Detecting Sharing of Partial Applications in Functional Programs

Benjamin Goldberg and Paul Hudak

Research Report YALEU/DCS/RR-526
March 1987

This research was supported in part by the Department of Energy under grant FG02-86ER25012 and a grant from Unisys Paoli Research Center.

Detecting Sharing of Partial Applications in Functional Programs *†

Benjamin Goldberg
Paul Hudak

Research Report YALEU/DCS/RR-526
March 1987

Yale University
Department of Computer Science

Abstract

A method is presented for detecting sharing of partial function applications in higher order functional programs. Such sharing occurs when there are several references to variables that are bound to a particular function application. In order to provide an interprocedural analysis, a non-standard semantics is defined for a lazy, higher-order functional language such that the meaning of a program is information about the sharing that occurred during its execution. An abstraction of this non-standard semantics is presented so that useful, although less complete, sharing information can be provided at compile-time.

In the second part of this paper, we utilize sharing detection in order to provide an efficient method for ensuring full laziness during program execution. A refinement of the method used to generate Hughes' super-combinators is discussed. Super-combinators insure that no unnecessary computation is performed when sharing occurs in a program. Unfortunately, the algorithm used to generate super-combinators assumes that every function application is shared and some unnecessary overhead is incurred while executing super-combinators. The refined super-combinators, called super-duper combinators, presented in this paper incur no unnecessary overhead in the cases where no sharing occurs.

*This research was supported in part by the Department of Energy under grant FG02-86ER25012 and a grant from the Unisys Paoli Research Center

†This paper also appeared in the proceedings of the Functional Programming and Computer Architecture Conference, September 1987

1 Introduction

One of the more promising general approaches to the execution of functional languages is *graph reduction*. In graph reduction, the program is represented, along with the data, as a graph which gets transformed, or *reduced*, according to the reduction rules of the lambda calculus [5,11]. Execution of the program is finished when the graph has been reduced to a normal form.

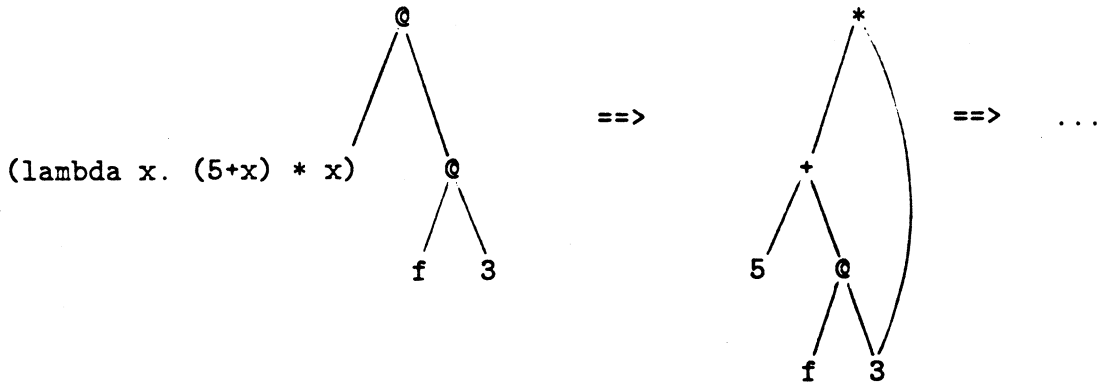
The most significant aspect of graph reduction is its ability to copy references to expressions (subgraphs) instead of the expressions themselves during beta substitution. For example, given the expression:

$$(\lambda x.\ (5 + x) * x)(f\ 3)$$

and some function f, the lambda calculus specifies that reduction proceeds via the substitution of $(f\ 3)$ for every occurrence of x within the expression $(x + 5) * x$. In a purely lexical implementation, the following transformation would take place:

$$(\lambda x.\ (5 + x) * x)(f\ 3) \Rightarrow (5 + (f\ 3)) * (f\ 3)$$

and $(f\ 3)$ would be computed once for each reference to x. Graph reduction, however, performs the reduction by substituting a pointer to $(f\ 3)$ for each occurrence of x in the lambda body. This can be represented graphically in the following way:



Therefore, the value of $(f\ 3)$ need only be computed once upon the first demand. Every subsequent reference sees only the resulting value.

In the case where there are several reference to an expression e we say that e is *shared*. In this paper the expressions that are of interest are *partial applications* of functions. Given a function definition of the form,

$$f\ x_1 \ldots x_n = \ \ldots$$

any application of f to k arguments, where $k < n$, is a partial application of f. An application of f to n arguments is a *complete* application.

This paper presents a method in which we can determine at compile time if a partial application of a given function may be shared during evaluation of the program. While this analysis is interesting for its own sake, we will show how it can be used to increase the efficiency of functional language implementations.

For each function f, as defined above, the analysis will determine the maximum number of times that a partial application of f to k arguments is referenced, for each value of $k < n$.

2 A Naive Approach

A naive approach to detecting sharing of partial applications would be to examine how each partial application is used. If a particular application of f, say $(f\ y_1 \ldots y_k)$, occurs only once and is not passed as an argument to another function, then we can be sure that it is not shared. However, if $(f\ y_1 \ldots y_k)$ is passed as an argument to another function, then we must conservatively assume that it is shared. A more sophisticated scheme, such as the one presented in section 3, would perform an "interprocedural" analysis to detect if $(f\ y_1 \ldots y_k)$ is shared even when passed as an argument to another function.

It may appear that when a partial application is passed as an argument to a function, we need only look at the number of times the corresponding formal parameter occurred *lexically* in the body of the function to determine if the partial application is shared. If this were the case, a syntactic analysis in which bound variables are simply counted would suffice. However, this is not the case for (at least) two reasons:

1. The programs are (assumed to be) written in a lazy functional language. Thus, even though a bound variable may occur several times in the body of a function, it may never actually be referenced. For example in the program,

```
{ f x y == x + y;
  h a b == 1;
  g c == h c c;
  result g (f 1);
  }
```

the bound variable corresponding to `(f 1)` in the body of **g** occurs lexically several times although the partial application is never actually referenced.

