Acta Univ. Sapientiae, Informatica, **5,** 2 (2013) 167–183

DOI: 10.2478/ausi-2014-0009

Stackless programming in Miller*

Boldizsár NÉMETH
Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary
email: nboldi@caesar.elte.hu

Zoltán CSÖRNYEI
Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary
email: csz@inf.elte.hu

Abstract. Compilers generate from the procedure or function call instruction a code that builds up an "activation record" into the stack memory in run time. At execution of this code the formal parameters, local variables, data of visibility and the scope are pushed into the activation record, and in this record there are fields for the return address and the return value as well. In case of intensive recursive calls this is the reason of the frequent occurrences of the stack-overflow error messages. The classical technique for fixing such stack-overflows is to write programs in stackless programming style using tail recursive calls; the method is usually realised by Continuation Passing Style. This paper describes this style and gives an introduction to the new, special purpose stackless programming language *Miller*, which provides methods to avoid stack-overflow errors.

1 Introduction

"The modern operating systems we have operate with what I call the 'big stack model'. And that model is wrong, sometimes, and motivates the need for 'stackless' languages." (Ira Baxter, 2009 [1])

Computing Classification System 1998: D.3.2, D.3.3.
Mathematics Subject Classification 2010: 68-02, 68N15, 68N18
Key words and phrases: tail recursion, continuation passing style, stackless programming, programming language Miller

*Supported by Ericsson Hungary and EITKIC 12-1-2012-0001.

Using the big stack model in case of intensive recursive calls stack-overflow error messages may occur frequently. The classical method to fix such stack-overflows is to write programs in stackless programming style, when tail recursive procedures are used to eliminate the cases of running out of the available stack memory. This method is usually realised by Continuation Passing Style (CPS).

2 Stackless programming

2.1 Recursion and iteration

Hereinafter the usual definition of the factorial function is given. It is obvious that at all recursive call `fac i` it is needed to save information for the next operation, namely what operation has to be executed when the `fac i` is finished and its value is available.

$$\begin{aligned} \mathtt{fac} \equiv \quad & \lambda x \,.\, \mathtt{if} \quad (\mathtt{zero}\ x) \\ & \qquad\quad 1 \\ & \qquad\quad (*\ x(\mathtt{fac}\ (-\ x\ 1))) \end{aligned}$$

For example, the action of calculating the value of `fac 3` as follows.

$\mathtt{fac}\ 3 \rightarrow$
$*\ 3\ (\mathtt{fac}\ 2) \rightarrow$
$*\ 3\ (*\ 2(\mathtt{fac}\ 1)) \rightarrow$
$*\ 3\ (*\ 2(*\ 1(\mathtt{fac}\ 0))) \rightarrow$
$*\ 3\ (*\ 2(*\ 1\ 1)) \rightarrow$
$*\ 3\ (*\ 2\ 1) \rightarrow$
$*\ 3\ 2 \rightarrow$
6

It is obvious that if the value of the call `fac i` is calculated then it is needed to return to the caller process to execute multiplications. It means that activation records have to be used and thus a stack memory has to be applied for registration the calculating processes.

This is a so called "*recursive-controlled behaviour*", and it is obvious that using this method the stack-overflow error may appear in the case of intensive, multiple recursive calls.

There is a simple method to avoid stack-overflow errors, it has the name

"*iterative-controlled behaviour*". It is a simple iteration, where there is no need to preserve long series of operations, the size of occupied memory is not increased in the course of execution of recursive calls, and the most important property is that all of the calls are on the same level. This method uses an accumulator for storing intermediate results [4].

The factorial function in the iterative-controlled style is as follows. In this definition variable r is the accumulator.

```
fac ≡   λn  . fac′ n 1
fac′ ≡  λxr . if  (zero x)
                  r
                  (fac′ (− x 1) (∗ x r))
```

It seems that in the calculating process there is only one level for recursive calls, for example the value of fac 3

```
fac 3 →
fac′ 3 1 →
fac′ 2 3 →
fac′ 1 6 →
fac′ 0 6 →
6
```

There is no need to large stack memory for activation records, only two variables are required, one for the value n and another variable for the accumulator r. The calculating process is described and controlled by these variables.

It is important to observe that the called process does not return to the caller process to execute any operation, since there is no operation to be executed.

