Proving Properties About Programs Which Share

Tony Clark
Formal Methods Group
Phoenix Building
University of Bradford
West Yorkshire
BD7 1DP
UK
a.n.clark@comp.brad.ac.uk

September 30, 1997

Keywords: alias analysis, abstract interpretation, operational semantics, verification.

Abstract

The analysis of program properties is essential to the production of high quality systems. Unfortunately, the analysis of imperative programs is difficult because they are not referentially transparent. This paper makes a contribution to the analysis of imperative programs by proposing a general semantic model for expressing programs which involve aliasing and using this to develop a deductive system for establishing program properties. The approach is not limited to first order languages since a λ-calculus is used as a universal model of imperative programming. The approach is parameterised with respect to the semantics of aliasing and can therefore be instantiated with respect to the semantics of any particular imperative language.

1 Introduction

The analysis of program properties is essential to the production of high quality systems. The category of programming languages which are referentially transparent (for example the so-called *pure* functional languages) are amenable to analysis techniques mainly because they are free from side-effects. However the majority of languages which are used today are *imperative*, for example the C family. Such languages permit values to be modified by side-effect and are much more difficult to analyse.

In order to facilitate the analysis of imperative programs we require a general model of the sharing (or *aliasing*) and update which occurs during program execution. Such a model must be highly flexible since nearly all imperative programming languages exhibit varying sharing and update characteristics.

The λ-calculus is a universal model of programming languages. Its simplicity makes it highly appealing as an analysis tool. The calculus is higher-order which enables it to elegantly model both data and control abstractions. This paper addresses the problem of using a λ-calculus to perform alias analysis. Given the hypothesis that λ-calculi represent a universal model of (sequential) computation then the result is a general framework for performing alias analysis with respect to any (sequential) programming language.

The initial point of departure is Landin's *sharing machine* [5] which enriches a λ-calculus with sharing and update features and presents a semantics in terms of a state transition machine. Although the sharing machine is suitable as a general model of imperative computation, it is unwieldy when used to prove program properties. We show that a transformation can be applied which results in programs whose imperative semantics can be expressed as a deductive system and then give examples of how this system can be used to prove some properties of imperative programs. We conclude with an analysis of the work and compare it with related research.

2 Definitions

The sharing machine is defined as a state transition system. The states are defined as terms in addition to information which describes the sharing of substructures within terms. This section gives the definitions which are necessary to define the sharing machine.

2.1 Terms

Let V be a set of variables and disjointly let F be a set of function symbols. Each function symbol has an arity which is a positive integer or zero. A *term* is either a variable $v \in V$ or a function symbol $f \in F$ applied to a sequence of terms $(t_1, \ldots, t_n)$ where n is the arity of f. If a term contains no variables then it is called a *ground term* and if $n = 0$ then the term is called an *atom*. The set of variables in a term is $vars(t)$.

A *variable substitution* θ is a partial function from variables to ground terms. A variable substitution can be uniquely extended to a homomorphism which may be applied to any term:

$$\theta(f(t_1,\ldots,t_n)) = f(\theta(t_1),\ldots,\theta(t_n))$$

Given a term $f(t_1,\ldots,t_n)$ each of the sub-terms t_i is at *position* i. A *term accessor* is a function which is applied to a term to yield one of its sub-terms. There is a distinct accessor for each possible sub-term position. For example:

$$\begin{array}{l} 1(f(t_1,t_2,t_3)) = t_1 \\ 2(f(t_1,t_2,t_3)) = t_2 \\ 3(f(t_1,t_2,t_3)) = t_3 \end{array}$$

A *path* p is one of the following:

- the identity path ι for which $\iota(t) = t$.
- a term accessor.
- the composition of two paths: $p_1 \circ p_2$ whose application is defined as follows:
$$(p_1 \circ p_2)(t) = p_1(p_2(t))$$

Given a term t, a *location* in t is a set of paths L such that there is a single term t' such that for each path $p \in L$: $p(t) = t'$. Intuitively, a location contains a single data item which is accessible via different paths through the term.

It will be useful to develop a method for graphically *displaying* a term. A term may be represented as a directed graph whose vertices are labelled with function symbols. A term $f(t_1,\ldots,t_n)$ is represented as a directed graph containing a root vertex v labelled with f and edges e_i leading from v to the root vertices of the graph of each sub-term t_i. Such a graph is easily drawn on paper or a computer screen.

2.2 Sharing

A *sharing* S on a term t is a set of locations in t. Let T be the complete set of sub-terms in t then a sharing on t contains a path p if and only if $p(t) \in T$. A sharing S on a term t is *well formed* when for each pair of paths $p \circ p_1$ and $p \circ p_2$ in t, if p_1 shares with p_2 in S then $p \circ p_1$ shares with $p \circ p_2$ in S.

A sharing is to be used to describe the operational semantics of an imperative language and to prove sharing properties of programs. The operational semantics is defined as a state transition system given as a collection of rules. The system states are ground terms and the rules define how the states are transformed as the program evaluates.

A sharing in the display of a term allows two or more edges to be incident on the same vertex. A special case occurs when an entire term shares with itself. This is to be drawn as a cycle leading to the root vertex of the term.

2.3 Rules

A rule is $t_1 \longmapsto t_2$ with a possible side condition defining how the rule affects the current sharing. The term t_1 is called the *antecedent* and t_2 is called the *consequent*.

A rule is used to rewrite ground terms. A ground term t matches a term t' if there is a variable substitution θ such that $\theta(t') = t$. A rule $t_1 \longmapsto t_2$ matches a ground term t if its antecedent matches t. The rule then rewrites t to the ground term $\theta(t_2)$.

