Static Analysis of Aliases and Side Effects in Higher-Order Languages

Anne Neirynck
Ph.D. Thesis

88-896
February 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

STATIC ANALYSIS OF ALIASES AND SIDE EFFECTS IN HIGHER-ORDER LANGUAGES

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Anne Denise Neirynck

January 1988

STATIC ANALYSIS OF ALIASES AND SIDE EFFECTS IN HIGHER-ORDER LANGUAGES

Anne Denise Neirynck, Ph.D.

Cornell University 1988

In recent years, there has been substantial interest in the development of programming languages for new parallel architectures. A basic design conflict arises because languages with simple semantics tend to use storage inefficiently, whereas languages allowing the programmer to access storage explicitely are difficult to analyze. We present a compile-time estimation scheme for determining whether an expression in an imperative language either uses or updates the store. We also determine the aliasing behavior of expressions and in general, we can tell whether the evaluation of two expressions interfere.

Current interprocedural dataflow techniques for aliasing and side effect inference are valid for first-order languages. Our inference schemes provide information about aliasing and side effects in a higher-order expression language with call-by-value semantics. The higher order character of the language represents only a partial obstacle. On the other hand, the presence of l-valued expressions has the consequence that aliasing information must be computed for all expressions, and cannot be represented as a relation among identifiers. Furthermore, the introduction of pointers make aliasing and side effects flow-dependent properties.

Abstract interpretation techniques allow us to define compositional static inference schemes for aliasing and side effects, which can be proved sound with respect to the standard semantics by structural induction. The abstract interpretation functions are easy to modify, in case a different type of information is requested. We also discuss how different language features may affect the static analyses, simplifying them or making them untractable.

The abstract interpretation functions implicitely define static inference algorithms, which can easily be implemented by an attribute grammar, or any other tool capable of performing computations on the abstract syntax tree. The accuracy of these algorithms is better than for the dataflow ones, because we make use of control flow information. Our algorithms also compare favorably in complexity, but the dataflow approach is probably cheaper in most practical settings. In addition, our schemes can give information even in the presence of dynamically allocated data structures.

Biographical Sketch

Anne Neirynck was born in Belgium in 1956. She received a B.S. in Mathematics in 1978 from the Ecole Polytechnique Fédérale de Lausanne, in Switzerland. She worked as a research assistant in the same mathematics department and received a M.S. in Computer Science from the University of California at Berkeley in 1980. Before coming to Cornell, she worked in Lausanne with Prof. Giovanni Coray on the design of a distributed environment for Parallel Pascal.

Acknowledgements

Although a tremendous number of people have helped me through graduate school, I would like to especially thank my first advisor, Alan Demers, who made semantics look like a lot of fun. Prakash Panangaden has supervised the work described in this thesis, and I am deeply endebted to him for his time, attention, advice and enthusiasm. It has been a pleasure to work with him.

Among my friends, I extend special thanks to Jim Hook who has been a well-informed source of papers, references, and advice, and to Aliza Wechsler, for moral support and congenial companionship during our three years in the same office. Keshav Pingali always found time to listen to my problems with fixed point computations and to provide helpful suggestions. Although we have never met, I thank Prof. Tennent for a careful reading of a previous version of this work and for greatly improving the form and content of this dissertation.

Charlie Van Loan and David Gries, as graduate field representative and chairman, made the CS department a friendly place for graduate students during my years at Cornell. Special thanks are also due to Lee Schruben for his work as the outside minor member on my committee. I am endebted to Margaret Schimizzi for a great job on the figures.

And finally, I would like to thank Ken for his love and patience, and our daughter Sarah, who made the last months of thesis writing much more fun than they would have been without her.

Table of Contents

List of Figures

Chapter 1

Introduction

In recent years, there has been substantial interest in the development of programming languages for new parallel architectures. A basic design conflict arises because languages with simple semantics tend to use storage inefficiently, whereas languages allowing the programmer to access storage explicitely are difficult to analyze. Thus, although it is easy to detect potential parallelism in a pure functional language, the lack of in-place updating causes the creation of numerous copies of data structures. In an imperative language, assignments can be used to update data structures incrementally, but this produces side effects which make parallelism difficult to detect.

We present a compile-time estimation scheme for determining whether an expression in an imperative language either uses or updates the store. We also determine the aliasing behavior of expressions and in general, we can tell whether the evaluation of two expressions interfere. This work is an elaboration and amendment of the work reported in [16].

In addition to parallelizing compilers, aliasing and side effect information is useful in a variety of other circumstances: optimizing transformations, incremental recompilation, and user-friendly programming environments. In particular, aliasing information is needed for most static inferences, and is often implicitely computed in combination with other information.

Current interprocedural dataflow techniques for aliasing and side effect inference are valid for languages of the Fortran and Pascal families. Here, we study the static inference of aliasing and side effects in an expression language with *higher-order features*, such as ML [9]. A language is called higher-order if functions are expressible values (that is, an expression may evaluate to a function) instead of just denotable values (values associated with identifiers). In an expression language, there is no distinction between expressions and commands (or statements); statements are viewed as expressions with side effects. Static analyses for such languages are difficult for two reasons: first, there is no static call graph. The second problem is that aliasing becomes a major issue, since the left-hand side of an assignment or the argument of a function call can be any expression. The higher-order character of the language puts the problem outside the reach of traditional dataflow techniques, because these rely on knowledge of the call graph of a program.

By using techniques borrowed from abstract interpretation, we are able to define compositional static inference schemes for aliasing and side effects. We can also prove the soundness of our inference schemes with respect to the standard semantics by structural induction. We also discuss how different language features may affect the static analyses, simplifying them or making them untractable. When restricted to first-order languages our method gives more accurate results than dataflow analyses, but at higher cost.

Overview

Chapter 2 of this dissertation contains the formal setting of the problem, with a specification of the class of languages to which our analyses apply and the difficulties associated with aliasing and side effect inferences in such languages. Chapter 3 presents a solution for aliasing and side effects for a sub-class of these languages. In chapter 4 we prove the soundness of these schemes relative to the standard semantics of the language. Chapter 5 gives the general solution, valid for languages with linked data structures. Chapter 6 presents some results concerning the computation of fixed points for the abstract interpretation functions.

Chapter 2

Formal specification of the problem

In this chapter we give a formal description of aliasing and side effects in higher-order languages. The first section is a description of the class of languages to which our analysis applies. The abstract syntax and the standard semantics present no particular difficulty, but the features of interest (l-valued expressions, higher-order functional terms) are explained in some detail. The next section contains a discussion of purity, aliases and side effects in those languages. Because we work with expression languages, aliasing analysis is more difficult than for the imperative languages considered by the dataflow approach (section 2.3). To solve our problem, we borrow techniques from abstract interpretation. Section 2.4 contains a very cursory introduction to this topic, and mentions its connections with strictness analysis and operational semantics. The last section regroups the notations used in this thesis, for the benefit of the reader unfamiliar with denotational semantics.

2.1 The language and its semantics

The analyses developed in this thesis are applicable to block-structured languages with imperative features – a store with explicit means of access to it: allocation and deallocation of variables, assignments, dereferencing. Within that framework, there is still a wide variety of languages to choose from. Command vs expression languages, typed or not, polymorphic or not, static or/and dynamic allocation, lexical scoping etc. The class of languages covered by this analysis can be described as higher-order simply typed expression languages with lexical scoping. Chapter 5 is dedicated to the extension of the analyses to dynamic data structures. The adaptation to an imperative language is straightforward; it would entail the separation between statements and expressions, and the use of two semantic functions, just as in a standard semantics. On the other hand, it is necessary to have a typed language. This point will be made clearer when discussing the proofs of the soundness theorems. We do not know whether our analyses apply to polymorphic languages with a general polymorphic type system [21]. They do apply to languages with a ML-style of polymorphism.

This section contains a formal description of the languages covered. An abstract syntax is a convenient way to describe a class of languages, without being concerned about concrete syntactic details. For the sake of readability, we will use explicit keywords rather than relying on the very condensed notation favored by functional languages community; this does not imply any commitment about the

$$
\begin{aligned}
e \;&::=\; c \mid x \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \mid \lambda x : t.\ e \mid e_1(e_2) \\
&\qquad \textbf{letrec } f_1 = \lambda x_1 : t_1.e_1\ ,\ \ldots\ , f_n = \lambda x_n : t_n.e_n \textbf{ in } e \\
&\qquad \textbf{new } x : \textbf{ref } t \textbf{ in } e \mid e_1 \leftarrow e_2 \mid e\uparrow \\
&\qquad e_1\ ;\ e_2 \\
t \;&::=\; \mathit{int} \mid \mathit{bool} \mid \cdots \\
&\qquad t_1 \to t_2 \\
&\qquad \textbf{ref } t
\end{aligned}
$$

Figure 2.1: Abstract syntax

concrete syntax. We then define the denotational semantics of the language and the store model in which the operational behavior is observed.

2.1.1 Abstract syntax

The complete abstract syntax for the language is displayed in Figure 2.1.1. The core of the language is the typed lambda calculus with a construct to define mutually recursive functions:

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

An expression is either a constant, an identifier, an alternative expression, a lambda term, an application, or an expression evaluated in the context of n mutually recursive functions. To this functional kernel, we add three imperative constructs: static allocation performed with the **new** construct, an assignment statement written $e_1 \leftarrow e_2$, and a dereferencing construct $e\uparrow$.

$$
e \;::=\; \textbf{new } x : \textbf{ref } t \textbf{ in } e \mid e_1 \leftarrow e_2 \mid e\uparrow
$$

Since this is an expression language, these three constructs are expressions. As in the programming language C, the value of an assignment is the value of its right hand side. The value of a dereferencing construct is the value of the variable denoted by e. Finally, the value of **new** $x : t$ **in** e is the value of e. The type rules and the complete denotational semantics can be found in appendix A.

Unlike the lambda calculus, we use the word *variable* to refer to an identifier bound by a **new** construct. The others are referred to simply as *identifiers*. A variable denotes a storage location; to denote the value associated with the variable we use the dereferencing construct. There is no implicit dereferencing, as is often the case with current programming languages. For instance, in Pascal, the x on the left hand side of the assignment $x := x$ is understood to mean the address of x, and the x on the right hand side to mean the value of variable x. In this language, the equivalent assignment must be written $x \leftarrow x\uparrow$. An expression can evaluate to a variable rather than to the value associated with the variable as in

$$\textbf{if } \mathit{true} \textbf{ then } x \textbf{ else } y$$

(assuming that x and y are global variables). Expressions which evaluate to a storage location are called *l-values*, the others are *r-values*. The separation between l-values and r-values is part of the type information, and l-values have a type of the form **ref** t. Only l-valued expressions are allowed on the left hand side of an assignment.

Finally, since we have side effects, a sequential operator is the last construct needed:

$$e \ ::= \ e_1 \ ; \ e_2$$

As syntactic sugar, a **let** construct will sometimes be used in the examples: **let** $x = e_1$ **in** e_2 is equivalent to $(\lambda x : t.e_2)e_1$. The type t of x is implicitely determined by the type of e_1.

The type system is defined inductively by:

$$t \ ::= \ bool \mid int \mid \ \ldots \ \mid t_1 \rightarrow t_2 \mid \mathbf{ref}\ t$$

It allows only simple types (no data structures), and provides an arrow type for functions and the **ref** construct for l-values. For the time being, we will restrict the syntax in order to simplify the abstract interpretations. Functions are not storable and pointers are not allowed (i.e. l-values are not storable either). These restrictions can be enforced by typechecking, and some will be lifted in a later chapter. Furthermore, because we have lexical scoping, l-values cannot be exported outside their scope. This property can be detected statically by the methods developed here.

2.1.2 Store model

It is traditional to find a model of the store, including store management, in the denotational semantics [10,24]. Although this results in a semantics which is not fully abstract [3], the abstract interpretation functions are insensitive to the store management policy, and rely only on some basic properties of the store. These properties can be expressed as a set of axioms which must be satisfied by the store model in the semantics used. Therefore, the soundness of the abstract interpretations is not bound to a particular store model in the denotational semantics.

A store can be viewed as a finite function from locations to values.

$$Store : Loc \rightarrow (Val + \underline{unused})$$

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

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

The store model must satisfy the locality axioms listed below.

Given a store, *Allocate* returns a store and the location of the newly allocated space. The new store must coincide with the old store over all addresses except the one just allocated (axiom Ia), and the new location should not have been already allocated (axiom Ib).

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

Axiom I

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

then

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

(b) $Eval(store, loc) = \underline{unused}$

$Eval(newstore, loc) = \underline{empty}$

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

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

Axiom II

$newstore \mid_{Loc-\{loc\}} = store \mid_{Loc-\{loc\}}$

where $newstore = DeAllocate(store, loc)$

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

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

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

The third axiom specifies that updating a location should not affect the content of other locations.

Axiom III

$Update(store, loc, value) \mid_{Loc-\{loc\}} = store \mid_{Loc-\{loc\}}$

One usually finds additional axioms specifying the behavior of $Update$ and $Eval$. In our case, because l-values are not stored, we will not need them until later.

Axiom IV

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

Axiom V

$Update(Update(store, loc, val_1), loc, val_2) = Update(store, loc, val_2)$

2.1.3 Semantics

The type system is standard and is discussed in appendix A. The semantic domains used in the denotational semantics are as follows.

$$Val = Triv + Bool + Nat + FVal + Loc$$

$$Env = Id \to Val$$

$$FVal = Val \to Store \to (Val \times Store)$$

The domains *Triv*, *Bool*, *Nat* and *Loc* contain the atomic values used in the language. We assume that the domains are flat and the sum being used is the coalesced sum. *Loc* represents the domain of storage locations. The domain *FVal* is used to represent higher-order constructs or "functional" values. 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.

The environment is a finite function mapping identifiers to their values. The usual operators on finite functions will be used to manipulate the environment. The reader is referred to section 2.5 for a definition of finite functions.

In defining the denotational semantics, we 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 (due to Demers [7]) 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 signatures of these functions are:

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

and

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

A full definition of the semantics is given in appendix A; a few clauses are discussed here in order to illustrate the notation and style of semantic definition.

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

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

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

$$\langle loc, store' \rangle = Allocate(store)$$

$$env' \ = \ env[x \ \leftarrow \ loc]$$

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

$$store''' = DeAllocate(store'', loc)$$

Within the scope of a new variable declaration, the variable is first allocated space, then the expression is evaluated in the context of the updated environment env' and the new store $store'$. This evaluation may also modify the store, resulting in $store''$. Deallocation occurs when the scope of the declaration is exited.

The following clauses specify the two other explicit imperative features – assignment and dereferencing.

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

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

where

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

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

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

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

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

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

where

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

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

2.2 Purity, aliases, and support sets

Originally, our problem was the detection of pure expressions at compile-time for a higher-order language. An expression is called *pure* if its evaluation is independent of the store, that is its value does not depend on the content of the store and its evaluation does not modify the store. A related notion is that of *side effect* for a function. A function produces a side effect if its application modifies the values of global variables. So, an expression is pure if its evaluation does not cause any side effect (here, the store is viewed as a single global variable). Side effects in function calls or boolean evaluation are considered poor programming style [24] and the existing methods to ensure purity at compile-time enforce good style rules which are sufficient but not necessary for purity and which are definitely inadequate for "real" programming. For instance, to ensure that functions are pure, some compilers did forbid parameters passed by reference, or the occurrence of global variables in declarations of functions. These restrictions are too conservative as the discussion below will show.

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

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

In order for our scheme to deem an expression impure only when global variables are affected we need to compute the set of global variables actually read or 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 using a compositional analysis we must actually know which variables are being affected by subexpressions, not just that the subexpressions do affect a variable. We call the set of global variables which are affected by or whose values affect the evaluation of an expression the *support* of that expression. A related notion appears in a paper by Halpern, Meyer and Trakhtenbrot [11], who define the cover of an expression as a set of locations instead:

$$Cover(e) = \{\mathbf{M_e}[\![x]\!]env\ store \mid x \in Support(e)\}$$

An expression is pure when its support set is empty.

The computation of the support set of an expression may require knowledge of the aliases 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 the variables possibly aliased by e. A similar phenomenon occurs in assignment expressions.

2.3 Related work

Static analysis is as old as optimizing compilers, and there has been a substantial body of work published on the static inference of aliasing and side effects, most recently [1,2,6,25] . The earliest algorithms use dataflow graphs to detect interfering expressions in a Fortran-like language. Although the limitations on the language are not explicitely spelled out, this approach is restricted to imperative first-order languages. The restriction to first-order languages is not just pragmatic: the dataflow approach is based on knowledge of the call graph of a program. The most sophisticated algorithms [1,2] perform inter-procedural analysis of block-structured languages by computing symbolically the procedure calls to detect identifier aliasing.

Barth's algorithm [2] applies to a Pascal-like language, with nested procedures, recursion and parameters passed by reference. The information is computed as relations over the sets of procedures and variables. For instance, the call graph is encoded as a relation between procedures and the scoping information is a relation between procedures and variables. Direct side effects (that is, without considering procedure calls) are estimated by scanning the definition of the procedure, and are represented as relations between procedures and variables. Thus, in one pass over the text of the program, Barth collects information about the side effects of procedures and computes the call graph and scoping information. The interprocedural information is then computed from the composition and transitive closure of these relations.

Banning's algorithm [1] applies to the same class of languages, but represents the information in graphs and tables instead. From these, he performs what is in fact symbolic execution, complete with dynamic allocation and symbolic locations.

Weihl [25] addresses the issue of functional parameters and pointers, in the presence of which the call graph is no more a static entity and aliasing and side effects become flow-dependent properties. He uses Barth's relations to compute an estimate of the call graph. Assignments involving pointers are translated in updates to relevant relations. Unfortunately his static analysis model is based on relations, more suited to the representation of flow-independent properties, and his analyses therefore lack precision.

We should also mention Kennedy and his group [6], who are developing a programming environment for Fortran. Although the choice of Fortran means that aliasing and side effects computation is greatly simplified, they discuss a related issue, which static information is needed and how is it used.

An important precursor of our work is a paper by Reynolds [20] in which he defined syntactic restrictions to eliminate interference in a higher-order language with *call-by-name* parameter passing. The soundness of his scheme was proved correct by Tennent [23]. Reynolds defines a symmetric, decidable binary relation # between program phrases such that when $P\#Q$ then 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:

$$\mathbf{let}\ use\_n = \lambda p : int \rightarrow int.n \leftarrow 0$$

and

$$\mathbf{let}\ use\_m = \lambda x : int.m \leftarrow 0$$

In Reynolds' scheme, $use\_n(use\_m)$ and $m \leftarrow 0$ interfere, whereas our scheme will realize that these two phrases are interference-free. We are able to do this by having the support sets and alias sets for higher-order expressions be pairs, whose second component keeps track of the possible aliases or interference which may result when the function is applied. This complicated the theory but is essential for languages with call-by-value semantics. Furthermore, it is not clear how one could relax the restrictions in Reynolds' paper to handle l-valued expressions (pointers).

The last development in the area is the work of Gifford and Lucassen [8,14].

As part of the development of a new language, FX, Lucassen integrates side effects information into the type system of a language. He draws the parallel between the type of an expression, which determines the sort of values the expression may evaluate to, and the effect of an expression, which encodes the operations on the store allowed during the evaluation of the expression. Effects are described in terms of partitions of the store, called regions. The description of the side effects is done by the programmer: the type of a function must include a specification of its side effects, and a type and region must be associated with each new location. An expression can be abstracted over either type, effect or region, and effect inference is part of the type inference mechanism. This approach yields an elegant solution to the problem of side effect verification, but leaves to the programmer the burden of effect specification.

2.4 Abstract interpretation

2.4.1 A simple application of abstract interpretation

Abstract interpretation is the generic name given to static analysis techniques using non-standard semantics domains approximating the standard semantics domains. The simplest example is the sign rule for integers. Suppose one is not interested in the result of a calculation, but only in its sign. Given the signs of the inputs, one might infer the sign of the result using the well-known rules:

$$positive + positive = positive$$

$$positive + negative = any$$

$zero * positive = zero$

etc ...

In a formal setting, it is as if *Integer*, the domain of integers (see figure 2.2) had been approximated by a simpler domain, called *Sign*, consisting of values *zero*, *negative*, *positive* and *any*, arranged in a lattice as shown in figure 2.3. The operations of addition, difference, and multiplication have a new meaning in the simpler domain, but are consistent with their traditional interpretation in the integer domain.

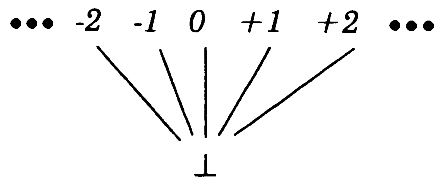

Figure 2.2: The *Integer* domain

The syntactical class *Exp* of legal expressions is given by the grammar:

$e ::= number \mid e * e \mid e + e \mid e - e \mid (e)$

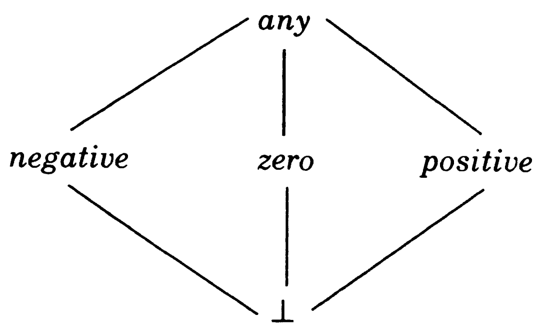

Figure 2.3: The *Sign* lattice

An expression is either a number constant, or the sum, difference, or product of two expressions. The standard semantics domain is *Integer*, the lattice of integers of figure 2.2, with the usual operations of addition, substraction, multiplication etc. To distinguish them from the syntactic symbols $+$, $-$, and $*$, they are indexed by I. The function *value* maps syntactic number constants to their values.

The standard semantics is defined by the function **Value**, mapping syntactical expressions to their values in the *Integer* domain:

$$\mathbf{Value} : Exp \rightarrow Integer$$

$$\mathbf{Value}[\![number]\!] = value(number)$$

$$\mathbf{Value}[\![e_1 + e_2]\!] = \mathbf{Value}[\![e_1]\!] +_{\mathsf{I}} \mathbf{Value}[\![e_2]\!]$$

$$\mathbf{Value}[\![e_1 - e_2]\!] = \mathbf{Value}[\![e_1]\!] -_{\mathsf{I}} \mathbf{Value}[\![e_2]\!]$$

$$\mathbf{Value}[\![e_1 * e_2]\!] = \mathbf{Value}[\![e_1]\!] *_{\mathsf{I}} \mathbf{Value}[\![e_2]\!]$$

The approximating semantics performs computations in the *Sign* domain. The operations in this domain are indexed by S. Instead of the *value* function, we now use a *sign* function, which returns *positive*, *zero*, or *negative* for syntactic constants.

$$\mathbf{Sign} : Exp \rightarrow Sign$$

$$\mathbf{Sign}[\![number]\!] = sign(number)$$

$$\mathbf{Sign}[\![e_1 + e_2]\!] = \mathbf{Sign}[\![e_1]\!] +_S \mathbf{Sign}[\![e_2]\!]$$

$$\mathbf{Sign}[\![e_1 - e_2]\!] = \mathbf{Sign}[\![e_1]\!] -_S \mathbf{Sign}[\![e_2]\!]$$

$$\mathbf{Sign}[\![e_1 * e_2]\!] = \mathbf{Sign}[\![e_1]\!] *_S \mathbf{Sign}[\![e_2]\!]$$

We can now formalize the notion of "consistent" semantics by stating a soundness theorem relating the standard semantics to the non-standard one. To continue the example above, one may write:

$$\mathbf{Sign}[\![e]\!] = positive \;\Rightarrow\; \mathbf{Value}[\![e]\!] > 0$$

Of course, the reverse is not true. An expression with a positive value does not necessarily evaluate to *positive* in the approximating semantics. It could as well evaluate to *any*.

