Intensional Equality `;=)` for Continuations

Andrew W. Appel
Princeton University
appel@princeton.edu

September 8, 1995

Abstract

I propose a novel language feature, *intensional continuation equality*, useful in languages with or without first-class continuations, and show how it enables *truly remarkable gains in efficiency* of ordinary user programs.

Continuations, expressing "what the program will do from now on," are a much-used tool of semantics, and sometimes show up as a user-accessible programming feature. But most use of continuations is *parametric*, in the sense that functions behave the same way independent of their continuation. I will show that nonparametric use of continuations allows very substantial, almost incredible gains in program speed. Furthermore, this technique is compatible with almost any style of programming language; imperative, functional, even object-oriented.

Introduction

Many programming languages allow variables to hold function values; some languages allow the programmer to test function values for equality. But what does such a test $f = g$ mean? One can imagine three possibilities:

Extensional: returns true if $f(x) = g(x)$ for all x. This test is not computable in an ordinary (Turing-equivalent) programming language, so an extensional $=$ must be partial, perhaps returning *false* accurately on some inputs, but necessarily returning *maybe* on most inputs. I know of no programming language with an extensional equality primitive.

Intensional: returns true if f is the same expression (has the same internal structure) as g. Intensional equality implies extensional equality, but the lack of intensional equality does not provide much information to the programmer. Interpreted Lisp 1.5 has a structural (intensional) equality test.

Reference: tests whether f points to the same location as g. This is a weak form of intensional equality; reference equality implies intensional equality. Many languages have reference equality.

A seemingly unrelated development in the field of programming languages is the notion of *continuation* to express "what the program will do next." Originally developed as a "behind-the scenes" tool to help express the semantics of control flow in programming languages, continuations a programmer-accessible feature of languages such as Scheme.

This *call-with-current-continuation* feature has been criticized as being too expensive to implement because entire control stacks need to be copied. Some researchers [1, 2] propose a "prompt" primitive to ameliorate the expense by manipulating continuations in a carefully controlled way. Here I will propose a new, limited continuation primitive that is cheap (even for stacks) and that can accomplish certain things for which prompts are too weak.

A programming contest

In the spring of 1995, I invited Guy Jacobson of AT&T Bell Laboratories to teach our undergraduate software engineering course, entitled "Advanced Programming Techniques." During the semester Jacobson gave his class several programming assignments, and he he formulated some of them in the form of contests. The post-facto discussion of different solutions implemented by the students was a useful and lively pedagogical exercise.

One of the early exercises was to implement an efficient integer cube-root function. He provided two program modules, a driver module `main.c` (figure 1) and a cube-root module `root.c` (figure 2). The task was to write an efficient module `fastroot.c` to be run on a Sparc workstation such that

1. The executable obtained by conventionally linking `fastroot.c` with `main.c` would run as fast as possible;
2. `fastroot.c` would match the behavior of `root.c` when linked with *any* test driver.

That is, `fastroot.c` must be both fast and correct.

To derive a good solution to this problem, I examined the continuation transform Q of the `quickroot` function:

```
main (int ac, char *av[])
{
   int i, j;
   (void) srandom (atoi (av[1]));
   for (i = 0; i < 10000000; i++) {
       j = quickroot (random ());
   }
   exit (0);
}
```

Figure 1: The driver module `main.c`

```
double cbrt (double);

int quickroot(int i)
{return (int) cbrt ((double) i);}
```

Figure 2: The cube-root module `root.c`
The library function `cbrt` computes floating-point cube roots.

$$Q : \quad (\mathit{Int} \times \mathit{Kont}) \rightarrow \mathit{Cont}$$

where *Cont* is a conventional continuation function of the form *Store* → *Answer* and *Kont* is an expression continuation of the form *Int* → *Cont*.

Here, *Store* is a snapshot of the machine's memory, and *Answer* is the sequence of all output (and other externally visible system calls) that the program will perform from "now" until the time it exits.

We are required to implement a function *extensionally* equal to Q_1, where

$$Q_1(i,k)s = k(\lfloor i^{1/3} \rfloor)s$$

Here it is very valuable that we have made the continuation argument explicit, because we can now apply a standard rule of software engineering: make the program fast on frequently occurring arguments. One frequently occurring argument is the particular continuation k_0 provided when the driver program `main.c` calls `quickroot`. Examination of the program reveals that `main` and `quickroot` produce no output and do no other system calls before calling `exit`. Thus, the continuation k_0 is extensionally equal to

$$\lambda x.\lambda s.\mathtt{exit}(0)$$

where `exit` is a library function that produces the empty answer.

