Assignment 1
Language Implementation (CC4023)

Pedro Vasconcelos

March 21, 2024

1. The assignments consist of studying one of the topics below that and extending a prototype SECD machine compiler/interpreter.
2. The assignment is individual; plagarism (i.e. copy of assignments from other students or from the internet) will be penalized by loss of classification in all assignments in this course.
3. Your should base your extensions on the code provided in the Github repository `https://github.com/pbv/cc4023-il/`.
4. Students should present the work developed and submit a short report (4–6 pages A4) possibly including fragments of code and examples.
5. Extra points will be awarded for using Haskell build tools (cabal) and including a parser for the Fun language and command-line interface to test your assignment. I recommend using either Alex/Happy or the *Parsec* parser combinator library [7].
6. The assignments should be submitted until 8th April using Moodle. The presentations will be done in the week 8-12th April.

1 Optimizations for the SECD machine

Study and implement some optimizations of SECD machine code and implement them by extending the compiler and C interpreter presented in lectures. The techniques to consider are described in Section 7.8 of [1]:

1. Avoid un-necessary construction of closures for let-expressions;
2. Simplify conditional instructions in a tail call position, for example, sequences such as "SEL [. . . JOIN] [. . . JOIN], RTN";
3. Last call optimization, e.g. code sequences such as "AP, RTN";
4. Tail recursion optimization, i.e. avoid growing the stack when a recursive call occurs in a tail position.

5. Combine the *stack* and *dump* of the C implementation into a single stack.

Include some examples of effects of optimizations you implement.

2 Implement pairs and lists

Extend the compiler for the SECD machine with pairs and lists:

1. Extend the abstract syntax; in particular add constructors, projections for lists and pairs:

$$
\begin{array}{rcll}
e & ::= & \cdots & \\
 & | & (e_1, e_2) & \\
 & | & \textbf{fst}\ e & \\
 & | & \textbf{snd}\ e & \\
 & | & [\,] & \\
 & | & e_1 : e_2 & \\
 & | & \textbf{null}\ e & \text{— should yield 0 or 1} \\
 & | & \textbf{head}\ e & \\
 & | & \textbf{tail}\ e &
\end{array}
$$

2. Use Church encodings to implement the above extensions to the SECD machine. Note that **null** should give an integer rather than a (Church-encoded) boolean: **null** e should evaluate to 0 if the list is empty and 1 otherwise, so that you can use **ifzero** for branching.

Include some examples of common list functions that illustrate these extensions, e.g. length, append, zip, reverse, map.

3 Implement I/O and imperative effects

Extend the compiler and C interpreter for the SECD machine with primitives for I/O and imperative programming [3].

$$
\begin{array}{rcl}
e & ::= & \cdots \\
 & | & e_1\,\mathtt{;}\,e_2 \\
 & | & \textbf{get\_char} \\
 & | & \textbf{put\_char}\ e \\
 & | & \textbf{ref}\ e \\
 & | & \mathtt{!}e \\
 & | & e_1\,\mathtt{:=}\,e_2
\end{array}
$$

The expression **get_char** will perform a side-effect of reading the next character from *stdin*; its result is the character code. The expression **put_char** e will evaluate e and output its value (an integer) as character to *stdout*. We can use the semicolon to sequence I/O operations.

The constructs **ref**, `!` and `:=` are used to introduce *mutable references* (i.e. state). See Section 4.1 of [3] for an exposition.

4 Implement type inference

Implement the Damas-Milner type inference algorithm for the Fun language. Consider simple types τ with a single base type `int` e functions $\tau_1 \to \tau_2$. Quantified types are simple types universally qualified in one or more variables:

$$
\begin{array}{rcl}
 & & \text{simple types} \\
\tau & ::= & \mathbf{int} \;\mid\; \alpha \;\mid\; \tau_1 \to \tau_2 \\
 & & \\
 & & \text{quantified types} \\
\sigma & ::= & \tau \;\mid\; \forall \alpha.\, \sigma
\end{array}
$$

For more information on the type inference read [4, 5, 6].

5 Compile SECD code to C

Modify the second version of the Haskell SECD compiler presented in lectures to output C code instead of bytecode [9, 8]:

1. the SECD machine components and registers should be globals;
2. each block of SECD code should be translated to a single C function with no arguments;
3. each function returns the address of continuation, i.e. a function pointer.

The result should be a valid portable C file that you can compile with a C compiler; running the executable should execute the SECD code and print the final value on top of the stack (i.e. same behaviour as the bytecode interpreter).

References

[1] *The Architecture of Symbolic Computers*, Peter M. Kogge. 1991, McGraw-Hill International.

[2] *Functional Programming: Application and Implementation*, Peter Henderson. 1980, Prentice-Hall International.

[3] *Programming languages*, Mike Grant, Zachary Palmer and Scott Smith. `http://pl.cs.jhu.edu/pl/book/`

[4] *Types and Programming Languages*, Benjamin Pierce, The MIT Press, 2002. Chapter 22 (*Type reconstruction*).

[5] *Polymorphic type checking*, Peter Hancock. Chapter 8 of *The Implementation of Functional Programming Languages*, Simon Peyton Jones, 1987.

[6] *A type checker*, Peter Hancock. Chapter 9 of *The Implementation of Functional Programming Languages*, Simon Peyton Jones, 1987.

[7] The *Parsec* library documentation, `http://hackage.haskell.org/package/parsec`

[8] *Implementação de uma linguagem functional usando combinadores compactos*, Pedro Vasconcelos, Master thesis, Universidade do Porto, 1998. `http://www.dcc.fc.up.pt/~pbv/research/tese_msc.ps.gz`

[9] Implementing lazy functional language on stock hardware: The Spineless Tagless G-machine (version 2.5). Simon Peyton Jones, 1992. Chapter 6.2.