2.4.2 Abstract interpretation and strictness analysis

Abstract interpretation has been applied mostly to functional languages, and there is extensive literature about strictness analysis. Among the recent papers, Hudak [13] showed that strictness analysis extends to higher-order languages, but that the cost is exponential in the worst case. Subsequently, a series of papers [4, 5,15,19] showed that although the worst case is exponential, the average case is tractable. The algorithms for higher-order abstract interpretation are exponential or worse and chapter 6 is devoted to improving the cost of the algorithms developed in this thesis.

2.4.3 Abstract interpretation and operational semantics

Since the soundness of an abstract interpretation is expressed with respect to a denotational semantics, it may seem at first that it it is not an adequate framework to analyze operational aspects of a language. In fact, the denotational semantics often contains some operational specifications as well, and it is possible therefore to use abstract interpretation for that purpose. Hudak does so for reference counting [12]. He goes further and claims that a denotational semantics together with the full specification of the operational aspects can be viewed as a non-standard semantics also. We would rather reconcile operational semantics and abstract interpretation in the manner described in section 4.5, where is discussed a general framework to show the soundness of an abstract interpretation with operational features.

2.5 Notational conventions

Most of the notational conventions used in this thesis are standard practice in the area of denotational semantics. They are collected here.

Semantic functions

Abstract interpretation functions are written in the form of semantic functions.

Given an expression, and an environment, and perhaps some additional argument (typically a store), a semantic function returns the meaning of the expression with respect to the environment (and the store). Thus, semantic functions have a signature of the form:

$$\mathbf{F} : Exp \rightarrow Env \rightarrow \ldots \rightarrow D$$

The function name is in uppercase bold font. Since we define more than one abstract interpretation function, we index the environments, domains etc with the name of the function:

$$\mathbf{F} : Exp \rightarrow Env_F \rightarrow \ldots \rightarrow D_F$$

When invoking the function, the expression argument is enclosed in double brackets, and the environment and other arguments are appended to the right:

$$\mathbf{F}[\![exp]\!]env \ldots$$

The image of an expression under a semantic function is called the *meaning* of the expression. Since we use many abstract interpretation functions, we reserve the word "meaning" for the standard semantics, and we call $\mathbf{F_i}[\![exp]\!]env$ the *F-image* of exp. If the range of the function is a product, we sometimes use subscripts next to the function name to denote the projection of the function onto a factor of its range. Thus, a declaration

$$\mathbf{F} : Exp \rightarrow Env \rightarrow \ldots \rightarrow (D_1 \times \ldots \times D_n)$$

induces a family of functions with signature

$$\mathbf{F_i} \; : \; Exp \rightarrow Env \rightarrow \ldots \rightarrow D_i \quad i = 1, \ldots, n$$

defined by

$$\mathbf{F_i}[\![exp]\!]env \;=\; (\mathbf{F}[\![exp]\!])_i \quad i = 1 \ldots n$$

Semantic functions are defined inductively over the structure of expressions. Definitions consist of a list of clauses, one for each syntactical construct in the language. Elements of a product domain are denoted by a list of terms enclosed in angle brackets $\langle e_1, \ldots, e_n \rangle$.

Finite functions

Environments and stores are defined in terms of finite functions. Finite functions are functions undefined everywhere except on a finite number of values of their arguments. They are built from ϵ, the function undefined everywhere, by means of an updating construct which extends the domain of the function:

$$(f[x \leftarrow v])(y) = \begin{cases} f(y) & y \neq x \\ v & y = x \end{cases}$$

The reverse operation restricts the domain of a finite function:

$$(f \backslash x)(y) = \begin{cases} f(y) & y \neq x \\ \perp & y = x \end{cases}$$

The symbol $\perp$ (bottom) denotes the undefined value.

In some cases, we implicitely use the pointwise extension of a finite function. Set values are denoted by uppercase letters. For instance, if we have finite function f with signature:

$$f : A \rightarrow B$$

$$\perp \sqsubseteq \mathit{false}$$
$$\perp \sqsubseteq \mathit{true}$$

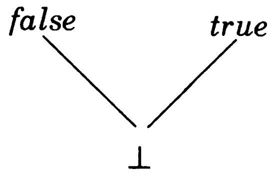

Figure 2.4: Boolean domain

Its pointwise extension $\overline{f}$ has signature:

$$\overline{f} : \mathcal{P}(A) \to \mathcal{P}(B)$$

and is defined by:

$$\overline{f}(X) = \{f(x) \mid x \in X\}$$

We write $f(X)$ for $\overline{f}(X)$.

<u>Domains</u>

The range of semantic functions are *domains* [22]. For our purposes, domains are specified as sets with a partial order denoted by $\sqsubseteq$. The least element of a domain D is denoted $\perp_D$, or simply $\perp$ if there is no ambiguity.

The domains used in this thesis are built from two base domains, *Bool* and $\mathcal{P}(S)$. *Bool* is the flat domain of Boolean values, completely defined by the equations of figure 2.4 page 26. The domain $\mathcal{P}(S)$, where S is a finite set, is the powerset of S, ordered by set inclusion:

$$s_1 \sqsubseteq s_2 \iff s_1 \subseteq s_2$$

The least element is the empty set. The ordering relation induces a lattice structure, and thus a domain.

The more elaborate domains are built from the base domains using domain constructors. The partial order is induced from the partial order of the sub-domains.

Ordinary product $D_1 \times D_2$

$D_1 \times D_2$ is the usual Cartesian product with the ordering relation applied componentwise:

$$\langle x_1, x_2 \rangle \sqsubseteq \langle y_1, y_2 \rangle \iff (x_1 \sqsubseteq y_1) \wedge (x_2 \sqsubseteq y_2)$$

Coalesced sum $D_1 + D_2$

The coalesced sum of D_1 and D_2 is the set

$$(D_1 - \{\perp_{D_1}\}) \cup (D_2 - \{\perp_{D_2}\}) \cup \{\perp_D\}$$

with the partial order induced from the partial orders in D_1 and D_2 and :

$$\perp_D \sqsubseteq x \quad \forall x \in D_1 + D_2$$

Chapter 3

Aliasing and side effect analysis

This chapter contains the definitions of the abstract interpretation functions, **P**, **A**, and **S**. The first one, **P**, detects pure expressions (for a restricted notion of purity) and is introduced solely to illustrate our basic techniques. The same techniques are then applied to define **A** and **S**, the alias and support abstract interpretation functions, which compute conservative estimates of aliasing and support information. These estimates are correct, except for one situation which can be detected by similar techniques.

3.1 Purity analysis – a first attempt

This section describes 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 sections.

We begin with an intentionally simplistic definition of purity, namely that an expression is pure if and only if its computation does not require any assignments to, or evaluations of, variables. This version of purity analysis uses two abstracted functions, $\mathbf{P}_1$ and $\mathbf{P}_2$. The first one, $\mathbf{P}_1$, has a simple interpretation:

$$\text{“}\mathbf{P}_1[\![e]\!]penv \;=\; true\text{” means “expression } e \text{ is pure ”}$$

Its signature is:

$$\mathbf{P_1} : Exp \rightarrow Env_P \rightarrow Bool$$

Given an expression and an environment (which provides the purity information associated with the free identifiers of the expression), $\mathbf{P_1}$ returns *true* if the expression is pure. This is only a conservative estimate, and if $\mathbf{P_1}$ returns *false*, the expression may nonetheless be pure.

If functions appear in the expression, we need to know whether their applications produce side effects. This is the information computed by the second function, $\mathbf{P}_2$, and saved in the environment. If e is not of function type, the value of $\mathbf{P}_2[\![e]\!]penv$ is the constant $\underline{ground}$. 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 function $\mathbf{P}$, whose signature is

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

$$\begin{aligned}
\perp_{D_P} &= \langle \perp_{Bool}, \perp_{Val_P} \rangle & (3.1)\\
\lambda u.\langle b_1, v_1\rangle \sqsubseteq \lambda u.\langle b_2, v_2\rangle &\iff b_1 \sqsubseteq b_2 \text{ and } v_1 \sqsubseteq v_2 & (3.2)\\
\lambda u.\langle b_1, v_1\rangle \sqcup \lambda u.\langle b_2, v_2\rangle &= \lambda u.\langle b_1 \sqcup b_2, v_1 \sqcup v_2\rangle & (3.3)\\
\lambda u.\langle b_1, v_1\rangle \wedge \lambda u.\langle b_2, v_2\rangle &= \lambda u.\langle b_1 \wedge b_2, v_1 \wedge v_2\rangle & (3.4)\\
\underline{ground} \wedge \underline{ground} &= \underline{ground} & (3.5)
\end{aligned}$$

Figure 3.1: The domain D_P

The domains are:

$$Env_P = Id \rightarrow Val_P$$

$$Val_P = \{\underline{ground}\} + Val_P \rightarrow D_P$$

$$D_P = Bool \times Val_P$$

The purity environment maps identifiers to purity values. The domain D_P pairs the information needed to define $\mathbf{P}_1$ with that needed for $\mathbf{P}_2$. The range of $\mathbf{P}_1$ is the boolean domain *Bool*. Val_P, the range of $\mathbf{P}_2$, contains the purity information necessary to compute the side effects of expressions when applied as functions. The element $\underline{ground}$ abstracts all nonfunctional values. For functional expressions, we must know the purity behavior of the argument to compute the side effects (thus, the subdomain $Val_P \rightarrow D_P$). D_P and Val_P are recursively defined in order to capture the relevant behavior of higher-order functions. The recursive structure of the domains is similar to the one used by Hudak for higher-order strictness analysis [13].

The domain structure of D_P is induced by the structure of its subdomains. Figure 3.1 contains the non-trivial equations defining D_P as a domain. The domain

$$
\begin{array}{l}
\mathbf{P}[\![c]\!]penv \;=\; \langle\; true,\; \underline{ground} \rangle \\
\mathbf{P}[\![x]\!]penv \;=\; \langle\; true,\; penv(x) \rangle \\
\mathbf{P}[\![\mathbf{if}\;\; e_1 \;\mathbf{then}\;\; e_2 \;\mathbf{else}\;\; e_3]\!]penv \;=\; \\
\qquad \langle\; \mathbf{P}_1[\![e_1]\!]penv \wedge \mathbf{P}_1[\![e_2]\!]penv \wedge \mathbf{P}_1[\![e_3]\!]penv\;, \\
\qquad\qquad \mathbf{P}_2[\![e_2]\!]penv \wedge \mathbf{P}_2[\![e_3]\!]penv \;\rangle \\
\mathbf{P}[\![\lambda x : t.e]\!]penv \;=\; \langle\; true\;,\; \lambda u.\mathbf{P}[\![e]\!]penv[x \leftarrow u] \;\rangle \\
\mathbf{P}[\![e_1(e_2)]\!]penv \;=\; \\
\qquad \langle\; \mathbf{P}_1[\![e_1]\!]penv \wedge \mathbf{P}_1[\![e_2]\!]penv \wedge (\mathbf{P}_2[\![e_1]\!]penv\mathbf{P}_2[\![e_2]\!]penv)_1\;, \\
\qquad\qquad (\mathbf{P}_2[\![e_1]\!]penv\mathbf{P}_2[\![e_2]\!]penv)_2 \;\rangle \\
\mathbf{P}[\![\mathbf{letrec}\;\; f_1 = e_1,\; \ldots\;, f_n = e_n \;\mathbf{in}\;\; e]\!]penv \;=\; \mathbf{P}[\![e]\!]penv' \\
\qquad \text{where } penv' \;=\; lfp(\lambda penv.penv[\ldots, f_i \leftarrow \mathbf{P}_2[\![e_i]\!]penv, \ldots]) \\
\mathbf{P}[\![\mathbf{new}\;\; x : t \;\mathbf{in}\;\; e]\!]penv \;=\; \mathbf{P}[\![e]\!]penv[x \leftarrow \underline{ground}] \\
\mathbf{P}[\![e_1 \leftarrow e_2\,]\!]penv \;=\; \langle\; false\;,\; \underline{ground} \;\rangle \\
\mathbf{P}[\![e\!\uparrow]\!]penv \;=\; \langle\; false\;,\; \underline{ground} \;\rangle \\
\mathbf{P}[\![e_1\;;\; e_2\,]\!]penv \;=\; \langle\; \mathbf{P}_1[\![e_1]\!]penv \wedge \mathbf{P}_1[\![e_2]\!]penv\;,\; \mathbf{P}_2[\![e_2]\!]penv \;\rangle
\end{array}
$$

Figure 3.2: The **P** function.

Val_P is built as a separated sum (equation 3.1), so the element $\underline{ground}$ is related only to $\perp$. This is reasonable, since once we definitely know that an expression is not a function it cannot become one as computation progresses. Functional terms in subdomain $Val_P \rightarrow D_P$ can be compared, possibly after α-conversion, if they have the same structure (equation 3.2). The join of two functional terms is defined accordingly (equation 3.3). Finally, we extend the boolean conjunction operator to the functional terms (equations 3.4 3.5).

The definition of **P** is given in figure 3.2 page 31. Groups of related clauses are explained in detail below.

<u>Constants and identifiers</u>

$$\mathbf{P}[\![c]\!]penv \;=\; \langle\; true,\; \underline{ground} \rangle$$

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

A constant is pure, and since all constants are non-functional, the second component of the pair is ground. An identifier by itself is pure and its potential for producing side effects during applications depends on whether it is bound to a function. The bindings of identifiers are kept in environment *penv*.

Conditional and sequential evaluations

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

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

A conditional expression is pure if its three subexpressions are pure. Since we cannot decide which branch of the expression will be executed, the second component of $\mathbf{P}$ is the conjunction of the two branches. By the same token, the sequential evaluation of two expressions is pure if both expressions are pure. Since the value of e_1 ; e_2 is e_2, the second component involves only e_2.

Abstraction and application

$$\mathbf{P}[\![\lambda x : t.e]\!]penv \;=\; \langle\ true\ ,\ \lambda u.\mathbf{P}[\![e]\!]penv[x \leftarrow u]\ \rangle$$
$$\mathbf{P}[\![e_1(e_2)]\!]penv \;=$$
$$\langle\ \mathbf{P}_1[\![e_1]\!]penv \wedge \mathbf{P}_1[\![e_2]\!]penv \wedge (\mathbf{P}_2[\![e_1]\!]penv\ (\mathbf{P}_2[\![e_2]\!]penv))_1\ ,$$
$$(\mathbf{P}_2[\![e_1]\!]penv\ (\mathbf{P}_2[\![e_2]\!]penv))_2\ \rangle$$

A lambda abstraction by itself is always pure because its evaluation does nothing, as a glance at the standard semantics confirms. It is during applications that the side effects concealed by lambda abstractions become manifest. Note also the contribution of the subexpression computations to the first component of $\mathbf{P}[\![e_1(e_2)]\!]penv$.

Recursive functions

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

A **letrec** block is pure when the evaluation of its body in the new environment is free of side effects. The environment is built using a least fixed point construction. A complete discussion of fixed point computations appears in chapter 6.

Imperative constructs

$$\mathbf{P}[\![\mathbf{new}\ x : t\ \mathbf{in}\ e]\!]penv \ = \ \mathbf{P}[\![e]\!]penv[x \leftarrow \underline{ground}]$$
$$\mathbf{P}[\![e_1 \leftarrow e_2\ ]\!]penv \ = \ \langle\ false\ ,\ \underline{ground}\ \rangle$$
$$\mathbf{P}[\![e\uparrow]\!]penv \ = \ \langle\ false\ ,\ \underline{ground}\ \rangle$$

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 <u>*ground*</u> since it cannot be a function. Of course, assignments and dereferencing are not pure. Since functions are not storable, we know that the type of the expressions is non-functional, and thus the second component is <u>*ground*</u>.

Unfortunately, the **P** function is unsatisfactory, since it declares an expression impure even if its evaluation affects only local variables. In general, this is too restrictive to be useful. A more standard definition of purity states that an expression is pure if its computation does not 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:

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

and

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

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 global variables involved in the evaluation of the expression. This motivates us to replace the boolean domain D_P with a richer structure capable of expressing this information. This is the objective of the next three sections.

3.2 Representation of alias information

Two identifiers are possible aliases if they may refer to the same location in some activation. Since this relation is reflexive, symmetric and transitive, aliasing information is usually represented as an equivalence class over the set of identifiers. This is the approach taken by the dataflow algorithms (see Barth [2] for instance).

Because our language allows dereferencing and assignments of expressions (instead of identifiers), our analysis requires knowledge of the aliases of any expression. We chose to present that information by defining an abstract interpretation function that maps an expression to the set of equivalence classes it may alias. Note that since variables are each allocated their own space in the store, they have distinct addresses and thus there is at most one variable in each equivalence class; the other elements must be identifiers. Furthermore (except in one particular case), because there are no pointers, the part of the store accessible to an expression is exactly covered by the variables visible in the scope of that expression. Therefore, at the condition that the variables have distinct names, we can use the identifiers denoting these variables to denote the locations. And we can also use the identifier denoting the unique variable in an alias equivalence class as its representative. We call *Var* the set of identifiers bound by a **new** construct in the program being analyzed.

The exception is the case where a function with local variables is called more than once in some activation, and the local variable instantiated in one call is passed to the other call. Consider the following example, where we use modified versions of the functions *apply* and *eval*:

$$\textbf{let } mod\_apply =$$

$$\lambda g : \textbf{ref } int \rightarrow int.\lambda x : \textbf{ref } int.\textbf{new } y : \textbf{ref } int \textbf{ in } y \leftarrow x\uparrow \; ; \; g(y)$$

$$\textbf{in}$$

$$\textbf{let } mod\_eval = \lambda y : \textbf{ref } int.mod\_apply(\lambda z : \textbf{ref } int.z\uparrow, y)$$

$$\textbf{in } mod\_apply(mod\_eval, a)$$

The application $mod\_apply(mod\_val, a)$ binds g to mod_apply and x to global variable a. The execution of mod_apply creates an instance of local variable y, at location l_1, assigns it the value of a, and then calls mod_eval with y, i.e. l_1, as parameter. All that mod_eval does is call mod_apply and pass it its parameter, in this case l_1. This second application of mod_apply creates a second instantiation of local variable y, at location l_2. Now, the support of the assignment $y \leftarrow x\uparrow$ consists of the two instantiations of y, at l_1 and l_2.

Except for this case, rare in practice, the representation of aliases by means of variable names is sound. Furthermore, the use of variable names leads to a simple and elegant solution to the static estimation of aliases, outlined in the next section. The general scheme to estimate aliases is much more elaborate and since we can also detect statically multiple calls to the same function within one activation (see appendix B), we feel that the simple scheme is useful and worth presenting.

These remarks justify the use of $\mathcal{P}(Var)$, the powerset of Var, as the base domain for our alias abstract interpretation function.

3.3 An abstract interpretation for computing aliases

As discussed in section 2.2, in order to calculate the support of an expression, we may need to determine the aliasing behavior of some of its subexpressions. This section describes the abstract interpretation function **A**, which gives a conservative estimate of the set of variables possibly aliased by an expression.

The recursive structure of the domains is similar to the ones used for the function **P**. This should not be a surprise, since it is the presence of higher-order functions which determines the recursive structure. For the reasons mentioned in the previous section, the base domain is $\mathcal{P}(Var)$, the powerset of the set of variable identifiers, instead of *Bool*.

$$\mathbf{A} : Exp \rightarrow Env_A \rightarrow D_A$$

$$Env_A = Id \rightarrow D_A$$

$$D_A = \mathcal{P}(Var) \times Val_A$$

$$Val_A = \{\underline{ground}\} + D_A \rightarrow D_A$$

The intuition behind this choice of domains is as follows. D_A is the domain of alias values. The first component of D_A is a set of identifiers denoting the variables possibly aliased, while the second contains the information needed to determine the aliasing behavior of other expressions which contain applications of the first expression. We refer to elements of D_A as *alias pairs*, and to the first component of an alias pair as an *alias set*.

The lattice structure over domain D_A is induced by the lattice $\mathcal{P}(Var)$. The partial order is defined as follows:

$$\mathbf{A}[\![c]\!]aenv \; = \; \langle\ \emptyset\ ,\ \underline{ground}\ \rangle$$
$$\mathbf{A}[\![x]\!]aenv \; = \; aenv(x)$$
$$\mathbf{A}[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]aenv \; = \; \mathbf{A}[\![e_2]\!]aenv \sqcup \mathbf{A}[\![e_3]\!]aenv$$
$$\mathbf{A}[\![\lambda x : t.e]\!]aenv \; = \; \langle\ \emptyset\ ,\ \lambda u.\ \mathbf{A}[\![e]\!]aenv[x \leftarrow u]\ \rangle$$
$$\mathbf{A}[\![e_1(e_2)]\!]aenv \; = \; \mathbf{A}_2[\![e_1]\!]aenv\ (\mathbf{A}[\![e_2]\!]aenv)$$
$$\mathbf{A}[\![\mathbf{letrec}\ f_1 = e_1\ ...\ f_n = e_n\ \mathbf{in}\ e]\!]aenv \; = \; \mathbf{A}[\![e]\!]aenv'$$
$$\text{where } aenv' \; = \; lfp(\lambda aenv.\ aenv[...,f_i \leftarrow \mathbf{A}[\![e_i]\!]aenv,...])$$
$$\mathbf{A}[\![\mathbf{new}\ x : t\ \mathbf{in}\ e]\!]aenv \; = \; \mathbf{A}[\![e]\!]aenv[x \leftarrow \langle\ \{''x''\}, \underline{ground}\rangle]$$
$$\mathbf{A}[\![e_1 \leftarrow e_2\ ]\!]aenv \; = \; \langle\ \emptyset\ ,\ \underline{ground}\ \rangle$$
$$\mathbf{A}[\![e\!\uparrow]\!]aenv \; = \; \langle\ \emptyset\ ,\ \underline{ground}\ \rangle$$
$$\mathbf{A}[\![e_1\ ;\ e_2\ ]\!]aenv \; = \; \mathbf{A}[\![e_2]\!]aenv$$

Figure 3.3: The **A** function

$$\lambda u.\langle a_1, v_1\rangle \sqsubseteq \lambda u.\langle a_2, v_2\rangle \; = \; \lambda u.\langle a_1 \subseteq a_2, v_1 \sqsubseteq v_2\rangle$$

It induces a lub operation, also defined only for terms of identical type:

$$\lambda u.\langle a_1, v_1\rangle \sqcup \lambda u.\langle a_2, v_2\rangle \; = \; \lambda u.\langle a_1 \cup a_2, v_1 \sqcup v_2\rangle$$

The operation of least upper bound in this domain is obtained by taking unions of possible alias sets. This reflects the conservative nature of the estimates we are making.

The clauses for the computation of aliases are summarized in figure 3.3 on page 38, and explained in details below. The aliasing behavior of global identifiers is provided by an alias environment called *aenv*.