2. Because programs are written in a higher-order functional language, a variable bound to a partial application may itself be passed as an argument to an "unknown" function (i.e. some other bound variable). For example, in the program,

```
{   f x y == x+y;
    h c d == c d + c 2;
    g a  b == b a 1;
    g (f 1) h;
    }
```

the variable **a** that is bound to `(f 1)` is passed as an argument to the function that **b** is bound to. A somewhat more sophisticated analysis is required to determine the behavior of the function that **b** is bound to.

3 Semantic-based sharing analysis

Our methodology for detecting sharing is one that has given promising results in recent work on other aspects of functional languages [1,6]. This method involves defining a *non-standard* denotational semantics for a functional language such that the result of executing a program using these semantics will be information about the sharing that occurred during the execution. However, since to obtain such complete information we need to actually run the program, this method is not a tool that can be used at compile time.

We therefore define an *abstraction* of the non-standard semantics that will provide us with useful, although less complete, sharing information. The compile-time "execution"

of the program using an abstraction of an exact semantics is called *abstract interpretation* [4,9] and has been previously applied to analyzing the strictness properties of functions in a program [2,3].

Before proceeding, an important point must be made. In graph reduction, as defined here, sharing can *only* occur when a variable that has been bound to a partial application is referenced several times. Our analysis does *not* perform common subexpression elimination (cse). Cse must have occurred prior to sharing detection and have been performed by the abstraction of common subexpressions from the expressions in which they occur. For example, $(f3) + (f3)$ must be translated into $(\lambda x.\ x + x)(f\ 3)$ for sharing of $(f3)$ to occur. Even though two expressions may be identical and occur within the same lexical scope, unless they are occurrences of the same bound variable no sharing will occur.

3.1 Representing Sharing Information

When "executing" a program using a non-standard semantics for determining sharing, the value of each expression must contain enough information to infer the sharing that occurred while the expression was being evaluated. The value that is returned by the program will therefore contain information about all the sharing that occurred while the program was executing. In this section we describe a domain S, called the *sharing domain*, of values that contain the necessary sharing information.

Each value in S must contain information indicating whether or not that value represents a partial application. When a variable that is bound to such a value is referenced several times it can then be determined whether or not sharing has occurred. If a variable is not bound to a partial application then no sharing will occur no matter how many times the variable is referenced. In addition, different partial applications must be represented by different values, even if the partial applications are lexically identical (and have the same value in the standard semantics). For example, during the evaluation of the program,

```
{ f x y == x+y;
  g a b == a 1 + b 1;
  result g (f 1) (f 1);
  }
```

the variables **a** and **b** in the body of **g** are bound to **different partial applications** and no sharing of **(f 1)** occurs.

When a value is returned as the result of evaluating an expression, the value must also contain information about which partial applications were shared while the expression was being evaluated. For example, given the following function definitions,

```
{ f x y == x+y;
  j z w == z+w;
  g a == h a (j 1);
  h c d == (c 1 + c 2) * (d 3 + d 4);
  result g (f 1);
  }
```

the value resulting from executing the program indicate that an application of **f** to a single argument and an application of **j** to a single argument were each referenced twice and therefore shared.

Because the result of evaluating an expression in the standard semantics may be a function, a value in S must be able to capture the behavior of a function over elements of the sharing domain. In section 2 it was seen that a purely lexical sharing analysis fails because partial applications may be passed to "unknown" functions.

Based on the above discussion, any value $s \in S$ resulting from the evaluation of an expression e is defined to be a triple of the form,

$$\langle p,\ l,\ f \rangle$$

where:

- The first element, p, indicates whether e represents a partial application and, if so, provides enough information to differentiate it from other partial applications. This information is called the p-value of e.
- The second element, l, is a list of partial applications that were referenced during the evaluation of e and is called the l-value of e.

- The third element, f, is a function over S that captures the higher order behavior of e and is called the f-value of e.

The three elements of a value in S are described in detail in the rest of this section.

The p-value

The p-value of an expression e is a tuple of the form

$$[id\ v_0 \ldots v_n]$$

where id is an identifier and $v_0 \ldots v_n$ are natural numbers. This tuple can be interpreted as follows: "Expression e represents the application of the bound variable id to n arguments, where id was itself bound to a partial application. The result of this application, and thus the value of e, is also a partial application."

For example, in the program,

```
{  f x y z == x+y+z;
   g b == b 1;
   result g (f 1);
   }
```

the p-value of `(b 1)` inside the body of `g` is [`b 1 1`] since the variable `b` is bound to a partial application and was applied to one argument. Likewise, the p-value of `(f 1)` in the expression `g (f 1)` would be [`f 1`] since `(f 1)` represents an occurrence of a partial application (namely `f`) applied to one argument.

In a p-value the value of each v_i is `1` (however, the same representation will be used when describing other partial applications in l and the value of each v_i will vary). When an expression e gets applied to an argument and the result is a partial application, then the p-value for the result is the p-value for e with an additional `1` in the tuple.

Since the p-value of `(b 1)` does not indicate which function is actually partially applied, how can it be determined that the partial application bound to `b` is really `(f 1)`? When `g` was called, the p-value of its argument was [`f 1`]. However, the p-value of the corresponding

formal parameter, b, was bound to [b 1]. After the body of *g* has been evaluated, [b 1] is replaced by [f 1] in the resulting *p*-value to show that that the occurrence of b was actually an occurrence of (f 1). After this substitution, the correct *p*-value for the program, namely [f 1 1] is returned.

In the standard semantics, when a function is applied the body of the function is evaluated with the values of the arguments substituted for (or bound to) the formal parameters. In the sharing semantics described in section 3.2, *two* substitutions occur during the evaluation of a function application. The first substitution occurs on entry to the function when the *p*-value of each formal parameter is bound to a "dummy value" (such as [b 1] above). The second substitution occurs (as described above) after the body of the function has been evaluated and any "dummy value" occurring in the result is replaced by the original value of the corresponding argument.

