WIKIPEDIA

SECD machine

The **SECD machine** is a highly influential (*See: #Landin's contribution*) virtual machine and abstract machine intended as a target for functional programming language compilers. The letters stand for **S**tack, **E**nvironment, **C**ontrol, **D**ump, the internal registers of the machine. These registers point to linked lists in memory.

The machine was the first to be specifically designed to evaluate lambda calculus expressions. It was originally described by Peter J. Landin as part of his ISWIM programming language definition in 1963. The description published by Landin was fairly abstract, and left many implementation choices open (like an operational semantics). Hence the SECD machine is often presented in a more detailed form, such as Peter Henderson's Lispkit Lisp compiler, which has been distributed since 1980. Since then it has been used as the target for several other experimental compilers.

In 1989 researchers at the University of Calgary worked on a hardware implementation of the machine.[1]

Contents

Landin's contribution

D. A. Turner (2012) [2] points out that *The Revised Report on Algol 60* (Naur 1963) specifies a procedure call by a copying rule which avoids variable capture with a systematic change of identifiers. This method works in the Algol 60 implementation, but in a functional programming language where functions are first-class citizens, a free variable on a call stack might be dereferenced in error.

Turner notes that Landin solved this with his SECD machine, in which a function is represented by a closure in the heap instead.[2]

Informal description

When evaluation of an expression begins, the expression is loaded as the only element of **C**. The environment **E**, stack **S** and dump **D** begin empty.

During evaluation of **C** it is converted to reverse Polish notation (RPN) with **ap** (for apply) being the only operator. For example, the expression F (G X) (a single list element) is changed to the list X:G:**ap**:F:**ap**.

Evaluation of **C** proceeds similarly to other RPN expressions. If the first item in **C** is a value, it is pushed onto the stack **S**. More exactly, if the item is an identifier, the value pushed onto the stack will be the binding for that identifier in the current environment **E**. If the item is an abstraction, a closure is constructed to preserve the bindings of its free variables (which are in **E**), and it is this closure which is pushed onto the stack.

If the item is **ap**, two values are popped off the stack and the application done (first applied to second). If the result of the application is a value, it is pushed onto the stack.

If the application is of an abstraction to a value, however, it will result in a lambda calculus expression which may itself be an application (rather than a value), and so cannot be pushed onto the stack. In this case, the current contents of **S**, **E**, and **C** are pushed onto **D** (which is a stack of these triples), **S** is reinitialised to empty, and **C** is reinitialised to the application result with **E** containing the environment for the free variables of this expression, augmented with the binding that resulted from the application. Evaluation then proceeds as above.

Completed evaluation is indicated by **C** being empty, in which case the result will be on the stack **S**. The last saved evaluation state on **D** is then popped, and the result of the completed evaluation is pushed onto the stack contents restored from **D**. Evaluation of the restored state then continues as above.

If **C** and **D** are both empty, overall evaluation has completed with the result on the stack **S**.

Registers and memory

The SECD machine is stack-based. Functions take their arguments from the stack. The arguments to built-in instructions are encoded immediately after them in the instruction stream.

Like all internal data-structures, the stack is a list, with the **S** register pointing at the list's *head* or beginning. Due to the list structure, the stack need not be a continuous block of memory, so stack space is available as long as there is a single free memory cell. Even when all cells have been used, garbage collection may yield additional free memory. Obviously, specific implementations of the SECD structure can implement the stack as a canonical stack structure, so improving the overall efficiency of the virtual machine, provided that a strict bound be put on the dimension of the stack.

The **C** register points at the head of the code or instruction list that will be evaluated. Once the instruction there has been executed, the **C** is pointed at the next instruction in the list—it is similar to an *instruction pointer* (or program counter) in conventional machines, except that subsequent instructions are always specified during execution and are not by default contained in subsequent memory locations, as it is the case with the conventional machines.

The current variable environment is managed by the **E** register, which points at a list of lists. Each individual list represents one environment level: the parameters of the current function are in the head of the list, variables that are free in the current function, but bound by a surrounding function, are in other elements of **E**.

The dump, at whose head the **D** register points, is used as temporary storage for values of the other registers, for example during function calls. It can be likened to the return stack of other machines.

The memory organization of the SECD machine is similar to the model used by most functional language interpreters:

a number of memory cells, each of which can hold either an *atom* (a simple value, for example *13*), or represent an empty or non-empty list. In the latter case, the cell holds two pointers to other cells, one representing the first element, the other representing the list except for the first element. The two pointers are traditionally named *car* and *cdr* respectively—but the more modern terms *head* and *tail* are often used instead. The different types of values that a cell can hold are distinguished by a *tag*. Often different types of atoms (integers, strings, etc.) are distinguished as well.