Constants and identifiers

$$\mathbf{A}[\![c]\!]aenv \; = \; \langle\ \emptyset\ ,\ \underline{ground}\ \rangle$$

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

There are no l-valued or functional constants, so the alias set of a constant is empty, and the second component is *ground*. The alias pair associated with an identifier is provided by the alias environment *aenv*.

Conditional and sequential evaluation

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

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

In the presence of a conditional, we compute a conservative estimate. The least upper bound operation computes the possible aliases as the union of the sets of possible aliases for each arm of the conditional. For a sequential evaluation, only the second expression contributes to the alias value.

Abstraction and application

$$\mathbf{A}[\![\lambda x : t.e]\!]aenv \;=\; \langle\ \emptyset\ ,\ \lambda u.\, \mathbf{A}[\![e]\!]aenv[x \leftarrow u]\ \rangle$$

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

A lambda abstraction cannot possibly be an identifier, so we compute its alias set as the empty set; we need the second component to compute the alias set of expressions involving applications of this function. This will become clearer in the last section of this chapter, where some alias computations are described in detail.

Recursive functions

$$\mathbf{A}[\![\mathbf{letrec}\ \ f_1 = e_1 \ldots f_n = e_n\ \mathbf{in}\ \ e]\!]aenv \;=\; \mathbf{A}[\![e]\!]aenv'$$

where $aenv' = lfp(\lambda aenv.\ aenv[..., f_i \leftarrow \mathbf{A}[\![e_i]\!]aenv, ...])$

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.

Imperative constructs

$$\mathbf{A}[\![\mathbf{new}\ \ x : t\ \mathbf{in}\ \ e]\!]aenv \ = \ \mathbf{A}[\![e]\!]aenv[x \leftarrow \langle\ \{''x''\}, \underline{ground}\rangle]$$

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

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

A new variable starts out being aliased only to itself. Since there are no variables of functional type, we can safely assert that the second component value is *ground*. 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. Similar remarks apply to the clause for the dereferencing operator.

3.4 Support sets as an abstract interpretation

The support set of an expression is estimated by the function **S**, which is defined in terms of **A**. The recursive equations defining the support domains D_S and Val_S are similar to those introduced earlier for aliases. There is one difference, however: an expression consisting of an identifier is always pure, and there is no need to keep track of the first component of its support. Thus, we use Val_S instead of D_S in the environment and as the source domain for functional values.

$$\mathbf{S} : Exp \to Env_S \to D_S$$

$$Env_S = Id \to Val_S$$

$$Val_S = \{\underline{ground}\} + Val_S \to D_S$$

$$D_S = \mathcal{P}(Var) \times Val_S$$

The definition of the function **S** is given below. Note that when we use the 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. The complete definition of **S** is in figure 3.4, page 42. A discussion of each of the clauses follows.

<u>Constants and identifiers</u>

$$\mathbf{S}[\![c]\!]senv = \langle\, \emptyset,\ \underline{ground}\rangle$$

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

Constants and identifiers are pure, and their support set is empty. As for purity, the second component of the support of an identifier is provided by the environment.

<u>Conditional and sequential evaluation</u>

$$\mathbf{S}[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]senv =$$

$$\langle\ \mathbf{S}_1[\![e_1]\!]senv \cup \mathbf{S}_1[\![e_2]\!]senv \cup \mathbf{S}_1[\![e_3]\!]senv\ ,$$

$$\mathbf{S}_2[\![e_2]\!]senv \sqcup \mathbf{S}_2[\![e_3]\!]senv\ \rangle$$

$$
\begin{aligned}
&\mathbf{S}[\![c]\!]senv \;=\; \langle\, \emptyset,\ \underline{ground} \rangle \\
&\mathbf{S}[\![x]\!]senv \;=\; \langle\, \emptyset,\ \overline{senv(x)} \rangle \\
&\mathbf{S}[\![\mathbf{letrec}\ \ f_1 = e_1 \ldots f_n = e_n\ \mathbf{in}\ \ e_2]\!]senv \;=\; \mathbf{S}[\![e]\!]senv' \\
&\quad\text{where } senv' \;=\; lfp(\lambda senv.senv[\ldots, f_i \leftarrow \mathbf{S}_2[\![e_i]\!]senv, \ldots]) \\
&\mathbf{S}[\![\mathbf{if}\ \ e_1\ \mathbf{then}\ \ e_2\ \mathbf{else}\ \ e_3]\!]senv \;=\; \\
&\qquad\quad \langle\ \mathbf{S}_1[\![e_1]\!]senv \cup \mathbf{S}_1[\![e_2]\!]senv \cup \mathbf{S}_1[\![e_3]\!]senv\ , \\
&\qquad\qquad \mathbf{S}_2[\![e_2]\!]senv \sqcup \mathbf{S}_2[\![e_3]\!]senv\ \rangle \\
&\mathbf{S}[\![\lambda x : t.e]\!]senv \;=\; \langle\ \emptyset\ ,\ \lambda u.\mathbf{S}[\![e]\!]senv[x \leftarrow u]\ \rangle \\
&\mathbf{S}[\![e_1(e_2)]\!]senv \;=\; \\
&\quad \langle\ \mathbf{S}_1[\![e_1]\!]senv \cup \mathbf{S}_1[\![e_2]\!]senv \cup (\mathbf{S}_2[\![e_1]\!]senv(\mathbf{S}_2[\![e_2]\!]senv))_1\ , \\
&\qquad\quad (\mathbf{S}_2[\![e_1]\!]senv(\mathbf{S}_2[\![e_2]\!]senv))_2\ \rangle \\
&\mathbf{S}[\![\mathbf{new}\ \ x : t\ \mathbf{in}\ \ e]\!]senv \;=\; \\
&\quad \langle \mathbf{S}_1[\![e]\!]senv[x \leftarrow \underline{ground}]\ -\ \{''x''\}\ ,\ \mathbf{S}_2[\![e]\!]senv[x \leftarrow \underline{ground}]\ \rangle \\
&\mathbf{S}[\![e_1 \leftarrow e_2\ ]\!]senv \;=\; \langle\ \mathbf{S}_1[\![e_1]\!]senv \cup \mathbf{S}_1[\![e_2]\!]senv \cup \mathbf{A}_1[\![e_1]\!]aenv\ ,\ \underline{ground}\ \rangle \\
&\mathbf{S}[\![e\!\uparrow]\!]senv \;=\; \langle\ \mathbf{S}_1[\![e]\!]senv \cup \mathbf{A}_1[\![e]\!]aenv\ ,\ \underline{ground}\ \rangle \\
&\mathbf{S}[\![e_1\ ;\ e_2\ ]\!]senv \;=\; \langle\ \mathbf{S}_1[\![e_1]\!]senv \cup \mathbf{S}_1[\![e_2]\!]senv\ ,\ \mathbf{S}_2[\![e_2]\!]senv\ \rangle
\end{aligned}
$$

Figure 3.4: The **S** function

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

Each subexpression contributes its support to the first component. For the conditional expression, the second component is the least upper bound of the two branches.

<u>Abstraction and application</u>

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

$$\mathbf{S}[\![e_1(e_2)]\!]senv \;=\;$$

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

$$(\mathbf{S}_2[\![e_1]\!]senv(\mathbf{S}_2[\![e_2]\!]senv))_2\ \rangle$$

The functional clauses are defined as for the **P** function.

Recursive functions

$$\mathbf{S}[\![\textbf{letrec } f_1 = e_1 \ldots f_n = e_n \textbf{ in } e_2]\!] senv \;=\; \mathbf{S}[\![e]\!] senv'$$

$$\text{where } senv' \;=\; lfp(\lambda senv.senv[\ldots, f_i \leftarrow \mathbf{S}_2[\![e_i]\!] senv, \ldots])$$

Imperative constructs

$$\mathbf{S}[\![\textbf{new } x : t \textbf{ in } e]\!] senv \;=$$

$$\langle \mathbf{S}_1[\![e]\!] senv[x \leftarrow \underline{ground}] \;-\; \{''x''\} \;,\; \mathbf{S}_2[\![e]\!] senv[x \leftarrow \underline{ground}] \;\rangle$$

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

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

When a new variable is declared, its support inside its scope is just itself. On the other hand, the support of the entire block must not include the new variable since the scope of the new variable ends when the block is exited, so we explicitly remove the new variable from the support computed for the body of the block. This particular equation is correct assuming all identifiers in the expression being analyzed are distinct. In determining the support of the explicitly imperative constructs, we need to know the aliases of some subexpressions. Thus in the assignment clause, we compute the union of 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.

If one wishes to distinguish between variables read and variables updated, the definition of **S** can be decomposed into two functions, **R** and **U**, identical to **S**, except for assignments and dereferences:

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

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

$\mathbf{U}[\![e_1 \leftarrow e_2]\!]uenv \;=\; \langle\; \mathbf{U}_1[\![e_1]\!]uenv \cup \mathbf{U}_1[\![e_2]\!]uenv \cup \mathbf{A}_1[\![e_1]\!]aenv \;,\; \underline{ground} \;\rangle$

$\mathbf{U}[\![e\uparrow]\!]uenv \;=\; \langle\; \mathbf{U}_1[\![e]\!]uenv \;,\; \underline{ground} \;\rangle$

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

$$\mathbf{S}_1[\![e]\!]senv = \mathbf{R}_1[\![e]\!]renv \cup \mathbf{U}_1[\![e]\!]uenv$$

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

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

3.5 Examples

To clarify the clauses for abstraction and application, we compute the alias and support sets of two simple functions, *id*, the identity function over the integer variables, and *eval*, an abstraction of the dereferencing operation. Thus, if a is a variable, one expects the alias of $id(a)$ to be the set $\{''a''\}$, and its support set to be empty. For $eval(a)$, it should be the reverse: its alias set should be empty, but its support set should be $\{"a"\}$.

Consider the following expression:

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

As mentioned before, the computation of aliases and support sets should be paired. We give the new domains, so that the reader can understand the calculations below.

$$\mathbf{AS} : Exp \rightarrow Env_{AS} \rightarrow D_{AS}$$

$$
\begin{aligned}
Env_{AS} &= Id \rightarrow \ D_{AS} \\
Val_{AS} &= \ \{\underline{ground}\} \ + \ D_{AS} \rightarrow D_{AS} \\
D_{AS} &= \mathcal{P}(Var) \times \mathcal{P}(Var) \times Val_{AS}
\end{aligned}
$$

An alias&support value ($\mathbf{AS}[\![e]\!]\rho$) is a triple, the first two components are the alias and support sets of the expression (formely $\mathbf{A}_1[\![e]\!]aenv$ and $\mathbf{S}_1[\![e]\!]senv$); the third component is $\underline{ground}$ if e is a non-functional value, and a function otherwise. The clauses defining the alias&support value of an application and of an abstraction are:

$\mathbf{AS}[\![\lambda x : t.e]\!]\rho = \langle\ \emptyset, \emptyset, \lambda u.\mathbf{AS}[\![e]\!]\rho[u \leftarrow u]\ \rangle$

$\mathbf{AS}[\![f(v)]\!]\rho$

$\quad = \langle\ (effect(f(v)))_1,\ \mathbf{AS}_2[\![f]\!]\rho \cup \mathbf{AS}_2[\![v]\!]\rho \cup (effect(f(v)))_2,\ (effect(f(v)))_3\ \rangle$

where

$$effect(f(v)) = (\mathbf{AS}_3[\![f]\!]\rho)(\mathbf{AS}[\![v]\!]\rho)$$

The initial alias&support environment, ρ, is empty. After a's declaration is processed, the environment contains the alias&support information associated with variable a:

$$\rho = [a \leftarrow \langle\{"a"\}, \emptyset, \underline{ground}\rangle]$$

The alias&support information for the identity function id is:

$$\mathbf{AS}[\![id]\!]\rho = \langle\emptyset, \emptyset, \lambda u.u\rangle$$

For the evaluation function $eval$, we have:

$$\mathbf{AS}[\![eval]\!]\rho = \langle\emptyset, \emptyset, \lambda u.\langle\emptyset, u_1, \underline{ground}\rangle\rangle$$

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

$$\begin{aligned} effect(id(a)) &= (\mathbf{AS}_3[\![id]\!]\rho')(\mathbf{AS}[\![a]\!]\rho') \\ &= (\lambda u.u)(\langle\{"a"\}, \emptyset, \underline{ground}\rangle) \\ &= \langle\{"a"\}, \emptyset, \underline{ground}\rangle \end{aligned}$$

$$\begin{aligned} effect(eval(a)) &= (\mathbf{AS}_3[\![eval]\!]\rho')(\mathbf{AS}[\![a]\!]\rho') \\ &= (\lambda u.\langle\emptyset, u_1, \underline{ground}\rangle)(\langle\{"a"\}, \emptyset, \underline{ground}\rangle) \\ &= \langle\emptyset, \{"a"\}, \underline{ground}\rangle \end{aligned}$$

We verify that $\mathbf{AS}$ computes the expected values:

$$\begin{aligned} &\mathbf{AS}[\![id(a)]\!]\rho' \\ &= \langle\, (effect(id(a)))_1,\ \mathbf{AS}_2[\![id]\!]\rho' \cup \mathbf{AS}_2[\![a]\!]\rho' \cup (effect(id(a)))_2,\ (effect(id(a)))_3\, \rangle \\ &= \langle\{"a"\}, \emptyset, \underline{ground}\rangle \end{aligned}$$

$$\begin{aligned}
&\mathbf{AS}[\![eval(a)]\!]\rho' \\
&\quad = \langle\ (\mathit{effect}(eval(a)))_1,\ \mathbf{AS}_2[\![eval]\!]\rho' \cup \mathbf{AS}_2[\![a]\!]\rho' \cup (\mathit{effect}(eval(a)))_2, \\
&\qquad (\mathit{effect}(eval(a)))_3\ \rangle \\
&\quad = \langle \emptyset, \{\text{"}a\text{"}\}, \underline{ground} \rangle
\end{aligned}$$

Chapter 4

Soundness

Besides the compositional flavor of the inference schemes for aliases and support sets, the abstract interpretation framework has a second major advantage: we can relate the abstract interpretation functions to the standard semantics and show the soundness of our inference schemes.

We state and discuss the soundness theorem for aliases in the next section. The proof relies on the axioms of the store model. The soundness theorem for support sets is similar to the one for aliases, and since support sets are defined in terms of aliases values, the proof uses the soundness theorem for aliases. The heart of both proofs is a joint induction on the structure of terms in the programming language as well as on their types. Hudak and Young prove a soundness theorem for strictness analysis which they claim holds for the untyped lambda calculus. In fact, their proof crucially uses induction on the type of lambda terms and, as they observe, they need to enforce a "weak type discipline" to guarantee termination.

Because we defined the standard semantics in a denotational form, our soundness theorems are not as strong as they could be. This point is discussed in the last section.

4.1 Soundness theorem for aliases

In the following sections, S denotes a subset of the variable set *Var*, and $\overline{S}$ its complement with respect to *Var*. 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 the abstract interpretation for aliases is defined formally in the following theorem:

Theorem 4.1

If the evaluation of e terminates,

$\forall\ S \in \mathcal{P}(Var)$,

$$\mathbf{A}_1[\![e]\!]aenv \subseteq S \ \Rightarrow\ \forall\ y \in \overline{S},\ \forall\ store\quad \mathbf{M_e}[\![e]\!]env\ store \neq \mathbf{M_e}[\![y]\!]env\ store$$

Because the non-standard semantics **A** cannot detect non-terminating computations, the theorem is valid only for converging expressions. Otherwise, in a context where a is a global variable, the diverging expression

letrec

$may\_diverge = \lambda n : int.\ \mathbf{if}\ n = 0\ \mathbf{then}\ a\ \mathbf{else}\ may\_diverge(n+1)$

in $may\_diverge(1)$

would alias to "a" , according to the computation of **AS**:

$$\mathbf{AS}[\![may\_diverge]\!]\rho = \langle\emptyset, \emptyset, \lambda u.\langle\{''a''\}, \emptyset, \underline{ground}\rangle\rangle$$

$$\mathbf{A}_1[\![may\_diverge(a)]\!]aenv = ((\mathbf{AS}_3[\![may\_diverge]\!]\rho)(\mathbf{AS}[\![1]\!]\rho))_1$$
$$= \{''a''\}$$

But since its argument is non-zero, the function call diverges, and its effect on the store, $\mathbf{M_s}[\![may\_diverge(1)]\!]env\ store$, is undefined. Note that the computation of aliases and support sets did not diverge.

By corresponding environments *env* and *aenv*, we mean environments in the standard and the non-standard semantics which result from the "same" computation, that is the expression e is evaluated in the same context. In that case, both environments contain bindings for the free identifiers of e, and their meanings correspond. A more formal definition follows.

As in Hudak and Young [13], we define partial application operators in both semantic domains, which facilitate the proof by providing a mechanism for carrying out induction on the type structure. The partial applicators define the reduction of a sequence of nested applications. The notation AAP_n is used for the abstract domain while AP_n is used in the standard domain.

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

$AP_n\ :\ (Store \rightarrow (Val \times Store))^{n+1} \rightarrow (Store \rightarrow Val \times Store)$

$AP_n(e, e^1, \ldots, e^n) =$

$$\begin{cases} e & n = 0 \\ AP_{n-1}(\lambda s.(e\ s)_1\ (e^1(e\ s)_2)_1\ (e^1(e\ s)_2)_2, e^2, \ldots, e^n) & n > 0 \end{cases}$$

The last line of the definition deserves an explanation: when passed a store s, one evaluates the function expression and gets a new store $(e\,s)_2$, which is used to evaluate the argument expression, yielding the value-store pair $(e^1(e\,s)_2)$. Finally, the application is performed, by applying the function expression value $((e\,s)_1)$ to the argument value $((e^1(e\ s)_2)_1)$ and the most recent store $((e^1(e\ s)_2)_2)$. This is just a rewriting of the standard semantics clause for application.

The operator AAP_n reflects the much simpler semantics of aliases.

$$AAP_n\ :\ (D_A)^{n+1} \rightarrow \mathcal{P}(Var)$$

$$AAP_n(a, a^1, \ldots, a^n) = \begin{cases} (a)_1 & n = 0 \\ AAP_{n-1}(((a)_2\ a^1)_2, \ldots, a^n) & n > 0 \end{cases}$$

The AP and AAP operators allow one to move down the type structure by applying the function component of a member of Val_S or Val. The fact that we have finite types only means that one can "reach" all types of interest by an inductive argument. Dually, it also means that computations in the approximating semantic domain must terminate. The induction on the type structure is reduced to an induction over the integers (the index of the partial application operators), since the number of arrows in the type of an expression is exactly the number of applications necessary to evaluate that expression.

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

$\forall\, m \leq n,\ a^i \in D_A,\ e^i \in D,\ a^i$ safe at level $n-1$ for e^i

$$AAP_m(a, a^1, \ldots, a^m) \subseteq S$$

$$\Rightarrow$$

$$\forall\, y \in \overline{S}, \forall\, store$$

$$AP_m(e, e^1, \ldots, e^m, store) \neq \mathbf{M_e}[\![y]\!]env\ store \tag{1}$$

Here, since y is a variable, $\mathbf{M_e}[\![y]\!]env\ store$ is the address of y, not its value ($\mathbf{M_e}[\![y\uparrow]\!]env\ store$).

$a \in D_A$ 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 follows in the next section.

4.2 Proof of soundness theorem for aliases

This proof exhibits the same sort of reasoning that one sees in proofs of strong normalization of the typed lambda calculus. There too, one goes down in the type structure by performing an application and producing in the process a structurally more complicated term.

We 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.

$\underline{e \equiv x}$

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 $env(x)$

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

By structural induction.

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

By fixed point 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 induction hypothesis on e', (1) 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 induction hypothesis on e_i, $aenv_n$ and env_n are corresponding environments.

By fixed point induction, $aenv'$ and env' are corresponding environments,

By induction hypothesis on e', (1) is true.

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

$\mathbf{A}_1[\![\lambda x : t.e']\!] aenv \ = \ \emptyset$

The value of e is an r-value, and it cannot be equal to any l-value.

This proves safety at level 0.

For safety at level n, note that

$AP_m(\mathbf{M_e}[\![e]\!] env\ store, e_1 \ldots, e_m, store)$

$= AP_m((\mathbf{M_e}[\![e]\!] env\ store\ e_1)_2\ store, e_2, \ldots, e_m, store)$

$= AP_{m-1}((\mathbf{M_e}[\![e']\!]\ env[u \leftarrow\ e_1]\ store, e_2, \ldots, e_m, store)$

And similarly for the alias expression:

$AAP_m(\mathbf{A}[\![e]\!] aenv, a_1 \ldots, a_m, store)$

$= AAP_{m-1}((\mathbf{A}[\![e']\!] env[u \leftarrow\ a_1] store, a_2, \ldots, a_m, store)$

By induction hypothesis on e', and because a_1 is safe for e_1, we have (1).

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

fix $n \geq 0$

We must show that $\mathbf{A}[\![e_1(e_2)]\!]aenv$ 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

$$AAP_m(\mathbf{A}[\![e]\!]aenv, a_1, \ldots, a_m) \subseteq S \quad \Rightarrow$$

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

By definition

$$AAP_m(\mathbf{A}[\![e_1(e_2)]\!]aenv, a_1, \ldots, a_m)$$

$$= AAP_{m-1}(((\mathbf{A}_2[\![e_1(e_2)]\!]aenv)\ a_1)_2, a_2, \ldots, a_m)$$

$$= AAP_{m-1}((((\mathbf{A}_2[\![e_1]\!]aenv)(\mathbf{A}[\![e_2]\!]aenv))_2 a_1)_2, \ldots, a_m)$$

$$= AAP_m(((\mathbf{A}_2[\![e_1]\!]aenv)(\mathbf{A}[\![e_2]\!]aenv))_2, a_1, \ldots, a_m)$$

$$= AAP_{m+1}(\mathbf{A}[\![e_1]\!]aenv, \mathbf{A}[\![e_2]\!]aenv, a_1, \ldots, a_m)$$

By induction hypothesis 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, by definition of AP

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

$$= AP_m((\mathbf{M_e}[\![e_1]\!]env\ store\ \mathbf{M_e}[\![e_2]\!]env\ store)_1, v_1, \ldots, v_m, store)$$

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

and therefore (1)

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

It is sufficient to show

$aenv[x \leftarrow \langle\ \{"x"\}, \underline{ground}\rangle]$ is safe for $env[x \leftarrow loc]$.

Since *aenv* and *env* are corresponding environments,

all that is left to prove is that

$aenv(x) = \langle\ \{"x"\}, \underline{ground}\rangle$ is safe at all levels for $env(x) = loc$

Since x is not a function type, only the case $n = 0$ is considered. That is:

$loc \neq \mathbf{M_e}[\![y]\!]env\ store$

which is true by axiom Ib.

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

In both the standard semantics and the aliasing semantics, the image of an assignment is that of its right hand side.

So, by induction hypothesis applied to e_2, (1) is true.

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

$\mathbf{A}_1[\![e'\uparrow]\!]aenv \ = \ \emptyset$

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

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

By induction hypothesis on e_2.

4.3 Soundness theorem for support sets

The soundness theorem for the function $\mathbf{S}$ states that the evaluation of an expression does not affect the store outside its support set, and that it depends solely on the values of variables in that set. In terms of operational semantics, this is equivalent to say that if two stores agree on the support set of an expression, the values of that expression with respect to each store are equal, and the side effects are identical.