Iterative behaviour seems to be a very nice method, but it is applicable for cases where the size of the accumulator is constant during the calculation, and what is more, unfortunately there are procedures for which there is no possibility to convert them into iterative-controlled behaviour forms. But the continuation passing style solves this problem.

2.2 Tail position and tail call

In the previous example it was shown that the called process does not return to the caller process, and for this case we say that a tail call was executed.

More precisely, the procedure E is in *tail position* of the enclosing procedure F if F does not have any action to performed after E is returned. This means

that the return value of E is the result of F. *Tail call* means a call to expression in tail position, and the recursion is said to be *tail recursion* if the recursive calls are tail calls.

Thus in the case of tail calls there is no need for extra memory to the control information, the result of the procedure E is the result of F. Namely, after executing E, the control of execution is passed to the process which is the continuation of F.

There is a general method to convert procedures into this form, we have to write procedures in continuation passing style where the continuation represents what is left to do.

3 CPS—Continuation Passing Style

Continuation is a function and the result of the procedure is applied to it. A new variable is introduced, this variable represents the *continuation*. It usually has the name k.

There are many methods to convert an expression into continuation passing style [2, 8]. For example, a formal method published by Plotkin is as follows.

$$\begin{aligned}
[\mathsf{x}]\ \mathsf{k} &= \mathsf{k}\ \mathsf{x} \\
[\mathsf{n}]\ \mathsf{k} &= \mathsf{k}\ \mathsf{n} \\
[\lambda \mathsf{x}\ .\ \mathsf{E}]\ \mathsf{k} &= \mathsf{k}\ (\lambda \mathsf{x} \mathsf{v}\ .\ [\mathsf{E}]\ \mathsf{v}) \\
[\mathsf{E}\mathsf{F}]\ \mathsf{k} &= [\mathsf{E}]\ (\lambda \mathsf{v}\ .\ [\mathsf{F}]\ (\lambda \mathsf{w}\ .\ \mathsf{v}\ \mathsf{w}\ \mathsf{k}))
\end{aligned}$$

where the expression $[\,.\,]$ is due to convert, x is a variable and n is a constant. For example, if the continuation is $\mathsf{k} \equiv \texttt{print}$, then using the second rule to form $[6]$ `print`, it results `print 6` as it was expected.

For reductions it is not too hard to prove that $\mathsf{E} \to \mathsf{F} \iff [\mathsf{E}]\ \mathsf{k} \to [\mathsf{F}]\ \mathsf{k}$.

It is known that $(\lambda \mathsf{x}\,\mathsf{y}\ .\ \mathsf{y})\ \mathsf{1} \to \lambda \mathsf{y}\ .\ \mathsf{y} \equiv \mathsf{Id}$. The next example shows that $[(\lambda \mathsf{x}\,\mathsf{y}\ .\ \mathsf{y})\ \mathsf{1}]\ \mathsf{k} \to [\lambda \mathsf{y}\ .\ \mathsf{y}]\ \mathsf{k}$.

$$\begin{aligned}
&[(\lambda \mathsf{x}\,\mathsf{y}\ .\ \mathsf{y})\ \mathsf{1}]\ \mathsf{k} = \\
&[\lambda \mathsf{x}\,\mathsf{y}\ .\ \mathsf{y}]\ (\lambda \mathsf{v}\ .\ [\mathsf{1}]\ (\lambda \mathsf{w}\ .\ \mathsf{v}\ \mathsf{k}\ \mathsf{w})) = \\
&[\lambda \mathsf{x}\,\mathsf{y}\ .\ \mathsf{y}]\ (\lambda \mathsf{v}\ .\ (\lambda \mathsf{w}\ .\ \mathsf{v}\ \mathsf{k}\ \mathsf{w})\ \mathsf{1}) = \\
&(\lambda \mathsf{v}\ .\ (\lambda \mathsf{w}\ .\ \mathsf{v}\ \mathsf{k}\ \mathsf{w})\ \mathsf{1})\ (\lambda \mathsf{p}\,\mathsf{x}\ .\ [\lambda \mathsf{y}\ .\ \mathsf{y}]\ \mathsf{p}) \\
&(\lambda \mathsf{v}\ .\ (\lambda \mathsf{w}\ .\ \mathsf{v}\ \mathsf{k}\ \mathsf{w})\ \mathsf{1})\ (\lambda \mathsf{p}\,\mathsf{x}\ .\ \mathsf{p}\ (\lambda \mathsf{q}\,\mathsf{y}\ .\ \mathsf{q}\ \mathsf{y})) \to \\
&(\lambda \mathsf{w}\ .\ (\lambda \mathsf{p}\,\mathsf{x}\ .\ \mathsf{p}\ (\lambda \mathsf{q}\,\mathsf{y}\ .\ \mathsf{q}\ \mathsf{y}))\ \mathsf{k}\ \mathsf{w})\ \mathsf{1}) \to \\
&(\lambda \mathsf{p}\,\mathsf{x}\ .\ \mathsf{p}\ (\lambda \mathsf{q}\,\mathsf{y}\ .\ \mathsf{q}\ \mathsf{y}))\ \mathsf{k}\ \mathsf{1} \to \\
&\mathsf{k}\ (\lambda \mathsf{q}\,\mathsf{y}\ .\ \mathsf{q}\ \mathsf{y}) =
\end{aligned}$$

