\zeta\theta = \\ &\quad\langle \underline{\mathit{func}}, \emptyset, \lambda u_3.\lambda s_3.\lambda q_3.\langle \underline{\mathit{rvalue}}, \{s_3\} \cup (u_3)_1 - \{s_3\}, \underline{\mathit{ground}}\rangle\rangle \end{aligned}$$

When evaluating the main expression, the result is an *r*-value, and the

support set consists of a, but during execution also included the two instantiations of b, at addresses 1 and 2:

$$\mathbf{AS}[\![mod\_apply(mod\_eval)(a)]\!]\,\rho\zeta\theta = \langle \underline{rvalue}, ((\{\{0\} \cup \{1\}\}) - \{1\}) \cup \{2\}) - \{2\}, \underline{ground} \rangle$$

7. SOUNDNESS OF THE ALIAS ESTIMATE

In this section we state and prove a soundness theorem for the alias estimation. Informally, we show that the set of possible locations associated, by the function **A**, with an expression includes the locations associated with all possible aliases of the given expression. If there were no imprecision in the estimates, **A** would associate a single location with each 1-valued expression; the aliases of a given expression could then be determined by finding all expressions mapped to the same location by **A**.

The heart of the proof is a joint induction on the structure of terms in the language and on their types. In the proof, the induction on the type structure is nested inside an overall structural induction. The *letrec* construct requires a fixed-point induction as well. The estimation scheme would not converge if we applied it to an untyped language.

We need a convenient notation for expressing application of higher-order expressions in the standard semantics. Since the language is not purely functional, every application involves stores being passed around appropriately. We define a notation that handles this automatically. We focus on denotable values, i.e., values in the semantic domain that are the denotations of expressions in the language, in a given environment.

We recall the definitions of the semantic domains from Section 2.

Definition 7.1. For each type expression of higher type $t \to t'$, we define the domain $Val^{(t \to t')}$ as

$$Val^{(t \to t')} = Store \to (Val^{t} \to (Val^{t'} \times Store)).$$

These domains are the domains of *values*. We define D^t as

$$D^t = Store \to (Val^t \times Store)$$

These are the domains of *denotable values*.

A denotable value, thus, has the form $\lambda\, store.\langle v, store' \rangle$ where v is a value and $store'$ is a store. We write

$$\mathbf{M}[\![e]\!]\, env = \lambda\, store.\langle \mathbf{M}_e[\![e]\!] env\ store, \mathbf{M}_s[\![e]\!] env\ store \rangle$$

pairing together the two semantic functions. We also need a notation for applying two denotable values; the following definition encapsulates the effects on the store of evaluating the expression in the function position and the subsequent effect of evaluating the expression in the argument position.

Definition 7.2. Suppose that p is an element of $D^{(t \to t')}$ and q is an element of D^t. We define

$$p * q \equiv \lambda s.[(p(s))_1(q((p(s))_2))_2](q((p(s))_2))_1$$

The expression $p * q$ belongs to the domain $D^{t'}$.

A more readable definition uses "let" notation

$$p * q = \lambda s.\text{let } \langle f, s' \rangle = p(s) \text{ in}$$
$$\text{let } \langle v, s'' \rangle = q(s') \text{ in } f(s'')(v)$$

This definition allow us to conceal the details of store manipulation when we apply denotable values to other denotable values. The standard semantics of application are written as

$$\mathbf{M}[\![e_1(e_2)]\!]env = \mathbf{M}[\![e_1]\!]env * \mathbf{M}[\![e_2]\!]env$$

We say that an alias or support domain and a standard domain *correspond* if they are of the same type.

As is fairly standard, we use the term *satisfy* for the correctness property. The following definition says that a value from the alias domains, henceforth called an *alias value*, safely estimates the possible aliases if the set of identifiers associated with any location is included in the estimate of the set of identifiers that are aliased together.

Definition 7.3. For any a, an alias value of ground type, a is said to be *safe* for value d from the corresponding domain, with respect to the pair of environments $\langle \alpha, env \rangle$, if, for any store s,

$$(a \neq \underline{rvalue}) \Rightarrow env^{-1}(d(s)_1) \subseteq \{i \in Id \mid \alpha(i) \cap a \neq \emptyset\}$$

and $(a = \underline{rvalue}) \Rightarrow (d(s)_1 \notin Loc)$. If the type of a is $t \to t'$, then a is said to be safe for value d of the corresponding domain, if $\forall a_1, \ldots, a_n$ of appropriate type such that $aa_1\zeta\theta \cdots a_n\zeta\theta$ is of ground type and for $d_1, \ldots, d_n$ of corresponding domains, with each a_i safe for d_i, we have that $aa_1\zeta\theta \cdots a_n\zeta\theta$ is safe for $d * d_1 * \cdots d_n$; where ζ is an arbitrary abstract store and θ is an abstract stack.

The definition of safety given here relates alias values with the denotable semantic values; we have not mentioned the syntactic entities of

the programming language as yet. The soundness theorem will take the form that, for a given expression, the value computed by the function **A** is safe for the value it denotes in the standard semantics. In order to state the theorem, however, we need to have some correspondence between the alias environments and the standard semantics environments. The following definition is essentially the pointwise extension of the safety property.

Definition 7.4. An alias environment α and an environment *env* are said to *correspond* if, (*i*) any identifier in the domain of *env* is also in the domain of α and (*ii*) $\forall x \in dom(env)\ \alpha(x)$ is safe for $\lambda s.\langle env(x), s\rangle$.

The next lemma states that the safety relation is preserved by taking least upper bounds. From this it follows at once that the relation of corresponding environments is ω-inclusive. Rather than showing this for the case of least upper bounds of directed sets we shall show it for least upper bounds of chains, which is simpler and equivalent. We begin with a simple lemma that states that safety is preserved by the ordering in the lattices of alias values. In the following proofs we will often apply an alias value at a higher type to values at lower type. Each one of these applications involves an abstract store and abstract stack as well. We will omit these since they clutter up the notation. Thus, we will write expressions like $aa_1 \cdots a_n$ when we mean $aa_1\zeta\theta \cdots a_n\zeta\theta$.