A rule also affects a sharing on a ground term as follows. Let S_1 be a well formed sharing on a ground term t and let $t_1 \longmapsto t_2$ be a matching rule given the variable substitution θ. For each variable $v \in \mathit{vars}(t_1)$ there is a unique path p_v such that $p_v(t_1) = v$. For each variable $v \in \mathit{vars}(t_1)$ there is a set of paths P_v such that for each path $p \in P_v$: $p(t_2) = v$. Let S_2 be a well formed sharing on $\theta(t_2)$. Two paths p_1 and p_2 share in S_2 if and only if:

- the side condition does not explicitly prevent it; and

  1. $p_1 = p \circ p'_1$ and $p_2 = p \circ p'_2$ such that $p'_1(t_2) = p'_2(t_2) = v$ for some variable v; or
  2. $p_1 = p \circ p_x \circ p_y$ and $p_2 = p \circ p_a \circ p_b$ such that there exist paths p_{v_1} and p_{v_2} in t_1 where $p_y \in P_{v_1}$ and $p_b \in P_{v_2}$ and the paths $p \circ p_a \circ p_{v_1}$ and $p \circ p_x \circ p_{v_2}$ share in S_1; or
  3. the side condition explicitly demands it.

Consider the rule $f(v_1, v_2) \longmapsto g(v_1, v_2, v_1)$. When this rule is used to rewrite any ground term which matches the antecedent the resulting sharing will contain the following location $\{1, 3\}$ due to the repeated use of the variable v_1 in the consequent.

Consider the rule $f(v_1, v_2) \longmapsto g(v_2, v_1)$. When this rule is used to rewrite the following ground term: $f(h(k_1, k_2), i(k_3, k_1))$ and the sharing:

$$\{\{1, 2\}, \{2 \circ 1, 2 \circ 2\}, \{2 \circ 1\}, \{1 \circ 2\}\}$$

then the resulting term is $g(i(k_3, k_1), h(k_1, k_2))$ with the following sharing:

$$\{\{1\}, \{2\}, \{2 \circ 1, 1 \circ 2\}, \{1 \circ 2\}, \{2 \circ 2\}\}$$

Rules describe modifications to the display of a term. In such cases it is convenient to think of the display as being achieved using pegs labelled with function symbols (vertices) and elastic string, coloured at the target end, tied between the pegs (edges).

A rule modifies a display *in place* by introducing new pegs and string and by untying the coloured end of existing strings and re-tying the string to a different peg.

The application of a rule to a display d is as follows. Each variable in the antecedent identifies a distinct peg. The display of the consequent is constructed, starting at the root of d and the root of the consequent term. Where the pegs match they are left unaltered and construction proceeds with the corresponding sub-terms. Where the consequent introduces a new function symbol, the display is modified to add a new appropriately labelled peg and the edge leading from the parent peg is untied and re-tied to the new peg. Where the consequent refers to a variable, the display string is untied and re-tied to the peg associated with the variable. When the application of a rule to a display is complete, any pegs and string which are unreachable from the root are removed.

2.4 Update

Given a term and two locations in the term we can *update* the first location with the value in the second by replacing all occurrences of the value in the first location with the value in the second location and arrange so that the two locations are left sharing.

Consider the display of a term. Locations in the term are pegs. Given two locations, if the first is to be updated to contain the value in the second then all strings tied to the first peg are untied and then re-tied to the second peg.

3 A Sharing SECD Machine

The SECD machine is a flexible system for providing an operational semantics for a λ-calculus. The machine is defined in terms of a collection of states and rules which perform calculations by rewriting the states.

In order to provide a model for an imperative language, an SECD machine must support sharing and update. This section describes such an SECD machine.

3.1 Machine States

The SECD machine consists of a set of states and a collection of rules. The states are terms which are constructed from the following term classes:

- a *dit* is an integer, a boolean, a tuple, or a closure.
- integers which are treated as term constants.
- booleans which are treated as term constants.
- tuples. Tuples of arity n are constructed using a function symbol *tuple-n* where each sub-term is a dit. When writing tuples we drop the function symbol, for example *tuple-3*$(1, 2, 3)$ becomes $(1, 2, 3)$.
- sequences which are either the constant *nil*(), written [], or a term *cons*(t, l) where t is a dit and l is a sequence. When writing non empty sequences we use infix notation, for example $1 : 2 : []$ instead of *cons*$(1,$ *cons*$(2, []))$, and $[t]$ instead of $1 : []$. A sequence $t_1 : t_2 : \ldots : t_n : []$ can also be written $[t_1, t_2, \ldots, t_n]$.

- environments which are sequences of tuples of arity 2 where the first component of each tuple is an identifier and the second is a dit.
- program expressions which are drawn from the following syntax definition:

$$E ::= I \mid \lambda I.E \mid EE \mid E \rightarrow E; E \mid (E, \ldots, E) \mid (E)$$

  where I is the syntactic category of program identifiers, $\lambda I.E$ is the syntactic category of functions, EE is the syntactic category of applications, $E \rightarrow E; E$ is the syntactic category of conditional expressions, and $(E, \ldots, E)$ is the syntactic category of tuples. Each non-atomic syntactic category has its own term constructor. For example the term $\mathit{lambda}(i, \mathit{apply}(f, i))$ is written $\lambda i.f(i)$.
- machine instructions which are: @; $\mathit{choose}(e_1, e_2)$ where e_1 and e_2 are program expressions, written $e_1 \rightarrow e_2$; $\mathit{constuple}(n)$ where n is an integer, written $[n]$; and, $\mathit{update}()$ written $:=$.
- closures which are terms $\mathit{closure}(i, \rho, e)$ where i is an identifier, ρ is an environment and e is a program expression. A closure is written $<i, \rho, e>$.
- machine states which are either the constant tuple () or a tuple of arity 4: (s, e, c, d) where s is a sequence of dits called the *stack*, e is an environment, c is a sequence of program expressions and machine instructions called the *control* and d is a machine state called the *dump*.