$k\ (\lambda q\, y\ .\ [y]\ q) =$
$[\lambda y\ .\ y]\ k$

There is a simple method to convert the expression to a tail call form, the method is presented by the calculation of $\mathtt{fac}\ 3$ [10]. We see an intermediate state:
$*\ 3(*\ 2\ (\mathtt{fac}\ 1))$.
Replace the call to $\mathtt{fac}\ 1$ with a new variable x,
$*\ 3\ (*\ 2\ x)$,
and create a λ-abstraction with this new variable:
$\lambda x.*\ 3\ (*\ 2\ x)$,
this is a continuation k of the expression $\mathtt{fac}\ 1$, that is
$k\ (\mathtt{fac}\ 1) =$
$(\lambda x.*\ 3(*\ 2\ x))\ (\mathtt{fac}\ 1) \rightarrow$
$*\ 3\ (*\ 2\ (\mathtt{fac}\ 1))$.
This means that the steps of calculation have form $k\ (\mathtt{fac}\ i)$. Now we create a new version of fac that takes an additional argument k for continuation and calls that function as the original body of $\mathtt{fac}$, that is
$\mathtt{fac\text{-}cps}\ n\ k \rightarrow k\ (\mathtt{fac}\ n)$.
The function $\mathtt{fac\text{-}cps}$ has the form as follows.

$$\begin{aligned}\mathtt{fac\text{-}cps} \equiv \lambda n k\ .\mathtt{if}\quad & (=\ n\ 0)\\ & (k\ 1)\\ & (\mathtt{fac\text{-}cps}\ (-\ n\ 1)\ (\lambda v\ .\ (k\ (*\ n\ v))))\end{aligned}$$

For example the value of $\mathtt{fac\text{-}cps}\ 3\ k$:

$\mathtt{fac\text{-}cps}\ 3\ k\ \ =$
$\mathtt{fac\text{-}cps}\ 2\ (\lambda v\ .\ (k\ (*\ 3\ v)))\ \ =$
$\mathtt{fac\text{-}cps}\ 1\ (\lambda v'\ .\ ((\lambda v\ .\ (k\ (*\ 3\ v)))\ (*\ 2\ v'))) \rightarrow$
$\mathtt{fac\text{-}cps}\ 1\ (\lambda v'\ .\ (k\ (*\ 3\ (*\ 2\ v')))) =$
$\mathtt{fac\text{-}cps}\ 0\ (\lambda v''\ .\ ((\lambda v'\ .\ (k\ (*\ 3\ (*\ 2\ v'))))(*\ 1\ v'')))\ \ \rightarrow$
$\mathtt{fac\text{-}cps}\ 0\ (\lambda v''\ .\ (k\ (*\ 3\ (*\ 2\ (*\ 1\ v''))))) =$
$(\lambda v''\ .\ (k\ (*\ 3\ (*\ 2\ (*\ 1\ v''))))))\ 1 \rightarrow$
$(k\ (*\ 3\ (*\ 2\ (*\ 1\ 1)))) \rightarrow$
$k\ 6$

If $k \equiv \mathtt{print}$, then $\mathtt{fac\text{-}cps}\ 3\ \mathtt{print} = \mathtt{print}\ 6$, as it was expected.

We remark that continuations can be used to implement for example branching, loops, goto, return, and give possibility to write such types of control flow that coroutines, exceptions and threads.