Lemma 7.5. If $a \sqsubseteq a'$ are alias expressions of a given type, d is a value of a corresponding type, and a is safe for d in $\langle \alpha, env\rangle$, then a' is also safe for d in $\langle \alpha, env\rangle$.

Proof. At the ground type, $a \sqsubseteq a'$ is just $a \subseteq a'$. For the ground type then, it is clear that if a is safe for d and $a \sqsubseteq a'$, then a' is also safe for d. At higher types, suppose that $a \sqsubseteq a'$, and that a is safe for d in $\langle \alpha, env\rangle$. Let $a_1,\ldots,a_n$ be chosen such that $aa_1 \cdots a_n$ is of ground type (by which we also imply that the types are such that the expression is well typed). Suppose that $d_1,\ldots,d_n$ are also chosen so that $d * d_1 * \cdots * d_n$ is of ground type and for each i, a_i is safe for d_i in $\langle \alpha, env\rangle$. By the definition of safety, $aa_1 \cdots a_n$ is safe for $d * d_1 * \cdots d_n$. Since $a \sqsubseteq a'$, and the order is defined pointwise in the functional types, we have $(aa_1 \cdots a_n) \sqsubseteq (a'a_1 \cdot a_n)$, which means $(aa_1 \cdots a_n) \subseteq (a'a_1 \cdots a_n)$. Thus $a'a_1 \cdots a_n$ is safe for $d * d_1 * \cdots * d_n$ also, but, since the a_i were arbitrary safe alias expressions, this means that a' is safe for d as well. ∎

The following lemma is not very profound but it is essential in order to show that fixed-point inductions are justified. The crux of the argument is that chains are finite, so if there is a failure of safety with the least upper

bound of a chain then this failure must have occurred at some finite stage in the chain.

Lemma 7.6. Suppose that $\langle \alpha, env \rangle$ are a pair of environments. Suppose that $a_1,..., a_i,...$ is a chain of alias values and that $d_1,..., d_i,...$ is a chain of values with each a_i safe for the corresponding d_i in $\langle \alpha, env \rangle$. Let a be the least upper bound of the a_i and let d be the least upper bound of the d_i. Then a is safe for d in $\langle \alpha, env \rangle$.

Proof. Throughout this proof we shall say "x is safe for y" without mentioning the pair $\langle \alpha, env \rangle$, which is always the pair we intend to refer to. First, we note that a is safe for all the d_i because of the previous lemma. Second, all the alias lattices are finite so the chain of alias value reaches its least upper bound at some finite point in the chain, call this value a_n. Thus a_n is safe for all the d_i.

Suppose that a_n is not safe for d. This means that for some choice of $k_1,..., k_m$ and some store s, such that $d * k_1 * \cdots * k_m(s)$ is of ground type (and denotes a location), there is some identifier, say **x**, bound by *env* to $d * k_1 \cdots * k_m(s)$ such that α does not associate **x** with a set of possible locations that intersects $ab_1 \cdots b_m$, where the b_i are alias values selected to be safe for the corresponding k_i. Consider the chain $\{d_i * k_1 \cdots * k_m(s) \mid i \in Int\}$. Elements in this chain can only take on one of two values, a specific location, say l, or $\perp$. If it always stays at $\perp$ then so does d and any alias value is safe for it. If it reaches some nonbottom value, then it does so at some finite stage in the chain, say at step j. This means that a_n cannot be safe for d_j, a contradiction since a_n was supposed to be safe for all the d_i. ∎

The soundness of our abstract interpretation for aliases is stated formally in the following theorem. We qualify the statement of soundness with the proviso that the evaluation of the expression actually terminates in the standard semantics and the alias estimate does not cause the finite locations to overflow. This does *not* mean that the evaluation of **A** may not terminate; alias computations always terminate, but soundness is only guaranteed when the evaluation of the actual expression terminates as well. This is when soundness matters, because one would not be interested in parallelizing nonterminating computations.

Theorem 7.7. $\forall \alpha$, *env*, pairs of corresponding environments, and $\forall$ *store*, ζ we have, if

$$\mathbf{M}_e[\![e]\!] env\ store \neq \perp$$

and the abstract store does not overflow then

$$\mathbf{A}[\![e]\!]\alpha\zeta\theta \text{ safe for } \mathbf{M}[\![e]\!]\alpha\zeta\theta$$

Proof. The proof proceeds by structural induction on e. We assume a fixed store, *store*, and alias store, ζ. We suppress the abstract stack except in the case of the allocation of local variables since it has no effect otherwise. The reader should consult the appendix for details of the standard semantics.

$\underline{e \equiv x}$

$\mathbf{A}[\![e]\!]\alpha\zeta\theta = \alpha(x)$ by the definition of $\mathbf{A}$, and $\mathbf{M}_e[\![e]\!]env\ store = env(x)$ by definition of $\mathbf{M}_e$. Since α and *env* are corresponding environments, by hypothesis, $\alpha(x)$ is safe for $\lambda s.\langle env(x), s\rangle$.

$\underline{e \equiv \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3}$

We do this case by induction on the type structure. Suppose that e, e_2, e_3 are all of ground type. The case when e is an r-value is trivial since there is no interesting aliasing behavior . Thus assume that e, e_2, e_3 are 1-valued expressions. By the definition of $\mathbf{A}$, $\mathbf{A}[\![e]\!]\alpha\zeta\theta$ is $\mathbf{A}[\![e_2]\!]\alpha\zeta\theta \cup \mathbf{A}[\![e_3]\!]\alpha\zeta\theta$. By the definition of the safety property, if a is safe for a value v in a pair of environments, then any superset of a is also safe for that value in the same pair of environments. This means that $\mathbf{A}_1[\![e]\!]\alpha\zeta\theta$ is safe for $\mathbf{M}[\![e_2]\!]env$ as well as for $\mathbf{M}[\![e_3]\!]env$ in the pair of environments $\langle\alpha, env\rangle$. In the standard semantics, the meaning of e is either the meaning of e_2 or the meaning of e_3; in either case, the alias estimate is safe.

