DTU Compute

Lyngby 22-10-20, 21-10-2020

Michael R. Hansen

Ongoing exercise: Simple compiler

Compilers from high-level languages like Java, C#, F#, C++ ... typically target an *abstract machine* that can execute programs in a so-called *intermediate language*. Compilation from intermediate languages to target architectures with register machines like x86, ARM and MIPS are then done by other compilers. A compiler from from a high-level language typically comprises a part that compiles expressions to stack-machine code. An aspect that is typically included in a compilation from an intermediate language is register allocation.

This exercise introduces semantics, symbolic manipulation and compilation to stack-machine code in the context of a very simple expression language. It can be completed when the topic *finite trees* is been studied. The aims are:

- to exercise functional programming concepts on finite trees,
- to introduce techniques that are based on the treatment of programs as data, and
- to provide an appetizer for a following course: Computer Science Modelling.

The themes of the various parts are:

- Part 1 focuses on construction of simple abstract stack-based machine. The program programmring language for this this machine is called the are *target language*. This part can be solve immediately as it is not based on finite trees; "just" disjoint union.
- Part 2 focuses on abstract syntax and semantics of a simple expression language. This language is called the *source language*. This part exercises understanding of finite trees and recursion following the structure of the tree.
- Part 3 focuses on a compilation from source language to the target language. This part exercises recursion following the structure of finite trees on the basis of a fundamental concept: postfix form (or reverse polish notation) for expressions.
- Part 4 focuses on optimization addressed through simplifications/symbolic manipulations of source programs. This part requires a more complicated form of recursion compared to the previous parts.

Furthermore, correctness issues involving compilation and optimization are addressed by using property-based testing.

Part 1: A simple stack machine

We consider a simple abstract stack machine with instructions for addition, subtraction, change sign and absolute value, and an instruction for pushing an integer to the top of the stack.

The *instruction set* of this stack machine is modelled by the following F# `type`:

```
type Instruction = | ADD | SUB | SIGN | ABS | PUSH of int
```

A *stack* for this machine is a list of integers.

The *execution* of an instruction maps a stack to a new stack:

- The execution of `ADD` with stack $\boxed{a\ b\ c\ \cdots}$ yields a new stack: $\boxed{(b+a)\ c\ \cdots}$, where the top two elements a and b on the stack have been replaced by the single element $(b+a)$. Similarly with regard to the instruction `SUB`.
- The execution of `SIGN` with stack $\boxed{a\ b\ \cdots}$ yields a new stack: $\boxed{-a\ b\ \cdots}$, where the top element a on the stack has been replaced by the single element $-a$ Similarly with regard to the instruction `ABS`.
- The execution of `PUSH` r with the stack $\boxed{a\ b\ c\ \cdots}$ *pushes* r on top of the stack, that is, the new stack is: $\boxed{r\ a\ b\ c\ \cdots}$.

Declare a type `Stack` for representing the stack, and declare an F# function to interpret the execution of a single instruction:

```
intpInstr: Stack -> Instruction -> Stack
```

A *program* for the stack machine is a list of instructions $[i_1, i_2, \ldots, i_n]$. A program is *executed* by executing the instructions $i_1, i_2, \ldots, i_n$ one after the other, in that order, starting with an empty stack. The result of the execution is the top value of the stack when all instructions have been executed.

Declare an F# function to interpret the execution of a program:

```
exec: Instruction list -> int
```

Remark: This stack machine is kept to the minimum so that it just can calculate the value of expressions constructed from constants using the operations for addition, subtraction, change sign and absolute value. Instructions for branching, for example, are not needed. Furthermore, since there are not function calls, memory management instructions are avoided as well.

Part 2: Expressions: Syntax and semantics

You shall make a type declaration for *expressions* that are formed from a single variable (constructor `X`) and integer constants (constructor `C`) using operations for addition (constructor `Add`), subtraction (constructor `Sub`), change sign (constructor `Minus`) and absolute value (constructor `Abs`).