To optimize for the common case, we can write

$$Q_2(i,k)s = \textbf{if } k = k_0 \textbf{ then } \mathtt{exit}(0) \textbf{ else } k(\lfloor i^{1/3} \rfloor)s$$

It is easy to check that Q_2 is extensionally equal to Q_1 by case analysis on k.

But the equality test $k = k_0$ is problematic. We cannot use an extensional check, as that will take far too long. However, for this purpose, intensional equality suffices. Let k_{main} be the actual continuation value passed to the compiled Q by the compiled `main` function; it is extensionally equal to k_0 but (unless the compiler is very smart) not necessarily intensionally equal.

We can just test $k = k_{\mathrm{main}}$, using intensional equality as a weak but sufficient approximation. To make this explicit, I will introduce the symbol `;=)` for intensional equality:
$Q_3(i,k)s = \textbf{if } k \;\mathtt{;=)}\; k_{\mathrm{main}} \textbf{ then } \mathtt{exit}(0) \textbf{ else } k(\lfloor i^{1/3} \rfloor)s$

Practical application

It is all very well to write lambda calculus, but for the contest I need a C program. Fortunately, C is powerful enough to allow the implementation of intensional equality on continuations. My `quickroot` can simply grab the return address register and see if it points within machine code structurally similar to the contest-driver `main`. If so, we have $k \;;\!=)\; k_{\mathrm{main}}$ and we can apply the `exit` continuation. If not, we must have some other continuation meant to test for correct cube root computation, and it would be wise to compute the cube root (slowly and carefully) and return it.

The complete solution is shown in figure 3.

The array `mycaller` is actually two Sparc instructions that will return the return address of its caller. Using the attractive and powerful *union* feature of C, we cast this to a function value.

The first **if** statement is so that we apply the intensional equality optimization only on the first call to `quickroot`, so that not much time is lost for unexpected test programs.

The second **if** tests whether `quickroot`'s continuation points within `main`, taking advantage of the fact that C function pointers are represented uniformly as addresses.

The **for** loop examines the instructions of `main` to see if they match our copy of the standard driver, which we call `copyOfMain`. The **if** statement inside the loop relocates certain jump instructions that contain absolute addresses.

Performance

The original program using `cbrt` runs in about 20 seconds. The driver alone (using an empty `quickroot` function) runs in about 2 seconds. My version, using intensional equality on continuations, runs in 0.0 seconds (rounded to the nearest tenth). This was sufficient to win the contest, had I been eligible to enter. Furthermore, my version gave correct answers on any test input that Jacobson was able to devise.

```
#include <stdio.h>

copyOfMain (int ac, char *av[])
{
   int i, j;

   (void) srandom (atoi (av[1]));

   for (i = 0; i < 10000000; i++) {
       j = quickroot (random ());
   }

   exit (0);
}
endMain(){}

double cbrt (double);

extern main();

unsigned mycaller[] =
    {0x81c3e008,0x9010001f};

int quickroot(int i)
{static x=0;
 if (x) return (int) cbrt ((double) i);
 x=1;
  { unsigned *p, *q, caller;
   union {unsigned *z; unsigned (*f)();} u;
   u.z=mycaller;
   caller = u.f();
   if (caller <= (unsigned)main ||
       caller >= (unsigned)main +
                 (unsigned)endMain -
                 (unsigned)copyOfMain)
     return quickroot(i);
   for(p=(unsigned*)copyOfMain,
        q=(unsigned*)main;
       p<(unsigned*)endMain;
       p++,q++)
    { unsigned px = *p, qx = *q;
      if ((px&0xf0000000) == 0x40000000 &&
          (qx&0xf0000000) == 0x40000000)
          {px += ((unsigned) p)>>2;
           qx += ((unsigned) q)>>2;}
      if (px != qx) return quickroot(i);
    }
   exit(0);
  }
}
```

Figure 3: Optimizing the common case

Conclusion

As I have shown, the use of an intensional equality test on continuations does not require the programming language to have full first-class continuation (i.e., *call-with-current-continuation*). But what I have done cannot be expressed using prompts, because I need to reason about the *full* answer produced by the program, not a partial answer up to the next prompt.

But my implementation is a bit clumsy in C. What we need is a primitive for testing intensional equality on continuations against specified constant values, to allow portable code to be efficient in the common case.

Disclaimer The opinions expressed in this paper are not those of Princeton University, nor even those of the author.

References

[1] Matthias Felleisen. The theory and practice of first-class prompts. In *Fifteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 180–90, New York, Jan 1988. ACM Press.

[2] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Proc. Seventh Int'l Conf. on Functional Programming and Computer Architecture*, pages 12–23. ACM Press, 1995.