SECD Machine - Optimizations and Extensions

Leonardo Santos, Mário Florido
Faculty of Science - University of Porto

Contents

1 Introduction

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.[1]

1.1 Subsection

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

2 Language `fun`

2.1 Introduction

Let us define the sample language we will be using throughout the paper to later describe compilation and semantics.

2.2 Syntax

The BNF of our language is as follows:

$$
\begin{array}{lcl}
\langle expr \rangle & ::= & \langle ident \rangle \\
 & | & \langle int \rangle \\
 & | & \text{`}\lambda\text{'}\ \langle ident \rangle\ \text{`}\texttt{->}\text{'}\ \langle expr \rangle \\
 & | & \langle expr \rangle\ \langle expr \rangle \\
 & | & \text{`}\texttt{let}\text{'}\ \langle ident \rangle\ \text{`}\texttt{=}\text{'}\ \langle expr \rangle\ \text{`}\texttt{in}\text{'}\ \langle expr \rangle \\
 & | & \langle expr \rangle\ \langle bop \rangle\ \langle expr \rangle \\
 & | & \text{`}\texttt{if}\text{'}\ \langle expr \rangle\ \text{`}\texttt{is 0 then}\text{'}\ \langle expr \rangle\ \text{`}\texttt{else}\text{'}\ \langle expr \rangle \\
 & | & \text{`}\texttt{fix}\text{'}\ \langle expr \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle bop \rangle & ::= & \text{`}\texttt{+}\text{'} \\
 & | & \text{`}\texttt{*}\text{'} \\
 & | & \text{`}\texttt{-}\text{'}
\end{array}
$$

$\langle int \rangle$ ::= (integer literals, e.g., 0, 1, 2, ...)

$\langle ident \rangle$ ::= (identifiers, e.g., x, y, foo, ...)

2.3 Haskell datatype

For the implementation, instead of using identifiers, we use De Bruijn indexing to refer to variables. The datatype of our language is defined with a GADT that captures the return type of each constructor definition:

```haskell
data Expr a where
  Var :: Int -> Expr a
  Int :: Int -> Expr Int
  Abs :: Expr b -> Expr (a -> b)
  App :: Expr (a -> b) -> Expr a -> Expr b
  Let :: Expr a -> Expr b -> Expr b
  Bop :: BopE -> Expr Int -> Expr Int -> Expr Int
  IfZ :: Expr Int -> Expr a -> Expr a -> Expr a
  FixP :: Expr (a -> a) -> Expr a

data BopE
  = Add
  | Mul
  | Sub
```

3 SECD Machine

3.1 Introduction

The SECD machine was introduced in 1964 by Landin. It has a simple instruction set that operates on stacks. You can think about this instruction set as a virtual machine of sorts, if you translate your language into this instruction set, through a process called **compilation**, then there is a standard way of running said instructions. Hopefully, given your compiled program is correct, the list of instructions is **interpreted**, producing the expected behaviour of the term written in your language.

The list of instructions is just the first part, to **interpret** them, the SECD machine keeps 4 data structures in memory, and sometimes another helper data structure to support function applications.

3.2 Data Structures

3.2.1 S: Stack

The Stack holds intermediate results from our interpreting process. It is a list of **values**. It is also where our final output will be stored.

The stack is a list of the `Val` data type:

```
data Val
  = VInt Int
  | Addr Int -- corresponds to the position of either a variable or a function inside the Env
```

3.2.2 E: Environment

The Environment associates the values that are bound to variables. It can be seen as a simple list of values, where the indexes are refer to the variables themselves.

3.2.3 C: Code

This is the list of instructions that we compiled previously, and are currently running.

3.2.4 D: Dump

The dump stores the contents of the other 3 registers temporarily, while a function is executing, for example. You can see these function calls as **detours** from the original execution. When they are finished, we can pop the head of the Dump to go retrieve the main execution path.

We will see later that we can combine the Stack and Dump into just one data structure.

3.2.5 Closures on a Store

Closures represent function applications that are yet to be processed. As environments are very dynamic and can change quickly, when creating a Closure, the current environment is captured inside of it. This makes sure the function has access to the unbounded variables it needs.