Why not simply bind the formal parameter to the value of the corresponding argument in a function application (as is done in the standard semantics and most non-standard semantics of functional languages)? A problem arises in the following program:

```
{  f x y z == x+y+z;
   g a b == a 1 + b 2;
   result g (f 1) (f 1);
   }
```

Although both arguments to g evaluate to the same value in the standard semantics, they represent different partial applications. Therefore, the corresponding formal parameters, a and b, must be recognized as being bound to different partial applications. If a and b were both bound to the value of (f 1) then it would incorrectly appear as though they represent the same partial application. If this were the case, it would seem that (f 1) was shared in the body of g, which is untrue. One solution to this would be to create a unique identifier for every partial application. However, as discussed in section 3.4 creating new identifiers creates a termination problem for the analysis.

Instead of creating a unique identifier, we use the names of the formal parameters of g as "dummy" names to distinguish between the two partial applications of f. The process

of binding the p-values of formal parameters to "dummy" values on entry to a function and back-substituting real values into the result is described in section 3.1.1.

The l-value

The l-value is the second element of a value in S for an expression e. It is a *set* of tuples, $\{t_1, \ldots, t_n\}$, where each tuple, t_i has the same form as a p-value, namely

$$[id\ v_0 \ldots v_n]$$

The value of each v_i is significant (unlike in a p-value) and represents the maximum number of times that an application of the variable id to i arguments occurred during the evaluation of e. The variable id must itself bound to a partial application.

For example, the tuple [b 1 2 1] indicates that the variable b is was bound to a partial application and that there was:

- one occurrence of b applied to no arguments,
- two occurrences of a partial application of b to one argument (and thus was shared), and
- one occurrence of a partial application of b to two arguments.

Given the program,

```
{ f x y == x+y;
  j z w == z+w;
  g a == h a (j 1);
  h c d == (c 1 + c 2) * (d 3 + d 4);
  result g (f 1);
  }
```

the l-value of the result expression, `g (f 1)`, would be:

{[g 1 1], [h 1 1 1], [f 1 2 1], [j 1 2 1]}

From this l-value, we can see that the only sharing that occurred was of a partial application of `f` and a partial application of `j`, each applied to a single argument.

In the body of `h`, the l-value of the expression `(d 3 + d 4)` would be $\{[\mathtt{d}\ 2\ 1]\}$ since `d` is bound to the partial application `(j 1)` and occurs twice. Although `d` is applied to a single argument twice, namely `(d 3)` and `(d 4)`, there is no sharing either of those applications.

Just like a p-value, the l-value of the body of a function may contain "dummy" partial application names corresponding to the names of the formal parameters (such as `d` above). Before the value of the function application returns, all such "dummy" names are replaced by inserting the p-values of the corresponding actual parameters into the appropriate tuples.

Merging l-values

During the evaluation of an expression e several applications of the same variable may have occurred. Since the l-value for the e must contain the maximum sharing information for partial applications of each bound variable, the sharing information for the same variables must be merged.

The function that merges two tuples representing applications of the same variable is called merge_tuple and is defined as follows:

$$\begin{aligned}&\text{merge\_tuple}([id\ v_0 \ldots v_n], [id\ v'_0 \ldots v'_m]) = \\ &\qquad [id\ (v_0 + v'_0)\ \max(v_1,\ v'_1)\ \ldots\ \max(v_n,\ v'_n)\ v'_{n+1} \ldots v'_m]\end{aligned}$$

where both tuples have the same id and it was assumed (wlog) that $m \geq n$. Note that the only additional sharing caused by merging the tuples is the sum of the number occurrences in each tuple of id applied to no arguments, namely $v_0 + v'_0$. The number of occurrences of id applied to i arguments is the maximum such number in the two tuples, namely $\max(v_i, v'_i)$. If two tuples do not represent application of the same id, then merge_tuple cannot be applied to them.

Since an l-value is a set of tuples, the function merge takes two sets of tuples and merges them as follows:

$$\text{merge}(l_1, l_2) =$$

$$\{\text{merge\_tuple}(t_i, t'_j) \mid t_i \in l_1, t'_j \in l_2 \text{ and } \text{id}(t_i) = \text{id}(t'_j)\}$$
$$\cup$$
$$\{t_i \mid t_i \in l_1 \text{ and } \forall t'_j \in l_2,\ \text{id}(t_i) \neq \text{id}(t'_j)\}$$
$$\cup$$
$$\{t'_j \mid t'_j \in l_2 \text{ and } \forall t_i \in l_1,\ \text{id}(t'_j) \neq \text{id}(t_i)\}$$

where each t_i and t'_i is a tuple and $id(t_i)$ is the bound variable associated with t_i.

Given the program,

```
{ f x == x 1 2 + x 2 3;
  g b c d == b + c + d;
  result f (g 1);
  }
```

the l-values of both `(x 1 2)` and `(x 2 3)` in the body of `f` will be {[x 1 1 1]}. Since both these expressions occur in `(x 1 2 + x 2 3)`, the resulting l-value is:

$$\text{merge}(\{[\text{x 1 1 1}]\}, \{[\text{x 1 1 1}]\}) \Rightarrow \{[\text{x 2 1 1}]\}$$

The f-value

The f-value is the third element of the value in S for an expression e and reflects the higher-order behavior of e. If, in the standard semantics, e evaluates to a function (i.e. partial application), then f-value for e is a function that operates over the sharing domain. While the precise definition of the f-value of an expression is presented in section 3.2, in the next section we describe how the f-value is used when the expression e is applied in the program.

3.1.1 Function Applications in the Sharing Domain

If an application of the expression e to an argument x still represents a partial application, then the f-value of e, when applied to the value of x, returns an element of S whose p-value is simply the p-value for e with an additional 1 in the tuple. For example, if the value of

e is $\langle [b\ 1], \{\}, f \rangle$ then the value in S of $(e\ x)$ would be $\langle [b11], \{\}, f' \rangle$ where f' is a function capturing the higher order behavior of $(e\ x)$.