So a list holding the numbers *1*, *2*, and *3*, usually written as "(1 2 3)", could be represented as follows:

```
Address   Tag       Content (value for integers, car & cdr for lists)

     9 [ integer |     2 ]
     8 [ integer |     3 ]
     7 [ list    | 8 | 0 ]
     6 [ list    | 9 | 7 ]
     ...
     2 [ list    | 1 | 6 ]
     1 [ integer |     1 ]
     0 [ nil             ]
```

The memory cells 3 to 5 do not belong to our list, the cells of which can be distributed randomly over the memory. Cell 2 is the head of the list, it points to cell 1 which holds the first element's value, and the list containing only *2* and *3* (beginning at cell 6). Cell 6 points at a cell holding 2 and at cell 7, which represents the list containing only *3*. It does so by pointing at cell 8 containing the value *3*, and pointing at an empty list (*nil*) as cdr. In the SECD machine, cell 0 always implicitly represents the empty list, so no special tag value is needed to signal an empty list (everything needing that can simply point to cell 0).

The principle that the cdr in a list cell must point at another list is just a convention. If both car and cdr point at atoms, that will yield a pair, usually written like "(1 . 2)"

Instructions

- **nil** pushes a nil pointer onto the stack
- **ldc** pushes a constant argument onto the stack
- **ld** pushes the value of a variable onto the stack. The variable is indicated by the argument, a pair. The pair's car specifies the level, the cdr the position. So "(1 . 3)" gives the current function's (level 1) third parameter.
- **sel** expects two list arguments, and pops a value from the stack. The first list is executed if the popped value was non-nil, the second list otherwise. Before one of these list pointers is made the new **C**, a pointer to the instruction following **sel** is saved on the dump.
- **join** pops a list reference from the dump and makes this the new value of **C**. This instruction occurs at the end of both alternatives of a **sel**.
- **ldf** takes one list argument representing a function. It constructs a closure (a pair containing the function and the current environment) and pushes that onto the stack.
- **ap** pops a closure and a list of parameter values from the stack. The closure is applied to the parameters by installing its environment as the current one, pushing the parameter list in front of that, clearing the stack, and setting **C** to the closure's function pointer. The previous values of **S**, **E**, and the next value of **C** are saved on the dump.
- **ret** pops one return value from the stack, restores **S**, **E**, and **C** from the dump, and pushes the return value onto the now-current stack.
- **dum** pushes a "dummy", an empty list, in front of the environment list.
- **rap** works like **ap**, only that it replaces an occurrence of a dummy environment with the current one, thus making recursive functions possible

A number of additional instructions for basic functions like car, cdr, list construction, integer addition, I/O, etc. exist. They all take any necessary parameters from the stack.

See also

- Krivine machine

References

1. A paper on the design, SECD: DESIGN ISSUES (http://hdl.handle.net/1880/46590) is available.
2. D. A. Turner "Some History of Functional Programming Languages" in an invited lecture **TFP12**, St Andrews University, 12 June 2012. See the section on Algol 60 (https://www.cs.kent.ac.uk/people/staff/dat/tfp12/tfp12.pdf)

Further reading

- Danvy, Olivier. *A Rational Deconstruction of Landin's SECD Machine* (http://www.brics.dk/RS/03/33/). BRICS research report RS-04-30, 2004. ISSN 0909-0878
- Field, Anthony J. Field and Peter G. Harrison. 1988 *Functional Programming*. Addison-Wesley. ISBN 0-201-19249-7
- Graham, Brian T. 1992 "The SECD Microprocessor: A Verification Case Study". Springer. ISBN 0-7923-9245-0
- Henderson, Peter. 1980 *Functional Programming: Application and Implementation*. Prentice Hall. ISBN 0-13-331579-7
- Kogge, Peter M. *The Architecture of Symbolic Computers*. ISBN 0-07-035596-7
- Landin, P. J. (January 1964). "The Mechanical Evaluation of Expressions". *Comput. J.* **6** (4): 308–320. doi:10.1093/comjnl/6.4.308 (https://doi.org/10.1093%2Fcomjnl%2F6.4.308).
- Landin, P. J. (March 1966). "The next 700 programming languages" (http://fsl.cs.uiuc.edu/images/e/ef/P157-landin.pdf) (PDF). *Comm. ACM*. **9** (3): 157–166. doi:10.1145/365230.365257 (https://doi.org/10.1145%2F365230.365257).

External links

- SECD collection (http://skelet.ludost.net/sec/)
