SECD Machine - Optimizations and Extensions

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

Contents

1 Introduction

This paper explores the SECD Machine, a simple yet powerful abstract machine designed to evaluate expressions in functional programming languages. We start by defining its core data structures and instruction set, then break down the compilation and interpretation processes. From there, we introduce several key optimizations — including improved handling of let declarations, combining stack and dump structures, and tail call optimization. Finally, we extend the machine to support new features like tuples and basic variants.

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:

⟨*expr*⟩ ::= ⟨*ident*⟩
| ⟨*int*⟩
| 'λ' ⟨*ident*⟩ '`->`' ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*expr*⟩
| '`let`' ⟨*ident*⟩ '`=`' ⟨*expr*⟩ '`in`' ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*bop*⟩ ⟨*expr*⟩
| '`if`' ⟨*expr*⟩ '`is 0 then`' ⟨*expr*⟩ '`else`' ⟨*expr*⟩
| '`fix`' ⟨*expr*⟩

⟨*bop*⟩ ::= '`+`'
| '`*`'
| '`-`'

⟨*int*⟩ ::= (integer literals, e.g., 0, 1, 2, ...)

⟨*ident*⟩ ::= (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:

```
1  data Expr a where
2    Var :: Int -> Expr a
3    Int :: Int -> Expr Int
4    Abs :: Expr b -> Expr (a -> b)
5    App :: Expr (a -> b) -> Expr a -> Expr b
6    Let :: Expr a -> Expr b -> Expr b
7    Bop :: BopE -> Expr Int -> Expr Int -> Expr Int
8    IfZ :: Expr Int -> Expr a -> Expr a -> Expr a
9    FixP :: Expr (a -> a) -> Expr a
10
11 data BopE
12   = Add
13   | Mul
14   | 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:

```
1 data Val
2   = VInt Int
3   | 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]`:

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

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 in table 1, 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:

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

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:

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

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:

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

Table 4 outlines the interpreting steps.

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 AP : c	e e	s v : VFixClos(c', e') : s	c c'	e v : VFixClos(c', e') : e'	VFixClos(c', e) : s VClos(c, e) : s
FIXC c' : c AP : c	e e	s v : VFixCClos(c', e') : s	c c'	e VFixClos(c', e') : e'	VFixCClos(c', e) : s 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:

$$f = \lambda x \to \ldots\ g\ (x) \ldots$$
$$g = \lambda y \to h\ (...)$$
$$h = \lambda z \to \ldots$$

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:

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

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:

```
1 data Instr
2   = LD Int -- loads the variable at the specified index
3   ...
4   | TAP -- tail application
5   | ENDLET
6   ...
```

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

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

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

Now for the semantics of `TAP`, it is very simple. They are a mirror of the `AP` rules, except that they don'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

Let's compare the difference between our tail optimized compiler, and our last compiler definition. Specifically, let's look at the number of items on the stack for an evaluation of the `fact 42 1` term above:

Step #	Non-Tail	Tail
...	...	...
552	86	45
553	87	46
554	86	45
555	85	43
556	86	44
557	86	44
558	87	45
...	...	...
603	42	1
...	...	0
645	1	0

Table 6: Stack Sizes for Non-Tail and Tail Calls

You can see how this scales exponentially.

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.

```
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, so we will need to keep them in our stack. For that reason, we shall extend `Val` with a constructor for them.

```
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...

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

And a new compilation rule.