However, if e represents a partial application of a function **g** and needs only one argument to become a complete application, then when e is applied, the body of **g** gets evaluated. The environment in which the body of **g** is evaluated binds the formal parameters of **g** to values in S that have "dummy" p-values but whose f-values are the same as the corresponding actual parameters. For example, in the program

```
{ h x y == x+y;
  f a == a 1 + b 2;
  g b == b (h 1);
  result g f;
  }
```

the variable `b` is actually bound to the function `f`. When `b` is applied in the expression `(b (h 1))`, the body of `f` is evaluated in an environment in which the variable `a` is bound to

$$\langle [a11], \{\}, f' \rangle$$

where f' captures the higher order behavior of `(h 1)`.

If e represents a partial application of **g** then the f-value of e is function that takes *two* arguments when an application of the form $(e\ x)$ is encountered. The first argument is the value of x that will be used when the body of **g** is executed. The second argument is the p-value for e itself which will be appended with a 1 and returned as the p-value of the result if $(e\ x)$ still represents a partial application.

Going back to the above program, when the body of f has been evaluated, the "dummy" name for `(h 1)`, namely `a`, in the p-value and l-value of the result have to be replaced by the p-values of the actual arguments to `f`, namely `(h 1)`.

The function backsub_p takes the p-value of the result of executing the body of a function and replaces the bound variable name with the p-value of the corresponding actual parameter. The second argument to backsub_p is a list of the p-values of the original arguments corresponding to each bound variable.

$$\begin{aligned}\text{backsub\_p}(p, sub\_list) = \ & \text{let } [id\ v_0 \ldots v_n] = p \\ & \text{in if look\_up}(id, sub\_list) = \{\} \text{ then } p \\ & \quad \text{else let } p' = \text{look\_up}(id, sub\_list) \\ & \quad\quad \text{in insert}(p',\ v_0 \ldots v_n)\end{aligned}$$

where

$$\text{insert}([id\ v'_0 \ldots v'_n], v_0 \ldots v_m)) = [id\ v'_0 \ldots v'_{n-1}\ (v'_n \times v_0)\ v_1 \ldots v_m]$$

The function backsub_l takes the l-value of the result of executing the body of a function and replaces the occurrence of a bound variable (i.e. "dummy") name in a tuple by the p-value of the actual argument. The second argument to backsub_l is a list of the l-values and p-values of the original arguments corresponding to each bound variable.

$$\begin{aligned}&\text{backsub\_l}(l, sub\_list) = \\ &\quad \text{let}\{[id_0\ v_{00} \ldots v_{0n_0}], \ldots,\ [id_m\ v_{m0} \ldots v_{mn_m}]\} = l \\ &\quad \text{in } \bigcup s_i,\ 0 \le i \le m \\ &\quad\quad \text{where if look\_up}(id_i, sub\_list) = \{\} \text{ then} \\ &\quad\quad\quad s_i = \{[id_i\ v_{i1} \ldots v_{in_i}]\} \\ &\quad\quad\quad \text{else } (p'_i,\ l'_i) = \text{look\_up}(id_i, sub\_list) \\ &\quad\quad\quad\quad s_i = \{\text{insert}(p'_i,\ v_{i_1} \ldots v_{in_i})\} \cup l'_i\end{aligned}$$

If the bound variable name corresponding to an actual argument occurs in the l-value of the result, then two things have to happen during the back-substitution.

1. Any bound variable name, id, is replaced by the p-value of the corresponding argument.

2. If the bound variable name occurs in the l-value of the result, it means that the value of corresponding argument was needed in the body of the function. Therefore, all the sharing that occurred during the evaluation of the argument, namely the l-value of the argument, should be included in the l-value of the result.

After the back-substitutions have occurred, the l-value of the result may now contain several tuples that contain the same variable name. This happens when different formal parameters are bound to the same partial application. Therefore, after the back-substitutions occur the function combine is applied to the l-value of the result in order to merge all tuples with the same variable name.

$$\text{combine}(\{t_1.\ t_2, \ldots t_n\} = \text{merge}(\{t_1\}, \text{combine}(t_2, \ldots, t_n))$$

3.2 An Exact Sharing Semantics

Before proceeding to the semantic definitions, we define the syntax of our lazy, higher-order functional language:

$$\begin{array}{rcll}
c & \in & Con & \text{constants} \\
x & \in & V & \text{variables} \\
e & \in & Exp \text{ where} & \\
 & & e ::= c \mid x \mid e_1\ e_2 \mid e_1 + e_2 \mid e_1 \rightarrow e_2,\ e_3 & \\
pr & \in & Prog \text{ where} & \\
 & & pr ::= \{\ f_1\ x_{11} \ldots x_{1k_1} = e_1; & \\
 & & \quad f_2\ x_{21} \ldots x_{2k_2} = e_2; & \\
 & & \quad \ldots & \\
 & & \quad f_n\ x_{n1} \ldots x_{nk_n} = e_n; & \\
 & & \quad \texttt{result}\ e; \qquad \} &
\end{array}$$

Note that in this language we assume that all nested lambda abstractions have been "lifted" to the top level. These top-level functions are precisely the ones whose sharing properties will be determined. We also assume that common subexpression elimination, if desired, has already been performed in the standard way (i.e., by lambda abstraction).

The semantics which we are about to give specifies the exact sharing that occurs during a program's execution. The semantic domains and functions are:

$$\begin{array}{rcll}
P & = & (V \times \mathcal{N}^+) & \text{the domain of tuples} \\
L & = & \mathcal{P}(P) & \\
F & = & (S \times P) \rightarrow S & \\
S & = & (P \times L \times F)) + \{error\} & \text{the sharing domain} \\
Env & = & V \rightarrow S & \\
E & : & Exp \rightarrow Env \rightarrow S & \text{the semantic function for expressions} \\
E_p & : & Prog \rightarrow S & \text{the semantic function for programs}
\end{array}$$

where $\mathcal{N}$ is the set of natural numbers and and $\mathcal{P}(P)$ denotes the power set of P. The semantic functions E and E_p are defined below.

Since a constant is not a partial application and does not contribute to the sharing of any other partial application,

$$E[\![c]\!]env = \langle [], \{\}, err \rangle$$

where c is a constant and err is a function that returns an error if ever applied.