Theorem 4.2

If the evaluation of e terminates,

$\forall$ *env*, *senv* corresponding environments, $\forall S \in \mathcal{P}(Var)$

$$\mathbf{S_1}[\![e]\!]senv \subseteq S \;\Rightarrow\; \forall\, store \;\; (\mathbf{M_s}[\![e]\!]env\ store)\mid_{\overline{S}} \;=\; store\mid_{\overline{S}}$$

and

$$\mathbf{S_1}[\![e]\!]senv \subseteq S \;\Rightarrow$$

$$\forall\, store,\ store' \text{ such that } store\mid_S = store'\mid_S$$

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

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

We use another partial application function to define formally corresponding environments for support sets.

$SAP_n \;:\; (D_S)^{n+1} \rightarrow \mathcal{P}(Var)$

$$SAP_n(s, s^1, \ldots, s^n) = \begin{cases} (s)_1 & n = 0 \\ (s)_1 \cup (s^1)_1 \cup SAP_{n-1}((s)_2(s^1)_2, \ldots, s^n) & n > 0 \end{cases}$$

$s \in D_S$ is <u>safe at level n for value $e \in D$</u> if

$\forall\, m \leq n,\; s^i \in D_S,\; e^i \in D,\; s^i$ safe at level $n-1$ for e^i

$$SAP_m(s, s^1, \ldots, s^m) \subseteq S$$

$$\Rightarrow$$

$$\forall\; store, store' \text{ such that } store\mid_S = store'\mid_S$$

$$[AP_m(e, e^1, \ldots, e^m) store]_1 = [AP_m(e, e^1, \ldots, e^m) store']_1 \tag{2a}$$

$$[AP_m(e, e^1, \ldots, e^m) store]_2 \mid_S = [AP_m(e, e^1, \ldots, e^m) store']_2 \mid_S \tag{2b}$$

and

$$\forall\; store\;\; [AP_m(e, e^1, \ldots, e^m, store)]_2 \mid_{\overline{S}} = store \mid_{\overline{S}} \tag{3}$$

If two stores agree on the support set of an expression, then the expression evaluates to the same value with respect to both stores (equation (2a)), its side effects are identical (equation (2b)), and the variables outside its support set are not modified by the evaluation (equation (3)). The base case ($n = 0$) simply reads as the soundness theorem we want to prove.

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

<u>$senv$ and env are corresponding environments</u> if

$$senv(x) \text{ is safe for } \lambda s.\langle env(x), s\rangle \;\; \forall\, x \in dom(senv)$$

Remark

The requirement that a pure expression always evaluates to the same value (equation 1a) may be too stringent for l-values, since it requires that variables be allocated to the same locations in the two stores. It could be replaced by a weaker one, but the proof of the alias soundness theorem would become more complicated.

4.4 Proof of soundness theorem for support sets

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

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

The following identities hold:

$$AP_{m+1}(\mathbf{M}[\![e_1]\!]env, \mathbf{M}[\![e_2]\!]env, v_1, \ldots, v_m)$$
$$= AP_m(\mathbf{M}[\![e_1(e_2)]\!]env, v_1, \ldots, v_m) \qquad \text{(ID1)}$$
$$SAP_{m+1}(\mathbf{S}[\![e_1]\!]senv, \mathbf{S}[\![e_2]\!]senv, s_1, \ldots, s_m)$$
$$= SAP_m(\mathbf{S}[\![e_1(e_2)]\!]senv, s_1, \ldots, s_m) \qquad \text{(ID2)}$$

Proof of ID1

Similar to the proof of ID2, see below.

Proof of ID2

$$SAP_m(\mathbf{S}[\![e_1(e_2)]\!]\rho, s^1, \ldots, s^m)$$
$$= \mathbf{S}_1[\![e_1(e_2)]\!]\rho \cup (s^1)_1 \cup SAP_{m-1}(\mathbf{S}_2[\![e_1(e_2)]\!]\rho\ (s^1)_2, s^2, \ldots, s^m)$$

The type of the term has been decreased by using the definition of SAP. The next equality follows by using the definition of **S** twice.

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

$$\cup (s^1)_1 \cup SAP_{m-1}((\mathbf{S}_2[\![e_1]\!]\rho(\mathbf{S}_2[\![e_2]\!]\rho))_2(s^1)_2, \ldots, s^m)$$

The next equality follows from the definition of SAP used in the reverse direction to go to a higher safety level but with the original application term broken down.

$$= \mathbf{S}_1[\![e_1]\!]\rho \cup \mathbf{S}_1[\![e_2]\!]\rho \cup SAP_m(\mathbf{S}_2[\![e_1]\!]\rho(\mathbf{S}_2[\![e_2]\!]\rho), s^1, \ldots, s^m)$$

$$= SAP_{m+1}(\mathbf{S}[\![e_1]\!]\rho, \mathbf{S}[\![e_2]\!]\rho, s^1, \ldots, s^m)$$

$\square$

The proof of the theorem proceeds by structural induction on e and by induction on the type of e. For the **letrec** construct we will need fixed point induction as well. Recall that all variable names are distinct. The proof of ($2a$) and ($2b$) are virtually identical; therefore, we only give the proof of ($2a$) and (3).

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

To show (3) for $n > 0$ and (2), it is sufficient to show that env' and $senv'$ are corresponding environments, and then use the induction hypothesis on e_1.

By definition,

$env' \;=\; env[x \;\leftarrow\; loc]$

$senv' \;=\; senv[x \;\leftarrow\; \underline{ground}]$

Since env and $senv$ are corresponding environments, and x is not of functional type, it is sufficient to show that $\langle \emptyset, \underline{ground} \rangle$ is safe at level 0 for $\lambda s.\langle loc, s\rangle$, which is true.

There is one equality left to show independently, that is (3) for $n = 0$, or

$\mathbf{M_s}[\![e]\!]env\ store\ |_S = store\ |_S$.

Assume $\mathbf{S_1}[\![\mathbf{new}\ \ x : t\ \mathbf{in}\ \ e_1]\!]\rho \subseteq S$

(a) $\mathbf{S_1}[\![e_1]\!]\rho \subseteq S \cup \{"x"\}$ by definition of $\mathbf{S}$

(b) $store'\ |_S = store\ |_S$ by Axiom Ia

(c) $store''\ |_{S\cup\{"x"\}} = store'\ |_{S\cup\{"x"\}}$ by induction hypothesis on e_1 and by (a)

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

(e) Since S is disjoint from $\{"x"\}$, when deallocating the space for x,

the S-restriction of the store is not affected:

$store'''\ |_S = DeAllocate(store'', loc)\ |_S$

$= store''\ |_S$ by axiom II

and we have our equality by transitivity of (b), (d), and (e)

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

Fix $n \geq 0$

We must show $\mathbf{S}[\![e_1(e_2)]\!]\rho$ is safe at level n for $\mathbf{M}[\![e_1(e_2)]\!]env$

Choose $m \leq n$, and s_i safe at level $n-1$ for v_i, $i = 1, \ldots, m$

Here we need induction on the term structure as well as on the type structure.

The latter is represented by the level n.

To prove (3) we proceed as follows:

By identity ID2,

$SAP_m(\mathbf{S}[\![e_1(e_2)]\!]\rho, s_1, \ldots, s_m) \subseteq S$

$\Rightarrow SAP_{m+1}(\mathbf{S}[\![e_1]\!]\rho, \mathbf{S}[\![e_2]\!]\rho, s_1, \ldots, s_m) \subseteq S$

Since e_1 and e_2 are structurally simpler than e, we get from the inductive hypothesis:

$[AP_{m+1}(\mathbf{M}[\![e_1]\!]env, \mathbf{M}[\![e_2]\!]env, v_1, \ldots, v_m)\ store]_2)\ |_{\overline{S}} = store\ |_{\overline{S}}$

But, by identity ID1:

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

and therefore (3) holds.

For (1a) we proceed as follows. By induction hypothesis on e_1, e_2:

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

$\Rightarrow$

$$[AP_{m+1}(\mathbf{M}[\![e_1]\!]env, \mathbf{M}[\![e_2]\!]env, v_1, \ldots, v_m)\ store]_1$$
$$= [AP_{m+1}(\mathbf{M}[\![e_1]\!]env, \mathbf{M}[\![e_2]\!]env, v_1, \ldots, v_m)\ store']_1$$

But using identity ID1 in the above equation, we get:

$$[AP_m(\mathbf{M}[\![e_1(e_2)]\!]env, v_1, \ldots, v_m)\ store]_1$$
$$= [AP_m(\mathbf{M}[\![e_1(e_2)]\!]env, v_1, \ldots, v_m)\ store']_1$$

that is, (1a).

The two cases we have shown illustrate the higher-order situations and an imperative construct. The remaining cases are much simpler.

$\underline{e \equiv x}$

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

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

$\mathbf{S}[\![x]\!]senv\ =\ senv(x)$

Since $senv$ and env are corresponding environments,

$senv(x)$ is safe for $\lambda s.\langle env(x), s\rangle$.

$e \equiv$ **letrec** $f_1 = e_1 \ \dots \ f_n = e_n$ **in** e'

By fixed point induction.

- base case:

$env_0 = env[\dots, f_i \leftarrow \bot_D, \dots]$ and $senv_0 = senv[\dots, f_i \leftarrow \bot_{D_S}, \dots]$ are corresponding environments.

By induction hypothesis on e', (2) and (3) are true.

- induction step:

Assume $senv_{n-1}$ and env_{n-1} are corresponding environments.

$env_n = env_{n-1}[\dots, f_i \leftarrow \mathbf{M_e}[\![e_i]\!] \ env_{n-1} \ store, \dots]$

$senv_n = senv_{n-1}[\dots, f_i \leftarrow \mathbf{S}[\![e_i]\!] senv_{n-1}, \dots]$

Then by induction hypothesis on e_i,

$senv_n$ and env_n are corresponding environments.

By fixed point induction, $senv'$ and env' are corresponding environments,

By induction hypothesis on e', (2) and (3) are true.

$e \equiv$ **if** e_1 **then** e_2 **else** e_3

By structural induction.

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

Showing (3) for $n = 0$ is trivial, since:

$\mathbf{M_s}[\![e \ = \ \lambda x : t.e']\!] env \ store = store$ by definition

To show (3) for $n > 0$ and (2), we use the induction hypothesis on e':

$\mathbf{M_e}[\![e \ = \ \lambda x : t.e']\!] env \ store$

$= \lambda v.\lambda s.\langle \mathbf{M_e}[\![e']\!]\ env[x \leftarrow v]\ s, \mathbf{M_s}[\![e']\!]\ env[x \leftarrow v]\ s\rangle$

$= \lambda v.\mathbf{M}[\![e']\!]\ env[x \leftarrow v]$

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

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

We need only to consider the case $n = 0$.

Show (1) for $n = 0$:

Assume $store_1 \mid_S = store_2 \mid_S$.

By induction hypothesis on e_1, we have:

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

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 (3) for $n = 0$:

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

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

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

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

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

By induction hypothesis on e_1 and (a),

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

By induction hypothesis on e_2 and (b),

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

By Axiom III and the soundness theorem for aliases,

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

and therefore (3)

$e \equiv e'\uparrow$

$\mathbf{M_e}[\![e'\uparrow]\!]env\ store = Eval(store, loc)$ cannot be applied, since its type is not functional.

Thus, it is sufficient to prove (2) and (3) for $n = 0$.

If two stores agree on S, by induction hypothesis, e' will evaluate to the same value loc 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, $\{loc\} \subseteq S$.

The equation $Eval(store'_1, loc) = Eval(store'_2, loc)$

follows from $(1b)$ on e'.

$\mathbf{M_s}[\![e\uparrow]\!]env\ store = store'$ by definition,

by induction hypothesis on e', $store' \mid_{\overline{S}} = store \mid_{\overline{S}}$ and we have (3).

$e \equiv e_1\ ;\ e_2$

By structural induction.

4.5 Discussion

The soundness theorems for aliases and support sets relate the approximate semantics and the standard semantics as shown in figure 4.1 page 67. A denotational semantics for a typed language is usually expressed in terms of a semantic meta-language, consisting of the typed lambda calculus plus some additional constants (the store etc). In our case, this is the role played by $\mathbf{M_e}$ and $\mathbf{M_s}$. The denotational semantics in fact also include, although not often explicitely, the lambda calculus semantics in terms of domains. A natural operational semantics is induced by the operational semantics of lambda calculus. Abstract interpretation, being a semantics in a simplified domain, obeys a analogous schema: the abstract interpretation function itself (**P**, **A** or **S**) maps expressions of the same typed language to an abstract semantic meta-language, which also consists of the typed lambda calculus. No additional constants are needed in this case. Then the lambda calculus semantics induces the approximate domains.

For strictness analysis, the soundness theorems relate the standard domain to the approximate domain. This is possible because the denotational semantics domain represent non-termination as a value. The soundness theorems for aliases and side effect analysis follow a different path. They create a connection between semantic meta-language expressions and the value of the approximate domain. This works because the store represents support and alias information as values.

Any kind of backward flow analysis (for instance, liveness analysis) would require a continuation semantics related to a collecting semantics. Incidentally, the resemblance of our soundness proofs to the proof of the strong normalization theo-

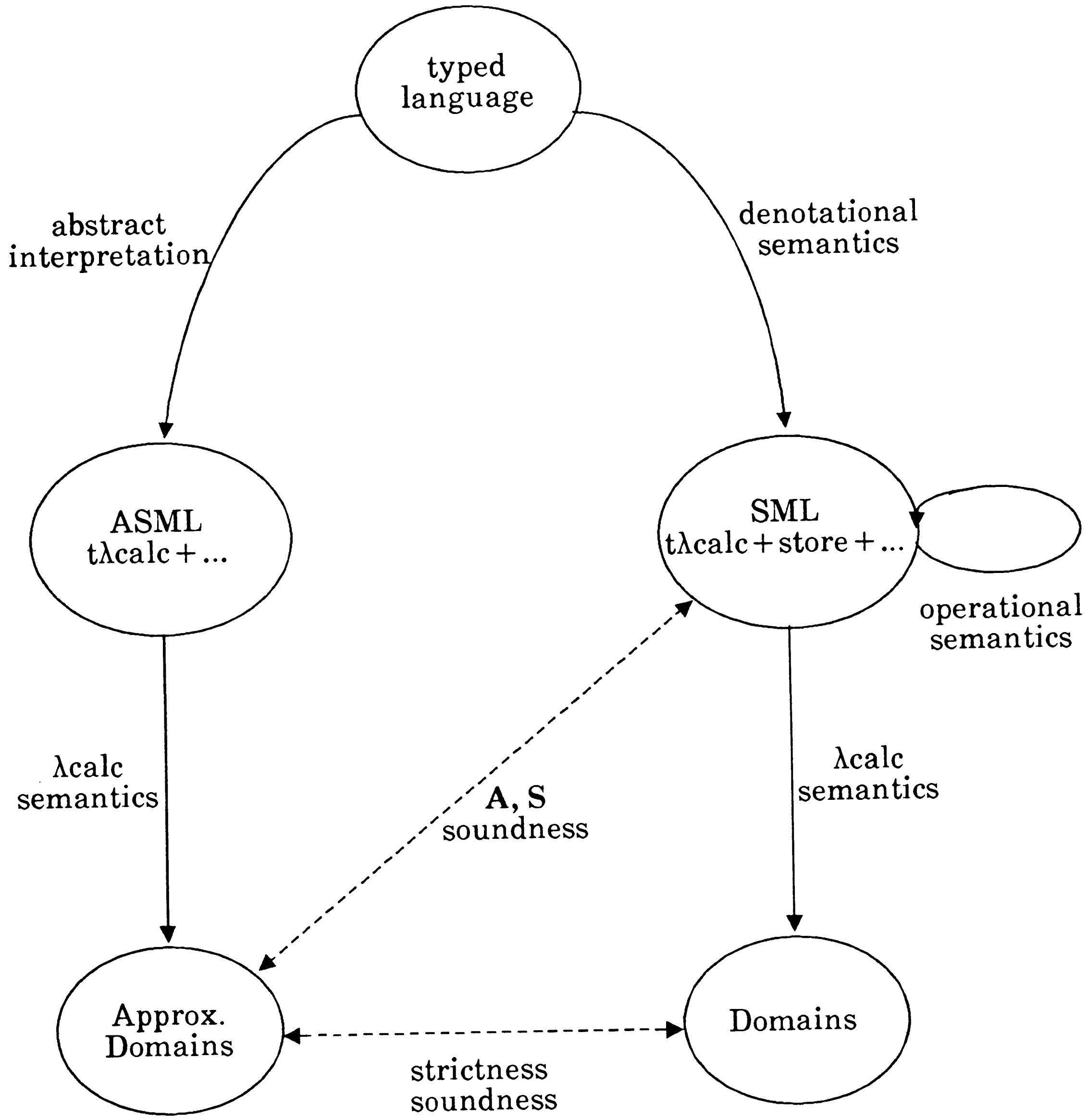

Figure 4.1: Abstract interpretation diagram

rem (which deals with questions of termination) is caused by the need to show that **A** and **S** computations do terminate.

Unfortunately, the fact that aliasing and support information is represented in the store has hidden the importance of operational semantics. Because we use a denotational form for the standard semantics, the soundness theorem for support sets is not as strong as it could be. A stronger statement would be that "variables outside the support set of an expression are never accessed *during* the evaluation of the expression". Instead, we can only show that the variables outside the support set are unchanged *after* the evaluation. This loss of operational information is inherent to the denotational semantics. For instance, the effect on the store of an expression such as $x \leftarrow x - 1 \ ; \ x \leftarrow x + 1$ is the identity, according the denotational semantics. However, our scheme safely recognizes this expression as impure.

Chapter 5

Data structures

This chapter covers the extension of aliasing and side effect analysis to a language with data structures. By *data structure* we mean a data object constructed as an aggregate of other data objects [18]. We are not concerned with abstract data types, where the definition of the data type and the operations to manipulate it are encapsulated so that the user need only to know the specification of the data type, not its implementation. From the point of view of aliasing and side effect analysis, the major problems associated with user-defined types come from the use of pointers and dynamic allocation. In fact, mechanisms for data abstraction make the task of static analysis easier by providing a compile-time structuring tool from which to infer information.

So far, only simple data types (integer, boolean, real, etc.) were allowed. Allocation was done through variables declared local to an expression, and deallocation was automatic when the evaluation of the expression terminated. Let us call *local* *variables* those variables whose lifetime is linked to the execution of an expression. The extension to data types of fixed size (arrays, records), for which local allocation is still possible, is the first step. The analyses described in chapter 3 apply directly to a language with local data structures, but yield information whose granularity is the same as that of the data structures involved. For a finer analysis, some additional work is required. Records can be handled easily, but arrays are difficult to partition into smaller structures because values of index expressions may not be easy to estimate at compile-time.

In addition to (or in place) of local variables, languages may also offer a programmer-controlled storage management option. The programmer can allocate and free storage for data structures during the execution of the program. We will refer to this form of storage allocation as *dynamic allocation* [1]. To link together dynamically allocated objects, the programmer uses and stores pointers (l-valued expressions). Lists, trees, stacks, and queues are examples of variable-size data structures which can be implemented with linked storage representation. These features will be modelled in the abstract syntax by a function $alloc(t)$, which allocates enough space to store an object of type t. Pointers come for free by lifting the restrictions on the storage of l-values. We will not be concerned with the garbage collection of inaccessible storage space.

With the introduction of pointers, the definition of the function **A** ceases to be valid. In fact, aliasing and side effects become flow-dependent properties, and the functions must be modified in consequence. We add an abstraction of the store,

[1] This is a misnomer, since technically, local variables are also dynamically allocated.

called the alias store, and split **A** into two functions, as for the standard semantic functions (sections 5.3 and 5.4). We will simulate dynamic allocation with a similar mechanism in the alias store, except in the context of recursive functions where such a scheme would lead to an infinite computation. We discuss various solutions to keep the approximate allocations within static bounds in section 5.5.

Although the aliasing function **A** is replaced by a pair of functions whose signatures and computations are much more elaborate, the support function **S** is hardly affected by these changes. This is due to the fact that the flow-dependency character of **S** is entirely captured in the new aliasing functions.

Section 5.7 compares our approach with the dataflow algorithms. The abstract interpretation approach to static analysis of operational properties provides a general framework in which we can express dataflow algorithms as special cases of our solution. An exact algorithmic comparison is hindered by lack of a common cost function, but the relative accuracy of the various solutions can be estimated.

5.1 Fixed-size data structures

From a semantic point of view, fixed-size data structures are finite functions, which, given an appropriate selector, return a value. For abstract interpretation, the finite functions fall into two classes. The ones where the domain of the finite function consists of a list of constants with no operations (Pascal's record type for example) can be directly incorporated in the alias base domain, as finite functions. It is just as if the non-standard domain of computation included these data structures.

However, arrays do not fall into this category, since the finite functions characterizing arrays have non-trivial domains. Estimating array index expressions requires the full strength of the standard semantics, and therefore, arrays cannot be adopted "as is" in the abstract interpretation. The key problem is the naming of aliases: how do we designate partitions of an array, so that we can do a static estimate of the aliases of array expressions. There is a direct tradeoff here: the domain of alias values must contain enough information, but not so detailed as to prevent the static computation of alias expressions. Alias values are determined by the refinements of arrays of interest to the programmer: these are most often linear subspaces of the indices space of values, such as rows, columns, diagonals, or blocks. Other partitions of the indices sets are sometimes useful, for instance in red/black algorithms. It is possible to define a domain which describes linear subspaces of N^n, and from this, to build an adequate alias domain for refinements of arrays. The alias semantic function is then defined in terms of another semantic function which estimates array index expressions. However, the usual applications of aliasing analysis cannot make use of such alias values. For instance, to determine whether two alias estimates are disjoint may involve determining whether the intersection of linear spaces is empty. Of course this is decidable, but it is not an inexpensive scheme suitable for static analysis.

The inability to define a fine-grain aliasing analysis for arrays is not very encouraging for interference analysis, an important tool for parallelizing compilers. The basic issue is that alias and support analyses provide too much information, and of a sort that is difficult to compute. Interference analysis needs to know facts of the form $e_1 \# e_2$, where $\#$ is a binary predicate over expressions which means that the evaluations of the two expressions do not interfere. For instance, two consecutive entries of an array do not interfere: $A[i] \# A[i+1]$. Aliasing analysis as developed here attempt to estimate the values taken by i, and then shows that the corresponding storage locations are distinct. But all that is necessary to know is that i and $i+1$ cannot be equal, whatever the value of i is.

There is still a hope to integrate interference analysis in an abstract interpretation context, by making use of negative information as is done by Plotkin in the T^ω universal domain [17], but this is beyond the scope of this thesis.

On the other hand, aliasing and side effect analysis can be defined in the presence of linked data structures, as we show in the rest of the chapter.

5.2 Language extensions

We present extensions of the abstract syntax to accommodate data structures and dynamic allocation. These extensions invalidate the analyses devised previously for alias estimation.

5.2.1 Language extensions

First, the restrictions on pointers are lifted. We allow types of the form **ref ref** t, where t itself can be any non-functional type, including a ref type. Storage of l-values in variables is legal too.

