Compiling Arithmetic Expressions

Gert Smolka, Saarland University

December 5, 2017

We specify and verify a compiler from arithmetic expressions to programs for a stack machine. This popular case study provides a perfect start for a course on the theory of programming languages. We base everything on constructive type theory and provide an accompanying Coq development.

1 Expressions

We consider an inductive type Exp of simple arithmetic expressions:

$$e : \mathsf{Exp} ::= n \mid e_1 + e_2 \mid e_1 - e_2 \qquad (n : \mathsf{N})$$

We fix a semantics for expressions with a recursive function

$$\mathcal{E} : \mathsf{Exp} \to \mathsf{N}$$

defined as follows:

$$\begin{aligned} \mathcal{E}\, n &= n \\ \mathcal{E}\,(e_1 + e_2) &= \mathcal{E}\, e_1 + \mathcal{E}\, e_2 \\ \mathcal{E}\,(e_1 - e_2) &= \mathcal{E}\, e_1 - \mathcal{E}\, e_2 \end{aligned}$$

2 Machine Language

We also define a machine language with the following commands:

$$c : \mathsf{Com} ::= n \mid \mathsf{add} \mid \mathsf{sub} \qquad (n : \mathsf{N})$$

We fix a semantics for the machine language with a tail-recursive function

$$R : \mathsf{L}\,(\mathsf{Com}) \to \mathsf{L}\,(\mathsf{N}) \to \mathsf{L}\,(\mathsf{N})$$

executing programs (i.e., lists of commands) on stacks of numbers:

$$\begin{aligned} R\ \mathsf{nil}\ B &= B \\ R\ (n :: A)\ B &= R\ A\ (n :: B) \\ R\ (\mathsf{add} :: A)\ (n_1 :: n_2 :: B) &= R\ A\ (n_1 + n_2 :: B) \\ R\ (\mathsf{sub} :: A)\ (n_1 :: n_2 :: B) &= R\ A\ (n_1 - n_2 :: B) \\ R\ \_\ \_ &= \mathsf{nil} && \text{otherwise} \end{aligned}$$

The function R may be understood as a machine that executes a program on a stack of numbers. If there are not enough arguments on the stack for an operation, the machine returns the empty stack.

Exercise 1 Write a more informative version of R returning a stack option where $\emptyset$ models the case that the machine has crashed since there were not enough arguments on the stack. Prove that $R\ A\ (n :: B) = \lfloor C \rfloor$ implies $C \neq \mathsf{nil}$.

3 Compiler

We define a recursive function

$$\gamma : \mathsf{Exp} \to \mathsf{L}(\mathsf{Com})$$

compiling expressions to machine programs:

$$\begin{aligned} \gamma n &= [n] \\ \gamma(e_1 + e_2) &= \gamma e_2 +\!\!+ \gamma e_1 +\!\!+ [\mathsf{add}] \\ \gamma(e_1 - e_2) &= \gamma e_2 +\!\!+ \gamma e_1 +\!\!+ [\mathsf{sub}] \end{aligned}$$

We now would like to show the correctness of the compiler:

$$R\ (\gamma e)\ \mathsf{nil} = [\mathcal{E} e] \tag{1}$$

The equation says that the machine applied to the compilation of an expression and the empty value stack yields a singleton stack consisting of the value of the expression.

4 Correctness Proof

We will show equation (1) by induction on e. As is, the induction does not go through since recursive calls of R employ more general programs and more general value stacks. The induction goes through if we generalise equation (1) to more general programs and general value stacks.

Theorem 2 $R\ (\gamma e +\!\!+ A)\ B = R\ A\ (\mathcal{E}e :: B)$.

Proof By induction on e. The case for addition proceeds as follows:

$$\begin{array}{rll}
& R\ (\gamma(e_1 + e_2) +\!\!+ A)\ B & \\
= & R\ (\gamma e_2 +\!\!+ \gamma e_1 +\!\!+ [\mathsf{add}] +\!\!+ A)\ B & \text{definition } \gamma \\
= & R\ (\gamma e_1 +\!\!+ [\mathsf{add}] +\!\!+ A)\ (Ee_2 :: B) & \text{inductive hypothesis} \\
= & R\ ([\mathsf{add}] +\!\!+ A)\ (Ee_1 :: Ee_2 :: B) & \text{inductive hypothesis} \\
= & R\ A\ ((Ee_1 + Ee_2) :: B) & \text{definition } R \\
= & R\ A\ (E(e_1 + e_2) :: B) & \text{definition } E
\end{array}$$

The equational reasoning implicitly employs certain standard laws for $+\!\!+$ (associativity and computational laws). ■

Corollary 3 $R\ (\gamma e)\ \mathsf{nil} = [\mathcal{E}e]$.

Proof Follows with Theorem 2 with $A = B = \mathsf{nil}$. ■

Exercise 4 Compare the paper proof of the correctness theorem with the proof generated by the proof script in the accompanying Coq development.

Exercise 5 (Decompilation) Write a decompilation function $\delta : \mathsf{L}(\mathsf{Com}) \to \mathsf{O}(\mathsf{Exp})$ such that $\delta(\gamma e) = \lfloor e \rfloor$. Prove the correctness of your function.

5 Discussion

The semantics of the expressions and programs considered here is particularly simple since evaluation of expressions and execution of programs can be accounted for by structural recursion.

The use of recursive functions for semantic purposes in Coq's type theory is rather limited because of the confinement to structural recursion. This already shows if one consider a tail-recursive interpreter for expressions. Since such an interpretation function is not structurally recursive, one is forced to model it as a functional inductive predicate.

We represented expressions as abstract syntactic objects using an inductive type. Inductive types are the canonical representation of abstract syntactic objects. A concrete syntax for expressions would represent expressions as strings. While concrete syntax is important for the practical realisation of programming systems, it has no semantic relevance.

Compilation of expressions appears as first example in Chlipala's textbook [1], where it is also used to get the reader acquainted with Coq.

References

[1] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press, 2013.