The Derivation of Functional Equivalents of Imperative Programs

Graham H.B. Roberts
School of Informatics and Engineering
The Flinders University of South Australia
Email: graham@infoeng.flinders.edu.au

Abstract

Denotational semantics is presented as a valuable theoretical tool, having many applications including language design, compiler generation and program analysis. In particular, a method is described for deriving a concise and useful functional representation of a program using a denotational definition of the source language's semantics.

Our aim is to translate a given program into a compact functional representation to facilitate its evaluation on functional hardware. The λ-expressions are first translated into Turner's combinator code (see [7]). We choose to use a fixed set of combinators as the resulting code is more amenable to analysis and there are many inherent advantages such as lazy evaluation and once only evaluation of reducible sub-expressions.

Semantic algebras relating to static semantics and the store algebra are "unfrozen" so they can be partially evaluated. The reduction machine that performs the evaluation includes simplification rules that allows a more compact functional representation (denotation) to be reached. If desired, some or all of the programs arguments can be supplied to produce a new denotation (result) using the same reduction machine.

1. Introduction

The problem we address in this paper is that of finding the "simplest" functional representation of a given program. We require the simplification method to be as close to language independent as possible so that it will be generally applicable. Our method is similar to program transformation except that rather than source code level transformations we transform functional representations of programs that we obtain using denotational semantics.

The denotational method of specifying the semantics of a programming language entails defining a mapping from a program, expressed as an abstract syntax tree, to its meaning or denotation. The denotation is a mathematical value (an element of a semantic domain). A semantic definition is expressed as *valuation* functions that map syntactic constructs to values. Unlike operational definitions, no computational steps are involved in the definition.

In order to perform meaning-preserving transformations on programs written in various languages, a unifying model of programming language semantics is required to act as a representation domain for transformations. Such a model is provided by λ-calculus-based denotational semantics ([6], [5]) which, although functional in nature, is convenient for expressing the semantics of most of the commonly used programming languages. The model has a well developed theory ([6]) which ensures that program constructs can be given sensible denotations. However, λ-expressions are still relatively syntactically too rich to be amenable to significant transformation without the help of complex analytical techniques (for example, abstract interpretation [1]) and so a simpler function representation is preferable, such as Turner's combinators [7].

Turner suggested the following fixed set of combinators as a basis for the representation and evaluation of functional programs: **S**, **K**, **I**, **B** and **C**. Combinators act as rewriting rules that can be interpreted in λ-expressions using the following correspondences:

$$\begin{aligned} \mathbf{S} &= \lambda f g x.(f\ x)\ (g\ x) \\ \mathbf{K} &= \lambda x y.x \\ \mathbf{I} &= \lambda x.x \\ \mathbf{B} &= \lambda f g x.f\ (g\ x) \\ \mathbf{C} &= \lambda f g x.(f\ x)\ g \end{aligned}$$

Each combinator can be described in terms of a transformation of its arguments. For example, $(S\ e_1\ e_2\ e_3)$ is transformed to $(e_1\ e_3)(e_2\ e_3)$. It is straightforward to transform a given λ-expression into a corresponding combinator expression (see [4]). The combinator representation of λ-expressions leads to many benefits which will be described later. A detailed account of of λ-calculus and combinatory logic can be found in [3].

1530-0900/01 $10.00 © 2001 IEEE

The translation of an imperative program, into an equivalent λ-expression, is achieved by "applying" semantic valuation functions to an abstract syntax tree. The process starts with the "program" valuation function being applied to the abstract syntax tree, yielding a λ-expression containing further valuation function applications. The process continues until all valuation function applications have been replaced. The resulting λ-expression is then translated to combinator form before being simplified.