Suppose now that e, e_2, e_3 are of type $t \to t'$. In the rest of this paragraph, whenever we say "x safe for y" we mean "x safe for y in the pair of environments $\langle\alpha, env\rangle$. Now let $d_4, \ldots, d_n$ be expressions such that the application $\mathbf{M}[\![e]\!]env * d_4 \cdots * d_n$ is well-typed and of ground type. Let the alias expression $\mathbf{A}[\![e]\!]\alpha\zeta\theta$ be a, and let a_i stand for alias values that are safe for the respective d_i. By the structural induction hypothesis, and the definition of safety at the higher types, $a_2 a_4 \cdots a_n$ is safe for $\mathbf{M}[\![e_2]\!]env * d_4 \cdots * d_n$ and similarly with e_3 and a_3 in place of e_2 and a_2 respectively. The definition of $\mathbf{A}$, at higher types, says that $a = a_2 \sqcup a_3$. The definition of $\sqcup$ says that the order is defined pointwise from the order at the ground type, which is ordinary inclusion. In other words,

$$a a_4 \cdots a_n = (a_2 a_4 \cdots a_n) \cup (a_3 a_4 \cdots a_n)$$

This means, as before, that a is safe for $\mathbf{M}[\![e_2]\!]env$ as well as for $\mathbf{M}[\![e_3]\!]env$. Thus a is safe for $\mathbf{M}[\![e]\!]env$.

This argument illustrates how one carries the proof of safety from the

ground type to the higher type. We will not repeat it in the later cases but will just give the argument at the ground type, which is where it is interesting.

$e \equiv \textbf{letrec } f_1 = e_1 \cdots f_n = e_n \textbf{ in } e'$

We need to show that the estimated alias value is safe, we do this by fixed point induction.

— base case:

$$env_0 = env[\ldots, f_i \leftarrow \perp_D, \ldots]$$

and

$$aenv_0 = aenv[\ldots, f_i \leftarrow \perp_{D_A}, \ldots]$$

are corresponding environments. Since any alias value is a safe estimate for $\perp_D$.
By structural induction on e', the theorem is true when applied to e', env_0 and $aenv_0$.
— induction step:
We do not perform allocation in the abstract semantics if the stack is *true*, only if it is *false*. Assume the latter. 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 structural induction on e_i, $aenv_n$ and env_n are corresponding environments. By fixed point induction, $aenv'$ and env' are corresponding environments, and the result follows by structural induction. Suppose the stack is *true*, then $aenv_n$ and $aenv_{n-1}$ are the same. env_n results from performing new allocations, it will be the case that any identifiers possibly aliased in env_n will have been possibly aliased in env_{n-1} already. Since $aenv_{n-1}$ makes a safe estimate for every identifier mentioned in env_{n-1} it will make a safe estimate for every identifier in env_n as well. Thus $aenv_n$ will correspond to env_n. The rest of the argument is the same as shown before.

$e \equiv \lambda x{:}t.e'$

Here we use the definition of safety at higher types. First we need an observation. Suppose that a is of type $t \to s$ and that $d \in D'^{\to s}$. Let a' be safe for d' for the environment pair $\langle \alpha, env \rangle$, and assume further that a' is of type t while $d' \in D'$. Then aa' is *always* safe for $d * d'$ in the environment pair $\langle \alpha, env \rangle$ *iff* a is safe for d. Now consider a pair $\langle \alpha, env \rangle$ of corre-

sponding environments. We must show that $\mathbf{A}[\![\lambda x{:}t.e']\!]\alpha\zeta\theta$ is safe for $\mathbf{M}[\![\lambda x{:}t.e']\!]env$ in the pair $\langle \alpha, env\rangle$. Suppose that a' is of type t and that $d' \in D^t$ and that a' is safe for d'.

By the definition of **A** and of the standard semantics we have the following equations

$$(\mathbf{A}[\![\lambda x{:}t.e']\!]\alpha\zeta\theta)a'\zeta\theta = \mathbf{A}[\![e']\!]\alpha[x \leftarrow a']\zeta\theta$$

and

$$(\mathbf{M}[\![\lambda x{:}t.e']\!]env) * d' = \lambda s.\mathbf{M}[\![e']\!]env[x \leftarrow (d'(s))_1](d'(s))_2$$

Since a' is safe for d' and $\langle \alpha, env\rangle$ are corresponding environments, we know that $\alpha[x \leftarrow a']$ and $env[x \leftarrow (d'(s))_1]$ are also corresponding environments. By the structural induction hypothesis, therefore, $\mathbf{A}[\![e']\!]\alpha[x \mapsto a']\zeta\theta$ is safe for $\mathbf{M}[\![e']\!]env[x \leftarrow (d'(s))_1](d'(s))_2$. Thus, we have that $(\mathbf{A}[\![\lambda x{:}t.e']\!]\alpha\zeta\theta)a'\zeta\theta$ is safe for $(\mathbf{M}[\![\lambda x{:}t.e']\!]env) * d'$ for any choice of ζ and with any pair a' and d' with the a' safe for d'. In view of this observation, this means that $\mathbf{A}[\![e]\!]\alpha\zeta\theta$ is safe for $\mathbf{M}[\![e]\!]env$.