The meaning of a variable is whatever it is bound to in the environment in which it occurs.

$$E[\![x]\!]env = env[\![x]\!]$$

The result of a binary operation is never a partial application, although sharing of partial applications may have occurred during the evaluation of the operands.

$$\begin{aligned} E[\![e_1 + e_2]\!]env = \ & \text{let } \langle p_1, l_1, f_1 \rangle = E[\![e_1]\!]env \\ & \quad \langle p_2, l_2, f_2 \rangle = E[\![e_2]\!]env \\ & \text{in } \langle [],\ \text{merge}(l_1, l_2),\ err \rangle \end{aligned}$$

In a well-typed program, no partial application can serve as an operand in a binary operation. Therefore, p_1 and p_2 above will both be [].

The conditional is handled as follows:

$$\begin{aligned} E[\![e_1 \rightarrow e_2, e_3]\!]env = \ & \text{let } \langle p_1, l_1, f_1 \rangle = E[\![e_1]\!]env \\ & \text{in if } (\text{Oracle}[\![e_1]\!] = True) \text{ then} \\ & \quad \text{let } \langle p_2, l_2, f_2 \rangle = E[\![e_2]\!]env \\ & \quad \text{in } \langle p_2,\ \text{merge}(l_1, l_2),\ f_2 \rangle \\ & \text{else let } \langle p_3, l_3, f_3 \rangle = E[\![e_3]\!]env \\ & \quad \text{in } \langle p_3, \text{merge}(l_1, l_3),\ f_3 \rangle \end{aligned}$$

In order to provide an exact semantics, conditionals must be resolved correctly (i.e. corresponding to the way the conditional would be resolved in the standard semantics during program execution). To do so, we defer to an oracle to determine the correct meaning of each conditional. In the next section we provide an abstract sharing semantics that does not rely upon such an oracle, but provides less precise sharing information.

Function application is defined as follows:

$$\begin{aligned} E[\![e_1\ e_2]\!]env = \ & \text{let } \langle p_1, l_1, f_1 \rangle = E[\![e_1]\!]env \\ & \quad \langle p_3, l_3, f_3 \rangle = f_1(E[\![e_2]\!]env, p_1) \\ & \text{in } \langle p_3, \text{merge}(l_3, l_1), f_3 \rangle \end{aligned}$$

Since f_1 is the function that captures the higher order behavior of e_1, f_1 is applied to the value of e_2. The sharing information gained from evaluating e_1 is then merged with the sharing information gained from performing the application. Notice the extra argument p_1 to f_1 which indicates to f_1 which partial application it represents.

The meaning of a program is the value of the result expression in an environment in which all function names are bound to values in S.

$$E_p[\![\ \{F_1\ x_{11} \ldots x_{1k_1} = e_1;$$
$$\ldots$$
$$F_n\ x_{n1} \ldots x_{nk_n} = e_n;$$
$$\texttt{result}\ e; \qquad\qquad \}\!] = E[\![e]\!]env'$$

whererec

$$env' = [s_1/F_1, \ldots, s_n/F_n]$$
$$s_i = \langle [F_i\ 1], \{\},$$
$$\lambda\langle px_1, lx_1, fx_1\rangle\ p_1.$$
$$\langle \mathrm{app}(p_1, 1),\ \{\},$$
$$\lambda\langle px_2, lx_2, fx_2\rangle p_2.$$
$$\langle \mathrm{app}(p_2, 1), \{\}, \lambda \ldots$$
$$\vdots$$
$$\lambda.\langle px_{k_i}, lx_{k_i}, fx_{k_i}\rangle\ p_{k_i}.$$
$$\text{let}\quad px'_j = \text{ if } (px_j = [])\text{then } []$$
$$\text{else}[x_{ij}\ 1] \quad \text{for} 1 \le j \le k_i$$
$$\langle p', l', f'\rangle =$$
$$E[\![e_i]\!]env'[\ \langle px'_1, \{\}, fx_1\rangle/x_{i1}, \ldots,$$
$$\ldots, \langle px'_{k_i}, \{\}, fx_{k_i}\rangle/x_{ik_i}]$$
$$p'' = \mathrm{backsub\_p}(p', [px_1/x_{i1}, \ldots, px_{k_i}/x_{ik_i}])$$
$$l'' = \{p_{k_i}\} \cup \mathrm{backsub\_l}(l', [\ (px_1, lx_1)/x_1,$$
$$\ldots (px_{k_i}, lx_{k_i})/x_{k_i}]))$$
$$\text{in } \langle p'', \mathrm{combine}(l''), f'\rangle \ldots\rangle$$

The utility functions backsub_p, backsub_l, and combine were defined in section 3.1.1. The function app$(p, 1)$ simply takes a p-value and appends a 1 to the end of the tuple.

3.3 Abstract Interpretation of the Sharing Semantics

Since we are unable to resolve conditionals at compile time, we define an abstract sharing semantics for our functional language such that the meaning of a program is information

about the maximum sharing that *could possibly* occur when the program is executed.

This is accomplished by defining an abstract sharing domain S' whose elements are sets representing alternate possibilities for the sharing occurring in an expression. The abstract semantic domains and functions are:

$$
\begin{array}{rcll}
P & = & (V \times \mathcal{N}^+) & \text{the domain of tuples} \\
L & = & \mathcal{P}(P) & \\
F' & = & (S' \times P) \rightarrow S' & \\
S' & = & \mathcal{P}(P \times L \times F') + \{error\} & \text{the abstract sharing domain} \\
Env' & = & V^* \rightarrow S' & \\
E' & : & Exp \rightarrow Env \rightarrow S' & \text{the abstract semantic function for expressions} \\
E'_p & : & Prog \rightarrow S' & \text{the abstract semantic function for programs}
\end{array}
$$

and E' and E'_p are defined below.