3.2 Transition Rules

The transition rules for the SECD machine with sharing are defined in figure 1. Each time a rule is used to transform the current machine state, the current sharing is modified using the definition given in §2.3.

Side conditions are required in order to define sharing issues which differ from the default mechanisms which are defined in §2.3. The following side conditions apply to the rules in figure 1:

- rule 2 must arrange for the location which contains the value of the identifier i in the environment to share with the top of the stack.
- rule 5 must arrange for the value of the identifier i in the consequent to share with the argument v in the antecedent[1]
- rule 13 must arrange for the dit v_1 to be updated with the dit v_2. The rule shows that all machine components can potentially be affected by the update; otherwise this rule behaves as normal.

[1] Note that in practice the *degree* of sharing depends upon the particular imperative language which we are modelling. For example we may have *call-by-value* or *call-by-reference* parameter passing.

$$(s, e, k : c, d) \longmapsto (k : s, e, c, d) \tag{1}$$

$$(s, e, i : c, d) \longmapsto (e(i) : s, e, c, d) \tag{2}$$

$$(s, e, (\lambda i.b) : c, d) \longmapsto (<i, e, b> : s, e, c, d) \tag{3}$$

$$(s, e, (e_1 e_2) : c, d) \longmapsto (s, e, e_2 : e_1 : @ : c, d) \tag{4}$$

$$(<i, e_1, b> : v : s, e_2, @ : c, d) \longmapsto ([], (i, v) : e_1, [b], (s, e, c, d)) \tag{5}$$

$$(v : \_, \_, [], (s, e, c, d)) \longmapsto (v : s, e, c, d) \tag{6}$$

$$(s, e, (e_1; e_2 \to e_3) : c, d) \longmapsto (s, e, e_1 : (e_2 \to e_3) : c, d) \tag{7}$$

$$(\mathit{true} : s, e, (e_1 \to e_2) : c, d) \longmapsto (s, e, e_1 : c, d) \tag{8}$$

$$(\mathit{false} : s, e, (e_1 \to e_2) : c, d) \longmapsto (s, e, e_2 : c, d) \tag{9}$$

$$(s, e, (e_1, \ldots, e_n) : c, d) \longmapsto (s, e, e_n : \ldots : e_1 : [n] : c, d) \tag{10}$$

$$(v_1 : \ldots : v_n : s, e, [n] : c, d) \longmapsto ((v_1, \ldots, v_n) : s, e, c, d) \tag{11}$$

$$(s, e, (e_1 := e_2) : c, d) \longmapsto (s, e, e_1 : e_2 : (:=) : c, d) \tag{12}$$

$$(v_2 : v_1 : s, e, (:=) : c, d) \longmapsto (v_2 : s', e', c', d') \tag{13}$$

Figure 1: SECD Transition Rules

Given a program expression x and an environment e, an initial machine state for program execution is $([], e, [x], ())$. By repeatedly applying the sharing machine transition rules (assuming no coding errors in x), the result will be a terminal state $([v], e, [], ())$ where v is the outcome delivered by performing program x. If $\sigma_1 - \sigma_n$ are the intermediate states then a *calculation* describes all the steps:

$$([], e, [x], ()) \longmapsto \sigma_1 \longmapsto \ldots \longmapsto \sigma_n \longmapsto ([v], e, [], ())$$

or equivalently $([], e, [x], ()) \longmapsto^* ([v], e, [], ())$. The following example shows a calculation which performs an update:

$$\begin{array}{l} (s, (i, 1) : e, (i := 0) : c, d) \longmapsto \\ \quad (s, (i, 1) : e, 0 : i : (:=) : c, d) \longmapsto^* \\ \quad\quad (1 : 0 : s, (i, 1) : e, (:=) : c, d) \longmapsto \\ \quad\quad\quad (0 : s, (i, 0) : e, c, d) \end{array}$$

when the value of the identifier i is pushed, the head of the stack shares with the value in the environment. When the value at the head of the stack is updated then all sharing values must be updated to produce a consistent state. The calculation shows that the update will modify the value of i in the environment.

4 A Simplified Semantics

We claim that the SECD machine with sharing is a universal tool in the analysis of imperative programming languages. However, it can prove unwieldy if used to prove program properties. In particular, the machine has a number of components each of which may share with any of the other components. A consequence is that program proofs potentially involve a large amount of book-keeping due to the number of machine components which can share.

In order to facilitate the proof of imperative program properties, we would like the proofs to be as simple as possible. This section describes the key features which complicate the sharing machine and the restrictions which can be imposed in order to define a simple deductive system for program proof.

4.1 Program Transformation

The SECD machine uses a stack in order to hold the results from intermediate calculations. For example, in order to construct a tuple, each tuple component is produced by performing an expression and leaving the result on the stack. When all components have been pushed on the stack then they are popped and the tuple is pushed.

A consequence of using a stack for intermediate results is that the stack may be modified by an update to a data item while it shares with a stack component. The following expression gives a simple example:

$$(x, 1(x) := 3)$$

where the value of x (a tuple) is pushed onto the stack and then modified as a side effect of evaluating $1(x) := 3$.