4 Tail Call Optimisation in various languages

Tail Call Optimisation (TCO) is a common technique for transforming some execution units of the program to operate in constant stack space. The function with the recursive call is transformed into a loop, while preserving the semantics with the appropriate condition.

We chose Scala, a functional language running on JVM and the purely functional language Haskell. We did so because the problem is most relevant to functional languages where recursive calls are the only source of iterative behaviour. We also wanted to compare the different approaches to this problem.

4.1 TCO in Scala

The Scala compiler implements a limited form of the TCO.

The problem with the recursive approach is that calls to non-static functions in the JVM are dynamically dispatched. There is a direct and an indirect cost of this, mostly studied in C++ programs [3]. However extensive research had been done on the resolution of virtual calls in Java programs [11].

The system only handles self-recursion, so two functions mutually calling each other will not be transformed into a single cycle. The designers of the compiler introduced this constraint because they didn't want to cause duplications in the generated code, that could cause the program to slow down.

TCO can only be used on non-overrideable functions, because the dynamic method invocation of the JVM prevents further optimisations [9].

With the `@tailrec` annotation, the programmer can ensure that the TCO can be performed by the compiler, otherwise the compilation fails.

Example

```
def factorialAcc(acc: Int, n: Int): Int = {
  if (n<=1) acc
  else factorialAcc(n*acc, n-1)
}
```

The program is compiled to the same bytecode as:

```
def factorialAcc(acc: Int, n: Int): Int = {
  while(n <= 1) {
    acc = n*acc
    n = n-1
  }
  acc
}
```

So in this example we can get the elegant functional solution with no additional costs.

4.2 TCO in Haskell

Tail call elimination in Haskell is a little different than in languages with strict execution. It allows not only tail call functions to be executed in constant stack space, but a wider class of functions, that are called *productive* functions [7].

Every function is productive if only contains recursive calls in a data constructor.

For example take the three function definitions below:

```
infinite_list = 1 : infinite_list

infinite_number = go 0
  where go n = (let n' = n+1 in n' `seq` go n')

infinite_number' = 1 + infinite_number'
```

The `infinite_list` function is productive, because the recursive call is a parameter of the `:` data constructor, therefore the execution will not result in a stack overflow. And it will generate an infinite list of 1's.

The `infinite_number` function has a tail call, where the result is accumulated as an argument, and strictly evaluated by the `seq` function. If we would allow the lazy execution of the accumulator parameter, the tail call would be eliminated, but because the parameter is constantly growing, the "out of memory" error would be inevitable as seen in the case of `infinite_number'`. See also the strict folding functions `foldr'` and `foldl'` in [6].

Let's see the result of the execution of the three statements above:

```
> infinite_list
[1,1,1,1,1... -- This goes forever, but does not result in stack overflow.
```

```
> infinite_number
              -- Goes on forever, but also in constant stack size
> infinite_number'
<interactive>: out of memory
```

5 Structures for stackless programming

5.1 The Haskell Cont monad

The `Cont` monad can be found in the Monad transformer library [5], in the `Control.Monad.Cont` module. It is a monadic structure for writing functions with continuation passing style.

```
newtype Cont r a = Cont {
    runCont :: (a -> r) -> r  -- Returns the value after applying
                              -- the given (final) continuation to it.
}

instance Monad (Cont r) where
    return a = Cont (λf -> f a)
      -- When returning, simple supply the result to the final continuation.

    m >>= k  = Cont $ λc -> runCont m (λa -> runCont (k a) c)
      -- When binding, set the continuation of the first expression to
      -- the second expression (that gets the final continuation).$
```

After making a `Monad` instance for the `Cont` datatype, we can use it to create a factorial calculation in a simple way. The `fac` function simply executes `fac_cont` with an identity transformation as the final continuation.