2. Functional Representation and Manipulation

The specification of the denotational semantics of a language can be viewed as a functional program, written in λ-calculus, which computes denotations. If the specification has the valuation functions V_0, V_1, ...,V_n, then the functional program which when applied to a program will compute its denotation would be:

letrec $V_0 = E_0$ $V_1 = E_1 \ldots V_n = E_n$
in V_0

where the "letrec" construction is the usual mechanism found in functional languages that allows the definition of mutually recursive functions. We assume that any domains and semantic algebras used in the definition of the valuation functions are primitives of the functional programming language or have been defined in it. Note that the result of the program, V_0, is an un-applied function while usually the expression which is the result of a functional program is a fully applied function that results in a primitive, printable value. We will denote the result of the program, that is the function that represents the denotation of a program, as D.

For a program P, $D(P)$ is the denotation of P and also a functional representation of P. If P takes m arguments then the functionality of D is:

$$D : P \times I_1 \times I_2 \times \ldots \times I_m \rightarrow O$$

where P is the syntax domain for programs of the language being defined, and $I_1, I_2, \ldots, I_m$ are the domains of the respective arguments, and O is the output domain. In order to accommodate partial application, the application of a function to only some of its arguments, we Curry D and so its functionality becomes:

$$D : P \rightarrow I_1 \rightarrow I_2 \rightarrow \ldots \rightarrow I_m \rightarrow O$$

A partial application of D to just the program P, $D(P)$ which we will denote as C, has the functionality:

$$C : I_1 \rightarrow I_2 \rightarrow \ldots \rightarrow I_m \rightarrow O$$

C is a functional representation of P which is amenable to manipulation for a number of purposes. For example, we can simplify it with meaning preserving transformation rules or translate it into machine code. The same is true of each of

$$C(i_1), C(i_1, i_2), \ldots, C(i_1, i_2, \ldots, i_m)$$

where $i_1, i_2, \ldots i_m$ are given input values. Note that there is no reason why we cannot simplify any of the partial applications before they are applied to further arguments. For example, if $C_{12} = C(i_1, i_2)$ then $C_{12}(i_3)$ is again a candidate for simplification. The system we desire embraces these ideas.

3. The D-Machine

One way in which our approach differs from others is that we translate denotations into combinator code that turns out to have advantages over other representations (for example, [2] use data-flow code), and provides a common representation for simplification and final execution. The basic structure of our system, called the D(enotational)-Machine, is described below.

Denotations and simplification rules are expressed in Denotational λ-Calculus (DLC). A translator converts them to Combinator Code (CC) so that the functions they define are available for use by the CC Reduction machine. The "program" valuation function can then be applied to a program, with parameters, to produce a CC denotation. Further simplification can take place by applying this denotation to more parameters and feeding it back to the CC reduction machine. The CC to DLC translator and the DLC reduction machine are *not* necessary parts of the D-Machine; they do however provide a method of obtaining more readable output.

Although we can in theory represent a denotation in untyped λ-calculus, we include a number of primitive domains and semantics algebras that are common to many denotational specifications. For example, numbers, booleans and tuples and their associated operations. We also include a number of primitive functions which are convenient for expressing denotations. For example, the function `select` replaces the implicit pattern matching associated with the definition of a valuation function. For example,

$\mathbf{C}[\![C_1 \ \% \ C_2]\!] = \lambda s.\ \mathbf{C}[\![C_2]\!]\ (\mathbf{C}[\![C_1]\!]\ s)$
$\mathbf{C}[\![I{:=}E]\!] = \ldots$

is written in DLC as

```
(lambda(syntax-object)
  (select syntax-object
```

```
(*s* (?c1 % ?c2))
  (lambda(s) (c (*s* ?c2)
                (c (*s* ?c1) s)))
(*s* (?i := ?e)) ...
```

Lack of space prevents us from describing DLC in detail. Briefly, DLC has a Lisp-like syntax. Primitive objects include syntax objects, that have the form `(*s* object)`, primitive Lisp objects, that have the form `(*p* object)`, pattern variables that are prefixed with "?" and tuples that have the form

`(*t* (`e_1 e_2 ... e_n`))`

where $n \geq 0$. Disjoint unions are not directly supported, however their elements are directly representable as 2-tuples with the associated construction function (tuple2) being the injection operator, `snd` the projection operator and `fst` the inspection operator (selects the tag). The D-machine has been implemented in both Scheme and Common Lisp.