Closures can appear on both the Stack and the Dump, but for this first implementation we will keep them in a separate place in memory, a Store.

3.3 Default Instructions

The default set of instructions is relatively small:

```
data Instr
  = LD Int -- loads the variable at the specified index
  | LDC Int -- loads an integer constant
  | CLO [Instr] -- loads a function
  | FIX [ Instr ] -- loads a recursive function
  | AP -- applies a function to an argument
  | RTN -- returns from function application
  | IF [ Instr ] [ Instr ] -- tests a condition and selects one branch accordingly
  | JOIN -- returns from whichever branch IF selected
  | ADD -- adds two values
  | SUB -- subtracts two values
  | MUL -- multiplies two values
  | HALT -- halts execution
```

3.4 Compilation from `fun`

We define the compilation function $\mathcal{C}$, with type `Expr a`$-\ >$ `[Instr]`:

$$
\begin{aligned}
\mathcal{C}(n) &= \texttt{LDC}\ n \\
\mathcal{C}(\underline{n}) &= \texttt{LD}\ n \\
\mathcal{C}(\lambda\ \_ \rightarrow \texttt{M}) &= \texttt{CLO}\ [\mathcal{C}(\texttt{M}); \texttt{RTN}]) \\
\mathcal{C}(\texttt{M}\ \texttt{N}) &= \mathcal{C}(\texttt{M});\ \mathcal{C}(\texttt{N});\ \texttt{AP} \\
\mathcal{C}(\texttt{let}\ \_ = \texttt{M}\ \texttt{in}\ \texttt{N}) &= \mathcal{C}((\lambda\ \_ \rightarrow \texttt{N})\ \texttt{M}) \\
\mathcal{C}(\texttt{M} \circ \texttt{N}) &= \mathcal{C}(\texttt{M});\ \mathcal{C}(\texttt{N});\ \texttt{bopToInstr}(\circ) \\
\mathcal{C}(\texttt{if}\ \texttt{B}\ \texttt{is}\ \texttt{0}\ \texttt{then}\ \texttt{M}\ \texttt{else}\ \texttt{N}) &= \mathcal{C}(\texttt{B});\ \texttt{IF}\ [\mathcal{C}(\texttt{M}); \texttt{JOIN}]\ [\mathcal{C}(\texttt{N}); \texttt{JOIN}] \\
\mathcal{C}(\texttt{fix}\ (\lambda\ \_ \rightarrow \lambda\ \_ \rightarrow \texttt{e})) &= \texttt{FIX}\ [\mathcal{C}(\texttt{e}); \texttt{RTN}]
\end{aligned}
$$

Where `boptoInstr` is a function that transforms a binary operation $\circ$, of the form $+, -, *$, into its associated instruction, and the underline denotes the De Bruijn index of a variable.
We also don't care about the names the user has given to variables, as they get substituted by their De Bruijn indexes by the parser. The language syntax is shown here instead of the AST representation for readability.

3.5 Interpreting Instructions

Now that we have the abstract machine instructions, all that is left is to run them!

The **Store** transitions are not shown here, but its easy to infer its transitions and use:

1. When a function is created (using instructions like CLO or FIX), a closure is built and saved in the store at a unique address (using the `Addr` value). This stored closure can later be retrieved during function application;
2. In the case of recursive functions, we store the function's own address inside the environment so that it can refer to itself;

Before				**After**			
Stack	**Env**	**Code**	**Dump**	**Stack**	**Env**	**Code**	**Dump**
s	e	LD $i : c$	d	$e[i] : s$	e	c	d
s	e	LDC $k : c$	d	VInt $k : s$	e	c	d
VInt v_2 : VInt $v_1 : s$	e	ADD : c	d	VInt $(v_1 + v_2) : s$	e	c	d
VInt v_2 : VInt $v_1 : s$	e	SUB : c	d	VInt $(v_1 - v_2) : s$	e	c	d
VInt v_2 : VInt $v_1 : s$	e	MUL : c	d	VInt $(v_1 * v_2) : s$	e	c	d
s	e	CLO $c' : c$	d	Addr $a : s$	e	c	d
s	e	FIX $c' : c$	d	Addr $a : s$	e	c	d
v : Addr $a : s$	e	AP : c	d	$[\,]$	$v : e'$	c'	$(s, e, c) : d$
$v : s$	e	RTN : c	$(s', e', c') : d$	$v : s'$	e'	c'	d
VInt $0 : s$	e	IF $c_1\ c_2 : c$	d	s	e	c_1	$([\,], [\,], c) : d$
VInt $(n \neq 0) : s$	e	IF $c_1\ c_2 : c$	d	s	e	c_2	$([\,], [\,], c) : d$
s	e	JOIN : c	$(s', e', c') : d$	s	e	c'	d
$b : s$	e	LET : c	d	s	$b : e$	c	d
s	e	HALT : c	d	s	e	$[\,]$	d