The `fac_cont n` applies the final continuation with return, or executes `fac_cont (n-1)` with the multiplication as a continuation.

```
fac :: Int -> Int
fac n = runCont (fac_cont n) id
  where fac_cont :: Int -> Cont Int Int
        fac_cont 0 = return 1
        fac_cont n = do fprev <- fac_cont (n-1)
                        return (fprev * n)
```

For demonstrating the power of our continuation-using factorial function, we also present a naïve recursive implementation.

```
fac_naive :: Int -> Int
fac_naive n = n * fac_naive (n-1)
```

Then we execute both for a number larger than the maximum stack size, and inspect the results:

```
> fac 1000000
0    (Because of the overflow)
> fac_naive 1000000
<interactive>: out of memory
```

If we follow the execution of the `fac_cont` function we can observe that this is identical to the previous `fac-cps` example in Section 3.

```
> fac_cont 3
runCont (fac_cps 3) id
runCont (fac_cps 2) (λa1 → runCont (return (a1*3)) id)
runCont (fac_cps 1) ((λa2 → runCont (return (a2*2)))
                      (λa1 → runCont (return (a1*3)) id))
runCont (fac_cps 0) (λa3 → runCont (return (a3*1))
                      ((λa2 → runCont (return (a2*2)))
                        (λa1 → runCont (return (a1*3)) id)))
runCont (return (1)) ((λa2 → runCont (return (1*2)))
                       (λa1 → runCont (return (a1*3)) id))
runCont (return (1*2))) (λa1 → runCont (return (a1*3)) id)
runCont (return (1*2*3)) id
1*2*3
6
```

6 Stackless elements of the Miller programming language

In this section of the article we present the aspects of the Miller* programming language that are related to stackless programming.

* George Armitage Miller (February 3, 1920 – July 22, 2012) was one of the founders of the cognitive psychology field. He also contributed to the birth of psycholinguistics and cognitive science in general. Miller wrote several books and directed the development of WordNet, an online word-linkage database usable by computer programs [12]. We chose his name for our language because of his great contribution for the understanding on the usage of human memory, while our language focuses on the usage of electronic memory.

6.1 The Miller programming language

The Miller programming language is an industry-oriented programming language, focused on the development of performance-critical applications for special hardware.

The simplest execution unit of Miller is the *bubble*. A bubble is a separate compilation unit. Bubbles can contain simple sequential code, but no control structures such as branches and cycles. At the end of the bubble there is a section where conditions decide which bubble will be executed next. These are the *exit points* of the bubble.

Bubbles can also form a network, interconnected by their exit points. A *graph bubble* embeds a network of bubbles into itself. The nested bubbles can only be used by transferring the control to one of the entry points of the graph bubble. If an exit point of an inner bubble is connected to the entry point of another bubble inside it is called a *local exit point*. Otherwise if it's connected to an exit point of the containing graph bubble it is a *far exit point*.

The graph bubble creates an encapsulation for its inner bubbles, and defines an interface through which they can be accessed. Graph bubbles can also be nested into other graph bubbles. Nested bubbles can use any program element defined in their graph bubbles.

It is easy to see that graph bubbles, bubbles and exit points are equivalent in expressive power to the control structures of structured programming. Branches and loops can be simulated using a network of bubbles.

7 Defining control flow with bubbles

The general case of defining the transfer of control is to set exit points of the bubbles. This also allows the programmer to create control cycles. We experimented with this mode of control, but found that it is too cumbersome for actual programming.

Non-sequential code (for example branches and cycles) will be transformed into a network of bubbles. However, the language does not require the programmer to manually create these bubbles and their connections. An imperative programming interface is specified from which the compiler creates the final bubble graph. This interface contains *if-then* branching, *if-then-else* branching, special branching operations, and a *do-while* loop.

Nested bubbles can be instantiated in their ancestor bubbles any number of times.

Example

The following example shows a typical conversion from imperative frontend to a network of bubbles. It shows how the while cycle is transformed to bubbles in a naïve algorithm to compute the greatest common divisor of two positive integers.

Compile[... while (b ≠ 0)
if(a > b) { a := a − b; b := b − a; }
...]

The while cycle will be transformed by a stateful transformation. The condition and the loop body are separated.

Condition(b ≠ 0)
Core (Compile[if(a > b) a := a − b; b := b − a;])
State(bubble previous_bubble{...}...)

Then another transformation will create bubbles from the condition and the loop core and combine them into the state.

State(
bubble _loop_condition ⇒ _loop_core, _exit {...}
bubble _loop_core ⇒ _loop_condition{
Compile[if(a > b) { a := a − b; b := b − a; }]})

Using bubbles for the transfer of control does not need a stack. The programmer cannot return control to the caller from an execution module, just pass the control to the next execution unit. If call-and-return behaviour is expected, limited depth function calls provide help.

The result of the execution of the bubble body decides through which exit the control flow is passed.

