CALCULATING A COMPILER

Graham Hutton and Nils Anders Danielsson
Background

Verifying a compiler for a simple language with exceptions (MPC 04).

Calculating an abstract machine that is correct by construction (TFP 05).
We show how to calculate a compiler for a simple language of arithmetic expressions;

Our new approach builds upon and refines earlier work by Wand and Danvy.

We make essential use of dependent types, and our work is formalised in Agda.
Arithmetic Expressions

Syntax:

\[
data \text{ Expr } = \text{ Val } \text{ Int } \mid \text{ Add } \text{ Expr } \text{ Expr}
\]

Semantics:

\[
\begin{align*}
\text{eval} & \quad :: \quad \text{Expr} \to \text{Int} \\
\text{eval} \ (\text{Val} \ n) & \quad = \quad n \\
\text{eval} \ (\text{Add} \ x \ y) & \quad = \quad \text{eval} \ x \ + \ \text{eval} \ y
\end{align*}
\]
Step 1 - Sequencing

Rewrite the semantics in combinatory form using a special purpose sequencing operator.

Raising a type to a power:

$$a^3 = a \rightarrow a \rightarrow a \rightarrow a$$

3 arguments
Sequencing:

\[(;); a^n \rightarrow a^{1+m} \rightarrow a^{n+m}\]

Example (n = 3, m = 2):

\[f ; g = g \circ f\]
We can now rewrite the semantics:

\[
\text{eval} :: \text{Expr} \rightarrow \text{Int}^0 \\
\text{eval} (\text{Val} \ n) = \text{return} \ n \\
\text{eval} (\text{Add} \ x \ y) = \text{eval} \ x ; (\text{eval} \ y ; \text{add})
\]

where

\[
\text{return} :: \text{Int} \rightarrow \text{Int}^0 \\
\text{return} \ n = n \\
\text{add} :: \text{Int}^2 \\
\text{add} = \lambda n \ m \rightarrow m + n
\]

Only uses the powers 0,1,2.
Step 2 - Continuations

Generalise the semantics to arbitrary powers by making the use of continuations explicit.

Definition:

A continuation is a function that is applied to the result of another computation.
Aim: define a new semantics

\[
\text{eval'} :: \text{Expr} \rightarrow \text{Int}^{1+m} \rightarrow \text{Int}^m
\]

such that

\[
\text{eval'} e c = \text{eval } e ; c
\]

and hence

\[
\text{eval } e = \text{eval'} e \text{ halt}
\]  
\[
\text{halt} = \lambda n \rightarrow n
\]
Case: e = Add x y

\[
\text{eval'} (\text{Add } x y) \ c \\
= \\
\text{eval} (\text{Add } x y) ; c \\
= \\
(\text{eval } x ; (\text{eval } y ; \text{add})) ; c \\
= \\
\text{eval } x ; (\text{eval } y ; (\text{add } ; c)) \\
= \\
\text{eval'} x (\text{eval } y ; (\text{add } ; c)) \\
= \\
\text{eval'} x (\text{eval'} y (\text{add } ; c))
\]
New semantics:

\[
\text{eval} :: \text{Expr} \rightarrow \text{Int}^{0} \\
\text{eval} e = \text{eval'} e \text{ halt}
\]

\[
\text{eval'} :: \text{Expr} \rightarrow \text{Int}^{1+m} \rightarrow \text{Int}^{m} \\
\text{eval'} (\text{Val } n) \ c = \text{return } n \ ; \ c \\
\text{eval'} (\text{Add } x \ y) \ c = \\
\text{eval'} x \ (\text{eval'} y (\text{add} ; c))
\]

The semantics now uses \textit{arbitrary powers}. 
**Step 3 - Defunctionalize**

Make the semantics *first-order* again, by applying the *defunctionalization* technique.

Basic idea:

Represent the functions of type \( \text{Int}^n \) we actually need using a datatype \( \text{Term} \ n \).
New semantics:

eval :: Expr → Term 0
eval e = eval' e Halt

eval' :: Expr → Term (1+m) → Term m
eval' (Val n) c = Return n c
eval' (Add x y) c =
eval' x (eval' y (Add c))

The semantics is now first-order again.
New datatype:

```
data Term n where
    Halt   :: Term 1
    Return :: Int → Term (1+n) → Term n
    Add    :: Term (1+n) → Term (2+n)
```

Interpretation:

```
exec :: Term n -> Int^n
exec Halt = halt
exec (Return n c) = return n ; exec c
exec (Add c) = add ; exec c
```
Example

\[
\text{Add} \ (\text{Val 1}) \ (\text{Add} \ (\text{Val 2}) \ (\text{Val 3}))
\]
Summary

- Purely calculational development of a compiler for simple arithmetic expressions;

- Use of dependent types arises naturally during the process to keep track of stack usage;

- Technique has also been used to calculate a compiler for a language with exceptions.
Ongoing and Further Work

- Using an explicit stack type;
- Other control structures;
- Relationship to other approaches;
- Further exploiting operads.