Table 1: SECD Machine Transitions

3. Finally, for applications, one must retrieve the closure that is under the `Addr` that was popped from the stack. The closure's code is denoted as c' and the environment as e' in table 1.

4 Optimizations

4.1 Optimizing `let` declarations

Recall that the original compilation for the `let` declarations was as follows:

$$\mathcal{C}(\texttt{let \_} = \texttt{M in N}) = \mathcal{C}((\lambda \ \_ \rightarrow \texttt{N}) \ \texttt{M})$$

This creates an extra closure from the new application, which is not as efficient as it could be. In fact, if we add a new instruction:

```
data Instr
  = LD Int -- loads the variable at the specified index
  ...
  | LET -- pops the top of the stack and prepends it to the environment
  ...
```

We can compile a `let` simply as:

$$\mathcal{C}(\texttt{let \_} = \texttt{M in N}) = \mathcal{C}(M);\ LET;\ \mathcal{C}(N)$$

And the interpreter can be extended as:

Before				**After**			
Stack	**Env**	**Code**	**Dump**	**Stack**	**Env**	**Code**	**Dump**
$v:\ s$	e	LET : c	d	s	$v : e$	c	d

Table 2: LET Transition

It is easy to see that now the closest binded De Bruijn indexed variable, will be whatever `M` evaluates to. This models the behaviour we want for `let`!

4.2 Modern SECD: Combining the Stack and the Dump

You might have noticed that, when we push to the Dump, we always add the rest of the code left to run, and its environment. Doesn't that sound like a closure? Well, as it turns out, **it is**! Let's implement the Dumpless, Storeless SECD machine, commonly called the CES machine.

4.2.1 A Simpler Version First

First things first, let's extend the `Val` data type to contain closures:

```
data Val
  = VInt Int
  | VClos ([Instr], Env)
```

We also don't need `Addr` anymore, which was only used to access the `Store`, because we will now be keeping all closures on the stack.

Our compile function stays the same for now, the magic happens while interpreting!

You might have noticed that we are missing the fixed point operation. Lets tackle this hurdle.

Before			**After**		
Code	**Env**	**Stack**	**Code**	**Env**	**Stack**
LD n ; c	e	s	c	e	$e(n) : s$
LDC k : c	e	s	c	e	$k : s$
ADD : c	e	VInt v_2 : VInt $v_1 : s$	c	e	$(v_1 + v_2) : s$
SUB : c	e	VInt v_2 : VInt $v_1 : s$	c	e	$(v_1 - v_2) : s$
MUL : c	e	VInt v_2 : VInt $v_1 : s$	c	e	$(v_1 * v_2) : s$
CLO c' : c	e	s	c	e	VClos(c', e) : s
AP : c	e	v : VClos(c', e') : s	c'	$v : e'$	VClos(c, e) : s
RTN : c	e	v : VClos(c', e') : s	c'	e'	$v : s$
IF c_0 c_1 : c	e	VInt $0 : s$	c_0	e	Clos(c, e) : s
IF c_0 c_1 : c	e	VInt $(n \neq 0) : s$	c_1	e	Clos(c, e) : s
LET : c	e	v : s	c	$v : e$	s
HALT : c	e	v : s	[ ]	e	s

Table 3: CES Machine Transitions, no FIX

4.2.2 Adding the Fixed Point

Instead of just the one case where our fixed point absolutely needed at least two abstractions inside (the recursive function itself, and a first argument), we can also handle the case where it only has one (just the recursive function). Let's call this case the *control* case.