$\underline{e \equiv e_1(e_2)}$

The proof of this case is trivial given to the previous observation. The structural induction hypothesis says that, given a pair $\langle \alpha, env\rangle$ of corresponding environments, $\mathbf{A}[\![e_1]\!]\alpha\zeta\theta$ is safe for $\mathbf{M}_e[\![e_1]\!]env\ store$ and $\mathbf{A}[\![e_2]\!]\alpha\zeta\theta$ is safe for $\mathbf{M}_e[\![e_2]\!]env\ store'$ in the pair of environments. The definition of safety, and our observation, allow us to conclude immediately that

$$(\mathbf{A}[\![e_1]\!]\alpha\zeta\theta)(\mathbf{A}[\![e_2]\!]\alpha\zeta\theta)\zeta\theta$$

is safe for

$$(\mathbf{M}[\![e_1]\!]env) * (\mathbf{M}[\![e_2]\!]env)$$

The first term is $\mathbf{A}[\![e_1(e_2)]\!]\alpha\zeta\theta$ and the second is $\mathbf{M}[\![e_1(e_2)]\!]env$.

$\underline{e \equiv \textbf{new}\ x{:}\textbf{ref}\ t\ \textbf{in}\ e_1}$

In the absence of a recursive context it is sufficient to show

$$\alpha[x \leftarrow \{l\}]$$

is safe for $env[x \leftarrow address]$. Since α and env are corresponding environments, all that is left to prove is that $\alpha(x) = \{l\}$ is safe for $\lambda s.\langle address, s\rangle$. Since x is not of a functional type, only the ground case need be con-

sidered. Now, from the store axiom I(b), we know that $env^{-1}(address)$ is just $\{x\}$. In order to satisfy the safety property we need to check that $\alpha[x \leftarrow \{l\}](x) \cap \{l\} \neq \emptyset$ which is immediate since $\alpha[x \leftarrow \{l\}](x) = \{l\}$.

The case when we have a recursive enviroment is trivial because we are aliasing more variables together. Suppose that we have a recursive context then $\alpha(x)$ will denote the same abstract location, say l, for *all* the instances of x in the standard semantics. In particular $env^{-1}(address)$ is included in $\alpha^{-1}(x)$.

$\underline{e \equiv e_1 \leftarrow e_2}$

In both the standard semantics and the aliasing semantics, the meaning of an assignment is that of its right-hand side. So, by the structural induction hypothesis applied to e_2, the safety property remains true.

$\underline{e \equiv e'\uparrow}$

$$\mathbf{A}_1[\![e'\uparrow]\!]\alpha\zeta\theta = \underline{rvalue}$$

Recall that the typing discipline enforces that e be an r-value. Therefore, in the standard semantics, e cannot be equal to any address.

$\underline{e \equiv e_1; e_2}$

By routine structural induction. ∎

8. SOUNDNESS THEOREM FOR SUPPORT SETS

In this section we state and prove a soundness theorem for our support set estimate. This theorem says that the evaluation of an expression depends solely on the values of variables in its support set and, in addition, does not affect the store outside that set. As for aliases, soundness can only be guaranteed when the evaluation of the original expression terminates. The computation of **S** always terminates.

We will need to refer to portions of the store. The notation $store|_L$ means the portion of the store restricted to L, where L is a set of locations. Since stores are functions from locations to values, we are using the usual notation for restricting functions. The notation $\bar{L}$ means the complement of L with respect to Loc.

As for aliases, we define safety at ground types, and then extend the definition to the higher types.

Definition 8.1. Suppose that s is a support value of ground type that and m is a denotable value of the same type. Let $\langle \alpha, env \rangle$ be a pair of environments and let

$$S = env(\{x \in Id \mid \alpha(x) \cap s_1 \neq \emptyset\})$$

We say that s is *safe* for m with respect to the environments $\langle \alpha, env \rangle$ if

$$\forall\, store, store'.store|_S = store'|_S$$
$$\text{(i)} \quad (m(store))_1 = (m(store'))_1$$
$$\text{(ii)} \quad (m(\text{store}))_2|_S = (m(store'))_2|_S$$
$$\text{(iii)} \quad (m(store))_2|_{\bar{S}} = store|_{\bar{S}}$$

The set S has a complicated definition; it stands for the set of actual locations that s_1 estimates as being accessed by m. In order to go from the "virtual" locations represented in the abstract "semantics" to the real locations, we need the alias environment α to take us to a set of identifiers and the actual environment *env* to take us to actual locations. We extend this to higher types as follows. We use the ':' notation introduction in Section 5.

Definition 8.2. Suppose that $s \in Val_S^{s \to t}$ and m is a denotable value of the same type. Let $\langle \rho, \alpha, env \rangle$ be a triple of environments. Let S be defined as before. We say that s is *safe* for m with respect to $\langle \rho, \alpha, env \rangle$ if, s is safe for m in exactly the sense shown previously and, in addition, if for any sequence of values, $s_1, \ldots, s_n$ from Val_S such that $s{:}s_1{:}\cdots{:}s_n$ is of ground type and such that $m_1, \ldots, m_n$ are denotable values of appropriate types with each s_i safe for e_i with respect to $\langle \rho, \alpha, env \rangle$ we have that $s{:}s_1 \cdots{:} s_n$ is safe for $m * m_1 * \cdots * m_n$ with respect to $\langle \rho, \alpha, env \rangle$.

Now we define corresponding environments as follows.

Definition 8.3. A triple of environments $\langle \rho, \alpha, env \rangle$ are said to correspond if $\forall x \in dom(env).\rho(x)$ is safe for $\lambda s.\langle env(x), s \rangle$ with respect to the given triple.

The proof that the relation is ω-inclusive follows in the same way as for aliases.

Theorem 8.4. Suppose that $\langle \rho, \alpha, env \rangle$ are a triple of corresponding environments. For any expression e, $\mathbf{S}[\![e]\!]\rho$ is safe for $\mathbf{M}[\![e]\!]env$, with respect to $\langle \rho, \alpha, env \rangle$ provided that $\mathbf{M}[\![e]\!]env$ is not $\perp$.

Proof. The proof of the theorem proceeds by structural induction on e.

$\underline{e \equiv x}$

This follows immediately from the definition of corresponding environments.

