Feldspar implementation
Guest lecture, IntroFP, 2011

Emil Axelsson

October 5, 2011

Representing embedded languages

Haskell makes it very convenient to represent expressions as recursive data types. For example:

Arithmetic expressions

```
data Expr
    = Num Int
    | Add Expr Expr
    | Mul Expr Expr
```

Evaluation

```
eval :: Expr → Int
eval (Num n)      = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 ∗ eval e2
```

Embedded language API

```
num :: Int → Expr
num = Num

(<+>) :: Expr → Expr → Expr
(<+>) = Add

(<*>) :: Expr → Expr → Expr
(<*>) = Mul

eval :: Expr → Int
```

Example

```
prog = (num 10 <*> num 4) <+> (num 34 <+> num 6)
```

Example

```
prog = (num 10 <*> num 4) <+> (num 34 <+> num 6)
```

```
*Main> eval prog
80
```

Simple code generation

```
var v = "v" ⧺ show v

assign v expr = var v ⧺ " = " ⧺ expr ⧺ "\n"

matchOp (Add e1 e2) = ("+",e1,e2)
matchOp (Mul e1 e2) = ("*",e1,e2)

compile' :: Int → Expr → (String,Int)
compile' v (Num n) = (assign v (show n), v)
compile' v0 e      = (code1 ⧺ code2 ⧺ code3, v3)
  where
    (op,e1,e2)  = matchOp e
    (code1,v1)  = compile' v0     e1
    (code2,v2)  = compile' (v1+1) e2
    v3          = v2+1
    code3       = assign v3 (var v1 ⧺ op ⧺ var v2)

compile :: Expr → IO ()
compile = putStrLn . fst . compile' 0
```

Example

```
prog = (num 10 <*> num 4) <+> (num 34 <+> num 6)
```

```
*Main> compile prog
v0 = 10
v1 = 4
v2 = v0*v1
v3 = 34
v4 = 6
v5 = v3+v4
v6 = v2+v5
```

High-level sugar

The language API can be extended without changing the Expr type. Simple example, loop construct:

```
loop :: Int → (Expr → Expr) → (Expr → Expr)
loop 0 f = id
loop n f = f . loop (n−1) f
```

- Using Haskell to *generate* embedded programs.
- Advantage: Can give advanced constructs (like loop) to the user, while keeping the expressions simple.

High-level sugar

```
*Main> compile (loop 4 (<+>num 5) (num 0))
v0 = 0
v1 = 5
v2 = v0+v1
v3 = 5
v4 = v2+v3
v5 = 5
v6 = v4+v5
v7 = 5
v8 = v6+v7
```

Feldspar

- Same basic idea as the simple Expr language
- A recursive data type to represent programs
- An evaluator and a C code generator
- Lots of high-level sugar!
  - Most constructs are based on sugar
  - E.g. the Vector type (see previous presentation) only exists as sugar