$$E'[\![c]\!]env = \{\langle [], \{\}, err\rangle\}$$

$$E'[\![x]\!]env = env[\![x]\!]$$

$$
\begin{array}{rll}
E'[\![e_1 + e_2]\!]env = & \text{let} & \{\langle p_0, l_0, f_0\rangle, \ldots, \langle p_n, l_n, f_n\rangle\} = E'[\![e_1]\!]env \\
 & & \{\langle p'_0, l'_0, f'_0\rangle, \ldots, \langle p'_m, l'_m, f'_m\rangle\} = E'[\![e_2]\!]env \\
 & & l''_{ij} = \text{merge}(l_i, l'_j) \text{ for all } i \leq n,\ j \leq m \\
 & \text{in} & \{\langle [], l''_{00}, err\rangle, \ldots, \langle [], l''_{nm}, err\rangle\}
\end{array}
$$

$$
\begin{array}{rll}
E'[\![e_1 \rightarrow e_2, e_3]\!]env = & \text{let} & \{\langle p_0, l_0, f_0\rangle \ldots \langle p_n, l_n, f_n\rangle\} = E'[\![e_1]\!]env \\
 & & \{\langle p'_0, l'_0, f'_0\rangle \ldots \langle p'_m, l'_m, f'_m\rangle\} = E'[\![e_2]\!]env \\
 & & \{\langle p''_0, l''_0, f''_0\rangle \ldots \langle p''_q, l''_q, f''_q\rangle\} = E'[\![e_3]\!]env \\
 & & z_{ij} = \langle p'_j, \text{merge}(l_i, l'_j), f'_j\rangle \text{ for all } i \leq n,\ j \leq m \\
 & & z'_{ij} = \langle p''_j, \text{merge}(l_i, l''_j), f''_j\rangle \text{ for all } i \leq n,\ j \leq q \\
 & \text{in} & \{z_{ij} | \text{ for all } i \leq n,\ j \leq q\ \} \cup \{z'_{ij} | \text{ for all } i \leq n,\ j \leq q\ \}
\end{array}
$$

$$
\begin{array}{rll}
E'[\![e_1\ e_2]\!]env = & \text{let} & \{\langle p_0, l_0, f_0\rangle \ldots \langle p_n, l_n, f_n\rangle\} = E'[\![e_1]\!]env \\
 & & \{\langle p'_0, l'_0, f'_0\rangle \ldots \langle p'_m, l'_m, f'_m\rangle\} = E'[\![e_2]\!]env \\
 & & \{\langle p''_{ij0}, l''_{ij0}, f''_{ij0}\rangle, \ldots, \langle p''_{ijq}, l''_{ijq}, f''_{ijq}\rangle\} = \\
 & & \quad f_i(\{\langle p'_j, l'_j, f'_{ij}\rangle\}, p_i), \text{ for each } i \leq n,\ j \leq m \\
 & \text{in} & \bigcup\{\langle p''_{ijk}, \text{merge}(l''_{ijk}, l_i), f''_{ijk}\rangle\} \text{ for all } i,j,k
\end{array}
$$

$$E'_p[\![\{\ F_1\ x_{11}\ldots x_{1k_1} = e_1;$$
$$\ldots$$
$$F_n\ x_{n1}\ldots x_{nk_n} = e_n;$$
$$\textbf{result}\ e; \qquad \}\!] = E'[\![e]\!]env';$$

whererec

$$env' = [s_1/F_1,\ldots,s_n/F_n]$$
$$s_i = \{\langle [F_i\ 1],\{\},$$
$$\lambda\{\langle px_1, lx_1, fx_1\rangle\}\ p_1.$$
$$\{\langle \text{app}(p_1,1),\ \{\},$$
$$\lambda\{\langle px_2, lx_2, fx_2\rangle\}\ p_2.$$
$$\{\langle \text{app}(p_2,1),\{\},\lambda\ldots$$
$$\vdots$$
$$\lambda\{\langle px_{k_i}, lx_{k_i}, fx_{k_i}\rangle\}\ p_{k_i}.$$
$$\text{let}\quad px'_j = \text{if } (px_j = [])\text{then } []$$
$$\text{else}[x_{ij}\ 1],\quad \text{for } 1 \le j \le k_i$$
$$\{\langle p'_0, l'_0, f'_0\rangle,\ldots,\langle p'_m, l'_m, f'_m\rangle\} =$$
$$E'[\![e_i]\!]env'[\{\ \langle px'_1,\{\},fx_1\rangle\}/x_{i1},\ldots,$$
$$\ldots,\{\langle px'_{k_i},\{\},fx_{k_i}\rangle\}/x_{ik_i}]$$
$$p''_j = \text{backsub\_p}(p'_j,[px_1/x_{i1},\ldots,px_{j_i}/x_{ik_i}]),$$
$$\text{for all } j \le m$$
$$l''_j = \{p_{k_i}\} \cup \text{backsub\_l}(l'_j,[\ (px_1,lx_1)/x_{i1},\ldots$$
$$\ldots(px_{k_i},lx_{k_i})/x_{ik_i}])),$$
$$\text{for all } j \le m$$
$$\text{in } \{\ \langle p''_0, \text{combine}(l''_0), f'_0\rangle,\ldots\langle p''_m, \text{combine}(l''_m), f'_m\rangle\}$$
$$\}\}\ldots\}$$

3.4 Termination

In order to guarantee that the interpretation of a program using the above abstract sharing semantics will terminate we have to show that every function in the subdomain

$$F' = (S' \times P) \to S'$$

reaches a fixpoint in a finite number of iterations. We can accomplish if we do all of the following:

1. Ensure that the subdomain P of p-values is finite. This is clearly the case since a p-value is a tuple consisting of a variable name (from the finite set V) and a collection of 1s. Since a p-value tuple represents a partial application, the size of the tuple (i.e. the number of 1's) is limited by the largest number of formal parameters that can occur in a function definition (clearly a finite number).

   In section 3.1 it was mentioned that it is undesirable to create a new identifier for each partial application created during execution of the program. If new identifiers were created in such a fashion it would be very difficult (if not impossible) to insure that the elements of a sharing domain (including all those new identifiers) were finite in number.

2. Ensure that the the subdomain L of l-values is finite. Each l-value is a set of tuples that describe the maximum number of occurrences of partial applications of variables. We can make the set of possible tuples finite by setting a limit on the maximum number of occurrences of a partial application that the analysis can detect. In most cases, we simply want to know if a partial application of some function occurred more than once. If the number of occurrences reaches the maximum value then any further occurrences will not be counted.