$\underline{e \equiv e_1 ; e_2}$

Recall that

$$\mathbf{M}[\![e_1 ; e_2]\!] env = \lambda store.\, \langle \mathbf{M}_e[\![e_2]\!] env(\mathbf{M}_e[\![e_1]\!] env\ store), \mathbf{M}_s[\![e_2]\!] env(\mathbf{M}_s[\![e_1]\!] env\ store) \rangle$$

and that

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

In this case

$$S = env(\{x \mid \alpha(x) \cap (\mathbf{S}_1[\![e_1]\!]\rho \cup \mathbf{S}_1[\![e_2]\!]\rho) \neq \emptyset\})$$

We define

$$S_1 = env(\{x \mid \alpha(x) \cap \mathbf{S}_1[\![e_1]\!]\rho \neq \emptyset\})$$

and

$$S_2 = env(\{x \mid \alpha(x) \cap \mathbf{S}_1[\![e_2]\!]\rho \neq \emptyset\})$$

Let *store*, *store'* be stores that agree on S. Then, since S_1 and S_2 are contained in S, they certainly agree on S_1 and S_2. Thus the properties (i), (ii) and (iii) follow immediately by the structural induction hypothesis. Now we need to consider higher types. Let $s^{(1)} \cdots s^{(k)}$ be a sequence of support values and $m^{(1)} \cdots m^{(k)}$ a sequence of denotable values with each $s^{(i)}$ safe for $m^{(i)}$ with respect to the triple of environments. Consider the following calculation:

$$\begin{aligned}
&\mathbf{S}[\![e_1 ; e_2]\!]\rho : s^{(1)} : \cdots : s^{(k)} \\
&\quad = \langle (\mathbf{S}_1[\![e_1]\!]\rho) \cup (\mathbf{S}_1[\![e_2]\!]\rho), \mathbf{S}_2[\![e_2]\!]\rho \rangle : s^{(1)} : \cdots : s^{(k)} \\
&\quad = \langle (\mathbf{S}_1[\![e_1]\!]\rho) \cup (\mathbf{S}_1[\![e_2]\!]\rho) \cup s_1^{(1)} \cup ((\mathbf{S}_2[\![e_2]\!]\rho)(s^{(1)}))_1, \\
&\qquad ((\mathbf{S}_2[\![e_2]\!]\rho)(s^{(1)}))_2 \rangle : s^{(2)} : \cdots : s^{(k)} \\
&\quad = \langle \mathbf{S}_1[\![e_1]\!]\rho \cup (\mathbf{S}[\![e_2]\!]\rho : s^{(1)} : \cdots : s^{(k)})_1, (\mathbf{S}[\![e_2]\!]\rho : s^{(1)} : \cdots : s^{(k)})_2 \rangle
\end{aligned}$$

Now we use the structural induction hypothesis to conclude that each half of the last union is safe for the corresponding denotable value.

$\underline{e \equiv \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3}$

This case is also a routine structural induction, very similar to the preceding case.

$\underline{e \equiv \textbf{letrec}\ f_1 = e_1 \cdots f_n = e_n\ \textbf{in}\ e'}$

We have already shown, in the proof of the last theorem, that α, defined as a least fixed-point, gives rise to a pair of corresponding environments.

Therefore, in this case, the α environment can be ignored.
By fixed-point induction:
— base case:

$$env_0 = env[\ldots, f_i \leftarrow \perp_D, \ldots]$$

and

$$\rho_0 = \rho[\ldots, f_i \leftarrow \perp_{Vals}, \ldots]$$

are corresponding environments since *env* and ρ were. By the structural induction hypothesis, $\mathbf{S}[\![e']\!]\rho_0$ is safe for $\mathbf{M}[\![e']\!]env$ with respect to $\langle \rho_0, env_0 \rangle$.
— induction step:
Assume ρ_{n-1} and env_{n-1} are corresponding environments.

$$env_n = env_{n-1}[\ldots, f_i \leftarrow \mathbf{M}_e[\![e_i]\!]env_{n-1}\ store, \ldots]$$

and

$$\rho_n = \rho_{n-1}[\ldots, f_i \leftarrow \mathbf{S}[\![e_i]\!]\rho_{n-1}, \ldots]$$

Then by the structural induction hypothesis applied to e_i, ρ_n and env_n are corresponding environments. By fixed-point induction, ρ' and *env'* are corresponding environments. By structural induction on e', the safety properties hold. If we are in a recursive context then the estimate is even more conservative and the soundness is immediate.

$\underline{e \equiv \lambda x{:}t.e'}$

In this case we have an expression that is not of ground type. Informally, the effect of evaluating a lambda-expression is to build a closure, the store is not affected. Showing (*i*), (*ii*) and (*iii*) is trivial, since $\mathbf{M}_s[\![\lambda x{:}t.e']\!]env\ store = store$ according to the standard semantics. To show safety at high type, we use the definition of the semantics.

$$\mathbf{M}_e[\![\lambda x : t.e']\!]env\ store = \lambda s.\lambda v.\mathbf{M}[\![e']\!]env[x \leftarrow v]$$

For the function **S** we have the following equation

$$\mathbf{S}[\![\lambda x{:}t.e']\!]\rho = \langle \emptyset, \lambda v.\mathbf{S}[\![e']\!]\ \rho[x \leftarrow v] \rangle$$

Now we note that the modified environments in which the body e' is evaluated, in the semantics and the calculation of **S**, are corresponding environments. Thus by the structural induction hypothesis applied to the subterm e' the safety properties hold.