We can add a new constructor to our instruction set:

```
data Instr
  = LD Int -- loads the variable at the specified index
  ...
  | FIX [Instr] -- loads a recursive function
  | FIXC [Instr] -- loads a recursive function (control case)
  ...
```

And we should extend our `Val` data type to accommodate our new closures:

```
data Val
  = VInt Int
  | VClos ([Instr], Env)
  | VFixClos ([Instr], Env) -- recursive closure
  | VFixCClos ([Instr], Env) -- recursive closure (control case)
```

We are now ready to both compile our new instructions, and interpret them!

The compilation cases are:

$$\mathcal{C}(\texttt{fix}\ (\lambda\ \_ \to \lambda\ \_ \to \texttt{e})) = \texttt{FIX}\ [\mathcal{C}(\texttt{e});\ \texttt{RTN}]$$
$$\mathcal{C}(\texttt{fix}\ (\lambda\ \_ \to \texttt{e})) = \texttt{FIXC}\ [\mathcal{C}(\texttt{e});\ \texttt{RTN}]$$

And, to interpret:

In short, the application for a fixed point not only applies the code, but also inserts the fixed point below the argument of the application (if there is one) in the environment, so that a recursive call can be executed correctly.

Before			**After**		
Code	**Env**	**Stack**	**Code**	**Env**	**Stack**
FIX c' : c	e	s	c	e	VFixClos(c', e) : s
AP : c	e	v : VFixClos(c', e') : s	c'	v : VFixClos(c', e') : e'	VClos(c, e) : s
FIXC c' : c	e	s	c	e	VFixCClos(c', e) : s
AP : c	e	v : VFixCClos(c', e') : s	c'	VFixClos(c', e') : e'	VClos(c, e) : s

Table 4: CES Machine Transitions for FIX

4.3 Tail Call Optimization

Now that we have successfuly eliminated the need for a Dump and a Store, doing tail call optimization is easy.

But what even are tail calls? Consider the following:

$$
\begin{aligned}
f &= \lambda x \to \ldots\ g\ (x) \ldots \\
g &= \lambda y \to h\ (...) \\
h &= \lambda z \to \ldots
\end{aligned}
$$

The call from g to h is a **tail call**: when h returns, g has nothing more to compute, it just returns immediately to f.

If we analyze the instructions this sequence would create, we would see that the code for g is of the form [...; `AP`; `RTN`]. This will create a closure with only `RTN` inside, consuming stack space.

It may seem that this stack space is unimportant at first, but consider a recursive function like the following one:

$$
\begin{aligned}
&\text{let fact} = \text{fix fact } (\lambda f \to (\lambda n \to (\lambda acc \to \\
&\qquad \text{if } n \text{ is } 0 \text{ then } acc \text{ else } f\ (n-1)\ (acc * n) \\
&))) \\
&\text{in fact}\ \ 42\ 1
\end{aligned}
$$

The recursive call to f is in tail position. If we don't eliminate the tail call, this code will run with $\mathcal{O}(n)$ stack space, which risks a stack overflow. If we do apply tail call optimization, it runs in $\mathcal{O}(1)$ stack space as expected.

So what is the trick? First, we need a new instruction specifically for tail applications. Then, we should split the compilation into two mutually recursive functions, $\mathcal{C}$ and $\mathcal{T}$, for normal terms and terms in tail call position respectively.

We also need a new instruction to signal the end of a `let` declaration:

```
data Instr
  = LD Int -- loads the variable at the specified index
   ...
  | TAP -- tail application
  | ENDLET
   ...
```