3. Ensure that the domain $F' = (S' \times P) \rightarrow S'$ contains a finite number of functions. Since the number of functions of a given arity (i.e. the number of arguments that a function can be applied to before the result is no longer a function) over a finite domain is finite, we can ensure that F' is finite by requiring that the arity of each function in F' is finite. This is a reasonable restriction and is often enforced by a type inferencing system.

In addition, we can define an ordering of the elements of S' and prove monotonicity properties about the functions in F' although such a discussion is beyond the scope of this paper.

4 An Application: Efficient Full Laziness

As discussed in the introduction, graph reduction is one way of implementing the lambda-calculus. Another approach, which also utilizes sharing of expressions, is *combinator reduction* [12], in turn based on the reduction rules of combinatory calculus [5,10]. Both of these approaches handle higher-order functions and lazy evaluation fairly naturally. However, graph reduction has the disadvantage of having to explicitly implement lambda calculus's implicit notion of "substitution," which is typically manifested as an *environment* for bound variables. In combinator reduction, the environment is eliminated in favor of a more fine-grained computation, in which operands and operators are paired up through the behavior of a fixed set of primitive combinators. This simplifies the approach considerably, but at the expense of potentially more reductions and a greater consumption of space.

As an improvement to combinator reduction, Hughes observed that instead of relying on a *fixed* set of combinators, one could derive a *different* set of combinators for each program [7]. He called these derived combinators *super-combinators.* Not only do super-combinators preserve the property of not needing an environment structure for evaluation (having no free variables or embedded lambda abstractions), but they also preserve the property of being *fully lazy.* Loosely speaking, we say that a function is fully lazy if shared uses of any of its partial applications do not result in the evaluation of the same subexpression more than once. (Examples of this will be given shortly.)

Unfortunately, the algorithm for generating super-combinators turns out to be excessively conservative in preserving the property of full laziness. As a result, the super-combinators are very often much more fine-grained than they need to be, resulting (as with a fixed set of combinators) in more reductions and greater consumption of space.

In this section we discuss a refinement of super-combinators that overcomes this conservatism, resulting in larger and more efficient combinators called *refined super-combinators.* We present an effective algorithm for translating a set of lambda expression definitions into refined super-combinators. The algorithm utilizes the sharing analysis of the previous section.

4.1 Super-combinators

Consider the function f defined by:

$$f = \lambda a.\lambda b.\lambda c. * (+ a^2\, b)\, c$$

from which we define the combinator α:

$$\alpha\, a\, b\, c = * (+ a^2\, b)\, c.$$

Now suppose the following expression is evaluated:

$$(\lambda g. (* (g\, 5)\, (g\, 6)))\ (f\, 3\, 4)$$

Since g occurs twice, $(f\, 3\, 4)$ is shared. Yet because of this particular choice of combinator, both $(\alpha\, 3\, 4\, 5)$ and $(\alpha\, 3\, 4\, 6)$ will be evaluated independently. As a result, $(+\, 3^2\, 4)$ will be computed twice. Hence the combinator α does not have the property of being fully lazy, and results in more computation than necessary.[1]

To improve this situation, one might generate *super-combinators* from f, in which full laziness is (conservatively) guaranteed by generating one combinator for every bound variable. To see how this works, we first define a *free expression* with respect to a particular bound variable v as an expression in which there are no free occurrences of v. A *maximally free expression* (mfe) with respect to v is a free expression which is not contained within any larger free expression (with respect to v). When the context is clear, we omit naming the bound variable with respect to which an expression is maximally free.

The algorithm for generating super-combinators begins with the innermost lambda expression and works out, abstracting at each level all mfe's with respect to the bound variable at that level. For example, for the definition of f above, we see that the mfe of the innermost lambda expression is $(+\, a^2\, b)$. This expression is abstracted to form the super-combinator:

$$\alpha\, x\, c = * x\, c$$

[1] This combinator definition is essentially what would result from *lambda lifting* [8].

and thus $f = \lambda a.\lambda b.\, \alpha\,(+\,a^2\,b)$. Next we note that a^2 is the mfe of the new innermost lambda expression, so it is abstracted, forming the combinator:

$$\beta\, y\, b = \alpha\,(y + b)$$

and thus $f = \lambda a.\, \beta\,(a^2)$. Since there is no (non-trivial) expression in f that is free with respect to a, the next (and final) super-combinator is:

$$\gamma\, a = \beta\,(a^2)$$

and all occurrences of f in the program are replaced by γ.

Note that the shared expression mentioned earlier, $(f\,3\,4)$, will reduce as follows:

$$\begin{array}{lll} (f\,3\,4) & \Rightarrow & (\gamma\,3\,4) \\ & \Rightarrow & (\beta\,9\,4) \\ & \Rightarrow & (\alpha\,13) \end{array}$$

and therefore $(+\,3^2\,4)$ is only computed once — thus achieving fully lazy evaluation.

Even if a function definition contains no explicit nesting of lambda expressions it can still be transformed into a set of super-combinators. This is possible because a definition of the form,

$$f \times_1\ x_2 \ldots\ x_n = e$$

can be transformed into

$$f = \lambda x_1.\ \lambda x_2.\ \ldots x_n.e$$

and the super-combinator algorithm can then be applied.

A formal algorithm for generating super-combinators from a program P is:

1. Find the leftmost, innermost lambda expression L, of the form $\lambda v.exp$.

2. Find the maximally free expressions, $e_1 \ldots e_n$, of exp with respect to v.

3. Create a new combinator (say α) defined by:

$$\alpha\, i_1 \ldots i_n\, v = exp[i_1/e_1, \ldots, i_2/e_2]$$

where formal parameters $i_1 \ldots i_n$ do not occur free in *exp*.

4. Substitute $(\alpha\, e_1 \ldots e_n)$ for L in P.

5. Repeat steps 1-4 until step 1 fails.

There are a few obvious optimizations to this algorithm, such as eliminating redundent combinators, as in: $\alpha\, a\, b = \beta\, a\, b$.