The use of a stack to hold intermediate calculations can be avoided if we force all values to be named. This has the effect of transferring sharing which would otherwise occur through the stack to occur through the environment. If we also define that control items are not shared (*i.e.* we cannot update code during execution) then the result is that *all* sharing occurs through the environment and the dump. Since the stack and the control are not used for sharing then the dump (with respect to sharing) becomes a stack of environments.

In order to ensure that all data items are named, an arbitrary λ-calculus expression must be transformed. The transformation ensures that an expression is evaluated as a single thread and that each dit produced by a sub-expression is named. A function which performs the transformation is defined in figure 2. Note that we assume that all program constants are named and occur at particular environment locations.

Given a λ-calculus expression e, the transformed expression is *trans*$(e, \mathbf{I})$, where $\mathbf{I}$ is the identity function. The first argument to *trans* is a λ-calculus expression and the second is a continuation mapping identifiers to expressions.

Consider the following λ-calculus expression $(f((1,2), \lambda x.x + 3), 4)$ which

$$
\begin{array}{l}
\mathit{trans}(i, s) = s(i) \\
\\
\mathit{trans}(\llbracket \lambda i.e \rrbracket, s) = \llbracket \textbf{let } v(i) = e_1 \textbf{ in } e_2 \rrbracket \\
\quad \textbf{where} \\
\qquad e_1 = \mathit{trans}(e_1, \mathbf{I}) \\
\qquad v = \mathit{newvar} \\
\qquad e_2 = s(v) \\
\\
\mathit{trans}(\llbracket e_1 e_2 \rrbracket, s) = \mathit{trans}(e_1, \lambda f.\mathit{trans}(e_2, \lambda v.\llbracket \textbf{let } v' = f(v) \textbf{ in } e \rrbracket)) \\
\quad \textbf{where} \\
\qquad v' = \mathit{newvar} \\
\qquad e = s(v') \\
\\
\mathit{trans}(\llbracket e_1 \to e_2; e_3 \rrbracket, s) = \mathit{trans}(e_1, \lambda v.\llbracket v \to e_4; e_5 \rrbracket) \\
\quad \textbf{where} \\
\qquad e_4 = \mathit{trans}(e_2, s) \\
\qquad e_5 = \mathit{trans}(e_3, s) \\
\\
\mathit{trans}(\llbracket (e_1, \ldots, e_n) \rrbracket, s) = \mathit{trans}(e_1, \lambda v_1. \ldots \mathit{trans}(e_n, \lambda v_n.\llbracket \textbf{let } v = (v_1, \ldots, v_n) \textbf{ in } e \rrbracket) \ldots) \\
\quad \textbf{where} \\
\qquad v = \mathit{newvar} \\
\qquad e = s(v) \\
\\
\mathit{trans}(\llbracket e_1 := e_2 \rrbracket, s) = \mathit{trans}(e_1, \lambda v_1.\mathit{trans}(e_2, \lambda v_2.\llbracket \textbf{let } v = v_1 := v_2 \textbf{ in } e \rrbracket)) \\
\quad \textbf{where} \\
\qquad v = \mathit{newvar} \\
\qquad e = s(v) \\
\\
\mathit{trans}(\llbracket e_1; e_2 \rrbracket, s) = \mathit{trans}(e_1, \lambda\_.\mathit{trans}(e_2, s))
\end{array}
$$

Figure 2: Single Threading Transformation

produces the following single threaded expression:

$$\begin{array}{l}\textbf{let } v_1 = (1,2) \textbf{ in} \\ \quad \textbf{let } v_2(x) = x+3 \textbf{ in} \\ \quad\quad \textbf{let } v_3 = (v_1, v_2) \textbf{ in} \\ \quad\quad\quad \textbf{let } v_4 = f(v_3) \textbf{ in} \\ \quad\quad\quad\quad \textbf{let } v_5 = (v_4, 4) \textbf{ in } v_5\end{array}$$

Given an expression produced by *trans*, its evaluation on the sharing machine will only use the head stack location. Furthermore, immediately after a value is constructed at the head of the stack it is popped and added to the environment. As a result, we can dispense with the machine stack and need only pay attention to the environment during program execution.

4.2 A Natural Semantics with Sharing

The operational semantics of program execution can be simplified by dropping the stack. There is still a requirement for serialising certain aspects of program evaluation, but rather than define the semantics in terms of a machine, it is given as a deductive system which defines a relation:

$$\rho, S \vdash e \Rightarrow v, L, \rho', S'$$

where e is a transformed λ-expression which is performed relative to a stack of identifier binding environments ρ and the sharing S to produce the data item v whose program location is L. The environment ρ' and sharing S' are derived from ρ and S after any updates in e have been performed.

The deductive system is given as a collection of rules which use the following definitions. Let S be a sharing, ρ be a sequence of environments, p be a path and L be a location.