Next, we address the problem of runtime memory allocation, by extending the language with two additional constructs:

$$e ::= \ldots \mid alloc(t) \mid nil \mid \ldots$$

$alloc(t)$ allocates storage space for a data object of type t and returns a pointer of type **ref** t. The l-valued constant nil is the polymorphic pointer of Pascal. We must also extend the type system to allow recursive types. We include a product construct and the corresponding expressions:

$$t ::= \ldots \mid t_1 \times t_2 \mid \ldots$$

$$e ::= \ldots \mid \langle e_1, e_2 \rangle \mid e.1 \mid e.2 \ldots$$

An element of product type is a pair (enclosed in angle brackets) and the first or second component of a pair is selected using an infix "." operator. Recursive type expressions are embedded in a binding construct:

$$e ::= \ldots \mid \textbf{letrectype}\ id = t\ \textbf{in}\ e \mid \ldots$$

This construct binds the recursive type defined by the equation $id = t$ to id within the scope of expression e. The other constraints on t are that t cannot contain an arrow (since functions are not storable values) and must contain at least a product (to rule out meaningless recursive definitions such as **letrectype** $t = t\ \ldots$ or **letrectype** $t =$ **ref** $t\ \ldots$). The typing rules for these constructs can be found in appendix C.

For instance, the definition of a $list$ type uses the recursive type binding contruct and the product type:

$$\textbf{letrectype}\ list = int \times \textbf{ref}\ list\ \textbf{in}\ \ldots$$

This declaration means that a list cell is a pair containing an integer value and a pointer to the next cell. A recursive function creating a list with n elements can then be defined as follows:

$$\begin{array}{l} \textbf{letrec}\ create\_list = \lambda n : int.\ \textbf{if}\ n = 0\ \textbf{then}\ nil \\ \qquad\qquad \textbf{else let}\ cell = alloc(list)\ \textbf{in}\ cell.2 \leftarrow create\_list(n-1)\ ;\ cell\ \textbf{ni} \\ \textbf{in}\ \ldots \end{array}$$

The functions *create_list* allocates a list of length n and returns the head of the list.

5.2.2 Consequences for alias analysis

It is sufficient to lift the restrictions on pointers to invalidate the definition of **A**, as the following example demonstrates:

$$\begin{array}{l} \textbf{new}\ x : \textbf{ref}\ int\ \textbf{in} \\ \quad \textbf{new}\ y : \textbf{ref}\ \textbf{ref}\ int\ \textbf{in} \\ \qquad x \leftarrow 0\ ;\ y \leftarrow x\ ;\ y\!\uparrow\ ;\ \ldots \end{array}$$

In the inner **new** expression the value of $y\!\uparrow$ is the address of x – in other words, $y\!\uparrow$ is an alias for x. But its alias estimate, according to the definition of **A**, is $\langle\emptyset, \underline{ground}\rangle$. We must modify **A** to reflect the possibility that the result from an assignment or a dereference may now be an l-value. Recall that we defined the alias pairs of the imperative expressions to be

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

$$\mathbf{A}[\![e_2\!\uparrow]\!]aenv = \langle\emptyset, \underline{ground}\rangle \qquad \textit{(old definition)}$$

In the case of an assignment, it is easy to fix the definition:

$$\mathbf{A}[\![e_1 \leftarrow e_2]\!]aenv = \mathbf{A}[\![e_2]\!]aenv \qquad \textit{(new definition)}$$

For a dereference, the alias set depends on the previous assignments to the location being dereferenced. This information is not part of the alias environment, since an environment usually defines the compile-time constant identifiers. So, we need a store, or more exactly, an abstraction of the store used in the standard semantics, where only the pointer relationships are represented. In section 5.3, we will define the structure of the alias store in detail.

5.2.3 Consequences for alias information

The presence of pointers invalidates another basis of the analyses of the previous chapters. It is not true anymore that an expression may access only the variables visible in its scope. A counterexample is the expression:

$$\begin{array}{ll} \textbf{new } x : \textbf{ref } \mathit{int},\ y : \textbf{ref ref } \mathit{int} \textbf{ in} & (1) \\ \quad y \leftarrow x; & (2) \\ \quad \textbf{new } x : \textbf{ref } \mathit{int} \textbf{ in} & (3) \\ \qquad (\textbf{if } p \textbf{ then } x \textbf{ else } y\uparrow) \leftarrow 0 & (4) \end{array}$$

The expression on the left hand side of the assignment of line 4 may alias to either of the two variables x, declared on lines 1 and 3. It is not sufficient to require variables to be bound to distinct identifiers, since a function with a local variable may be invoked twice in the same activation (see the expression page 35).

Therefore, we cannot use the names of the variables to denote the portions of the store allocated to them. This is exactly why the notion of location was introduced in denotational semantics [22]. We will define an approximation of locations, by using a counter as an abstraction of the free list. When allocating new space, whether for a local variable through a **new** declaration, or for a dynamic variable by using $alloc(t)$, we use the value of the counter as the location of the variable. The counter is then incremented, so that allocated variables can all be distinguished.

$$Loc_A = Int$$

$$AFreeList = Int$$

5.3 The alias store

The alias store is an approximation of the "real" store, in the sense that it contains less information. We want the alias store to reflect information of the form "variable x may point to variables y or z". If a variable is not a pointer, its value can be ignored; all r-values are lumped into one constant, $\underline{rvalue}$.

Let us define the alias store domain as:

$$Store_A = (Loc_A \rightarrow (\underline{rvalue} + \mathcal{P}(Loc_A))) \times AFreeList$$

The first factor is the finite map describing the content of the store. To each location is associated either the constant $\underline{rvalue}$ (if the content of the location is not a pointer) or the set of locations possibly referred to. The domain structure is induced by the one of the base domain and the domain constructs.

We now define operations over $Store_A$, to approximate the operations over the standard semantics store. The initial alias store is $\langle \varepsilon,\ 0 \rangle$, i.e. the function undefined everywhere, and no location has already been allocated. The traditional operations over a store – allocation, deallocation, updating, and dereferencing – have the same signatures as in the standard semantics.

$Alloc_A : Store_A \rightarrow (Store_A \times Loc_A)$

Allocation of new space is done by extending the domain of the function with a new location l:

$Alloc_A(\langle s, c \rangle) = \langle\ \langle s[c \leftarrow \underline{unused}], c+1 \rangle, c\ \rangle$

An assignment in the alias store involves assigning a set V, the aliases of the right hand side, to the range of L, all the possible aliases of the left hand side.

$Update_A : Store_A \rightarrow \mathcal{P}(Loc_A) \rightarrow \mathcal{P}(Loc_A) \rightarrow Store_A$

$Update_A(\langle s, c \rangle, L, V) = \langle s[L \leftarrow V], c \rangle$

Note that the right hand side is the pointwise extension of the finite function updating construct:

$$(f[X \leftarrow V])(y) = \begin{cases} f(y) & y \notin X \\ V & y \in X \end{cases}$$

Dereferencing a variable in the standard semantics translates into dereferencing the set of possible aliases in the alias semantics, and therefore the abstract dereferencing operator takes a set of identifiers as its second argument:

$Deref_A : Store_A \rightarrow \mathcal{P}(Loc_A) \rightarrow \mathcal{P}(Loc_A)$

$Deref_A(\langle s, c \rangle, L) = s(L)$

When deallocating a variable, it must be removed from the domain of the function:

$$DeAlloc_A : Store_A \rightarrow Loc_A \rightarrow Store_A$$

$$DeAlloc_A(\langle s, c\rangle, l) = \langle s \backslash l, c\rangle$$

Finally, since $Store_A$ is a domain, we specify a partial order:

$$\langle s_1, c_1\rangle \sqsubseteq \langle s_2, c_2\rangle \iff (dom(s_1) \subseteq dom(s_2))\ and\ (\forall l \in dom(s_1)\ ,\ s_1(l) \subseteq s_2(l)\ )$$

The partial order induces a join operator:

$$(\langle s_1, c_1\rangle \sqcup \langle s_2, c_2\rangle)(l) = \langle \left\{ \begin{array}{ll} s_1(l) \cup s_2(l) & l \in dom(s_1),\ l \in dom(s_2) \\ s_1(l) & l \in dom(s_1),\ l \notin dom(s_2) \\ s_2(l) & l \notin dom(s_1),\ l \in dom(s_2) \end{array} \right\}, max(c_1, c_2)\rangle$$

The join of two stores represents the "union" of the stores resulting from a conditional expression. The two stores may not have the same domain, because we allow dynamic allocation. The free list counter is set to the maximum of the two stores' counters, to prevent reallocation of space used by only one of the alternative branches.

The dereferencing of pointers in the alias store must also be approximated when we are in the context of a recursive call. We define an operation on the alias store called *reach*, which returns the set of variables that can be reached by following pointer relations from the argument.

$$reach : Loc_A \rightarrow Store_A \rightarrow \mathcal{P}(Loc_A)$$

$reach(l, store)$ is the minimal set satisfying:

$$l \in reach(l, store)$$

$$store_1(l) \subseteq reach(l, store)$$

$$y \in reach(l, store) \Rightarrow store_1(y) \subseteq reach(l, store)$$

reach is an idempotent function:

$$reach(reach(l, store), store) = reach(l, store)$$

In other words, if we interpret the function component of the store as the adjacency list of a directed graph, $reach(l, store)$ is the set of nodes of the largest connected subgraph with node l as its unique source.

5.4 The $\mathbf{A_e}$ and $\mathbf{A_s}$ functions

The definition of **A** must be entirely rewritten to include the alias store computation. We adopt a presentation similar to that of $\mathbf{M_e}$ and $\mathbf{M_s}$, the standard semantics functions, with one function describing the alias value of an expression, and the second one the effect on the alias store of the evaluation of the expression. The new functions are indexed by **e** and **s**. Their signatures match those of the standard semantic functions:

$$\mathbf{A_e} : Exp \rightarrow Env_A \rightarrow Store_A \rightarrow D$$

$$\mathbf{A_s} : Exp \rightarrow Env_A \rightarrow Store_A \rightarrow Store_A$$

The domains are built along the same recursive structure as the one used for **A** and **S**. Just as in the standard semantics, a store is an additional parameter of the functional terms.

$$Env_A = Id \to D$$

$$D = \mathcal{P}(Loc_A) \times Val$$

$$Val = \{\underline{ground}\} + (D \to Store_A \to (D \times Store_A))$$

Except for the additional store argument, the $\mathbf{A_e}$ function is almost identical to the $\mathbf{A}$ function. The $\mathbf{A_s}$ function resembles the $\mathbf{M_s}$ function, in that each subexpression evaluation contributes intermediary stores, labelled *astore'*, *astore''*, etc. As in the standard semantics, the imperative constructs are the only ones to make use of the alias store access functions. The definitions of $\mathbf{A_e}$ and $\mathbf{A_s}$ follow, except for the dynamic allocation clause, treated in the next section.

<u>Constants and identifiers</u>

$\mathbf{A_e}[\![c]\!]aenv\ astore = \langle \underline{rvalue}, \underline{ground} \rangle$

$\mathbf{A_s}[\![c]\!]aenv\ astore = astore$

$\mathbf{A_e}[\![nil]\!]aenv\ astore = \langle \emptyset, \underline{ground} \rangle$

$\mathbf{A_s}[\![nil]\!]aenv\ astore = astore$

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

$\mathbf{A_s}[\![x]\!]aenv\ astore = astore$

Constants are non-functional r-values, except for *nil*, the only l-valued constant. The aliases of identifiers are saved in the environment. Both constants and identifiers are pure expressions and their evaluation does not affect the alias store.

Conditional and sequential evaluations

In the case of a conditional evaluation, we must insure that the two branches of the conditional perform allocations on distinct areas of the stores. We prevent the evaluation of e_3 from doing allocations in the same area as e_2 by passing the updated value of the free list counter, c'.

$\mathbf{A_e}[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] aenv\ astore =$

$$\mathbf{A_e}[\![e_2]\!] aenv\ \langle s, c\rangle \sqcup \mathbf{A_e}[\![e_3]\!] aenv\ \langle s, c'\rangle$$

$\mathbf{A_s}[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] aenv\ astore =$

$$\mathbf{A_s}[\![e_2]\!] aenv\ \langle s, c\rangle \sqcup \mathbf{A_s}[\![e_3]\!] aenv\ \langle s, c'\rangle$$

where

$$\langle s, c\rangle = \mathbf{A_s}[\![e_1]\!] aenv\ astore$$

$$\langle s', c'\rangle = \mathbf{A_s}[\![e_2]\!] aenv\ \langle s, c\rangle$$

$\mathbf{A_e}[\![e_1\ ;\ e_2]\!] aenv\ astore = \mathbf{A_e}[\![e_2]\!] aenv\ astore'$

$\mathbf{A_s}[\![e_1\ ;\ e_2]\!] aenv\ astore = \mathbf{A_s}[\![e_2]\!] aenv\ astore'$

where

$$astore' = \mathbf{A_s}[\![e_1]\!] aenv\ astore$$

The equations for sequential evaluation are consistent with the standard semantics: the value of the first expression is discarded, but its side effects are recorded in an intermediary store.

Abstraction and application

$\mathbf{A_e}[\![\lambda x : t.e]\!] aenv\ astore =$

$$\langle \emptyset\ ,\ \lambda u.\lambda s.\langle \mathbf{A_e}[\![e]\!] aenv[x \leftarrow u]\ s, \mathbf{A_s}[\![e]\!] aenv[x \leftarrow u]\ s\rangle\rangle$$

$\mathbf{A_s}[\![\lambda x : t.e]\!] aenv\ astore = astore$

$\mathbf{A_e}[\![e_1\ e_2]\!] aenv\ astore = (fun\ arg\ astore'')_1$

$\mathbf{A_s}[\![e_1\ e_2]\!] aenv\ astore = (fun\ arg\ astore'')_2$

where

$fun = (\mathbf{A_e}[\![e_1]\!] aenv\ astore)_2$

$astore' = \mathbf{A_s}[\![e_1]\!] aenv\ astore$

$arg = \mathbf{A_e}[\![e_2]\!] aenv\ astore'$

$astore'' = \mathbf{A_s}[\![e_2]\!] aenv\ astore'$

There is nothing new here. The abstraction mechanism follows the one designed for simple aliases. The alias store must be passed as an additional parameter, just as for the standard semantics. For an application, the functional term is evaluated first, the side effects of that evaluation producing an intermediary store $astore'$. Then the argument is evaluated in the context of $astore'$, and its side effects recorded in $astore''$. Finally, the application is performed.

Imperative constructs

Assignments in the alias store must take into account the possibility that both sides of the assignment expressions may be sets of identifiers. On the other hand, functional variables are not permitted, and therefore an alias pair associated with a variable will always be of the form $\langle \textit{set of identifiers}\ , \underline{\textit{ground}} \rangle$.

$\mathbf{A_e}[\![\mathbf{new}\ x : t\ \mathbf{in}\ e]\!] aenv\ astore = \mathbf{A_e}[\![e]\!] aenv'\ astore'$

$\mathbf{A_s}[\![\mathbf{new}\ x : t\ \mathbf{in}\ e]\!] aenv\ astore = DeAlloc_A(\mathbf{A_s}[\![e]\!] aenv'\ astore', l)$

where

$aenv' = aenv[x \leftarrow \langle l, \underline{ground} \rangle]$

$\langle astore', l \rangle = Alloc_A(astore)$

$\mathbf{A_e}[\![e\uparrow]\!] aenv\ astore = Deref_A(astore', alias)$

$\mathbf{A_s}[\![e\uparrow]\!] aenv\ astore = astore$

where

$astore' = \mathbf{A_s}[\![e]\!] aenv\ astore$

$alias = (\mathbf{A_e}[\![e]\!] aenv\ astore)_1$

$\mathbf{A_e}[\![e_1 \leftarrow e_2]\!] aenv\ astore = \mathbf{A_e}[\![e_2]\!] aenv\ astore'$

$\mathbf{A_s}[\![e_1 \leftarrow e_2]\!] aenv\ astore =$

$Update_A(astore'', (\mathbf{A_e}[\![e_1]\!] aenv\ astore)_1, (\mathbf{A_e}[\![e_2]\!] aenv\ astore')_1)$

where

$astore' = \mathbf{A_s}[\![e_1]\!] aenv\ astore$

$astore'' = \mathbf{A_s}[\![e_2]\!] aenv\ astore'$

5.5 Dynamic allocation in a recursive context

In the abstract domain, we cannot simulate dynamic allocation in a recursive context since we do not know how many times a recursive function may call itself. This section discusses two schemes to simulate a conservative estimate of dynamic allocation, so that fixed point computations are feasible. The formal treatment of recursive functions is the object of the next chapter.

5.5.1 A rough approximation

As a first approach, we approximate the multiple allocations caused by one syntactical occurrence of *alloc* with one allocation. The rationale behind this approximation is that a single recursive procedure is often used to generate a whole data structure. In the alias store, the whole data structure is merged into one cell.

To distinguish the different syntactical occurrences of *alloc* in the expression being analyzed, each occurrence is decorated with an index.

$$\mathbf{A_e}[\![alloc^i(t)]\!]aenv\ astore = \langle\{\text{"}alloc^{i}\text{"}\}, \underline{ground}\rangle$$

$$\mathbf{A_s}[\![alloc^i(t)]\!]aenv\ astore = astore[\text{"}alloc^{i}\text{"} \rightarrow \{\text{"}alloc^{i}\text{"}\}]$$

The domain of alias locations, Loc_A, should now be defined by $Loc_A = Int + Id$. To return to the *create_list* function defined on page 75, its alias expression is :

$\mathbf{A_e}[\![create\_list]\!]aenv\ astore =$

$$\langle \emptyset, \lambda u.\lambda s.\langle \{\text{"}alloc^1\text{"}\}, s[alloc^1 \leftarrow \text{"}alloc^1\text{"}]\rangle, \underline{ground}\rangle\ \rangle$$

$\mathbf{A_s}[\![create\_list]\!]aenv\ astore = astore$

5.5.2 A finer approximation

Unfortunately, substituting static allocation in the alias semantics for dynamic allocation in the standard semantics means that all the data structures created by calls to a recursive function f are merged into one cell.

Consider for instance a program where the functions *create_list* and *tail_of_list* are part of a data type to manipulate lists.

new p, q : **ref** *list*

  in

    $p \leftarrow create\_list(3);$

    $q \leftarrow create\_list(2);$

    ...

The two lists created by this program (see figure 5.1(a)) are treated as a single cell of the alias store, as shown in figure 5.1(b), when in fact we would like to have the situation of figure 5.1(c). The collapsing of a linked data structure into one cell of the alias store is necessary, since we cannot do an estimate of the size or the kind of data structure at compile-time. However, a useful scheme ought to be able to distinguish distinct data structures. We can modify the purely static allocation scheme designed above to obtain the situation of figure 5.1(c).

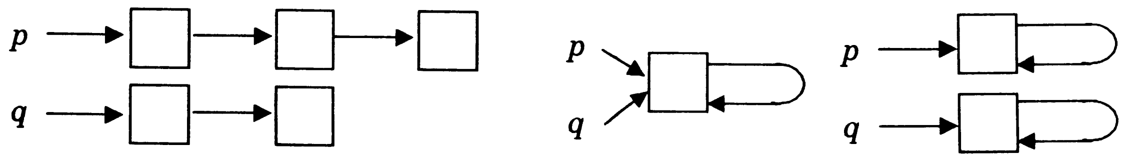

Figure 5.1: The real store, a rough approximation, a finer approximation

Recall that the objection to dynamic allocation in recursive functions is that we cannot estimate the number of recursive calls. But if we define the fixed points of recursive functions so that dynamic allocation in recursive calls is approximated by only one call to $alloc(t)$, the total number of calls to $alloc(t)$ can still be determined statically. This scheme yields an alias store where a whole data structure created by a recursive function is still merged into one cell, but distinct data structures can be differentiated.

$$\mathbf{A_e}[\![alloc(t)]\!] aenv\ astore = \langle l, \underline{ground} \rangle$$

$$\mathbf{A_s}[\![alloc(t)]\!] aenv\ astore = astore'$$

where

$$\langle astore', l \rangle = Alloc_A(astore)$$

The main shortcoming of this approach is that we cannot do any analysis inside a linked data structure. We are limited here by the low-level primitives used to define data types: pointers and an explicit allocation function. For instance, we cannot determine that the left and right children of a binary tree node are not aliases of each other. Such an analysis would be best undertaken in a language with abstract data types.

5.6 The S function

As mentioned in the introduction to this chapter, the support set computation is hardly affected by the presence of data structures.

The function **S** of section 3.4 was defined inductively over the structure of expressions using some clauses of the function **A**. Here, **A** is replaced by $\mathbf{A_e}$ in the clauses for assignment and dereferencing.

$$\mathbf{S}[\![e_1 \leftarrow e_2\,]\!]senv \;=\;$$

$$\langle\; \mathbf{S}_1[\![e_1]\!]senv \cup \mathbf{S}_1[\![e_2]\!]senv \cup (\mathbf{A_e}[\![e_1]\!]aenv\ astore)_1\;,\; \underline{ground}\;\rangle$$

$$\mathbf{S}[\![e\uparrow]\!]senv \;=\; \langle\; \mathbf{S}_1[\![e]\!]senv \cup (\mathbf{A_e}[\![e]\!]aenv\ astore)_1\;,\; \underline{ground}\;\rangle$$

The other clauses defining **S** are defined in terms of **S** – that is, $\mathbf{S}[\![e]\!]senv$ is defined solely in terms of $\mathbf{S}[\![e_i]\!]senv$, where the e_i are subexpressions of e – and thus are still correct. The base domain is $\mathcal{P}(Loc_A)$ instead of $\mathcal{P}(Var)$. No other changes are necessary.

5.7 Comparison with dataflow algorithms

Since the abstract interpretation approach can handle a larger class of languages than the dataflow algorithms can, our approach is clearly more general. The most sophisticated dataflow algorithms do interprocedural analysis in the presence of pointers, reference parameters and recursion. They do not handle higher-order functions, dynamic allocation or l-valued expressions, except for Weihl's method which treats higher-order functions, but is less accurate than ours, as will be shown further down. The other dataflow analyses rely on knowledge of the call graph of a program, and therefore cannot be extended to higher-order languages.