Declare a type `Exp` for expressions so that

- `X`, `C -2`, `C 7`
- `Abs X`, `Minus(C 7)`,
- `Add(Abs(Minus(C 7)), Sub(X, Minus(Add(C 2, X))))`

are six values of type `Exp`.

The type declaration for `Exp` defines a so-called *abstract syntax* for expressions. Since values of type `Exp` are trees, details that appear in concrete textual representations of expressions can be ignored, such as use of parenthesis and concrete textual representation of functions.

The *semantics* (or meaning) of expressions is given by a function:

$$\texttt{sem} : \texttt{Exp} \to \texttt{int} \to \texttt{int}$$

where the value of `sem` e x is the integer computed from e using that variable `X` has value x and the constructors `Add`, `Sub`, `Abs`, `Minus` correspond in an obvious way to operations on integers.

Part 3: Compilation to stack-machine code

You should now construct a compiler that transforms an expression into a list of instructions. Execution of the list instructions must give the same value as the semantics of the expressions.

The compilation function is a curried function in order to have a simple setting for handling the variable `X`:

```
compile:  Exp -> int -> Instruction list
```

where the value of the expression `compile` e x is a list of instructions *prg* satisfying

$$\texttt{exec}\ prg = \texttt{sem}\ e\ x$$

for every expression e and integer x.

The idea behind the compilation of an expression e is that the instruction list for e corresponds to a *postfix form* of e. In a postfix form of an arithmetical expression e each operator is preceded by its operands. For example:

$3 + 7$	has postfix form	$3\ 7\ +$
$4 - 5$	has postfix form	$4\ 5\ -$
$(3 + 7) + (4 - 5)$	has postfix form	$3\ 7\ +\ 4\ 5\ -\ +$

Observe that execution of `[PUSH 3; PUSH 7; ADD; PUSH 4; PUSH 5; SUB; ADD]` gives the value of the expression $(3 + 7) + (4 - 5)$.

Postfix form is also called *reverse polish notation* and has, for example, the property that any expression can be expressed without using parentheses.

Give a declaration of the `compile` function and use property-based testing to validate that it is in accordance with the semantics of expressions as described above.

Part 4: Optimization: Expression reductions

Optimizations are typically performed in all phases of a compiler. In this part we shall study optimization at the source-language level in the form of *reduction of expressions*.

In particular, we shall consider the below reductions that all are characterized by the fact that the number constructors in an expression is decreased. Other reductions may be considered; but we focus on the these ones.

Reductions where `Add` is the "outermost" constructor:

`Add`(`C` i, `C` j)	is reduced to	`C`$(i + j)$
`Add`(e, `C` 0)	is reduced to	e
`Add`(`C` 0, e)	is reduced to	e

Reductions where `Sub` is the "outermost" constructor:

`Sub`(`C` i, `C` j)	is reduced to	`C`$(i - j)$
`Sub`(e, `C` 0)	is reduced to	e
`Sub`(`C` 0, e)	is reduced to	`Minus` e

Reductions where `Minus` is the "outermost" constructor:

`Minus`(`C` i)	is reduced to	`C`$(-i)$
`Minus`(`Minus` e)	is reduced to	e

Reductions where `Abs` is the "outermost" constructor:

`Abs`(`C` i)	is reduced to	abs(i)
`Abs`(`Minus` e)	is reduced to	`Abs` e
`Abs`(`Abs` e)	is reduced to	`Abs` e

Declare a function `red: Exp -> Exp` that performs the above reductions and does it in a manner so that no further reduction is possible for the returned expression.

Use property-based testing to validate that your implementation of `red` is *sound* in the sense that

$$\texttt{sem}\ e\ x = \texttt{sem}(\texttt{red}\ e)\ x$$

for every expression e and integer x.

While it is not too difficult to do some reductions it is more complicated to make sure that the result cannot be reduced. Analyze your program to convince yourself that this indeed is the case.

Declare a function `reducible: Exp -> bool` that checks whether an expression can be reduced, that is, some reduction is possible somewhere in an expression.

Use property-based testing to validate that your implementation of `red` is *complete* in the sense that

$$\neg(\texttt{reducible}(\texttt{red}\ e))$$

for every expression e.