3.1. Reduction

The CC reduction Machine is essentially a standard SK-reduction engine with the added abilities of being able to reduce partial applications and to mix rule-based evaluation with reduction. With a normal order λ-calculus reduction engine, partial application presents no problem, however, with combinators, all the arguments are required for a combinator to be applied. Our approach is to reduce the (incomplete) arguments to a combinator if there are no other reductions possible.

The form of the resulting code depends on which semantic algebras are "frozen" ([5]). A semantic algebra is "unfrozen" if we allow each occurrence of an operator to be replaced with its definition and "frozen" otherwise. For example, if we required code for a non-functional architecture we would simply remove the definitions of store operations, causing them to be treated as primitive symbols and to appear in the resulting code. In order to obtain a simple functional representation we simply leave all algebras "unfrozen".

To prevent the non-termination of recursively defined functions, all recursion must be expressed with a least fixed point operator, `fix`. For partial applications of a program, `fix` is left undefined. If the denotation of a fully applied program is required then a definition of `fix` must be included.

Chao and Bryant [2] claim their system achieves a "degree of optimization" (simplification) not before achieved by "denotational semantics-based techniques". Our system achieves a similar level of simplification and has advantages due to the combinator representation such as the automatic extraction of loop invariants, once only evaluation of constant expressions and lazy evaluation. Note that CC correctly implements denotational definitions since the evaluation of an expression gives the same result as normal order reduction would (strict functions need to be treated differently). In addition, a denotation that results from a partial application can be further evaluated with additional arguments on the same reduction engine.

3.2. Simplification

One of our aims is to minimise the number of simplification rules required to simplify denotations (partially or fully applied) and so functions such as `select` and `case` are implemented as macros. Thus the number of distinct functions for which simplification rules have to be specified is reduced. We currently employ only a small, incomplete set of rules that still provide significant improvements in the level of simplification which can be achieved. These are:

1.	(sim (+ 0 ?x)	?x)
2.	(sim (+ ?x 0)	?x)
3.	(sim (+ (+ ?n ?n1) ?n2)	(+ ?n (+ ?n1 ?n2)))
4.	(sim (- ?x 0)	?x)
5.	(sim (extend-tuple (*t* ()) x)	(*t* (?x)))
6.	(sim (if ?b ?x ?x)	?x)
7.	(sim (if (not ?b) ?x ?y)	(if ?b ?y ?x))
8.	(sim ((if ?b ?n ?m) ?z)	(if ?b (?n ?z) (?m ?z)))

The respective arguments to `sim` specify the left and right-hand sides of a simplification rule respectively. Whenever an expression which matches the left-hand side of a rule is encountered, it is replaced with the corresponding instantiated right-hand side. The rules are applied during reductions of the CC Reduction Machine to ensure the final denotation is as simplified as possible.

For example, the the simplification rule 5 indicates that a call to `extend-tuple`, where the first argument is an empty tuple and the second argument is an arbitrary expression, can be replaced with (simplified to) a tuple of length 1 whose contents is the second argument.

The simplification of partially applied combinator expressions can be problematic. For example, consider the DLC expression:

```
(lambda(x y)
   ((if (= x 1) (lambda(x) (+ x y))
        (lambda(x) x))
        y))
```

We can simplify the expression using the last rule to:

```
(lambda(x y)
   (if (= x 1)
       ((lambda(x) (+ x y)) y)
       ((lambda(x) (x)) y)))
```

that can then be reduced further. However, if we look at corresponding combinator code it is not at all obvious that the rule is applicable:

```
((C ((B S)
     ((C ((B C)
          ((C ((B B)
               ((B if)
                ((C =) 1))))
           (C +))))
  I)))
 I)
```