Control flow of bubbles currently cannot follow continuation passing style, since it is not possible to transfer the exit point. A language feature is under planning which allows compile time passing of control. This is the parametrisation of the bubble exit points.

8 Parameter passing with interfaces

The bubble states that through an exit point which variables are passed to the next bubble. This is the exit specification. The bubble also declares the

variables that it uses, this is the entry specification. The interface checker verifies that the exit and entry specifications match.

It is very important to clarify that no copy operations happen, for this is not the traditional way of parameter passing. All variables declared in the exit and entry specification must appear in one of the ancestor bubbles.

The interface check theoretically prevents that the program has access to uninitialised variables. In practice this does not always happen. For example, it cannot be statically proven that a loop cycle running on an array gives all elements a starting value.

Nevertheless, the interface check gives us the same confidence that can be given by inspecting the initial assignment of variables, and does not require any data copy to be made for parameter passing. In addition, it also enables to create variables with constant values locally, in the scope of one bubble.

Example: factorial function in Miller

We present two approaches to calculate the factorial function. The first approach is a naïve recursive function, using stack. It is presented in a C-like pseudo-code. The second is a stackless approach, with bubbles accessing global variables, and using iterative control structure. It is presented with the pseudo-code version of the language Miller. We give the sequence of evaluation for each implementation.

Please note that, although the algorithm is the same as the example presented in the first part of the article, it is written in procedural and not in functional style.

```
int32 fac( int32 n ) {
    if( n ≤ 1 ) {
        return 1;
    } else {
        return n * fac(n-1);
    }
}
```

The next table shows the content of the stack after each step of execution. The ‘n’ columns show the value of the variable `n` in the given context. The `ret` columns show the points where the control is returned after the return call.

step #	ret	n	ret	n	ret	n
1	caller	3				
2	caller	3	fact	2		
3 (return 1)	caller	3	fact	2	fact	1
4 (return 2)	caller	3	fact	2		
5 (return 6)	caller	3				

The second part of the example shows the factorial function implemented with stackless bubbles of Miller. A few notes on the implementation:

- The language syntax may change, the purpose of the example is to give an idea about the implementation.

- The while cycle is needed because we have to connect the again exit point with the entry point of the cycle.

```
int32 n;
int32 val;

bubble fact(n)
  exits out(val) {
    val = 1;
    while(true)  cycle;
}
bubble fact::cycle(n,val)
   exits again(n,val)
   far exits out(val) {
     if( n ≤ 1 ) {
        exit out(val);
     } else {
        exit again(n-1, val*n);
     }
}
```

The next table shows the evaluation of the `fact`(3) expression in Miller with only two global variables. The two columns represent the values of the corresponding global variables.

step	n	val
1 (in fact)	3	
2 (in cycle)	3	1
3 (in cycle)	2	3
4 (in cycle)	1	6
5 (in out)		6

As can be seen, the expression is evaluated in constant stack size.

8.1 Limited depth function calls

There are situations where calling functions and returning the control is highly preferred to low level passing of control. The Miller language provides limited function calls in such situations.

Currently it's the programmers responsibility to return the control to the caller from the called method. This approach enables the function to be a complete system of bubbles, and any of them can return, if appropriate.

An important aspect of the functions is that the depth of the call chain is limited. We can calculate it in the following way:

- The bubble that calls no other bubbles have the call depth of zero.
- When a bubble calls other bubbles, its call depth is greater by one than the maximum of the call depths of the called bubbles.

Because the depth of all units must be bounded, the function calls cannot be recursive. This kind of control flow would require a stack to implement.

Thanks to the limitations on the function calls it becomes possible to store all function arguments and return addresses in registers, which is a common practice in performance critical systems. For example, if we limit our call chains to a depth of five, at most five registers would be enough to store the return addresses of the calls.

Of course, if values are passed to the called functions, more registers will be needed. To implement calls inside the bubble system, the compiler creates a calling bubble, which executes the call operation.

9 Evaluation of expressions using sandbox

The sandbox is a tool for generating instructions to evaluate complex expressions. It has a finite amount of registers and a larger amount of memory locations.

The compiler always optimises the usage of registers and memory locations to minimise the number of temporary variables. This means that the subexpressions to be evaluated first are the subexpressions that need more registers.