$\underline{e \equiv e_1(e_2)}$

We must show $\mathbf{S}[\![e_1(e_2)]\!]\rho$ is safe for $\mathbf{M}[\![e_1(e_2)]\!]env$. First, we verify the three conditions in Definition 8.1. Condition (i) states that, if *store* and *store'* agree on S then, $(\mathbf{M}[\![e_1(e_2)]\!]env\ store)_1 = (\mathbf{M}[\![e_1(e_2)]\!]env\ store')_1$. Using the definition of the semantics, we can write this equation as

$$(f\,h\ store_2) = (f'\,h'\ store_2')$$

where $f = \mathbf{M}_e[\![e_1]\!]env\ store$, $h = \mathbf{M}_e[\![e_2]\!]env\ store_1$, $store_1 = \mathbf{M}_s[\![e_1]\!]env\ store$, $store_2 = \mathbf{M}_s[\![e_2]\!]env\ store_1$ and the primed versions are the same with *store'* used instead of *store*. The set S in this case is

$$S = env(\{x \mid \alpha(x) \cap (\mathbf{S}_1[\![e_1]\!]\rho \cup \mathbf{S}_1[\![e_2]\!]\rho \cup (\mathbf{S}_2[\![e_1]\!]\rho\mathbf{S}[\![e_2]\!]\rho)_1) \neq \emptyset\})$$

Let

$$S_1 = env(\{x \mid \alpha(x) \cap \mathbf{S}_1[\![e_1]\!]\rho \neq \emptyset\})$$
$$S_2 = env(\{x \mid \alpha(x) \cap \mathbf{S}_1[\![e_2]\!]\rho \neq \emptyset\})$$

and

$$S_3 = env(\{x \mid \alpha(x) \cap ((\mathbf{S}_2[\![e_1]\!]\rho)(\mathbf{S}[\![e_2]\!]\rho))_1 \neq \emptyset\})$$

From the structural induction hypothesis, and from the fact that $S_1, S_2, S_3 \subseteq S$ we have $f = f'$, $h = h'$ and $store_2|_S = store_2'|_S$. Thus condition (*i*) is verified. Similar arguments apply to show that conditions (*ii*) and (*iii*) are met. To show safety at higher types we use the definitions of $*$ and :. Suppose that we have a sequence of safety values $s^{(1)} \cdots s^{(k)}$ and of denotable values $m^{(1)} \cdots m^{(k)}$ with each $s^{(i)}$ safe for $m^{(i)}$. Then we need to check that

$$(\mathbf{S}[\![e_1(e_2)]\!]\rho) : s^{(1)} \ldots s^{(k)}$$

is safe for the corresponding expression expressed in terms of the m values. The above expression is equal to

$$(\mathbf{S}[\![e_1]\!]\rho) : (\mathbf{S}[\![e_2]\!]\rho) : s^{(1)} : \ldots : s^{(k)}$$

Now we use the structural induction hypothesis to infer that $(\mathbf{S}[\![e_1]\!]\rho)$ and $(\mathbf{S}[\![e_2]\!]\rho)$ are safe for $\mathbf{M}[\![e_1]\!]env$ and $\mathbf{M}[\![e_2]\!]env$. From the definition of safety at higher types, this means that

$$(\mathbf{S}[\![e_1]\!]\rho) : (\mathbf{S}[\![e_2]\!]\rho) : s^{(1)} : \ldots : s^{(k)}$$

is safe for

$$(\mathbf{M}[\![e_1]\!]env) * m^{(1)} * \cdots * m^{(k)}$$

Finally, we note that this is the same as

$$(\mathbf{M}[\![e_1(e_2)]\!]env) * m^{(1)} * \cdots * m^{(k)}$$

$\underline{e \equiv \textbf{new } x : \textbf{ref } t \textbf{ in } e_1}$

We first show that the effects of the declaration on the environments result in corresponding environments. Let env', α' and ρ' be the new environments. By definition,

$$env' = env[x \leftarrow address]$$
$$\rho' = \rho[x \leftarrow \langle \emptyset, \underline{ground} \rangle]$$
$$\alpha' = \alpha[x \leftarrow a]$$

Since env, α and ρ are corresponding environments, and x is not of functional type, it is sufficient to show that $\langle \emptyset, \underline{ground} \rangle$ is safe for $\lambda s. \langle address, s \rangle$. This however, is trivially true. Then condition (i) follows from the fact that $\lambda s. \langle address, s \rangle$ returns $address$ in any store and (ii) and (iii) follow from the fact that $\lambda s. \langle address, s \rangle$ is the identity on any store.

Now the structural induction hypothesis gives us the fact that all the conditions are satisfied for the evaluation of the body. We only need to check that the exit from the block does not invalidate this. Suppose that S is the set of actual locations that may have been affected during the evaluation of e_1. On entry to the block, $address$ gets allocated and, on exit, it is removed; no other location is added or removed from the store. Thus the set S of possibly accessed locations is not affected. In the estimate computed by S, the symbolic location a is removed on exit from the block. We know that $address$ is the only actual address that could have been referred to by a so the removal of a will not affect the rest of the support estimate.

For the case where the expression is being evaluated in the context of a recursive call we do not perform the deallocation on exit so the proof of soundness does not even require the argument of the last paragraph.

$\underline{e \equiv e_1 \leftarrow e_2}$

Since we do not have storable functions we only need consider the ground type. We establish the three safety properties (i), (ii) and (iii) from Definition 8.1.

- (i) According to the standard semantics, the value of this expression is the value of e_2, so by the structural induction hypothesis applied to e_2 we see that condition (i) holds.
- (ii) Using the structural induction hypothesis on e_1, we see that $\mathbf{M_e}[\![e_1]\!]env\ store$ and $\mathbf{M_e}[\![e_1]\!]env\ store'$ are equal; let us call this value $address$. The effect on the store in either case is that

address gets updated by the same value, by part (*i*). Thus condition (*ii*) is met.

(*iii*) We note that by the structural induction hypothesis and from

$$env(\{x \mid \alpha(x) \cap (\mathbf{S}_1[\![e_1]\!]\rho) \neq \emptyset\}) \subseteq S$$

we have $\mathbf{M}_s[\![e_1]\!]env\ store|_{\bar{S}} = store|_{\bar{S}}$. Similarly, from

$$env(\{x \mid \alpha(x) \cap (\mathbf{S}_1[\![e_2]\!]\rho) \neq \emptyset\}) \subseteq S$$

we conclude that

$$\mathbf{M}_s[\![e_2]\!]env\ store_1|_{\bar{S}} = store_1|_{\bar{S}} = store|_{\bar{S}}$$

where $store_1 = \mathbf{M}_s[\![e_1]\!]env\ store$. By axiom III, the update operation only affects the given address. The soundness theorem for aliases says that all possible values of this address are included in $\mathbf{A}_1[\![e_1]\!]\alpha\zeta\theta$ which contributes to the definition of S. Thus outside S the store is not affected; i.e., condition (*iii*) holds.