For an algorithmic comparison, the alias and support functions can be simplified to cover the same class of languages as is normally considered for dataflow analyses. The language is imperative, so there is a distinction between statements and expressions and in particular, expressions are pure. The abstract syntax becomes

$s ::= \ldots \mid id \leftarrow e \mid \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \mid \textbf{let}\ f = \lambda x : t.e\ \textbf{in}\ s \mid id(e) \mid s_1;\ s_2$

$e ::= c \mid id \mid id\!\uparrow \mid id(e)$

We had defined the stratified domains for alias and support functions

$$D_A = \mathcal{P}(Loc_A) \times (\{\underline{ground}\} + D_A \rightarrow Store_A \rightarrow (D_A \times Store_A))$$

$$D_S = \mathcal{P}(Loc_A) \times (\{\underline{ground}\} + D_S \rightarrow D_S)$$

Without higher-order functions, the domains collapse to simple sums:

$$D_A = \mathcal{P}(Loc_A) \times (\{\underline{ground}\} + \mathcal{P}(Loc_A) \rightarrow Store_A \rightarrow (\mathcal{P}(Loc_A) \times Store_A))$$

$$D_S = \mathcal{P}(Loc_A) \times (\{\underline{ground}\} + \mathcal{P}(Loc_A) \rightarrow \mathcal{P}(Loc_A))$$

Since functions cannot be direct aliases, the first component of their alias pairs is the empty set; similarly, non-functional expressions have a trivial second component ($\underline{ground}$). We can thus distribute the sum over the product and simplify:

$$D_A = \mathcal{P}(Loc_A) + (\mathcal{P}(Loc_A) \rightarrow Store_A \rightarrow (\mathcal{P}(Loc_A) \times Store_A))$$

In the same manner, since functions can be denoted only by identifiers, their support set is empty, and we need to keep track only of the second term of the **S** function:

$$D_S = \mathcal{P}(Loc_A) + (\mathcal{P}(Loc_A) \rightarrow Store_A)$$

The signatures of **A** and **S** are unchanged:

$$\mathbf{A} : exp \rightarrow Env_A \rightarrow Store_A \rightarrow D_A$$

$$\mathbf{S} : exp \rightarrow Env_S \rightarrow D_S$$

$$Env_A = Id \rightarrow D_A$$

$$Env_S = Id \rightarrow D_S$$

$$Store_A = Loc_A \rightarrow \mathcal{P}(Loc_A)$$

Since expressions are pure, $\mathbf{S}[\![e]\!]senv$ is empty for any e, except for applications of functions. The definitions for the clauses with a simplified syntax become:

Statements

$$\mathbf{A_s}[\![id \leftarrow e]\!]aenv\ astore = Update_A(astore, aenv(id), \mathbf{A}[\![e]\!]aenv)$$

$$\mathbf{S}[\![id \leftarrow e]\!]senv = aenv(id) \cup \mathbf{S}[\![e]\!]senv$$

$$\mathbf{A_s}[\![\mathbf{let}\ f = \lambda x : t.e\ \mathbf{in}\ s]\!]aenv\ astore =$$

$$\mathbf{A_s}[\![e]\!]aenv[f \leftarrow \mathbf{A}[\![\lambda x : t.e]\!]aenv]\ astore$$

Expressions

$$\mathbf{A_e}[\![e\uparrow]\!]aenv\ astore = Eval_A(astore, \mathbf{A}[\![e]\!]aenv)$$

$$\mathbf{S}[\![e\uparrow]\!]senv = \mathbf{A}[\![e]\!]aenv$$

$$\mathbf{A_e}[\![\lambda x : t.e]\!] aenv\ astore = \lambda u.\lambda s.\langle \mathbf{A}[\![e]\!] aenv[x \leftarrow u]s, \mathbf{A_s}[\![e]\!] aenv[x \leftarrow u]\ s\rangle$$

$$\mathbf{A_e}[\![id(e)]\!] aenv\ astore = \emptyset$$

$$\mathbf{A_s}[\![id(e)]\!] aenv\ astore = aenv(id)(\mathbf{A_e}[\![e]\!] aenv\ astore)$$

$$\mathbf{S}[\![id(e)]\!] senv = senv(id)(\mathbf{S_2}[\![e]\!] senv)$$

The algorithm induced by the above definitions yields a more precise analysis than Weihl's algorithm. Because he works with relations, he must ignore control-flow information (procedures are sets of statements). Weihl himself points out the loss of accuracy in the higher-order computation incurred by ignoring "call-site" information. His algorithm cannot handle accurately sequential evaluation, and computes the aliasing and side effect information resulting of the consecutive execution of statements in any possible order within one procedure. For instance, where we write:

$\mathbf{A_s}[\![s_1; s_2]\!] aenv\ astore = \mathbf{A_s}[\![s_2]\!] aenv\ (\mathbf{A_s}[\![s_1]\!] aenv\ astore)$

his algorithm computes :

$\mathbf{A_s}[\![s_1; s_2]\!] aenv\ astore \sqcup \mathbf{A_s}[\![s_2; s_1]\!] aenv\ astore$

Barth and Banning both work with languages where side effects and aliases are flow-independent predicates. There is no need for an alias store in that case, only for the abstract free list. The definitions of the abstract interpretations can be further simplified.

Comparing the asymptotic cost of these algorithms may be a futile exercise, since all of them must rely on an efficient implementation to be of any use. The dataflow algorithms encode the call graph in relations, represented as boolean matrices. The algorithms to compute aliasing information are matrix algorithms performing transitive closures, whose cost depend mainly on the size of the matrix, typically the product of the number of procedures by the number of identifiers. These algorithms can be optimized by using the sparse character of the matrix. Overall, an accurate estimate of the cost is implementation dependent. In terms of number of operations, the tally is in favor of the abstract interpretation approach, because the algorithms implicitely defined by the abstract interpretation functions perform minimal computations: the evaluation of applications reproduces exactly the applications that would be performed in the standard semantics domain (with the exception of recursive calls). The equivalent computation in dataflow algorithms is a transitive closure of some matrix, where one would compute only the non-zero entries. A good sparse matrix transitive closure algorithm may thus perform as well as the abstract interpretation in the number of steps. On the other hand, the cost of operations is cheaper for the dataflow approach. It would take a very efficient implementation of aliases values to make the cost of application comparable to the cost of boolean operations.

5.8 Discussion

To summarize, the aliasing relations between the locations are computed statically in the alias store. Dynamic allocation in a static context is simulated exactly, but linked data structures created by calls to recursive functions are collapsed into single cells in the alias store.

The loss of accuracy incurred by the functions is not just due to the approximations introduced by conditional expressions and allocation in a recursive context. Another source of loss is caused by the use a single alias store to collect the side effects of multiple execution paths. A more costly approach would be to define a collecting abstract interpretation, and to use a set of stores with exact alias information. Such approach could be advantageous for interference analysis. For instance, consider the expression

$$\textbf{if } p \textbf{ then } x \leftarrow a \, ; \, y \leftarrow b \textbf{ else } y \leftarrow a \, ; \, x \leftarrow b$$

From our analysis, we cannot infer that x and y do not interfere, because their alias sets will be identical. The computation of fixed points may introduce additional approximations, but this is discussed in the next chapter.

We will not state the soundness theorem for $\mathbf{A_e}$ and $\mathbf{A_s}$, but the similarity between the standard semantics and the abstract interpretation functions should provide an intuitive argument in favor of the correctness of our inference schemes.

Chapter 6

Computing fixed points

In previous chapters, the questions pertaining to computations of fixed points in the various abstract semantics domains were given little attention. Although these domains were carefully defined to ensure that the fixed points do exist, we brushed the algorithmic issues under the rug.

This chapter addresses computations of fixed points in alias analysis. For a self-contained presentation, the first section is a short explanation of the necessity of fixed point computations. The methods developed for strictness analysis are discussed in section 6.2. We show in section 6.3 that some of them can be applied directly to alias analysis, but that the cost is prohibitive because the base domain is more complex. Section 6.4 presents an alternative for the **A** function. In the first-order case, the recursive equations defining the fixed point present patterns that can be analyzed to infer a lower bound of the fixed point. By working in a smaller lattice of values, the fixed points of n mutually recursive functions can be estimated in $n + 1$ steps. The same ideas are then applied to the more complex functions $\mathbf{A_e}$ and $\mathbf{A_s}$ in section 6.5. Section 6.6 contains a discussion on the correctness and complexity costs.

6.1 Necessity of fixed point computations

In all the abstract interpretations defined in this work, the semantics of recursive functions is defined in terms of least fixed points of a functional, just as it is done for the standard semantics [22]. Consider for example the recursive function *factorial*:

letrec

$$factorial = \lambda n : int \text{ .}\mathbf{if}\ \ n = 0\ \mathbf{then}\ \ 1\ \mathbf{else}\ \ n * factorial(n - 1)$$

in $\cdots$

The computation of the A-image of *factorial* results in a recursive equation, due to the recursive call to itself in the body of the function:

$$\mathbf{A}[\![factorial]\!]aenv = \langle\emptyset, \lambda u.\langle\emptyset, \underline{ground}\rangle \sqcup \mathbf{A}_2[\![factorial]\!]aenv[n \leftarrow u]u\rangle$$

In this particular case, the A-image of the recursive function can be found without solving the recursive equation: since the type of *factorial* is $int \rightarrow int$, the result of a call to *factorial* cannot be an alias and

$$\mathbf{A}[\![factorial]\!]aenv = \langle\emptyset, \lambda u.\langle\emptyset, \underline{ground}\rangle\rangle$$

When parameters are passed by reference or the results are l-values, the situation is not so simple. Consider the recursive function

letrec

$scan = \lambda A : \textbf{ref array } 1..n \textbf{ of } int\ .\lambda i : int\ .$

$\textbf{if } A[i]\!\uparrow = 0 \textbf{ then } A[i] \textbf{ else } scan(A)(i+1)$

in $\cdots$

We note, parenthetically, that we follow the usage in programming languages for array selector functions: when selecting an element of an l-valued array expression, we return the address of the element ($A[i]$). The function call $scan(A)(i)$ returns the address of the first zero-valued entry in array A at or after $A[i]$. Its aliasing behavior is described by the recursive equation:

$$\mathbf{A}[\![scan]\!]aenv = \langle \emptyset, \lambda u.\langle \emptyset, \lambda v.u \sqcup (\mathbf{A}_2[\![scan]\!]aenv[A \leftarrow u, i \leftarrow v]u)_2\ v\rangle\rangle$$

A solution of the above equation is:

$$solution_1 = \langle \emptyset, \lambda u.\langle \emptyset, \lambda v.u\rangle\rangle$$

There are other solutions to this recursive equation, for instance:

$$solution_2 = \langle \emptyset, \lambda u.\langle \emptyset, \lambda v.u \sqcup \langle s, \underline{ground}\rangle\rangle\rangle$$

where s is any set of variables. The "best" estimate of the aliasing behavior of $scan$ is the one that gives the smallest set of aliases, and thus we choose for $\mathbf{A}[\![scan]\!]aenv$ the smallest solution of the recursive equation, which is $solution_1$. It is also the least fixed point of the functional

$$F = \lambda f.\mathbf{A}[\![scan]\!]aenv[scan \leftarrow f]$$

since it satisfies the equation

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

In standard semantics, the solution of such recursive equations is obtained by computing an ascending chain $A^{(0)}, A^{(1)}, A^{(2)}$ etc. of more and more refined approximations to $\mathbf{A}[\![scan]\!]aenv$, using the following inductive definition:

$$A^{(0)} = \perp$$

$$A^{(j)} = F(A^{(j-1)})$$

The least fixed point of this chain is $A^{(k)}$, where k is the smallest integer such that $A^{(k+1)} = A^{(k)}$. That is we "unroll" the definition of $\mathbf{A}[\![scan]\!]aenv$ until it is unchanged:

$$\begin{aligned}
A^{(0)} &= \perp \\
A^{(1)} &= \mathbf{A}[\![scan]\!]aenv[scan \leftarrow \perp] \\
&= \langle \emptyset, \lambda u.\langle \emptyset, \lambda v.u \sqcup \perp \rangle\rangle \\
&= \langle \emptyset, \lambda u.\langle \emptyset, \lambda v.u \rangle\rangle \\
A^{(2)} &= \mathbf{A}[\![scan]\!]aenv[scan \leftarrow A^{(1)}] \\
&= \langle \emptyset, \lambda u.\langle \emptyset, \lambda v.u \sqcup ((A^{(1)})_2\, u)_2\, v \rangle\rangle \\
&= \langle \emptyset, \lambda u.\langle \emptyset, \lambda v.u \sqcup u \rangle\rangle \\
&= \langle \emptyset, \lambda u.\langle \emptyset, \lambda v.u \rangle\rangle \\
&= A^{(1)}
\end{aligned}$$

The abstract computations have been defined over domains precisely to ensure that these fixed points exist. Thus, from a purely theoretical perspective, the problem is solved. However, from a practical point of view, there remain two questions:

- can we specify an algorithm to perform fixed point computations ?
- if so, what is the cost of this algorithm ?

The answer to the first question is yes, because we work in finite domains. The depth of an alias expression, that is the number of lambda terms in it, is determined by the type of the expression. There is a lambda term for each arrow in the type. For example, the *scan* function is of type (**ref array** $1..n$ **of** *int*) $\rightarrow$ *int* $\rightarrow$ **ref** *int*, and its alias expression contains two lambda bindings. Therefore, an ascending chain consists of aliasing expressions of identical depth. The base domain component at each level is finite. Therefore, there are only a finite (although large) number of alias expressions of given depth, the ascending chain enumerates elements of a finite domain in increasing order, and eventually two consecutive elements must be equal. Equality between two elements is decidable because the domain is finite.

The second question is answered in the last section of this chapter. Complexity results for strictness analysis suggest that the cost for alias analysis may be high.

6.2 Fixed point computations for strictness analysis

Since recursive constructs in the language are at the root of the need for fixed point computations, this is not the first time the problem of fixed point computations poses itself in an abstract interpretation. We survey briefly the solutions used in strictness analysis, because the framework is similar to the one used here:

the abstract syntax contains higher-order constructs, and the abstract interpretation is defined over parse trees instead of flow graphs (see for instance Hudak [12]).

Hudak uses an iterative algorithm to determine the fixed point corresponding to a recursive construct. The iteration terminates when two consecutive elements of the ascending chain (called strictness pairs) are equal. Because we are in a higher-order context, these strictness pairs may contain functional terms, and thus the decision procedure for equality is non-trivial. The equality of functional terms is based on extensional equality, which means that two functions are said to be equal if and only if they have the same domain and range, and they agree on their domain.

$$f =_e g \iff domain(f) = domain(g) \text{ and } f(x) = g(x)\ \forall\ x \in domain(f)$$

For example,$(x - y)^2 =_e x^2 - 2xy + y^2$ and $x - x =_e 0$. From a computational point of view, extensional equality corresponds to a "black-box" view: functions are determined by their I/O behavior.

Testing for extensional equality of functions defined over finite domains is simple: we enumerate the elements of the domain and evaluate the functions for these elements. For first-order strictness analysis, the strictness abstract interpretation domain is **2**, the two-valued lattice consisting of bottom($\perp$) and top ($\top$). This algorithm works also for higher-order strictness analysis. The domains are larger, but they can still be enumerated. Hudak has shown that in the first-order case, the problem is complete in deterministic exponential time in the number of arguments to the function being analyzed.

In practice, the exponential behavior is rarely achieved, and there are two algorithms to handle the average case efficiently. The first one is due to Clack and Peyton-Jones [5,15], who advocate the use of an explicit representation of the strictness functions in terms of frontiers. The frontier of a strictness function $f : \mathbf{2}^n \rightarrow \mathbf{2}$ is defined as the set of values separating $f^{-1}(\bot)$ from $f^{-1}(\top)$, in the lattice $\mathbf{2}^n$ For a monotonic f, all the values of $f^{-1}(\top)$ are above those of $f^{-1}(\bot)$ and the frontier is the "line" separating these two sets. Functional arguments are represented by a sublattice, which is recursively explored. Functions returning functional results are extended to include the invisible arguments, so that the range domain is still $\mathbf{2}$.

Recently, Young and Hudak [26] observed that because the base domain of strictness values is flat, nested recursive calls can be compressed: $f(f(\bot))$ can safely be replaced by $f(\bot)$, since either $f(\bot) = \bot$ and the two expressions are equal, or $f(\bot) = \top$, and the fixed point is reached. They use this fact to develop an algorithm, called *pending analysis*, which is applicable to first-order recursive monotone boolean functions. Pending analysis directly applies to strictness analysis, via the interpretation of $\mathbf{2}$ as a Boolean set.

Since there is no consensus on what constitutes an average sample of functional programs, the comparative merits of the two approaches in terms of efficiency are not easy to quantify. Let us just repeat that both have exponential behavior in the worst case, but perform well in the average case.

6.3 Computing fixed points with ascending chains

The traditional algorithm to compute fixed points of strictness recursive equations can be carried over to alias analysis without theoretical pitfalls. However, even in the average case, the cost of such an algorithm is prohibitive, as we show now. The cost of finding a fixed point can be estimated as the product of the number of steps required to reach the fixed point and the cost of testing for equality of two alias terms.

6.3.1 Length of iteration

In the case of the function *scan* in section 6.1, the fixed point was reached in two steps. This is generally not true, especially with multiple mutually recursive functions. For n mutually recursive functions, the iterative process may take n steps. Consider the case:

letrec

$h_0 = \lambda i : int.\textbf{if } i = 0 \textbf{ then } x_0 \textbf{ else } h_1(i-1)$

$\ldots$

$h_j = \lambda i : int.\textbf{if } i = 0 \textbf{ then } x_j \textbf{ else } h_{j+1}(i-1)$

$\ldots$

$h_n = \lambda i : int.\textbf{if } i = 0 \textbf{ then } x_n \textbf{ else } h_0(i-1)$

in $\ldots$

The computation of the fixed point environment goes as follows:

$aenv_0 = [h_0 \leftarrow \perp, \cdots, h_n \leftarrow \perp]$

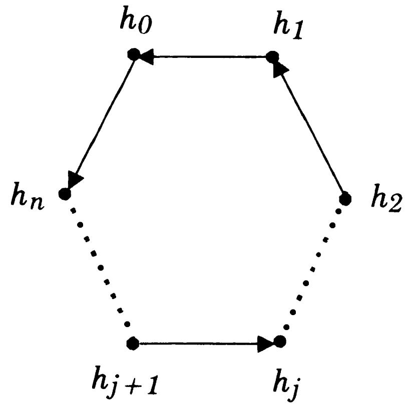

Figure 6.1: Dependency graph for the f_i's

$aenv_1 = [h_0 \leftarrow \mathcal{A}(\{x_0\}), \cdots, h_n \leftarrow \mathcal{A}(\{x_n\})]$

$aenv_2 = [h_0 \leftarrow \mathcal{A}(\{x_0, x_1\}), \cdots, h_n \leftarrow \mathcal{A}(\{x_n, x_0\})]$

The size of the alias sets increases by one at each iteration until:

$aenv_n = [h_0 \leftarrow \mathcal{A}(\{x_0, .., x_n\}), \cdots, h_n \leftarrow \mathcal{A}(\{x_0, .., x_n\})]$

where $\mathcal{A}(S)$ is an abbreviation for $\langle \emptyset, \lambda u.\langle S, \underline{ground} \rangle\rangle$.

This is the worst case for first-order functions. The dependencies between the recursive functions can be expressed in a graph G, with n nodes for the n recursive functions f_i, and an edge from n_i to n_j if f_j appears in the alias expression for f_i. So, for the example above, we have the graph of figure 6.1 page 102. A graph with n nodes cannot contain a path (without a cycle) of length greater than n. Attached to each node is a *load* value, which is a set of identifiers defined below. The iterative process to compute the fixed points is interpreted as an iterative process to compute the maximum load of the nodes of the graph. Initially, we define the load of node i to be:

$$l_i^1 = \mathbf{A}_1[\![f_i]\!]aenv_1$$

To each iteration computing a new environment, we associate node n_i with a new load, defined as

$$l_i^j = l_i^{j-1} \overset{\cup}{incident}\, k\; l_k^{j-1}$$

After at most n iterations, l_i should be maximal for any i, since the load of any node will have propagated to all the nodes of the graph in the same connected component. Therefore, we need at most n iterations to compute the fixed point of example 2.

For a function of k-th order, reaching the fixed point may take $k+1$ steps. An example is the function

$$\textbf{letrec}\ f = \lambda x_1 : \textbf{ref}\ int. \ldots .\lambda x_k : \textbf{ref}\ int.\textbf{if}\ p\ \textbf{then}\ x_1\ \textbf{else}\ \ f(x_2)\ldots(x_k)(a)$$

In this case, it takes $k+1$ iterations before the computation in the alias domain has "unrolled" the permutation of arguments in the recursive call and included all arguments and a in the alias set.

6.3.2 Cost of equality test

Hudak showed that the cost of the equality test for first-order strictness functions is exponential in the number of arguments of the function. The base of the exponential is 2, the size of domain of interpretation, **2**. For alias analysis, the size of the base domain $\mathcal{P}(Var)$ is itself an exponential in the number of global variables. In the presence of higher-order functions, the cost of extensional equality testing becomes prohibitive.

Call $C_e(D)$ the cost of equality testing in set D, and $size(D)$ the cardinality of set D. We have:

$size(D_1 \rightarrow D_2) = size(D_2)^{size(D_1)}$

$C_e(D_1 \rightarrow D_2) = size(D_1) * C_e(D_2)$

For example, extensional equality for the aliasing function f, of type (**ref** $int \rightarrow$ **ref** $int) \rightarrow$ **ref** int, in an environment with five global identifiers, would cost:

$C_e((\mathcal{P}(Var) \rightarrow \mathcal{P}(Var)) \rightarrow \mathcal{P}(Var)) = size(\mathcal{P}(Var) \rightarrow \mathcal{P}(Var)) * C_e(\mathcal{P}(Var))$

$= 32^{32} * C_e(\mathcal{P}(Var))$

This means that we may have to perform around 10^{48} comparisons of set values to decide that two successive iterations in the ascending chain are equal.

6.4 Fixed point computations in finite lattice

This section discusses an alternative method to discover fixed points of recursive equations for the **A** semantics. Instead of blindly computing the fixed point with a chain of approximations, this algorithm examines the type of the function and analyzes the patterns in the recursive equation defining the fixed point to guess its value. In the first-order case, this method discovers the least fixed point. For higher-order functions, the guess can be improved by heuristics, in case it is not the required fixed point. If the heuristics fail, we choose a trivial fixed point, and loose some accuracy in the analysis.

Despite their lenghty computations, the fixed points of all the recursive functions defined so far are simple expressions. This is not a random effect. The alias domain is simple enough that it does not reflect the complexity of recursive computations. Especially in the absence of dynamic allocation, there is little that can be done by recursive functions in terms of aliases, as we show below.

6.4.1 First-order functions

Let start with the first-order case: the type of a recursive function contains only one arrow, or in other words, the argument and the result are non-functional. Furthermore, we are not concerned with functions whose result is not of ref type, since their alias behavior is trivial: any application of such functions results in an r-value, whose alias set is empty.

$$f : t_1 \rightarrow t_2 \ , \ t_2 \neq \mathbf{ref}\ t \quad \Rightarrow \quad \mathbf{A}[\![f]\!]aenv = \lambda u.\langle \emptyset, \underline{ground} \rangle$$

Since there is no dynamic allocation and local variables cannot be exported outside their scope, the result of a function application is either an alias of a global variable or of its argument. Furthermore, if the argument is not passed by reference, the result can only be an alias of a global variable. Because we compute static estimates, the alias values may be a combination of both cases as well.

Let us denote by E, P, and G the alias terms:

$E = \lambda u.\langle \emptyset, \underline{ground} \rangle$

$P = \lambda u.\langle u_1, \underline{ground} \rangle$

$G = \lambda u.\langle s, \underline{ground} \rangle$ where $s \subseteq Var$

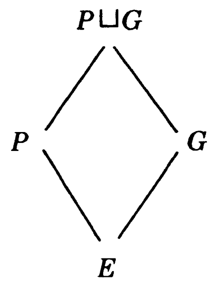

Figure 6.2: The E-P-G lattice

E stands for "empty set", P for "parameter" and G for "global". These three alias terms induce a lattice, described in figure 6.2. For a recursive first-order function f, of type $t_1 \rightarrow t_2$, we have the following possibilities:

$t_2 \neq \mathbf{ref}\ t \Rightarrow \mathbf{A}[\![f]\!]aenv = E$

$t_2 = \mathbf{ref}\ t_2'$

t_1 non ref type $\Rightarrow \mathbf{A}[\![f]\!]aenv = G$

$t_1 = \mathbf{ref}\ t_1' \Rightarrow \mathbf{A}[\![f]\!]aenv = G$ or $\mathbf{A}[\![f]\!]aenv = P$ or $\mathbf{A}[\![f]\!]aenv = P \sqcup G$

The distinction between the 3 possibilities is given by the pattern of the recursive equation defining the fixed point. The most general form the recursive equation can take is:

$$\mathbf{A}_2[\![f]\!]aenv = \lambda u.\langle \mathcal{A}_0 \cup (\mathbf{A}_2[\![f]\!]aenv(\mathcal{A}_1))_1 \cup \ldots \cup (\mathbf{A}_2[\![f]\!]aenv(\mathcal{A}_n))_1, \underline{ground}\rangle$$

The first component of the alias set consists of the "constant" contribution (global variables and parameters), unioned with one or more applications of the function. Let us call $\mathcal{A}_0$ the "constant" term, and $\mathcal{A}_i$ the alias argument in the i-th application.