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

Now that we have a way to construct 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.

```
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	VTuple$(v_1, v_2) : s$
FST : c	e	VTuple$(v_1, v_2) : s$	c	$v : e$	$v_1 : s$
SND : c	e	VTuple$(v_1, v_2) : s$	c	$v : e$	$v_2 : s$

Table 7: Tuple Machine Transitions

5.2 Untyped Variants

Variants are a type of data structure that can represent one of several possible values, each tagged with a unique constructor. We should be able to match on each constructor individually - this is the destructor of variants. They are also akin to values, just like tuples as before.

A way to encode variants in our language would be to extend the AST with the following:

```
data Expr a where
  ...
  Variant :: (String, Expr a) -> Expr a -- (name, expression) creates a variant.
  Match :: (Expr a, [(String, Expr a)]) -> Expr a -- match the input with a list of constructors;
                                                  -- "Var 0" will contain the inner expression.
```

As before, we need some way to tell the abstract machine to **construct** the variant, and another to start the **destruction** process. Let's extend our instruction set and value data type with the required elements:

```
data Val
  ...
  | VVariant String Val

data Instr
  ...
  | VARIANT String
  | MATCH [(String, [Instr])]
  ...
```

Compilation is a bit harder for the match case, we have to use some list magic and compile the inner expressions of the branches.

$$\mathcal{C}(\texttt{Variant (Name, E)}) = \mathcal{C}(E);\ \texttt{VARIANT Name}$$

$$\mathcal{C}(\texttt{match E with branches}) = \mathcal{C}(E);\ \texttt{MATCH branches'}$$

$$\texttt{where branches'} = \mathit{map}\ (\lambda(\texttt{name},\ \texttt{branch}) \to (\texttt{name},\ \mathcal{C}(\texttt{branch})))\ \texttt{branches}$$

Now, the fun part! Interpreting is quite easy, the interesting case is for the `match` expression:

The operator !! is the lookup operation for lists. If we find the name of the constructor we are matching for in the association list, then we execute the code relevant to that branch. If we

Before			After		
Code	**Env**	**Stack**	**Code**	**Env**	**Stack**
VARIANT name : c	e	$v : s$	c	e	VVariant$(name, v) : s$
MATCH branches : c	e	VVariant $name, v : s$	branches !! name : c	$v : e$	s

Table 8: Variant Machine Transitions

don't find a match, then we should exit with an error, warning the user that the pattern-match they wrote was not extensive.

We can now encode lists, peano numbers, and a whole lot more in our language. The combination of tuples and variants is extremely expressive!

As this is quite a bit abstract, I urge you to check the tests in CITE APPENDIX for a better intuition.

5.2.1 Why both Tuples and Variants

While it is true that we can encode tuples using variants, I personally found that having an abstraction specifically for tuples in our language is a lot nicer than having to write something like this to encode a pair:

```
1 let pair = Variant "Pair" (Variant "First" (Int 3), Variant "Second" (Int 5))
```

As it stands, to write functions that operate on this pair, you would have to write two match expressions. This is just a bit cumbersome, and for the purposes of this paper, it was also a good opportunity to show how easy tuples are to implement.

5.2.2 Encoding Records with Variants

As a side note, variants can be used to implement records, they can be seen as the opposite of one another:

- a variant is this field *or* that field *or* that field;
- a record is this field *and* that field *and* that field.

So, if we "invert" a variant, we should get something that behaves like a record.

We can encode records using variants like so (CITE Principles of Programming Languages):

$$\{l_1 = e_1; \ldots; l_n = e_n\} = (\lambda k_1 \to \ldots \lambda k_n \to \lambda s \to \text{match } s \text{ with } \begin{cases} 'l_1(x) \to k_1 \\ \quad \vdots \qquad\qquad \\ 'l_n(x) \to k_n \end{cases})(e_1) \ldots (e_n)$$

$$e.l_k = e\ 'l_k(0)$$

A bit complicated, but it works! See (CITE Principles of Programming Languages) for further explanation.

6 Conclusions

The code is online at github.com/zazedd/secd. We've outlined the structure and operation of the SECD Machine, from its data handling and instruction set to its compilation and interpretation processes. By implementing optimizations like streamlined let declarations, stack-dump unification, and tail call optimization, we improve both performance and memory efficiency. Additionally, extensions such as tuple support and untyped variants enhance the machine's versatility. These improvements demonstrate how the SECD Machine can evolve to support more complex language features while maintaining its core simplicity.

References

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