

Re: CAS, Lambda Calculus, Continuation and Functional Programming

Source: <http://sci.tech-archive.net/Archive/sci.math.symbolic/2005-03/0171.html>

From: Jens Axel Sjøgaard (*usenet_at_soegaard.net*)

Date: 03/16/05

Date: Wed, 16 Mar 2005 22:45:52 +0100

Robert A Duff wrote:

> Joachim Durchholz <*jo@durchholz.org*> writes:

>> *Recursion that's equivalent to loops can always be optimized back into*

> *^^^^*

>> *loop form when it comes to emitting code. This is not a problem with any*

>> *single functional programming language that I know.*

> *I don't think that's *quite* true. I agree that tail calls can be*

> *eliminated, and that tail calls are equivalent to loops.*

> *But there are recursive algorithms that are not tail recursive,*

> *but that are equivalent to loops.*

You and Joachim are using different definitions of equivalent :-)

If a loop is formulated in a recursive way (via tail calls) then

the compiler will generate the same code. An recursive algorithm

that are not tail recursive is not equivalent to a loop.

>>> *Is there anything that would*

>>> *prohibit using the same optimizations in an imperative language.*

>>

>> *Partly. Imperative aspects of a language may require global program*

>> *analysis to make sure that a tail call can indeed be eliminated, and*

>> *that makes tail call elimination less attractive for compiler writers.*

>> *I don't recall the full range of things that can prevent tail call*

>> *elimination, but the two that I do recall are exception handling and*

>> *setjmp/getjmp. I dimly recall that there were also issues with*

>> *destructors in C++, but I may be confusing things here.*

>

> *I don't know why exception handling should prevent tail-call*

> *elimination. Would someone care to enlighten me?*

>

> *Setjmp/longjmp is similar to exception handling. And destructors*

> *require something like an implicit exception handler -- when an*

> *exception occurs, catch it, call the destructor, and then re-raise the*

> *same exception. So these all seem like the same issue.*

>

- > *I think garbage collection makes tail call elimination easier,*
- > *and many imperative languages don't have GC, so that might be*
- > *part of the problem.*

Good point. Implementing Proper Tail Recursion [1] implies that iterative algorithms formulated recursively should run in constant space. I wouldn't attempt to do that with some form of GC.

- > *In any case, I know that some compilers for some imperative languages do*
- > *tail-call elimination in some cases.*
- >
- > *On the other hand, some functional languages *require* tail-call*
- > *elimination. Is this really a formal requirement?*

An example of a language standard that mandates it, is the R5RS standard for Scheme.

Will Clinger has formalized the requirement in [1] "Proper tail recursion and space efficiency" see <<ftp://ftp.ccs.neu.edu/pub/people/will/tail.ps.gz>>.

- > *I mean, can one write a test case that fails if and only if the compiler*
- > *doesn't eliminate tail calls?*

That is tricky. One can write programs that are likely to blow the stack, if the compiler doesn't handle the bounded-space requirement. E.g. this program

```
(define (f) (g))
(define (g) (f))
(f)
```

is an infinite loop if Proper Tail Recursion is supported, otherwise one will get an out-of-memory-error.

I think, that it is impossible to write a program that, say, prints yes if and only if Proper Tail Recursion is supported.

--
Jens Axel Søgaard