Def 1 *The sharing $\bar{S}$ represents pushing a new location onto S. All paths p in S become $p \circ 2$ in $\bar{S}$. The new location may have structure and the context of $\bar{S}$ will determine any new paths which must be present. For example, if the pair $(i, 2)$ is pushed then paths 1, $1 \circ 1$ and $2 \circ 1$ are present in $\bar{S}$.*

Def 2 *The sharing $\bar{\bar{S}}$ represent popping the head location in S. All paths $p \circ 1$ are removed from S and all paths $p \circ 2$ in S become p in $\bar{\bar{S}}$.*

Def 3 *The expression $S(\rho, i)$ is the location of the value of the identifier i in the environment sequence ρ.*

Def 4 *The expression $S[p_1 = p_2]$ is a sharing which is the same as S except that the path p_1 has been modified to share with the path p_2. Note that the path p_1 is added to the sharing if it is not already present. In order to produce a well formed sharing, for any path $p_3 \circ p_2$ in S the expression $S[p_1 = p_2]$ implies that $S[p_3 \circ p_1 = p_3 \circ p_2]$.*

Def 5 $S[e_1, \ldots, e_n] = ((S[e_1]) \ldots)[e_n]$

Def 6 *The expression $S[p = L]$ is defined as follows:*

$$S[p = L] = \begin{cases} S[p = p_1, \ldots, p = p_n] & \textit{when } L \neq \emptyset \wedge p_i \in L \\ S \cup \{\{p\}\} & \textit{otherwise} \end{cases}$$

Def 7 *The expression $\rho(i)$ produces the data item which is bound to the identifier i in the first environment in the sequence ρ. The expression $\rho(S(\rho, i_1), \rho(i_2))$ is a new environment sequence which is the same as ρ except that all paths in the location $S(\rho, i_1)$ contain the data item $\rho(i_2)$.*

Def 8 *The expression $S[L_1 = L_2]$ is a sharing which is defined as follows:*

$$S[L_1 = L_2] = \begin{cases} S[p_1 = L_2, \ldots, p_n = L_2] & \textit{when } L_1 \neq \emptyset \wedge p_i \in L_1 \\ S & \textit{otherwise} \end{cases}$$

Def 9 $\bar{L} = v$ *such that* $v \in \{\bar{L}\}$

Def 10 $\bar{\bar{L}} = v$ *such that* $v \in \{\bar{\bar{L}}\}$

Def 11 *The path to the data item at the head of the environment is* $h = 2 \circ 1 \circ 1$.

The deduction rules for performing transformed λ-expressions are defined in figure 3. In addition to providing a means to perform a program, these rules may also be used to test theorems about programs which share. The ability to define the context of expression evaluation via the environment ρ and the sharing S allows the same program expression to be tested in a variety of contexts. The rest of this section shows some examples of how the rules are used to prove sharing theorems about imperative programs.

5 Examples of Program Proof

The natural semantics can be used as a deductive system for proving properties about programs which share. This section gives a number of examples of such proofs. Note that some of the proofs omit the program transformation when it is not required. For example, the expression: **let** $i = f(x)$ **in** i is equivalent to the expression $f(x)$. Note also that locations which are unimportant to the proof, for example which do not share, are omitted. Since the calculus is intended to be the target of a program transformation from an imperative language, each example is given as a program fragment in a C-like language.

Theorem 1 *The result of performing the following program is to print the value* 20*:*

```
{ int i = 20;
  print(i);
}
```

$$\rho, S \vdash i \Rightarrow \rho(i), S(\rho, i), \rho, S \tag{14}$$

$$\frac{[(i_1, <i_2, \rho, e_1>)] : \rho, \bar{S}[2 \circ h = 2] \vdash e_2 \Rightarrow v, L, \_ : \rho', S'}{\rho, S \vdash \textbf{let } i_1(i_2) = e_1 \textbf{ in } e_2 \Rightarrow v, \bar{\bar{L}}, \rho', \bar{\bar{S}}'} \tag{15}$$

$$\frac{[(i_1, \rho(i_2))] : \rho, \bar{S}[2 \circ 1 = S(\rho, i_1) \circ 2] \vdash e \Rightarrow, v, L, \_ : \rho', S'}{\rho, S \vdash \textbf{let } i_1 = i_2 \textbf{ in } e \Rightarrow v, \bar{\bar{L}}, \rho', \bar{\bar{S}}'} \tag{16}$$

$$\frac{[(i, (\rho(i_1), \ldots, \rho(i_n)))] : \rho, \bar{S}[k \circ h = S(\rho, i_k) \circ 2] \vdash e \Rightarrow v, L, \_ : \rho', S'}{\rho, S \vdash \textbf{let } i = (i_1, \ldots, i_n) \textbf{ in } e \Rightarrow v, \bar{\bar{L}}, \rho', \bar{\bar{S}}'} \tag{17}$$

$$\frac{\begin{array}{c} \rho_1(i_2) = <i_4, \rho_2, e_2> \\ S_4 = \bar{S}_1[2 \circ 1 \circ 1 = 2 \circ (S_1(\rho_1, i_2)) \circ 2, 2 \circ 1 = S_1(\rho_1, i_3) \circ 2] \\ ((i_4, \rho_1(i_3)) : \rho_2) : \rho_1, S_4 \vdash e_2 \Rightarrow v_1, L_1, \_ : \rho_3, S_2 \\ [(i_1, v_1)] : \rho_3, \bar{\bar{S}}_2[h = \bar{\bar{L}}_1 \circ 2] \vdash e_1 \Rightarrow v_2, L_2, \_ : \rho_4, S_3 \end{array}}{\rho_1, S_1 \vdash \textbf{let } i_1 = i_2(i_3) \textbf{ in } e_1 \Rightarrow v_2, \bar{\bar{L}}_2, \rho_4, \bar{\bar{S}}_3} \tag{18}$$

$$\frac{\begin{array}{c} \rho(i_2) = p \\ [(i, p(\rho(i_3)))] : \rho, \bar{S}[h = p \circ (S(\rho, i_3)) \circ 2] \vdash e \Rightarrow v, L, \_ : \rho', S' \end{array}}{\rho, S \vdash \textbf{let } i_1 = i_2(i_3) \textbf{ in } e \Rightarrow v, \bar{\bar{L}}, \rho', \bar{\bar{S}}'} \tag{19}$$