This completes the three conditions needed for the proof of this case.

$\underline{e \equiv e_1\uparrow}$

$\mathbf{M}_e[\![e_1\uparrow]\!]env\ store = Eval(store_1, address)$ by the standard semantics, where $store_1$ is the store resulting from the evaluation of e_1. Since we so not have storable functions we need only consider the ground type. If two stores agree on S, by structural induction, e_1 will evaluated to the same value *address* in both. According to the definition of **S**, we have

$$\mathbf{S}_1[\![e]\!]\rho = (\mathbf{S}_1[\![e_1]\!]\rho) \cup (\mathbf{A}_1[\![e_1]\!]\alpha\zeta\theta)$$

By the soundness theorem on aliases, $address \in S$. By the structural induction hypothesis, applied to e_1, (*ii*) holds since the effect of e_1 is the same as the effect of $e_1\uparrow$. Now, to show that (*i*) holds, we need to check that the equation

$$Eval(store_1, address) = Eval(store_1', address)$$

holds, where $store_1'$ is the store resulting from the evaluation of e_1 in the store $store_1$. This equation follows immediately from (*ii*) applied to ed_1. Finally, we note that dereferencing does not modify the store, so, by the structural induction hypothesis applied to e_1, we immediately have (*iii*).

This completes all the cases we have to consider. ∎

9. 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 compositional fashion. 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. In particular, Banning,[11] Barth,[12] and Weihl,[13] have studied interprocedural interference analysis. The work of Barth addresses the problem of interference analysis in the presence of recursive procedures and aliasing but he does not consider higher-order functions or pointer variables. Banning also considers only first-order block-structured languages and does not allow 1-valued expressions. Weihl considers parameters of functional type, but his language is otherwise first-order. All three papers just cited use various flow-analysis algorithms rather than a compositional scheme. These techniques are based on doing flow analysis on the call-graph of the program. Thus they would not extend to higher-order languages where one cannot determine the call-graph in advance. Also, the correctness of their schemes is hidden in algorithmic details.

The paper by Reynolds[16] cited in the introduction is the starting point for our work and the work of Gifford and Lucassen[18] mentioned. In his paper, Reynolds defines a symmetric, decidable binary relation # between program phrases such that when $P \# Q$, P and Q can be executed in parallel without interference. The elegance and simplicity of his scheme derive from the fact that with call-by-name semantics one can determine interference by examining whether the sets of free identifiers are disjoint. However, this can also lead to imprecision if the argument to a procedure is not used. Suppose that we have the following procedures:

```
procedure use-n(p); procedure p;
  begin n := 0 end;
```

and

```
procedure use-m
begin m := 0 end;
```

Now it is easy to see that in Reynolds' scheme he would say that use-n

(use-m) and $m := 0$ interfere whereas our scheme would realize that these two phrases were interference-free. We are able to do this by representing the support sets and alias sets for higher-order expressions as pairs with the second component of the pair storing information that keeps track of the possible aliases or interference that may result when the function is applied. This complicates the theory but is essential if one needs to do interference analysis for languages with call-by-value semantics. Furthermore, it is not clear how one could relax the restrictions in Reynolds' paper to handle pointers.

Recently, Gifford and Lucassen[18,19] related effect checking (as they call it) to polymorphic type checking. Their scheme incorporates the relevant information into the type system and their effect inference mechanism becomes part of the type inference mechanism. The programmer is expected to divide the store into regions. The type inference mechanism checks that expressions manipulate disjoint regions and thus ensures interference freedom. The precision of the inference mechanism is now under the control of the programmer, which is an attractive feature. Their scheme, however, needs programmer supplied annotations in order to obtain useful interference analysis. The fundamental difference is that they have a mechanism for the user to express interference constraints via the type system whereas we have an automatic scheme which could be one phase of a parallelizing compiler.

We are currently working on the questions left open in this paper.[20] We removed the restrictions on pointers, recursive data types and dynamic allocation. The two semantic functions then require an abstract store and the cost of their computation increases accordingly. We are also developing heuristics to compute fixed points, with which we reduce the more than exponential cost of the current algorithms for fixed-point computations.

ACKNOWLEDGMENTS

We would like to thank Ken Birman for proofreading the manuscript. We have benefited from helpful remarks made by Carl Gunter, Paul Hudak, Dieter Maurer and especially Robert Tennent.

A STANDARD SEMANTICS

To keep the definitions simple, we do not explicitely give the cases dealing with nontermination. The reader should assume that the semantic functions are strict.

$\mathbf{M}_e[\![x]\!]env\ store = env(x)$
$\mathbf{M}_s[\![x]\!]env\ store = store$

$$\mathbf{M}_e[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]env\ store = \begin{cases} \mathbf{M}_e[\![e_2]\!]env\ store' \text{ if } val = true \\ \mathbf{M}_e[\![e_3]\!]env\ store' \text{ if } val = false \end{cases}$$

$$\mathbf{M}_s[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]env\ store = \begin{cases} \mathbf{M}_s[\![e_2]\!]env\ store' \text{ if } val = true \\ \mathbf{M}_s[\![e_3]\!]env\ store' \text{ if } val = false \end{cases}$$

where

$val = \mathbf{M}_e[\![e_1]\!]env\ store$
$store' = \mathbf{M}_s[\![e_1]\!]env\ store$

$\mathbf{M}_e[\![\lambda x{:}t.e]\!]env\ store = \lambda s.\lambda v.\langle \mathbf{M}_e[\![e]\!]env[x \leftarrow v]s, \mathbf{M}_s[\![e]\!]env]x \leftarrow v]s\rangle$
$\mathbf{M}_s[\![\lambda x{:}t.e]\!]env\ store = store$