The sandbox works according to a simple strategy. As long as there is space the intermediate values are stored on registers, then it puts them on the specified memory areas. For now, this is enough, when it will be necessary, we design an algorithm that takes into account the different types of memory as well.

Example

In this example we present two methods to evaluate a simple expression. The first method uses the stack to evaluate expressions of any size, while the second uses a sandbox with a finite amount of registers to evaluate expressions.

The evaluation of the expression $\mathtt{a} \wedge \mathtt{b}$ to the lower part of the `ax` register with a very simple code generator is based on stack operations.

```
... instructions for the evaluation of a
    to the lower part of ax register ...
push ax
... instructions for the evaluation of b
    to the lower part of ax register ...
pop bx       (loading the previously stored value of a)
and al,bl    (executing the instruction on
              the lower part of ax and bx registers)
```

The evaluation of $\mathtt{a} \wedge \mathtt{b}$ to the lower part of ax register with our sandbox model:

```
... instructions for the evaluation of a
    to the lower part of ax register ...
... instructions to acquire a new register r
    from the sandbox ... (that may cause moving previously stored data
                          from a register to the memory.)
... instructions for the evaluation of b
    to the allocated r register ...
and ax,r
... instructions to release the allocated r register ...
```

If more registers are needed for calculating the subexpressions then more registers can be acquired. When the sandbox has temporary registers, the allocation of registers doesn't generate any instructions.

10 Summary

We investigated the techniques of generating programs that do not use a runtime stack for the calls and the evaluation of complex expressions.

We followed two different paths to address this problem. The first way was the formal method of continuation passing style, that solved the problem in a functional way. In functional languages this formal method can be implemented easily.

The second path was a practical method, implemented in the imperative Miller language. It replaced function calls with passing of control between bubbles, and function arguments with controlled global variables. This method can be used on the special hardware, which is not equipped with an efficient stack implementation. Our research was motivated by these problems.

Acknowledgements

We wish to thank the head of our research group Gergely Dévai and academic staff of the research group at ELTE Software Technology Lab for their useful and constructive work on this project and we are really thankful for making our publication possible.

We would like to thank the support of Ericsson Hungary and the grant EIT-KIC 12-1-2012-0001 that is supported by the Hungarian Government, managed by the National Development Agency, and financed by the Research and Technology Innovation Fund.

References

[1] I. Baxter, Answer # 1, http://stackoverflow.com/questions/1016218/how-does-a-stackless-language-work, 2013. ⇒167

[2] O. Danvy, K. Millikin, *On One-Pass CPS Transformations*, BRICS, Department of Computer Science, University of Aarhus, RS-07-6, 2007. ⇒170

[3] K. Driesen, U. Hölzle, The direct cost of virtual function calls in C++, *SIGPLAN Not.*, **31**, 10 (1996) 306–323. ⇒172

[4] D. P. Friedman, M. Wand, *Essentials of Programming Languages* (3rd edition), The MIT Press, Cambridge, MA, 2008. ⇒169

[5] A. Gill, *Hackage, The Monad Transformer Library package*, (16. 07. 2013.), http://hackage.haskell.org/package/mtl-2.0.1.0 ⇒174
[6] The Glasgow Haskell Team, *Haskell, The Data.List module*, (16. 07. 2013.), http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-List.html ⇒173
[7] Haskell Wiki, *Tail recursion*, (16. 07. 2013.), http://www.haskell.org/haskellwiki/Tail_recursion ⇒173
[8] S. Krishnamurthi, *Programming Languages: Application and Interpretation* (2nd edition), ebook, http://cs.brown.edu/~sk/Publications/Books/ProgLangs/, 2007. ⇒170
[9] T. Lindholm, F. Xellin, G. Bracha, A. Buckley, *The Java Virtual Machine Specification*, (28. 02. 2013.), http://docs.oracle.com/javase/specs/jvms/se7/html ⇒172
[10] A. Myers, *Continuation-passing style*, Lecture notes for CS 6110, Cornell University, 2013. ⇒171
[11] W. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, C. Godin, Practical Virtual Method Call Resolution for Java, *SIGPLAN Not.*, **35**, 10 (2000) 264–280. ⇒172
[12] Wikipedia, *George Armitage Miller*, (16. 07. 2013.), http://en.wikipedia.org/wiki/George_Armitage_Miller ⇒175

Received: August 6, 2013 • Revised: October 27, 2013