$$\frac{\begin{array}{c} \rho(i) = \mathit{true} \\ \rho, S \vdash e_1 \Rightarrow v, L, \rho', S' \end{array}}{\rho, S \vdash i \rightarrow e_1; e_2 \Rightarrow v, L, \rho', S'} \tag{20}$$

$$\frac{\begin{array}{c} \rho(i) = \mathit{false} \\ \rho, S \vdash e_2 \Rightarrow v, L, \rho', S' \end{array}}{\rho, S \vdash i \rightarrow e_1; e_2 \Rightarrow v, L, \rho', S'} \tag{21}$$

$$\frac{\begin{array}{c} S' = S[h = S(\rho, i_3) \circ 2, S(\rho, i_2) \circ 2 = S(\rho, i_3) \circ 2] \\ [(i_1, \rho(i_3))] : (\rho(S(\rho, i_2), \rho(i_3))), S' \vdash e \Rightarrow v, L, \_ : \rho', S'' \end{array}}{\rho, S \vdash \textbf{let } i_1 = i_2 := i_3 \textbf{ in } e \Rightarrow v, \bar{\bar{L}}, \rho', \bar{\bar{S}}''} \tag{22}$$

Figure 3: Natural Semantics of Transformed Expressions

or equivalently:

$$[[(i,10)]],\{\{h\}\} \vdash i := 20 \Rightarrow 20, \{\{h\}\}, [[(i,20)]], \{\{h\}\}$$

The proof of this theorem is a single application of rule 22, QED.

The following theorem shows how sharing is introduced between components of the environment and the result of updating an environment component which shares.

Theorem 2 *The result of performing this program is* 20*:*

```
{ int i1 = 10;
  int *i2 = &i1;
  *i2=20;
  print(i1);
}
```

or equivalently:

$$[[(i_1,10)]],\{\{h\}\} \vdash \mathbf{let}\ i_2 = i_1\ \mathbf{in}\ i_2 := 20 \Rightarrow 20, \{\{h\}\}, [[(i_1,20)]], \{\{h\}\}$$

To prove this theorem we apply rule 16 and then prove the following:

$$\begin{array}{l} [(i_2,10)] : [[(i_1,10)]], \{\{h, h \circ 2\}\} \vdash \\ \quad i_2 := 20 \Rightarrow 20, \\ \{h, h \circ 2\}, [(i_2,20)] : [[(i_1,20)]], \{\{h, h \circ 2\}\} \end{array}$$

which is true by rule 22, QED.

The following theorem shows how sharing arises in structures and how accessor functions can be used to side effect a structure component.

Theorem 3 *The result of performing the following theorem is* 20*:*

```
{ int i1 = 10;
  struct {
    int *fst;
    int *snd;
  } i2 = { &i1, &i1 };
  int *i3 = i2.fst;
  *i3 = 20;
  print(i1);
}
```

or equivalently:

$$\begin{array}{l} [], \emptyset \vdash \\ \quad \mathbf{let}\ i_1 = 10\ \mathbf{in} \\ \quad\quad \mathbf{let}\ i_2 = (i_1, i_1)\ \mathbf{in} \\ \quad\quad\quad \mathbf{let}\ i_3 = 1(i_2)\ \mathbf{in} \\ \quad\quad\quad\quad \mathbf{let}\ \_ = i_3 := 20\ \mathbf{in}\ i_2 \Rightarrow (20,20), \\ \emptyset, [], \emptyset \end{array}$$

To prove this theorem we firstly apply rule 16 to produce the following theorem:

$$\begin{array}{l} [[(i_1, 10)]], \{\{h\}\} \vdash \\ \quad \textbf{let } i_2 = (i_1, i_1) \textbf{ in} \\ \quad\quad \textbf{let } i_3 = 1(i_2) \textbf{ in} \\ \quad\quad\quad \textbf{let } \_ = i_3 := 20 \textbf{ in } i_2 \Rightarrow (20, 20), \\ \emptyset, [[(i_1, 20)]], \{h\} \end{array}$$

then apply the rule 17 to produce the following theorem:

$$\begin{array}{l} [(i_2, (10, 10))] : [[(i_1, 10)]], \{\{1 \circ h, h \circ 2, 2 \circ h\}\} \vdash \\ \quad \textbf{let } i_3 = 1(i_2) \textbf{ in let } \_ = i_3 := 20 \textbf{ in } i_2 \Rightarrow (20, 20), \\ \{h\}, [(i_2, (20, 20))] : [[(i_1, 20)]], \{\{1 \circ h, 2 \circ h, h \circ 2\}\} \end{array}$$

then apply rule 19 to produce the following theorem:

$$\begin{array}{l} [[(i_3, 10)], [(i_2, (10, 10))], [(i_1, 10)]], \{\{h, 2 \circ h \circ 2, 1 \circ h \circ 2, h \circ 2 \circ 2\}\} \vdash \\ \quad \textbf{let } \_ = i_3 := 20 \textbf{ in } i_2 \Rightarrow (20, 20), \\ \{h \circ 2\}, [[(i_3, 20)], [(i_2, (20, 20))], [(i_1, 20)]], \{\{h, 2 \circ h \circ 2, 1 \circ h \circ 2, h \circ 2 \circ 2\}\} \end{array}$$

which is true by an application of rule 22 and then rule 14, QED.

The following theorem shows how induction can be used to prove an invariant which involves sharing within a structure. Consider a table represented as a pair (l_1, l_2). Both l_1 and l_2 are sequences of pairs. Each pair in both sequences is of the form (k, v) where k is a key and v is an entry.