Depending on the values of the A_i, we can infer information about the solutions of the equation.

Case 1

$$\mathcal{A}_0 = \emptyset \Rightarrow \mathbf{A}[\![f]\!]aenv = E$$

If the only contribution to the alias set comes from the recursive calls, the least fixed point is E. The recursive equation for $factorial$ (page 95) fits this pattern.

Case 2

$$u_1 \in \mathcal{A}_0 \Rightarrow P \sqsubseteq \mathbf{A}[\![f]\!]aenv$$

This is the case of a function returning an alias to its argument, for instance the function *scan* defined on page 95.

Case 3

$$a \in \mathcal{A}_0, a \neq u_1 \Rightarrow G \sqsubseteq \mathbf{A}[\![f]\!]aenv$$

If a global appears explicitely in the constant set, then the fixed point must be either G or $G \sqcup P$.

Case 4

$$u_1 \in \mathcal{A}_0, (\mathcal{A}_1 \cup \ldots \cup \mathcal{A}_n) \cap \{u\} \neq \emptyset \Rightarrow \mathbf{A}[\![f]\!]aenv = G \sqcup P$$

This pattern covers the case of a recursive function which may return an alias to its parameter or call itself with some global variable. A simple example of this use of recursion is:

$\textbf{letrec } f = \lambda x : \textbf{ref } int.\textbf{if } x\!\uparrow = 0 \textbf{ then } f(A) \textbf{ else } x \textbf{ in } \ldots$

Case 5

$\exists\, k \mid \mathcal{A}_k$ is a call to f.

If there are nested calls, then we must compute two terms of the ascending chain (i.e. unroll the recursive equation once). After two iterations, the arguments used in the nested calls are absorbed in the "constant" term, if the function result may alias to the parameter.

We have listed all possible patterns. The only problem left is how to translate the values computed for the fixed points into valid D_A terms. For E and P, there is a direct correspondance. For G, we must specify which global variables are involved. These appear explicitly in the equation defining the fixed point.

In the case of n recursive functions simultaneously defined, we have a bound on the number of iterations for the ascending chain method. So we have the choice between two methods: either the $n+1$ iterations are performed without comparing consecutive estimates. Or, by using the heuristics outlined above, one can shorten the computation by starting at a higher point in the lattice than bottom, and compare successive estimates of the functions symbolically.

6.4.2 Higher-order functions

In the case of higher-order functions, the same idea applies, but the patterns are more complicated and the guess may not succeed in finding the fixed point, but only an approximation. We can either revert to ascending chain computation starting from that guess to find the least fixed point or choose a trivial fixed point, the top element for functions of that type.

Again, we can first take advantage of the type information of the functions. A function with an r-value result has E as its alias interpretation. In the rest of this section, we assume that the result of a recursive function is either of functional or ref type. Similarly, non-functional and non-ref type arguments are ignored.

If the recursive equation contains a call with the formal parameter as argument, the recursive call can be erased from the equation:

$$\mathbf{A}_2[\![f]\!]aenv = \lambda u.\mathcal{A}_0 \sqcup (\mathbf{A}_2[\![f]\!]aenv(u))$$

has $\lambda u.\mathcal{A}_0$ for fixed point.

For the higher-order function resulting from currying a first-order function of k arguments, the ascending chain method reaches the fixed point in at most $k+1$ steps. This is because the function may only alias some of its arguments and perhaps the global variables appearing explicitely in the function. The worst case is achieved by a function where the permutation of arguments is of order k:

$$\textbf{letrec } f = \lambda x_1 : \textbf{ref } int. \ldots \lambda x_k : \textbf{ref } int.\textbf{if } \ldots \textbf{ then } x_1 \textbf{ else } f(x_2)(\ldots)(x_n)(a)$$

If the recursive equation does not fit these patterns, the last resort consists in comparing symbolically two consecutive terms in the ascending chain, to see if extensional equality can be proved by examination.

6.5 The same with an abstract store

With the addition of a store parameter and component, the alias fixed point computations defined by the equation below become much more complicated.

$$\mathbf{A_e}[\![\textbf{letrec } f_1 = e_1, \ldots, f_n = e_n \textbf{ in } e]\!]aenv\ astore = \mathbf{A_e}[\![e]\!]aenv'\ astore$$

$\mathbf{A_s}[\![\mathbf{letrec}\ f_1 = e_1, \ldots, f_n = e_n\ \mathbf{in}\ e]\!] aenv\ astore = astore$

where

$$aenv' = lfp(\lambda aenv.aenv[\ldots, f_i \leftarrow \mathbf{A_e}[\![e_i]\!] aenv\ astore, \ldots])$$

We consider first a simple example, the function which returns the tail of a list.

$$tail\_of\_list = \lambda l.\mathbf{if}\ l.2 = nil\ \mathbf{then}\ l\ \mathbf{else}\ tail\_of\_list(l.2)$$

The alias function is defined by a recursive equation:

$$(\mathbf{A_e}[\![tail\_of\_list]\!] aenv\ astore)_2 =$$

$$\lambda u.\lambda s.\langle u \sqcup ((\mathbf{A_e}[\![tail\_of\_list]\!] aenv\ astore)_2\ reach(u)\ s)_1\ ,$$

$$((\mathbf{A_e}[\![tail\_of\_list]\!] aenv\ astore)_2\ (reach(u))\ s)_2 s\rangle$$

If we compute the ascending chain of approximations, we get:

$$A^{(0)} = \lambda u.\lambda s.\langle\langle\emptyset, \underline{ground}\rangle, s\rangle$$

$$A^{(1)} = \lambda u.\lambda s.\langle\langle u_1, \underline{ground}\rangle, s\rangle$$

$$A^{(2)} = \lambda u.\lambda s.\langle\langle u_1 \cup reach(u_1), \underline{ground}\rangle, s\rangle$$

$$= \lambda u.\lambda s.\langle\langle reach(u_1), \underline{ground}\rangle, s\rangle$$

$$A^{(3)} = \lambda u.\lambda s.\langle\langle reach(u_1) \cup reach(reach(u_1)), \underline{ground}\rangle, s\rangle$$

$$= \lambda u.\lambda s.\langle\langle reach(u_1) \cup reach(u_1), \underline{ground}\rangle, s\rangle$$

$$= \lambda u.\lambda s.\langle\langle reach(u_1), \underline{ground}\rangle, s\rangle$$

$$= A^{(2)}$$

As an example of recursive function with side effects, let us compute the fixed point of *create_list*:

$\mathbf{letrec}\ create\_list =$

$\lambda n : int.\ \mathbf{if}\ n = 0$

then *nil*

else let *cell* = *alloc*(*list*) **in** *cell*.2 ← *create_list*($n - 1$) ; *cell* **ni**

in ...

The recursive equation defining its aliasing behavior is:

$\mathbf{A_{e2}}[\![create\_list]\!]\rho\ \sigma =$

$\lambda u.\lambda s.\langle \mathbf{A_e}[\![\mathbf{if}\ \ldots]\!]\rho[n \leftarrow u]\ s\ ,\ \mathbf{A_s}[\![\mathbf{if}\ \ldots]\!]\rho[n \leftarrow u]\ s\rangle$

We ignore the parameter n, since it is passed by value. The A-image of a conditional expression is the join of the two branches of the conditional. Since the A-image of *nil* does not contribute information ($\mathbf{A_e}[\![nil]\!]aenv\ astore \sqcup \mathbf{A_e}[\![e]\!]aenv\ astore = \mathbf{A_e}[\![e]\!]aenv\ astore$), we compute only the contribution of the **else** branch. Let us call ρ' the environment resulting from the binding of *cell*:

$\rho' = \rho[cell \leftarrow \langle l, \underline{ground}\rangle]$

The effect on the store s of the allocation of the new cell is given by:

$Update_A(s, l, \emptyset)$

We now evaluate the right hand side of the recursive equation defining the A-image of *create_list*:

$\mathbf{A_{e2}}[\![create\_list]\!]\rho\ \sigma =$

$= \lambda s.\langle \mathbf{A_e}[\![cell.2 \leftarrow create\_list()\ ;\ cell]\!]\rho'\ Update_A(s, l, \emptyset)\ ,$

$\mathbf{A_s}[\![\ldots]\!]\rho'\ Update_A(s, l, \emptyset)\rangle$

$= \lambda s.\langle \mathbf{A_e}[\![cell]\!]\rho'\ \mathbf{A_s}[\![cell.2 \leftarrow create\_list()]\!]\rho'\ Update_A(s, l, \emptyset)\ ,$

$\mathbf{A_s}[\![cell.2 \leftarrow create\_list()]\!]\rho'\ Update_A(s, l, \emptyset)\rangle$

$= \lambda s.\langle\ \langle l, \underline{ground}\rangle\ ,$

$$Update_A(\mathbf{A_s}[\![create\_list()]\!]\rho'\; Update_A(s,l,\emptyset)$$

$$l,$$

$$(\mathbf{A_e}[\![create\_list()]\!]\rho'\; Update_A(s,l,\emptyset))_1\;\rangle$$

$$= \lambda s.\langle\langle l, \underline{ground}\rangle\;,$$

$$Update_A(\mathbf{A_{e2}}[\![create\_list]\!]\rho'\; Update_A(s,l,\emptyset)_2, l, l)\rangle$$

The last equality makes explicit the recursive nature of the equation. As usual, we compute the ascending chain of approximations. For the first approximation, assume functions are pure:

$$A^{(0)} = \lambda s.\langle\langle\emptyset, \underline{ground}\rangle, s\rangle$$

$$A^{(1)} = \lambda s.\langle\langle l, \underline{ground}\rangle\;,$$

$$Update_A(Update_A(s,l,\emptyset),l,l)\rangle$$

$$= \lambda s.\langle\langle l, \underline{ground}\rangle\;,\; Update_A(s,l,l)\rangle$$

$$A^{(2)} = \lambda s.\langle\langle l, \underline{ground}\rangle\;,$$

$$Update_A(Update_A(Update_A(s,l,\emptyset),l,l),l,l)\rangle$$

$$= \lambda s.\langle\langle l, \underline{ground}\rangle\;,\; Update_A(s,l,l)\rangle$$

$$= A^{(1)}$$

The point of this exercise in lambda calculus is to show that the alias values are simple, even though the fixed point computation may be lengthy. The programming paradigms associated with recursive data structures produce "simple" aliasing behavior and we can use heuristics similar to the ones defined in the previous section. In addition, we have to consider the case of a recursive function which returns an alias to storage dynamically allocated during the call. Like *create_list*, these functions are characterized by a recursive equation with explicit allocation in the $\mathbf{A_s}$ component.

Because linked data structures are represented by one location in the alias store, recursive functions traversing them are characterized by recursive equations which converge after a small number of iterations. This is the case for *tail_list*, because *reach* is idempotent. Instead of testing for extensional equality, one can also compare two successive iterations symbolically, after simplification. The simplification rules are induced by the properties of sets and of the store model. This scheme fails in the case of nested calls to a functional argument. This rare case can be detected symbolically as well, and a trivial fixed point can be used.

6.6 Discussion

In conclusion, to obtain tractable fixed point computations, one must rule out the ascending chain + extensional equality algorithm. But, because the alias store contains only a finite number of locations and linked data structures have a simple representation, the recursive equations can be simplified. In the first-order case, the least fixed point of a single function can be inferred from the pattern of the recursive equation. If there are n multiple recursive functions, the number of iterations is bounded by $n+1$. A similar bound applies for the first-order function of k arguments. For higher-order case, the cheapest method is to use heuristics to guess a fixed point, and then to check that it satisfies the recursive equation. The correctness of the guess is thus verified. Although there is no agreement on what constitutes the normal distribution of programs, it is fair to say that the heuristics listed here cover the most frequent programming paradigms.

Chapter 7

Conclusions

Until recently, aliasing and side effect inference in higher-order expression languages has not been addressed because the problem lies beyond the range of traditional dataflow algorithms. Reynolds [20] defined syntactic restrictions to prevent interferences in a higher-order language with call-by-name semantics. Along similar lines, Lucassen [14] proposed to integrate side effect specification into the type system of a language, under explicit control of the programmer. In this thesis, we have seen that inference of aliases and side effects can be accomplished in a much less restrictive setting. The higher-order character of the language is shown to represent only a partial obstacle. On the other hand, the presence of l-valued expressions has the consequence that aliasing information must be computed for all expressions, and cannot be represented as a relation among identifiers. Furthermore, the introduction of pointers make aliasing and side effects flow-dependent properties, and a meaningful static inference scheme must take into account flow control information as well.

Our inference schemes provide information about aliasing and side effects in a higher-order expression language with call-by-value semantics. Abstract interpretation techniques allow us to give a compositional definition, which can be proved sound with respect to the standard semantics by structural induction. The abstract interpretation functions are easy to modify, in case a different type of information is requested. In particular, a function defined over a lattice base domain automatically has a "dual" function, obtained by reversing the partial order in the base lattice. The dual of our support set function would compute the variables that are definitely modified by the evaluation of an expression.

Except for the restriction on storable functions, there are only mild constraints on the language. Theoretically, our model could deal with storable functions, by using a more elaborate model for the alias store. Unfortunately, an efficient implementation would be difficult, and fixed point computations would probably reduce to educated guesses.

There is some controversy over the use of abstract interpretation to analyze operational properties of a language. Since this is largely a matter of definition, we agree in advance with the reader who would like to call **A** and the other functions something other than an "abstract interpretation" function. Overall, although we have demonstrated the feasibility of such techniques to analyze the operational behavior of programs, it is only in the process of doing so that we understood the importance of the operational semantics. The soundness theorems of chapter 4 still have a strong denotational flavor, and a true operational soundness theorem would state that variables outside the support set are never accessed *during* the computation, etc. Instead, with a denotational semantics, we can only prove that the variables outside the support set are unchanged *after* the computation. This suggests that it would be interesting and worthwhile to formalize static analysis in terms of operational semantics.

The abstract interpretation functions implicitely define static inference algorithms, which can easily be implemented by an attribute grammar, or any other tool capble of performing computations on the abstract syntax tree. In fact, we have implemented the functions **A** and **S** of chapter 3 in ML, and the result is a short and simple program. A wider implementation effort is currently in progress.

The accuracy of these algorithms is better than for the dataflow ones, because we make use of control flow information. Our algorithms also compare favorably in complexity, but the dataflow approach is probably cheaper in most practical settings, since it operates on Boolean matrices instead of the lambda calculus terms. In addition, our schemes can give information even in the presence of dynamically allocated data structures, although the utility of this information remains to be demonstrated in a a "real-life" system. An interesting extension of this work would be to consider abstract data types and the use of negative information to obtain even more accurate information.

Appendix A

Standard semantics

This appendix contains the standard semantics of the language defined in chapter 2.

Type system

The type rules for the abstract syntax are presented in the Gentzen style. There are two kinds of formula, $t : Type$, which reads as "t is a legal type", and $e : t$, which reads "expression e has type t". Hypotheses have the form

$$H ::= \emptyset \mid H, x : t$$

The hypothesis set must be consistent: it cannot contain more than one type assignment per identifier. When a type assignment hypothesis $x : t$ is added to the set H, any type assignment to x already in H is removed:

$$H, x : t \text{ means } (H \setminus \{x : t'\}) \cup \{x : t\}$$

Legal types

$\vdash int : Type$

$\vdash bool : Type$

$\vdash \mathbf{ref}\ int : Type$
$\vdash \mathbf{ref}\ bool : Type$

$H_1, t : Type, H_2 \vdash t : Type$

$$\frac{H \vdash t_1 : Type \qquad H \vdash t_2 : Type}{H \vdash t_1 \to t_2 : Type}$$

Typed expressions

$\vdash n : int$ for any integer n

$\vdash true : bool$
$\vdash false : bool$

$H_1, x : t, H_2 \vdash x : t$

$$\frac{H \vdash e_1 : bool \qquad H \vdash e_2 : t \qquad H \vdash e_3 : t}{H \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : t}$$

$$\frac{H, x : t_1 \vdash e : t_2}{H \vdash \lambda x : t_1.e : t_1 \to t_2}$$

$$\frac{H \vdash e_1 : t_1 \to t_2 \qquad H \vdash e_2 : t_1}{H \vdash e_1(e_2) : t_2}$$

$$\frac{\begin{array}{l} H, f_1 : t_1 \rightarrow t_1', \ldots, f_n : t_n \rightarrow t_n' \vdash \lambda x_i : t_i.e_i : t_i \rightarrow t_i' \quad i = 1, n \\ H, f_1 : t_1 \rightarrow t_1', \ldots, f_n : t_n \rightarrow t_n' \vdash e : t \end{array}}{H \vdash \textbf{letrec } f_1 = \lambda x_1 : t_1.e_1, \ldots, f_n = \ldots \textbf{ in } e : t}$$

$$\frac{H, x : \textbf{ref}\, t \vdash e : t'}{H \vdash \textbf{new } x : \textbf{ref}\, t \textbf{ in } e : t'}$$

$$\frac{\begin{array}{l} H \vdash e_1 : \textbf{ref}\, t \\ H \vdash e_2 : t \end{array}}{H \vdash e_1 \leftarrow e_2 : t}$$

$$\frac{H \vdash e : \textbf{ref}\, t}{H \vdash e\uparrow : t}$$

$$\frac{\begin{array}{l} H \vdash e_1 : t_1 \\ H \vdash e_2 : t_2 \end{array}}{H \vdash e_1 \,;\, e_2 : t_2}$$

Denotational semantics

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

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

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

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

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

where

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

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

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

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

$$\mathbf{M_e}[\![e_1(e_2)]\!] env\ store = (f\ e\ store'')_1$$

$$\mathbf{M_s}[\![e_1(e_2)]\!] env\ store = (f\ e\ store'')_2$$

where

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

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

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

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

$$\mathbf{M_e}[\![\textbf{letrec } f_1 = \lambda x_1 : t_1.e_1, \ldots, f_n = \ldots \textbf{ in } e]\!] env\ store = \mathbf{M_e}[\![e]\!]\ env'\ store$$

$$\mathbf{M_s}[\![\textbf{letrec } f_1 = \lambda x_1 : t_1.e_1, \ldots, f_n = \ldots \textbf{ in } e]\!] env\ store = \mathbf{M_s}[\![e]\!]\ env'\ store$$

where

$$env' = lfp(F\ env)$$

$$F : Env \rightarrow Env =$$

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

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

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

where

$$\langle loc, store' \rangle \ = Allocate(store)$$

$$env' \ = \ env[x \ \leftarrow \ loc]$$

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

$$store''' = DeAllocate(store'', loc)$$

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

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

where

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

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

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

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

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

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

where

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

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

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

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

Appendix B

Static detection of multiple calls

In a first-order language, multiple calls to the same function within one activation can be detected by searching for loops in the call graph of a program. With higher-order functions, the call graph of a program is not a static entity anymore, and the above method would fail. Using the same techniques developed for alias and support analysis, we define an abstract interpretation function, called **C**, to detect multiple calls to a function during the evaluation of an expression. The function requires auxiliary computations carried by two other abstract interpretations, **F** and **Y**.

The F function

The first function, **F**, computes the function aliases of an expression of function type. If the expression is not of function type, it returns the constant <u>*ground*</u>. If the expression is of function type, it returns the set of functions that the expression may evaluate to. Again, because the language has higher-order functions, there is a second component that gives the same information for the result of applying the expression.

However, the first question to solve is how to denote function constants. Unlike static variables which are bound to a name by syntax (the **new** contruct), the functions appearing in an expression are not necessarily named. For instance, in

$$(\lambda x : int.x)((\lambda x : int.x)(13))$$

there are two occurrences of the identity function over integers ($\lambda x : int.x$), and we would like to distinguish them. A solution is to decorate each λ appearing in the expression being analyzed with an integer value. The example above becomes:

$$(\lambda^1 x : int.x)((\lambda^2 x : int.x)(13))$$

We use the tag to name the functions: the first function is "λ^1", and the second "λ^2". This convention guarantees that distinct occurrences of identical functions are distinct in our semantics. Let us call FId the set of "λ^i" which appears in the program. The interpretation of $\mathbf{F}$ is as follows:

$$\mathbf{F}_1[\![e]\!]fenv = S \;\Rightarrow\; e \text{ may evaluate to any of the functions in set } S$$

The lambda terms are the constants of interest to the function $\mathbf{F}$. Non-functional constants and all variables evaluate to the constant <u>*ground*</u>. The recursive structure of the domains is similar to the ones used for the other abstract interpretations.

$$\mathbf{F} : Exp \rightarrow Env_F \rightarrow D_F$$

$$Env_F = Id \rightarrow D_F$$

$$D_F = \underline{ground} + (\mathcal{P}(FId) \times (D_F \rightarrow D_F))$$

The lattice structure over domain D_F is induced by the lattice $\mathcal{P}(FId)$.

The clauses for the computation of function aliases are summarized in figure B.1 on page 131, and explained in details below. The aliasing behavior of global identifiers is provided by an alias environment called $fenv$.

Constants and identifiers

$$\mathbf{F}[\![c]\!]fenv = \underline{ground}$$

$$\mathbf{F}[\![x]\!]fenv = fenv(x)$$

There are no functional constants, so the value of a constant is $\underline{ground}$. The functional aliases associated with an identifier are provided by the environment $fenv$.

Conditional and sequential evaluation

$$\mathbf{F}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]fenv = \mathbf{F}[\![e_2]\!]fenv \sqcup \mathbf{F}[\![e_3]\!]fenv$$

$$\mathbf{F}[\![e_1 \,;\, e_2\,]\!]fenv = \mathbf{F}[\![e_2]\!]fenv$$

These clauses are the same as for the alias function **A**.

Abstraction and application

$$\mathbf{F}[\![\lambda^i x : t.e]\!]fenv = \langle\, \{\text{"}\lambda^i\text{"}\} \,,\, \lambda u.\, \mathbf{F}[\![e]\!]fenv[x \leftarrow u] \,\rangle$$

$$\mathbf{F}[\![e_1(e_2)]\!]fenv = (\mathbf{F}_2[\![e_1]\!]fenv)\,(\mathbf{F}[\![e_2]\!]fenv)$$

These clauses are identical to the ones for the alias abstract interpretation function, except that the value of a lambda term is its name.

Recursive functions

$$\mathbf{F}[\![\mathbf{letrec}\ f_1 = e_1 \ldots f_n = e_n \ \mathbf{in}\ e]\!]fenv \ = \ \mathbf{F}[\![e]\!]fenv'$$

$$\text{where } fenv' \ = \ \mathit{lfp}(\lambda fenv.\ fenv[\ldots, f_i \leftarrow \mathbf{F}[\![e_i]\!]fenv, \ldots])$$

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.