$\mathbf{M}_e[\![e_1(e_2)]\!]env\ store = (f\ store_2\ h)_1$
$\mathbf{M}_s[\![e_1(e_2)]\!]env\ store = (f\ store_2\ h)_2$

where

$f = \mathbf{M}_e[\![e_1]\!]env\ store$
$h = \mathbf{M}_e[\![e_2]\!]env\ store_1$
$store_1 = \mathbf{M}_s[\![e_1]\!]env\ store$
$store_2 = \mathbf{M}_s[\![e_2]\!]env\ store_1$

$\mathbf{M}_e[\![\textbf{letrec } f_1 = \lambda x_1 : t_1.e_1, ..., f_n = \cdots \textbf{ in } e]\!]env\ store = \mathbf{M}_e[\![e]\!]env'\ store$
$\mathbf{M}_s[\![\textbf{letrec } f_1 = \lambda x\mathbf{1} : t_1.e_1, ..., f_n = \cdots \textbf{ in } e]\!]env\ store = \mathbf{M}_s[\![e]\!]env'\ store$

where

$env' = lfp(F)$
$F{:}Ev \rightarrow Env =$
$\lambda\rho.env[\ldots. f_i \leftarrow \mathbf{M}_e[\![\lambda x_i{:}t_i.e_i]\!]\rho\ store, \ldots]$

$\mathbf{M}_e[\![\textbf{new } x{:}t \textbf{ in } e]\!]env\ store = \mathbf{M}_e[\![e]\!]env'\ store'$
$\mathbf{M}_s[\![\textbf{new } x{:}t \textbf{ in } e]\!]env\ store = store'''$

where

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

$\mathbf{M}_e[\![e_1 \leftarrow e_2]\!]env\ store = value$
$\mathbf{M}_s[\![e_1 \leftarrow e_2]\!]env\ store = Update(store'', loc, value)$

where

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

$$\mathbf{M_e}[\![e_1\uparrow]\!]env\ store = Eval(store', \mathbf{M_e}[\![e_1]\!]env\ store)$$
$$\mathbf{M_s}[\![e_1\uparrow]\!]env\ store = store'$$

where

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

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

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

REFERENCES

1. A. Neirynck, P. Panangaden, and A. J. Demers, Computation of Aliases and Support Sets, In *Proc. of the Fourteenth Annual ACM Symp. on Principle of Programming Languages*, pp. 274–283 (1987).
2. M. J. Gordon, A. J. R. Milner, and C. P. Wadsworth, *Edinburgh LCF*, LNCS 78. Springer-Verlag (1979).
3. M. Burke and R. Cytron, Interprocedural Dependence Analysis and Parallelization, *ACM Sigplan Notices*, **21**(7):162–175 (1986).
4. K. Kennedy, J. R. Allen, and D. Callahan, An Implementation of Inter-procedural Analysis in a Vectorizing Fortran Compiler, Technical Report TR86-38, Rice University (1986).
5. K. Cooper, Analyzing Aliases of Reference Formal Parameters, In *Conf. Record of the Twelfth Annual ACM Symp. on Principles of Programming Languages, ACM*, pp. 281–290 (1985).
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 Symp. on Principles of Programming Languages* (1977).
7. P. Cousot and R. Cousot, Static Determination of Dynamic Properties of Programs, In *Proc. of the 2nd Int'l Symp. on Programming* (1976).
8. A. Mycroft and F. Nielson, Strong Abstract Interpretation Using Power Domains, In *Proc. ICALP 1983, Lecture Notes in Computer Science*, **154**:536–5447. Springer-Verlag (1983).
9. G. L. Burn, C. L. Hankin, and S. Abramsky, The Theory and Practice of Strictness Analysis for Higher Order Functions, *Proc. of Programs as Data Objects, Springer Lecture Notes in Computer Science*, Vol. 217 (1986).
10. A. J. Bernstein, Analysis of Programs for Parallel Processing, *IEEE Transactions on Electronic Computers*, **EC-15**(5):757–763 (October 1966).

11. J. P. Banning, An Efficient Way to Find the Side-effects of Procedure Calls and the Aliases of Variables, In *Proc. of the Sixth Annual ACM Symp. on Principles of Programming Languages*, pp. 29–41 (1979).
12. J. Barth, A Practical Interprocedural Data Flow Analysis Algorithm, *CACM*, **21**:724–736 (1978).
13. W. Weihl, Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables, In *Proc. of the Seventh Annual ACM Symp. on Principles of Programming Languages*, pp. 83–94 (1980).
14. J. R. Larus and P. N. Hilfinger, Detecting Conflicts Between Structure Accesses, In *Proc. of the SIGPLAN 88 Conf. on Programming Language Design and Implementation*, pp. 21–34 (1988).
15. L. J. Hendren and A. Nicolau, Parallelizing Programs with Recursive Data Structures, In *Proc. of the Third ACM Int'l Conf. on Supercomputing*, pp. 205–214 (1989).
16. J. C. Reynolds, Syntactic Control of Interference, In *Proc. of the Fifth Annual ACM Symp. on Principles of Programming Languages*, pp. 39–46 (1978).
17. R. D. Tennent, Semantics of Interference Control, *Theoretical Computer Science*, **27**:297–310 (1983).
18. D. K. Gifford and J. M. Lucassen, Integrating Functional and Imperative Programming, In *Proc. of ACM Conference on Lisp and Functional Programming*, pp. 28–38 (1986).
19. J. M. Lucassen, *Types and Effects*, PhD thesis, Massachusetts Institute of Technology (1987).
20. A. Neirynck, *Static Analysis of Aliases and Side-Effects in Higher-Order Languages*, PhD thesis, Cornell University (January 1988).
21. R. D. Tennent, *Principles of Programming Languages*, Prentice-Hall (1981).
22. M. J. C. Gordon, *The Denotational Description of Programming Languages*, Springer-Verlag (1979).