An invariant on a table is that every entry in the list l_2 shares with some entry in the list l_1. We require an operation which inserts a new pair into l_1 given the key of the l_2 entry which is to share. The implementation shown in figure 4 is proposed. This is equivalent to the following functional implementation using the proposed calculus:

$$\begin{array}{l} \textbf{let } \mathit{insert}(k_1, k_2, t) = \\ \quad\quad \textbf{letrec } \mathit{find}(k, l) = \mathit{null}(l) \rightarrow \epsilon; (1 \circ 1(l) = k \rightarrow 2 \circ 1(l); \mathit{find}(k, 2(l))) \\ \quad\quad \textbf{in} \\ \quad\quad\quad \textbf{let } v = \mathit{find}(k_2, 1(t)) \\ \quad\quad\quad \textbf{in } 2(t) := (k_1, v) : 2(t); t \end{array}$$

Theorem 4 *Given a table v_1 for which the invariant condition is true then after $insert(v_1, v_2, t)$ the condition is still true. This theorem can be stated in terms of the performance relation as:*

$$\begin{array}{l} [(t, v_1), (k_1, v_2), (k_2, v_3)] : \rho, S \vdash \\ \quad e \Rightarrow (1(v_1), ((v_2, z) : 2(v_1))), \\ \{2 \circ 1 \circ 2 \circ h, 2 \circ 1 \circ p \circ 1 \circ h\}, \\ [(t, (1(v_1), ((v_2, z) : 2(v_1)))), (k_1, v_2), (k_2, v_3)] : \rho, \\ S[2 \circ 1 \circ 2 \circ h = 2 \circ 1 \circ p \circ 1 \circ h] \end{array}$$

where e is the body of the function insert, z is the entry in the first component of the table v_1 associated with key v_3, and p is an accessor.

```
struct Rec { Key k; Entry *e };
struct List { Rec hd; List *tl; };
struct Table { List *l1; List *l2; };

Entry *find(Key k, List *l)
{
  if(l == NULL)
    error("cannot find key.");
  else if(l->hd.k == k)
    return l->hd.e;
  else
    return find(k,l->tl);
}

Table insert(Key k1, Key k2, Table t)
{
  Entry *v = find(k2,t.l1);
  t.l2 = new List(Rec(k1,v),t.l2);
  return t;
}
```

Figure 4: A Program for Inserting an Entry in a Table

The following lemma is used to prove this theorem:

Lemma 1 *Given a key v_1 and a list of pairs v_2 which contains v_1 as a key then the following theorem is true:*

$$\begin{array}{l}[(k, v_1), (l, v_2)] : \rho, S \vdash \\ \qquad \mathit{null}(l) \to \epsilon; (1 \circ 1(l) = k \to 2 \circ 1(l); \mathit{find}(k, 2(l))) \Rightarrow z, \\ \{2 \circ 1 \circ p \circ 2 \circ 1 \circ 2 \circ 1\}, [(k, v_1), (l, v_2)], S\end{array}$$

The proof is by induction on the length of the list l. Since we know that list v_2 contains key v_1 then the list is not empty; by applying rule 21 we get the following theorem:

$$\begin{array}{l}[(k, v_1), (l, v_2)] : \rho, S \vdash \\ \qquad 1 \circ 1(l) = k \to 2 \circ 1(l); \mathit{find}(k, 2(l)) \Rightarrow z, \\ \{2 \circ 1 \circ p \circ 2 \circ 1 \circ 2 \circ 1\}, [(k, v_1), (l, v_2)], S\end{array}$$

Now since either $1 \circ 1(v_2) = v_1$ or $1 \circ 1(v_1) \neq v_1$ we proceed by case analysis:

- if $1 \circ 1(v_2) = v_1$ then by applying rule 20 we get the following theorem:

$$\begin{array}{l}[(k, v_1), (l, v_2)] : \rho, S \vdash \\ \qquad 2 \circ 1(l) \Rightarrow z, \\ \{2 \circ 1 \circ p \circ 2 \circ 1 \circ 2 \circ 1\}, [(k, v_1), (l, v_2)], S\end{array}$$

  where $z = 2 \circ 1(v_2)$ and $p = \iota$.

- if $1 \circ 1(v_2) \neq v_1$ then by applying rule 21 we get the following theorem:

$$\begin{array}{l}[(k,v_1),(l,v_2)] : \rho, S \vdash \\ \qquad \mathit{find}(k, 2(l)) \Rightarrow z, \\ \{2 \circ 1 \circ p \circ 2 \circ 1 \circ 2 \circ 1\}, [(k,v_1),(l,v_2)], S\end{array}$$

and then by rule 19 we get the following theorem:

$$\begin{array}{l}[(k,v_1),(l,2(v_2))] : [(k,v_1),(l,v_2)] : \rho, \bar{S}[2 \circ 1 \circ 2 \circ 1 = 2 \circ 1 \circ 2 \circ 1 \circ 2] \vdash \\ \qquad \mathit{null}(l) \to \epsilon; (1 \circ 1(l) = k \to 2 \circ 1(l); \mathit{find}(k, 2(l))) \Rightarrow z, \\ \{2 \circ 1 \circ p \circ 2 \circ 1 \circ 2 \circ 1, 2 \circ 1 \circ p \circ 2 \circ 1 \circ 2 \circ 1 \circ 2\}, \\ [(k,v_1),(l,2(v_2))] : [(k,v_1),(l,v_2)], \bar{S}[2 \circ 1 \circ 2 \circ 1 = 2 \circ 1 \circ 2 \circ 1 \circ 2]\end{array}$$

which is true by induction for some p and z, QED.