4.2 Refined super-combinators

Although preserving full laziness is a worthy goal, the super-combinator approach is too conservative. To see this, note in the previous example that the original single-combinator definition for f would be perfectly satisfactory if no partial application of f were ever shared, for then there would be no partial result that might be computed more than once. And because one combinator is used instead of three, the single-combinator solution would be more efficient with respect to both time and space, as argued earlier. If one could infer from a given program whether or not a partial application of f was shared, then one could choose either a one-, two-, or three-combinator implementation for it, whichever is appropriate.

Not surprisingly, this is the key improvement that refined super-combinators make over ordinary super-combinators. The optimization turns out to be applicable for a great majority of all user-defined functions, since, despite the elegance of higher-order functions created through partial application, they are in reality used only a small percentage of the time, and an even smaller percentage are shared. Thus a large improvement in performance can generally be expected on most programs.

Assuming that the sharing information is given, we write "$g\,(w, x)\,(y, z) = \ldots$" to indicate, for example, that g applied to two arguments is shared, but not to one or three arguments. We then generalize, in the obvious way, the notion of a maximally free expression with respect to a single variable, to a maximally free expression with respect to

a set of variables. Let $MFE(exp, S)$ be the set of mfe's of exp with respect to the set of identifiers S.

When generating refined super-combinators for a lambda expression such as

$$g\,(w, x)\,(y, z) = \ldots$$

we treat each "group" of formal parameters as a single unit by abstracting mfe's with respect to each group, working innermost out as before. Given the previous discussion, the rationale for doing this should be obvious – we cannot make the groups any larger, for that might violate full laziness, nor is there any reason to make them any smaller, since no finer partial application is shared.

As an optimization we can treat each application of a function separately, by generating a different set of combinators for each application, with each set being tailored to the sharing properties at that point. Although this creates a potential for code explosion, it is probably a reasonable thing to do, for two reasons. First, the sharing properties typically do not vary much, and thus code explosion is not a problem. Second, in some sense it is unreasonable to penalize a programmer's use of a function in one place because of a use of the same function somewhere else.

These ideas form the basis for the following algorithm for generating refined super-combinators:

1. Let $f_1 \ldots f_n$ be the names of the defined functions in the program.

2. Let $f_i^1 \ldots f_i^k$ be the k occurrences of the function variable f_i in the program. For each f_i, replicate the function definition:

   $$f_i\, x_{i1} \ldots x_{im} = exp_i$$

   k times, yielding:

   $$\begin{aligned} f_{i1}\, x_{i1} \ldots x_{im} &= exp_{i1} \\ &\vdots \\ f_{ik}\, x_{i1} \ldots x_{im} &= exp_{ik} \end{aligned}$$

where initially $exp_{i1} = \ldots = exp_{ik}$.

3. Partition the formal parameters of each definition f_{ij} so as to reflect the sharing of partial applications at occurrence f_i^j.

4. For each definition f_{ij}, repeat this step until there is just one partition of bound variables remaining. Let $(x_{il} \ldots x_{in})$ be the right-most partition; i.e.,

   $$f_{ij}\ (x_{i1} \ldots) \ldots (\ldots x_{i(k-1)})\ (x_{ik} \ldots x_{in}) = e_{ij}$$

   Define a new combinator (say α) by:

   $$\alpha\ v_1 \ldots v_p\ x_{ik} \ldots x_{in} = exp_{ij}[v_1/e_1, \ldots, v_p/e_p]$$

   where $v_1 \ldots v_p$ are new variable names not occurring free in exp_{ij}, and:

   $$\{e_1, \ldots, e_p\} = MFE(exp_{ij}, \{x_{ik} \ldots x_{in}\})$$

   Then replace the previous definition of f_{ij} by:

   $$f_{ij}\ (x_{i1} \ldots) \ldots (\ldots x_{i(k-1)}) = \alpha\ exp_1 \ldots exp_p$$

5. Replace the occurrence f_i^j by the variable f_{ij}.

5 Conclusion

In this paper we have defined a non-standard denotational semantics for a functional language that provides information about a very "operational" property of programs, namely which objects get shared during execution. The abstraction of these semantics provides a valuable compiler tool increasing the efficiency of functional language implementations.

6 Acknowledgements

I would like to thank Paul Hudak for his careful reading of earlier drafts and helpful suggestions. Much of what I have learned about semantics and abstract interpretation has been a result of discussions with Jonathan Young and Adrienne Bloss as well as Paul Hudak. I would also like to thank Wendy Goldberg for helping out when time was short.

References

[1] Adrienne Bloss and Paul Hudak. Variations on strictness analysis. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pages 132–142, ACM SIGPLAN/SIGACT/SIGART, Cambridge, Massachusetts, August 1986.

[2] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In *LNCS 217: Programs as Data Objects*, pages 42–62, Springer-Verlag, 1985.

[3] C. Clack and S.L. Peyton Jones. Strictness analysis – a practical approach. In *Functional Programming Languages and Computer Architecture*, pages 35–49, Springer-Verlag LNCS 201, September 1985.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Sym. on Prin. of Prog. Lang.*, pages 238–252, ACM, 1977.

[5] H.K. Curry and R. Feys. *Combinatory Logic.* Noth-Holland Pub. Co., Amsterdam, 1958.

[6] P. Hudak and J. Young. Higher-order strictness analysis for untyped lambda calculus. In *12th ACM Sym. on Prin. of Prog. Lang.*, pages 97–109, January 1986.

[7] R.J.M. Hughes. Super-combinators: a new implementation method for applicative languages. In *Proc. 1982 ACM Conf. on LISP and Functional Prog.*, pages 1–10, ACM, August 1982.

[8] T. Johnsson. *The G-machine: an abstract machine for graph reduction.* Technical Report, PMG, Dept. of Computer Science, Chalmers Univ. of Tech., February 1985.

[9] A. Mycroft. *Abstract Interpretation and Optimizing Transformations for Applicative Programs.* PhD thesis, Univ. of Edinburgh, 1981.

[10] M. Schonfinkel. Uber die bausteine der mathematischen logik. *Mathematische Annalen*, 92:305, 1924.

[11] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Mass., 1977.

[12] D.A. Turner. A new implementation technique for applicative languages. *Software — Practice and Experience*, 9:31–49, 1979.