The solution is to apply the expression to dummy arguments, apply the simplification rule, reduce the result and then abstract the dummy arguments. Thus,

```
((if (= dummy-1 1)
     (C + dummy-2)
     I)
 dummy-2)
```

simplifies to

```
(if (= dummy-1 1)
    (C + dummy-2 dummy-2)
    (I dummy-2))
```

and after reduction and abstraction becomes:

```
((C ((B S)
     ((C ((B B)
          ((B if)
           ((C =) 1))))
  ((S +) I))))
 I)
```

3.3. Examples

The abstract syntax, and excerpts from a DLC version of the denotational semantics, of a simple procedural language is given below (due to lack of space the full definition is not given, however the language defined is very similar to that used in [2] with the exception that the processing of errors which result from projection and selection operations on disjoint unions have been made explicit). Another minor difference is the use of "%" as the statement separator, rather than a semicolon.

Abstract Syntax:

P ϵ Program
K ϵ Block
D ϵ Declaration
C ϵ Command
E ϵ Expression
I ϵ Id
N ϵ Numeral

P ::= K
K ::= begin D % C end
D ::= D_1 % D_2 | const I=N | var I
| proc $I_1(I_2)$ = C
C ::= C_1 % C_2
| I:=E
| **if** B **then** C_1 **else** C_2
| **while** B **do** C
| read I
| write E
| K
| **call** I E
E ::= E_1+E_2 | I | N
B ::= E_1 = E_2 | E_1 >= E_2 | **not** B

DLC Semantics:

```
; disjoint union operators
 (def in tuple2) (def tag-of fst)
 (def value-of snd)
; Configurations
(def config tuple3) (def cstore fst)
(def cinfile snd) (def coutfile trd)
; Files
(def emptyfile empty-tuple)
(def append-to-file extend-tuple)
; Storage locations
; location = Nat
(def fst-locn 0)
(def next-locn  (lambda (l) (+ 1 l)))
(def equal-locn
  (lambda (l1 l2) (equal? l1 l2)))
; Denotable-value = Errvalue + Location
;                   + Nat + Procedure
; Expressible-value = Nat + Errvalue
; Environments : Location -> Denotable-value
 ... code omitted

; Valuation functions
; p : Program -> File -> Post-File
(def p (lambda (ob)
  (select ob (*s* ?k)
     (lambda (f)
        (nlet (result
                (k (*s* ?k)
                   (emptyenv 0)
                   (config newstore
                     f emptyfile)))
          (in (tag-of result)
              (coutfile (value-of result))))))))
; k : Block -> Environment
      -> Configuration -> PostConfiguration
(def k (lambda (ob)
  (select ob
    (*s* (begin ?d % ?c end))
      (lambda (env)
        (c (*s* ?c) (d (*s* ?d) env))))))
    (*s* (call ?i ?v))
    (lambda(env con)
      (nlet (proc (accessenv (*s* ?i) env))
        (case (tag-of proc)
          ((*p* d-procedure)
```

```
          (let (arg (e (*s* ?v) env
                       (cstore con)))
            (case (tag-of arg)
              ((*p* e-nat) ((value-of proc) con
                            (value-of arg)))
              (else (signalerr con)))))
         (else (signalerr con)))))

...

(*s* (while ?b do ?c))
 (lambda (env)
   (fix (lambda (f con)
        (nlet (b-val (b (*s* ?b) env
                        (cstore con)))
          (case (tag-of b-val)
            ((*p* b-tr) ((if (value-of b-val)
                             (lambda (n)
                               ((check f)
                                (c (*s* ?c) env n)))
                             return)
                         con))
            ((*p* b-errvalue) (signalerr s)))))))
            ...

;e : Expression -> Environment -> Store
;    -> Expressible-value
(def e (lambda (ob)
  (select ob
    (*s* (?e1 + ?e2))
     ...

; b : Boolean-exp -> Environment
;     -> Store -> (Tr + Errvalue)
(def b (lambda (ob)
  (select ob
    (*s* (?e1 = ?e2))
      (lambda (env s)
        ...
```

A program maps an input file (an n-tuple of values) to an output file that is tagged with a status to indicate if the program was erroneous, either due to static semantic errors or to potential run-time errors. For the following examples, the CC reduction machine has had both the DLC code above and and the simplification rules defined previously translated to CC and loaded into the CC reduction machine.