Imperative constructs

$$\mathbf{F}[\![\mathbf{new}\ x : t\ \mathbf{in}\ e]\!]fenv \ = \ \mathbf{F}[\![e]\!]fenv[x \leftarrow \underline{ground}]$$

$$\mathbf{F}[\![e_1 \leftarrow e_2\,]\!]fenv \ = \ \underline{ground}$$

$$\mathbf{F}[\![e\!\uparrow]\!]fenv \ = \ \underline{ground}$$

Since there are no variables of functional type, we can safely assert that the value associated with a variable is $\underline{ground}$. Similarly, the result of an assignment or of a dereferencing cannot be of functional type, and the value of those expressions is $\underline{ground}$ as well.

The Y function

We also need to know which function expressions denote recursive functions. We use an auxiliary function called **Y** [1] to this end. Again, the interpretation is

[1] In honor of the lambda calculus fixed point operator, called "Y".

simple:

$$\mathbf{Y}_1[\![e]\!]yenv = \mathit{true} \ \Rightarrow \ e \text{ may be a recursive function}$$

and

$$\mathbf{Y}_1[\![e]\!]yenv = \mathit{false} \ \Rightarrow \ e \text{ is not a recursive function}$$

The domains used by **Y** have the same recursive structures as D_A and D_S. The underlying lattice of values is the Boolean domain. See figure B.2 page 131 for the definition of **Y**. The $\vee$ operator is the Boolean disjunction operator, extended to the recursive domain.

$$\mathbf{Y} : \mathit{Exp} \rightarrow \mathit{Env}_Y \rightarrow D_Y$$

$$\mathit{Env}_Y = \mathit{Id} \rightarrow D_Y$$

$$D_Y = \{\underline{\mathit{ground}}\} + (\mathit{Bool} \times (D_Y \rightarrow D_Y))$$

The **Y** computation is purely functional, since functions are not storable values.

The C function

Now that we can determine the functional aliases of an expression and detect expressions denoting recursive functions, we are ready to compute the set of functions in the activation stack while evaluating a function, and check that a function is not called twice in the same activation. We define the abstract interpretation function **C** (for "Calls_at_most_once"), which returns *false* if the evaluation of the expression may involve two calls to a function in the same activation. Since the function must keep track of which functions already appear in the activation stack, the signature of **C** includes, along with the environment, an abstraction of the stack as parameter. Because the order in which order the functions are called is not relevant, the abstract stack is a set of function identifiers instead of a more refined data structure. The domain equations follow the pattern of the ones for the support abstract interpretation.

$$\mathbf{C} : Exp \rightarrow Env_C \rightarrow Stack\_Set \rightarrow D_C$$

$$\begin{aligned} Env_C &= Id \rightarrow Val_C \\ D_C &= Bool \times Val_C \\ Val_C &= \{\underline{ground}\} + Val_C \rightarrow Stack\_Set \rightarrow D_C \\ Stack\_Set &= \mathcal{P}(FId) \end{aligned}$$

The complete definition of **C** is in figure B, page 132. Like the support abstract interpretation, this function is operational in flavor and is closely related to the $\mathbf{M_s}$ semantic function: the computation of subexpressions contributes to the first component, and there is an additional parameter, an approximation of the activation stack. A discussion of the clauses follows.

<u>Constants and identifiers</u>

$$\mathbf{C}[\![c]\!]cenv\ c = \langle\ true,\ \underline{ground}\rangle$$

$$\mathbf{C}[\![x]\!]cenv\ c = \langle\ true,\ cenv(x)\rangle$$

The evaluation of a constant or an identifier does not require any function calls, so the first component is *true*. There are no functional constants, hence *ground*, but an identifier may denote a function, and the environment is consulted.

Conditional and sequential evaluation

$$\mathbf{C}[\![\mathbf{if}\ \ e_1\ \mathbf{then}\ \ e_2\ \mathbf{else}\ \ e_3]\!]cenv\ c\ =$$

$$\langle\ \mathbf{C}_1[\![e_1]\!]cenv\ c \wedge \mathbf{C}_1[\![e_2]\!]cenv\ c \wedge \mathbf{C}_1[\![e_3]\!]cenv\ c\ ,$$

$$\mathbf{C}_2[\![e_2]\!]cenv\ c \sqcup \mathbf{C}_2[\![e_3]\!]cenv\ c\ \rangle$$

$$\mathbf{C}[\![e_1\ ;\ e_2\ ]\!]cenv\ c\ =\ \langle\ \mathbf{C}_1[\![e_1]\!]cenv\ c \wedge \mathbf{C}_1[\![e_2]\!]cenv\ c\ ,\ \mathbf{C}_2[\![e_2]\!]cenv\ c\ \rangle$$

These clauses are similar to the ones for the function **S**.

Abstraction and application

$$\mathbf{C}[\![\lambda x : t.e]\!]cenv\ c\ =\ \langle\ true\ ,\ \lambda u.\lambda c'.\mathbf{C}[\![e]\!]cenv[x \leftarrow u]\ c'\ \rangle$$

$$\mathbf{C}[\![e_1(e_2)]\!]cenv\ c\ =$$

$$\langle\ \mathbf{C}_1[\![e_1]\!]cenv\ c \wedge \mathbf{C}_1[\![e_2]\!]cenv\ c \wedge (\mathbf{F}_1[\![e_1]\!]fenv \cap c = \emptyset) \wedge (\neg\mathbf{Y}_1[\![e_1]\!]yenv)\ ,$$

$$\wedge((\mathbf{C}_2[\![e_1]\!]cenv\ c)(\mathbf{C}_2[\![e_2]\!]cenv\ c)\ c')_1\ ,$$

$$((\mathbf{C}_2[\![e_1]\!]cenv\ c)(\mathbf{C}_2[\![e_2]\!]cenv\ c)\ c')_2\ \rangle$$

where

$$c' = c \cup \mathbf{F}_1[\![e_1]\!]fenv$$

A lambda term is a constant, so the first component is *true*. The second component is the function giving the information when the lambda term is applied. This function takes two arguments: the first one is the information relating to the argument x of the lambda term, the second argument is the activation stack at the time of the call. For an application, we get the usual terms contributed by the computations of subexpressions, we check that the function was not already called once ($\mathbf{F}_1[\![e_1]\!]fenv \cap c = \emptyset$), that e_1 does not denote a recursive function ($\neg\mathbf{Y}_1[\![e_1]\!]yenv$) and finally we compute the contribution of the application itself. Note how the stack is updated ($c' = c \cup \mathbf{F}_1[\![e_1]\!]fenv$).

<u>Recursive functions</u>

$$\mathbf{C}[\![\mathbf{letrec}\ \ f_1 = e_1 \ldots f_n = e_n\ \mathbf{in}\ \ e_2]\!]cenv\ c \ = \ \mathbf{C}[\![e]\!]cenv'\ c$$

$$\text{where } cenv' \ = \ lfp(\lambda cenv.cenv[\ldots, f_i \leftarrow \mathbf{C}_2[\![e_i]\!]cenv\ c, \ldots])$$

<u>Imperative constructs</u>

$$\mathbf{C}[\![\mathbf{new}\ \ x : t\ \mathbf{in}\ \ e]\!]cenv\ c \ = \ \mathbf{C}[\![e]\!]cenv[x \leftarrow \langle true\ ,\ \underline{ground}\rangle]\ c$$

$$\mathbf{C}[\![e_1 \leftarrow e_2\ ]\!]cenv\ c \ = \ \langle\ \mathbf{C}_1[\![e_1]\!]cenv\ c \wedge \mathbf{C}_1[\![e_2]\!]cenv\ c,\ \underline{ground}\ \rangle$$

$$\mathbf{C}[\![e\!\uparrow]\!]cenv\ c \ = \ \langle\ \mathbf{C}_1[\![e]\!]cenv\ c,\ \underline{ground}\ \rangle$$

A variable is a non-functional identifier, so its value is the pair *true* (no function call) and <u>*ground*</u> (non-functional). Assignments and dereferencing also have for second component <u>*ground*</u>.

$$
\begin{array}{l}
\mathbf{F}[\![c]\!]fenv \;=\; \underline{ground} \\
\mathbf{F}[\![x]\!]fenv \;=\; fenv(x) \\
\mathbf{F}[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]fenv \;=\; \mathbf{F}[\![e_2]\!]fenv \sqcup \mathbf{F}[\![e_3]\!]fenv \\
\mathbf{F}[\![e_1\ ;\ e_2\ ]\!]fenv \;=\; \mathbf{F}[\![e_2]\!]fenv \\
\mathbf{F}[\![\lambda^i x : t.e]\!]fenv \;=\; \langle\ \{\text{"}\lambda^i\text{"}\}\ ,\ \lambda u.\,\mathbf{F}[\![e]\!]fenv[x \leftarrow u]\ \rangle \\
\mathbf{F}[\![e_1(e_2)]\!]fenv \;=\; (\mathbf{F}_2[\![e_1]\!]fenv)\,(\mathbf{F}[\![e_2]\!]fenv) \\
\mathbf{F}[\![\mathbf{letrec}\ f_1 = e_1 \ldots f_n = e_n\ \mathbf{in}\ e]\!]fenv \;=\; \mathbf{F}[\![e]\!]fenv' \\
\qquad \text{where } fenv' \;=\; lfp(\lambda fenv.\, fenv[\ldots, f_i \leftarrow \mathbf{F}[\![e_i]\!]fenv, \ldots]) \\
\mathbf{F}[\![\mathbf{new}\ x : t\ \mathbf{in}\ e]\!]fenv \;=\; \mathbf{F}[\![e]\!]fenv[x \leftarrow \underline{ground}] \\
\mathbf{F}[\![e_1 \leftarrow e_2\ ]\!]fenv \;=\; \underline{ground} \\
\mathbf{F}[\![e\uparrow]\!]fenv \;=\; \underline{ground}
\end{array}
$$

Figure B.1: The **F** function

$$
\begin{array}{l}
\mathbf{Y}[\![c]\!]yenv \;=\; \underline{ground} \\
\mathbf{Y}[\![x]\!]yenv \;=\; yenv(x) \\
\mathbf{Y}[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]yenv \;=\; \mathbf{Y}[\![e_2]\!]yenv \vee \mathbf{Y}[\![e_3]\!]yenv \\
\mathbf{Y}[\![e_1\ ;\ e_2\ ]\!]yenv \;=\; \mathbf{Y}[\![e_2]\!]yenv \\
\mathbf{Y}[\![\lambda x : t.e]\!]yenv \;=\; \langle\ false\ ,\ \lambda u.\,\mathbf{Y}[\![e]\!]yenv[x \leftarrow u]\ \rangle \\
\mathbf{Y}[\![e_1(e_2)]\!]yenv \;=\; (\mathbf{Y}_2[\![e_1]\!]yenv)\,(\mathbf{Y}[\![e_2]\!]yenv) \\
\mathbf{Y}[\![\mathbf{letrec}\ f_1 = e_1 \ldots f_n = e_n\ \mathbf{in}\ e]\!]yenv \;=\; \mathbf{Y}[\![e]\!]yenv' \\
\qquad \text{where } yenv' \;=\; lfp(\lambda yenv.\, yenv[\ldots, f_i \leftarrow \langle true\ ,\ \mathbf{Y}_2[\![e_i]\!]yenv\rangle, \ldots]) \\
\mathbf{Y}[\![\mathbf{new}\ x : t\ \mathbf{in}\ e]\!]yenv \;=\; \mathbf{Y}[\![e]\!]yenv[x \leftarrow \underline{ground}] \\
\mathbf{Y}[\![e_1 \leftarrow e_2\ ]\!]yenv \;=\; \underline{ground} \\
\mathbf{Y}[\![e\uparrow]\!]yenv \;=\; \underline{ground}
\end{array}
$$

Figure B.2: The **Y** function

$$\mathbf{C}[\![c]\!]\mathit{cenv}\ c \;=\; \langle\ \mathit{true},\ \underline{\mathit{ground}}\rangle$$
$$\mathbf{C}[\![x]\!]\mathit{cenv}\ c \;=\; \langle\ \mathit{true},\ \mathit{cenv}(x)\rangle$$
$$\mathbf{C}[\![\mathbf{letrec}\ \ f_1 = e_1 \ldots f_n = e_n\ \mathbf{in}\ \ e]\!]\mathit{cenv}\ c \;=\; \mathbf{C}[\![e]\!]\mathit{cenv}'\ c$$
$$\text{where } \mathit{cenv}' \;=\; \mathit{lfp}(\lambda \mathit{cenv}.\mathit{cenv}[\ldots, f_i \leftarrow \mathbf{C}[\![e_i]\!]\mathit{cenv}\ c, \ldots])$$
$$\mathbf{C}[\![\mathbf{if}\ \ e_1\ \mathbf{then}\ \ e_2\ \mathbf{else}\ \ e_3]\!]\mathit{cenv}\ c \;=$$
$$\langle\ \mathbf{C}_1[\![e_1]\!]\mathit{cenv}\ c \wedge \mathbf{C}_1[\![e_2]\!]\mathit{cenv}\ c \wedge \mathbf{C}_1[\![e_3]\!]\mathit{cenv}\ c\ ,\ \mathbf{C}_2[\![e_2]\!]\mathit{cenv}\ c \sqcup \mathbf{C}_2[\![e_3]\!]\mathit{cenv}\ c\ \rangle$$
$$\mathbf{C}[\![e_1\ ;\ e_2\ ]\!]\mathit{cenv}\ c \;=\; \langle\ \mathbf{C}_1[\![e_1]\!]\mathit{cenv}\ c \wedge \mathbf{C}_1[\![e_2]\!]\mathit{cenv}\ c\ ,\ \mathbf{C}_2[\![e_2]\!]\mathit{cenv}\ c\ \rangle$$
$$\mathbf{C}[\![\lambda x : t.e]\!]\mathit{cenv}\ c \;=\; \langle\ \mathit{true}\ ,\ \lambda u.\lambda c'.\mathbf{C}[\![e]\!]\mathit{cenv}[x \leftarrow u]\ cc'\ \rangle$$
$$\mathbf{C}[\![e_1(e_2)]\!]\mathit{cenv}\ c \;=$$
$$\langle\ \mathbf{C}_1[\![e_1]\!]\mathit{cenv}\ c \wedge \mathbf{C}_1[\![e_2]\!]\mathit{cenv}\ c \wedge (\mathbf{F}_1[\![e_1]\!]\mathit{fenv} \cap c = \emptyset) \wedge (\neg \mathbf{Y}_1[\![e_1]\!]\mathit{yenv})$$
$$\wedge((\mathbf{C}_2[\![e_1]\!]\mathit{cenv}\ c)(\mathbf{C}_2[\![e_2]\!]\mathit{cenv}\ c)c')_1\ ,$$
$$((\mathbf{C}_2[\![e_1]\!]\mathit{cenv}\ c)(\mathbf{C}_2[\![e_2]\!]\mathit{cenv}\ c)c')_2\ \rangle$$
$$\text{where } c' = c \cup \mathbf{F}_1[\![e_1]\!]\mathit{fenv}$$
$$\mathbf{C}[\![\mathbf{new}\ \ x : t\ \mathbf{in}\ \ e]\!]\mathit{cenv}\ c \;=\; \mathbf{C}[\![e]\!]\mathit{cenv}[x \leftarrow \underline{\mathit{ground}}]\ c$$
$$\mathbf{C}[\![e_1 \leftarrow e_2\ ]\!]\mathit{cenv}\ c \;=\; \langle\ \mathbf{C}_1[\![e_1]\!]\mathit{cenv}\ c \wedge \mathbf{C}_1[\![e_2]\!]\mathit{cenv}\ c\ ,\ \underline{\mathit{ground}}\ \rangle$$
$$\mathbf{C}[\![e\!\uparrow]\!]\mathit{cenv}\ c \;=\; \langle\ \mathbf{C}_1[\![e]\!]\mathit{cenv}\ c\ ,\ \underline{\mathit{ground}}\ \rangle$$

Figure B.3: The **C** function

Appendix C

Semantics for extensions

This appendix contains the semantics of the language extensions in chapter 5.

Type System

The type system is modified to allow pointers, but must still forbid storage of functions. To the two formulas of appendix A ($e : t$, and $t : Type$), we add $t : Storable$, which reads "t is a storable type", and implies that t contains no arrow constructor. Hypotheses are consistent sets of formulas of the form

$$H ::= \emptyset \mid H, x : t \mid H, t : Type \mid H, t : Storable$$

Legal types

There are no restrictions on types, except that functions are not storable. The rules for legal types enforce this constraint, by keeping track of the storable types. For recursive types, we use a type fixpoint operator *rec*.

$$\begin{array}{l} \vdash int \;:\; Type \\ \vdash int \;:\; Storable \end{array}$$

$$\begin{array}{l} \vdash bool \;:\; Type \\ \vdash bool \;:\; Storable \end{array}$$

$$\frac{H \vdash t \;:\; Storable}{\begin{array}{l} H \vdash \mathbf{ref}\; t \;:\; Type \\ H \vdash \mathbf{ref}\; t \;:\; Storable \end{array}}$$

$$\frac{\begin{array}{l} H \vdash t_1 \;:\; Storable \\ H \vdash t_2 \;:\; Storable \end{array}}{\begin{array}{l} H \vdash t_1 \times t_2 \;:\; Type \\ H \vdash t_1 \times t_2 \;:\; Storable \end{array}}$$

$$H_1,\, t \;:\; Type\,,\; H_2 \vdash t \;:\; Type$$

$$\frac{\begin{array}{l} H \vdash t_1 \;:\; Type \\ H \vdash t_2 \;:\; Type \end{array}}{H \vdash t_1 \rightarrow t_2 \;:\; Type}$$

$$\frac{H, t : Storable \vdash \sigma : Storable}{\begin{array}{l} H \vdash rec(t, \sigma) : Type \\ H \vdash rec(t, \sigma) : Storable \end{array}}$$

Typed expressions

To the rules of appendix A, we add:

$$\frac{H \vdash t : Storable}{H \vdash alloc(t) : \mathbf{ref}\ t}$$

$$\frac{\begin{array}{l} H \vdash\ e_1\ :\ t_1 \\ H \vdash\ e_2\ :\ t_2 \end{array}}{H \vdash\ \langle e_1\ ,\ e_2 \rangle\ :\ t_1 \times t_2}$$

$$\frac{H \vdash\ e\ :\ t_1 \times t_2}{\begin{array}{l} H \vdash\ e.1\ :\ t_1 \\ H \vdash\ e.2\ :\ t_2 \end{array}}$$

$$\frac{H \vdash\ e\ :\ \mathbf{ref}\ (t_1 \times t_2)}{\begin{array}{l} H \vdash\ e.1\ :\ \mathbf{ref}\ t_1 \\ H \vdash\ e.2\ :\ \mathbf{ref}\ t_2 \end{array}}$$

$$\frac{\begin{array}{l} H \vdash rec(t, \sigma) : Type \\ H \vdash e[t \rightarrow rec(t, \sigma)] : t' \end{array}}{H \vdash \mathbf{letrectype}\ t = \sigma\ \mathbf{in}\ e\ :\ t'}$$

$$\frac{H \vdash e : \sigma[t \rightarrow rec(t, \sigma)]}{H \vdash e : rec(t, \sigma)}$$

Domains

The domain of locations is redefined as:

$$Loc = Int + nil$$

and the domain of values must now include products :

$$D = \ldots + D \times D$$

Standard semantics

$\mathbf{M_e}[\![alloc(t)]\!]env\ store \ = \ loc$

$\mathbf{M_s}[\![alloc(t)]\!]env\ store \ = \ store'$

where, for $store = \langle map, loc\rangle$

$$store' = \langle map[loc \leftarrow \underline{unused}, \ldots, loc + size(t) - 1 \leftarrow \underline{unused}], loc + size(t)\rangle$$

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

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

$\mathbf{M_e}[\![\langle e_1, e_2\rangle]\!]env\ store \ = \ \langle \mathbf{M_e}[\![e_1]\!]env\ store, \mathbf{M_e}[\![e_2]\!]\ env\ store'\rangle$

$\mathbf{M_s}[\![\langle e_1, e_2\rangle]\!]env\ store \ = \ \mathbf{M_s}[\![e_2]\!]\ env\ store'$

where

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

The projection functions for pairs are overloaded operators. In an l-value context, they should return the address of the component, instead of its value. We give only the clause for the r-value context.

$\mathbf{M_e}[\![e.1]\!]env\ store \ = \ (\mathbf{M_e}[\![e]\!]env\ store).1$

$\mathbf{M_s}[\![e.1]\!]env\ store \ = \ \mathbf{M_s}[\![e]\!]env\ store$

$\mathbf{M_e}[\![\mathbf{letrectype}\ t = \sigma\ \mathbf{in}\ e]\!]env\ store = \mathbf{M_e}[\![e]\!]env\ store$

$\mathbf{M_s}[\![\mathbf{letrectype}\ t = \sigma\ \mathbf{in}\ e]\!]env\ store = \mathbf{M_s}[\![e]\!]env\ store$

Bibliography

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

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

[3] S.D. Brookes. *A Fully Abstract Semantics and a Proof System for an Algol-like Language with Sharing.* Technical Report CMU-CS-84-118A, Carnegie-Mellon University, 1984.

[4] 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-Verlag LNCS 217, 1986.

[5] 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-Verlag LNCS 201, 1985.

[6] K.D. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: eliminating unnecessary recompilation. In *13th ACM Symposium on Principles of Programming Languages*, 1986.

[7] A.J. Demers. CS611 course notes. Cornell University, unpublished.

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

[9] M.J. Gordon, A.J.Milner, and C.P.Wadsworth. *Edinburgh LCF.* Springer-Verlag LNCS 78, 1979.

[10] M.J.C. Gordon. *The Denotational Description of Programming languages.* Springer-Verlag, 1979.

[11] J.Y. Halpern, A.R. Meyer, and B.A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free ? In *11th ACM Symposium on Principles of Programming Languages*, 1984.

[12] P. Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, 1986.

[13] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *13th ACM Symposium on Principles of Programming Languages*, 1986.

[14] J. M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming.* Ph.D. thesis, Massachusetts Institute of Technology, 1987.

[15] C. Martin and C. Hankin. Finding fixed points in finite lattices. In *Functional Programming Languages and Computer Architecture*, Springer-Verlag LNCS 274, 1987.

[16] A. Neirynck, P. Panangaden, and A.J. Demers. Computation of aliases and support sets. In *14th ACM Symposium of Principles of Programming languages*, 1987.

[17] G.D. Plotkin. T-omega as a universal domain. *Journal of Computer and System Sciences*, 17:209–236, 1978.

[18] T.W. Pratt. *Programming Languages - Design and Implementation.* Prentice-Hall, 1984.

[19] P.Wadler and R.J.M. Hughes. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture*, Springer-Verlag LNCS 274, 1987.

[20] J. C. Reynolds. Syntactic control of interference. In *5th ACM Symposium on Principles of Programming Languages*, 1978.

[21] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In *IFIP Congress*, September 1983.

[22] J.E. Stoy. *The Scott-Strachey Approach to Programming Languages Theory.* MIT Press, 1977.

[23] R. D. Tennent. Semantics of interference control. *Theoretical Computer Science*, 27:297–310, 1983.

[24] R.D. Tennent. *Principles of Programming Languages.* Prentice-Hall, 1981.

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

[26] J. Young and P.Hudak. *Finding Fixpoints on Function Spaces.* Technical Report YALEU/DCS/RR-505, Yale University, December 1986.