The proof of theorem 4 is as follows. We ignore the binding of the function *find* in the body of *insert* since the function is closed and does affect the sharing in the table. By applying 19 and using lemma 1 we get the following theorem:

$$\begin{array}{l}[(v,z)] : [(t,v_1),(k_1,v_2),(k_2,v_3)] : \rho, \bar{S}[h = 2 \circ 1 \circ p \circ 1 \circ h \circ 2] \vdash \\ \quad \mathbf{let}\ \_ = 2(t) := (k_1, v) : 2(t)\ \mathbf{in}\ t \Rightarrow 1(v_1) : ((v_2, z) : 2(v_1)), \\ \{1 \circ h, 2 \circ 1 \circ 2 \circ h \circ 2, 2 \circ 1 \circ p \circ 1 \circ h \circ 2\}, \\ (\bar{S}[1 \circ h = 2 \circ 1 \circ p \circ 1 \circ h \circ 2])[2 \circ 1 \circ 2 \circ h \circ 2 = 2 \circ 1 \circ p \circ 1 \circ h \circ 2]\end{array}$$

Since the expression binds the identifier _ and then ignores it, we can use rule 22 without adding a binding to the environment, producing the following theorem:

$$\begin{array}{l}[(v,z)] : [(t,(1(v_1),(v_2,z) : 2(v_1)))),(k_1,v_2),(k_2,v_3)] : \rho, \bar{S}[h = 2 \circ 1 \circ p \circ 1 \circ h \circ 2] \vdash \\ \quad t \Rightarrow 1(v_1) : ((v_2, z) : 2(v_1)), \\ \{1 \circ h, 2 \circ 1 \circ 2 \circ h \circ 2, 2 \circ 1 \circ p \circ 1 \circ h \circ 2\}, \\ (\bar{S}[1 \circ h = 2 \circ 1 \circ p \circ 1 \circ h \circ 2])[2 \circ 1 \circ 2 \circ h \circ 2 = 2 \circ 1 \circ p \circ 1 \circ h \circ 2]\end{array}$$

which is true by rule 14, QED.

The theorems in this section show that the use of a λ-calculus defined using natural semantics with sharing is a flexible way of analysing imperative programs. In particular we have shown that the approach can handle both open and closed programs (*i.e.* with and without the concrete dits).

The approach is flexible with respect to the semantics of sharing. In order to tailor the calculus to a particular programming language, the definition of parameter passing, identifier binding and builtin function calling are readily modified.

6 Conclusion

This paper has proposed a general model for the analysis of imperative programs in terms of a λ-calculus. Since λ-calculi are claimed to be a universal model of computation we claim that the model is suitable for analysing any concrete imperative programming language given a suitable transformation to the calculus.

The operational semantics of a *sharing* λ-calculus expression is conveniently expressed using a sharing SECD machine and we have shown that the semantics can be greatly simplified (with respect to a collection of assumptions) by performing a source-to-source transformation on the expression.

The resulting semantics can be used as a deductive system in order to prove sharing theorems about programs. A number of examples of proof have been given including both open and closed programs. The final example has shown how induction can be used with respect to arbitrary sized data items.

This work is closely related to the abstract interpretation of programs in order to analyse aliasing properties, for example see [2] for a general overview of this area. The semantic models used to analyse programs are essentially first order, for example see [8] and [3]. Higher-order languages, such as λ-calculi, can be used to encode a wide variety of data and control abstractions without requiring extra programming constructs. In this sense we claim that the model presented in this paper is a universal framework for analysing imperative programs.

One of the main reasons for performing alias analysis is to establish *may-alias* and *must-alias* properties. We have given an example of establishing a must-alias property (theorem 4). See [7] and [4] for more details.

Another approach to modelling programming languages with sharing is *denotational semantics*. Here a program is viewed as a function over a semantic domain. The domain includes a store which associates memory addresses with dits. We believe the approach adopted here is more amenable to program proof. See [9] and [1] for more details about denotational semantics.

We have used a *natural semantics* approach which is derived from a *transition machine* approach. ML is a higher-order language with sharing which uses a natural semantics, see [6] for more details.

References

[1] Allison, L. 1986 *A Practical Introduction to Denotational Semantics.* Cambridge Computer Science Texts, 23.

[2] Deutsch, A. 1994 *Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting.* ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, Orlando FL, June 20 –24, pp 230 – 241.

[3] Deutch, A. 1992 *A Storeless Model of Aliasing using Finite Representations of Right Regular Equivalence Relations.* Proceedings of the IEEE 1992 International Conference on Computer Languages, Oakland California, pp 2 – 13.

[4] Landi, W. 1992 *Undecidability of Static Analysis.* Letters on Programming Languages and Systems, 1(4).

[5] Landin, P. J. 1965 *A Correspondence Between Algol 60 and Church's Lambda-Notation.* Communications of the ACM, 8, pp 89 – 101.

[6] Milner, R., Tofte, M., Harper, R. 1990 *The Definition of Standard ML.* The MIT Press.

[7] Ramalingam G. 1994 *The Undecidability of Aliasing.* ACM Transactions on Programming Languages and Systems, 16(5), pp 1467 – 1471.

[8] Ruf, E. 1995 *Context Insensitive Alias Analysis Reconsidered.* ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, La Jolla CA, June 18 – 21, pp 13 – 22.

[9] Stoy, J. 1977 *Denotational Semantics: The Scott-Stratchey Approach to Programming Language Theory.* The MIT Press.