3.4. Example 1

Our first simple example demonstrates the translation of an imperative program to a functional representation and includes static semantic processing and constant folding. We partially apply the valuation function for programs, `p`, to a program that will result in a function with functionality `File` $\rightarrow$ `Post-File`.

```
(p (*s* (begin
          ((const one = 1) %
           (var i)) %
           (if (one = 1) then
               ((read i) % (write i))
            else (write one))
          end)))
```

The bracketing of the program indicates the structure of the abstract syntax tree. The result after reduction is:

```
(*t* ((K p-ok) (*t* (fst))))
```

The function `fst` selects the first element of a tuple. The DLC equivalent (found using the CC to DLC translator and DLC reduction machine) of this CC expression is:

```
(*t* ((lambda(x0) p-ok) (*t* (fst))))
```

To understand this result we must realise that the application of a tuple

```
(*t* (e1 e2 ... en))
```

to an argument, say x, reduces to

```
(*t* ((e1 x) (e2 x) ... (en x))).
```

Consequently, if the result is applied to a file (a tuple) then it will return a `Post-File` (an element of the disjoint union `File+{p-ok p-err}`) that is tagged to indicate that no errors occurred and whose file component contains one element which will be the same as the first element of the input file. This is clearly the most concise functional description of the program obtainable.

3.5. Example 2

This example demonstrates the inline expansion and simplification of procedure calls. The program comprises a variable declaration, a procedure declaration, a read statement and a procedure call.

```
(p (*s*
   (begin
      ((var y) %
       (proc sum(x) =
          ((y := x) %
           (write (x + y))))) %
       ((read y) %
        (call sum (y + 1)))
 end)))
```

The result after reduction is:

```
(*t*
    ((K p-ok)
      (*t* (((C ((B +)
                 ((S ((B +)
                      ((C ((B +)
                           fst))
                       1)))
                  fst)))
                1)))))
```

The DLC equivalent is:

```
(*t* ((lambda(x1) p-ok)
      (lambda(x2)
         (+
           (+ (fst x2) 1)
           (+ (fst x2) 1))))))
```

The resulting

4. Conclusions

We have demonstrated that since the denotational definition of the semantics of a programming language can be viewed as a functional program, we immediately have a functional representation for any program written in that language. It is obtained by partial application of the program valuation function to the program. The translation to combinator code and the intermixing of partial evaluation and rule-based simplification by the D-machine, results in a compact, optimized representation. If all the semantic algebras are "unfrozen" then the representation will contain no imperative operations. It has been found that the degree of compaction and optimization declines with an increase in the complexity of programs, primarily due to the small number of simplification rules and how they are applied. Ongoing research aims at improving performance by the use of typed pattern variables and heuristics.

The resulting functional representation, even in the case of imperative languages, can be evaluated on functional hardware or transformed to another representation, such as λ-calculus, for some other purpose. This work has particular significance as part of a broader project that seeks to provide an environment in which abstract, partially specified programs can be refined by successive, machine aided transformations until a concrete, fully specified program results.

References

[1] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[2] S. Chao and B. Bryant. Denotational semantics for program analysis. *SIGPLAN Notices*, 23(1), January 1988.

[3] J. Hindley and J. Seldin. *Introduction to Combinatory Logic and λ-Calculus*. Cambridge University Press, 1986.

[4] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[5] D. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[6] D. Scott. Lattice-theoretic models for various type-free calculi. *Logic, Methodology and Philosophy of Science*, IV:157–187, 1973.

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