Now $\mathcal{C}$ and $\mathcal{T}$:

$$\begin{aligned}
\mathcal{C}(n) &= \texttt{LDC}\ n \\
\mathcal{C}(\underline{n}) &= \texttt{LD}\ n \\
\mathcal{C}(\lambda\ _ \to \texttt{M}) &= \texttt{CLO}\ (\mathcal{T}(\texttt{M})) \\
\mathcal{C}(\texttt{M N}) &= \mathcal{C}(\texttt{M});\ \mathcal{C}(\texttt{N});\ \texttt{AP} \\
\mathcal{C}(\texttt{let}\ _ = \texttt{M in N}) &= \mathcal{C}(\texttt{M});\ \texttt{LET};\ \mathcal{C}(\texttt{N});\ \texttt{ENDLET} \\
\mathcal{C}(\texttt{M} \circ \texttt{N}) &= \mathcal{C}(\texttt{M});\ \mathcal{C}(\texttt{N});\ \texttt{bopToInstr}(\circ) \\
\mathcal{C}(\texttt{if B is 0 then M else N}) &= \mathcal{C}(\texttt{B});\ \texttt{IF}\ \ (\mathcal{T}(\texttt{M}))\ \ (\mathcal{T}(\texttt{N})) \\
\mathcal{C}(\texttt{fix}\ (\lambda\ _ \to \lambda\ _ \to \texttt{e})) &= \texttt{FIX}\ \ (\mathcal{T}(\texttt{e})) \\
\mathcal{C}(\texttt{fix}\ (\lambda\ _ \to \texttt{e})) &= \texttt{FIXC}\ \ (\mathcal{T}(\texttt{e}))
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}(\texttt{let}\ _ = \texttt{M in N}) &= \mathcal{C}(\texttt{M});\ \texttt{LET};\ \mathcal{T}(\texttt{N}) \\
\mathcal{T}(\texttt{M N}) &= \mathcal{C}(\texttt{M}); \mathcal{C}(\texttt{N}); \texttt{TAP} \\
\mathcal{T}(a) &= \mathcal{C}(a); \texttt{RTN}
\end{aligned}$$

Now for the semantics of `TAP`, it is very simple. They are a mirror of the `AP` rules, except that it doesn't bother to push a new closure onto the stack.

Before			**After**		
Code	**Env**	**Stack**	**Code**	**Env**	**Stack**
TAP : c	e	v : Clos(c', e') : s	c'	v : e'	s
TAP : c	e	v : VFixClos(c', e') : s	c'	v : VFixClos(c', e') : e'	s
TAP : c	e	v : VFixCClos(c', e') : s	c'	VFixClos(c', e') : e'	s

Table 5: TAP Machine Transitions

5 Extensions

5.1 Tuples

Tuples are an easy addition to our language.

We can extend our GADT to support tuples easily, and lets assume `fun` has been extended to parse tuples as well.

```haskell
data Expr a where
  ...
  Tup :: Expr a -> Expr b -> Expr (a, b)
```

Tuples are akin to values. They can be passed as arguments and returned from functions, se we will need to keep them in our stack. For that reason, we shall extend `Val` with a contructor for them.

```haskell
data Val
  ...
  | VTuple (Val, Val)
```

We also need something that tells the abstract machine that it should build this value, that it should glue these two arbitrary things together. We need a new instruction and a new compilation rule for that.

```haskell
data Instr
  ...
  | TUP
  ...
```

$$\mathcal{C}((M, N)) = \mathcal{C}(M);\ \mathcal{C}(N);\ \texttt{TUP}$$

Now that we have a way to contruct tuples, we also need a way to destruct them. We can extend our language with a "unary operation" construct, and encode `fst` and `snd` with it. We also need the corresponding instructions such that the interpreter can run them.

```haskell
data Expr a where
  ...
  Op :: OpE -> Expr a -> Expr a
  ...

data OpE
  = Fst
  | Snd

data Instr
  ...
  | FST
  | SND
  ...
```

We are pretty much done! We just need to define how the interpreter will handle the new cases, which are pretty straight-forward:

Before			**After**		
Code	**Env**	**Stack**	**Code**	**Env**	**Stack**
TUP : c	e	$v_2 : v_1 : s$	c	e	$\text{VTuple}(v_1, v_2) : s$
FST : c	e	$\text{VTuple}(v_1, v_2) : s$	c	$v : e$	$v_1 : s$
SND : c	e	$\text{VTuple}(v_1, v_2) : s$	c	$v : e$	$v_2 : s$

Table 6: TAP Machine Transitions

5.2 Simple Type-Inference

6 Conclusions

The code is online at github.com/zazedd/secd

References

[1] Mike Bowker and Phil Williams. Helsinki and west european security. *International Affairs*, 61(4):607–618, 1985.