FUNCTIONAL PEARL

Typed Quote/Antiquote Or: Compile-time Parsing

RALF HINZE
Computing Laboratory, University of Oxford,
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
ralf.hinze@comlab.ox.ac.uk

1 Introduction

Haskell (Peyton Jones, 2003) is often used as a host language for embedding other languages. Typically, the abstract syntax of the guest language is defined by a collection of data type declarations; parsers and pretty-printers convert between the concrete syntax and its abstract representation. A quote/antiquote mechanism permits a tighter integration of the guest language into the host language by allowing one to use phrases in the guest language's *concrete syntax*.

For a simple example, assume that the abstract syntax of the guest language is given by the following data type of binary trees.

```
data Tree = Leaf | Fork Tree Tree
```

To dispense with the need for parentheses we choose prefix notation for the concrete syntax. The following interactive session illustrates the use of quotations.

```
Main⟩  « fork fork leaf leaf leaf »
Fork (Fork Leaf Leaf) Leaf
Main⟩  size (« fork fork leaf leaf leaf ») + 1
4
```

A *quotation* is delimited by guillemets (« and ») and consists of a phrase in concrete syntax, in our case, a prefix expression. The concrete syntax is a sequence of terminal symbols, written in typewriter font. A quotation evaluates to abstract syntax and can be freely mixed with ordinary Haskell expressions. In our example, a quotation yields a value of type *Tree* and may therefore serve as an argument to *size*, which computes the size of a tree.

Perhaps surprisingly, our quote mechanism guarantees that the guest-language phrase is well-formed: the malformed quotations « `fork` » and « `leaf leaf` » are both rejected by Haskell's type-checker. This is a big advantage over the use of strings, which often serve as an approximation to quotations.

The relationship between host and guest language also suggests a notion of antiquotation: the ability to splice a host-language expression into the middle of a guest-

language phrase. Continuing the example above, here is a session that demonstrates the use of anti-quotations:

```
Main⟩  « fork ` (full 2) leaf »
Fork (Fork (Fork Leaf Leaf) (Fork Leaf Leaf)) Leaf
Main⟩  let foo t = « fork ` t leaf »
Main⟩  foo (« fork leaf fork leaf leaf »)
Fork (Fork Leaf (Fork Leaf Leaf)) Leaf
```

An *anti-quotation* is written as a back-quote (`) followed by an atomic Haskell expression, for instance, an identifier or a parenthesised expression. The Haskell expression typically generates a piece of abstract syntax, for instance, in the first expression above, a fully balanced binary tree of depth 2.

A quote/antiquote mechanism usually requires an extension of the host language. The purpose of this pearl is to show that one can program such a mechanism within Haskell itself. The technique is based on Okasaki's flattening combinators (2002; 2003), which we shall review in the next section. To make the idea fly, I assume that we can use an arbitrary terminal symbol in `typewriter` font *as a Haskell identifier*. If you think that this assumption undermines the argument, then you should read the pearl as an exercise in compile-time parsing.

2 Background: the other CPS monad

To illustrate the basic idea consider a very simple example, which implements concrete syntax for the naturals.

```
Main⟩  (« | | | », « | | | | | » + 7)
(3, 12)
```

We have only one terminal symbol, the vertical bar, where a sequence of n bars represents the number n.

The succession of symbols « | | | » looks like a sequence of terminals enclosed in guillemets. But, of course, this is an illusion; the sequence is, in fact, a nested application of functions. If we take '«', '»', and '|' as aliases for *quote*, *endquote*, and *tick*, then « | | | » abbreviates the fully parenthesised expression (((*quote tick*) *tick*) *tick*) *endquote*. In what follows, we shall use '«' and *quote*, '»' and *endquote*, '`' and *antiquote* interchangeably.

Now, if Haskell used postfix function application, then we could simply define $quote = 0$, $tick = succ$, $endquote = id$ and we would be done. For Haskell's prefix function application we must additionally arrange that functions and arguments are swapped:

$$\begin{aligned} quote\ \ \ & f = f\ 0 \\ tick\ \ \ & n\ f = f\ (succ\ n) \\ endquote\ & n\ \ = n \end{aligned}$$

The stepwise evaluation of « | | | » shows that *tick* increments the counter, initialised to 0 by *quote*, and then passes control to the next function, which is either another *tick* or *endquote*.

$$
\begin{aligned}
& \mathit{quote}\ \mathit{tick}\ \mathit{tick}\ \mathit{tick}\ \mathit{endquote} \\
= {} & \mathit{tick}\ 0\ \mathit{tick}\ \mathit{tick}\ \mathit{endquote} \\
= {} & \mathit{tick}\ 1\ \mathit{tick}\ \mathit{endquote} \\
= {} & \mathit{tick}\ 2\ \mathit{endquote} \\
= {} & \mathit{endquote}\ 3 \\
= {} & 3
\end{aligned}
$$

The evaluation is solely driven by the terminal symbols, which is why we call them *active terminals*. This technique of passing control to a function argument is reminiscent of *continuation-passing style* (CPS). And indeed, if we call the *CPS* Monad to mind[1]

$$
\begin{aligned}
& \textbf{type}\ \mathit{CPS}\ \alpha = \forall \mathit{ans}\ .\ (\alpha \to \mathit{ans}) \to \mathit{ans} \\
& \textbf{instance}\ \mathit{Monad}\ \mathit{CPS}\ \textbf{where} \\
& \quad \mathit{return}\ a = \lambda\kappa \to \kappa\ a \\
& \quad m \gg\!= k\ = \lambda\kappa \to m\ (\lambda a \to k\ a\ \kappa)
\end{aligned}
$$

we can identify *quote* as *return* 0 and *tick* as *lift succ* where *lift* turns a pure function into a monadic one:

$$
\begin{aligned}
& \textbf{type}\ \alpha \rightarrowtriangle \beta = \alpha \to \mathit{CPS}\ \beta \\
& \mathit{lift} \qquad :: (\alpha \to \beta) \to (\alpha \rightarrowtriangle \beta) \\
& \mathit{lift}\ f\ a = \mathit{return}\ (f\ a)
\end{aligned}
$$

However, the bind of the monad, '$\gg\!=$', seems unrelated: in the *CPS* monad the continuation represents the 'rest of the computation' whereas in our example the continuation only stands for the next parsing step.

It may come as a surprise that the instance declaration above is not the only possibility for turning *CPS* into a monad. Here is a second instance introducing the monad of *partial continuations*.

$$
\begin{aligned}
& \textbf{instance}\ \mathit{Monad}\ \mathit{CPS}\ \textbf{where} \\
& \quad \mathit{return}\ a = \lambda\kappa \to \kappa\ a \\
& \quad m \gg\!= k\ = m\ k
\end{aligned}
$$

The definition of *return* is unchanged; '$\gg\!=$' is now a type-restricted instance of function application. Actually, it is a combination of type application—the universally quantified variable *ans* in the type of m is instantiated to $\mathit{CPS}\ \beta$—and function application, but this is not visible in Haskell. Since '$\gg\!=$' is postfix application of 'effectful' functions, this CPS monad implements postfix function application! Consequently, the quotation « | | | » can be seen as a monadic computation in disguise:

$$(((\mathit{quote}\ \mathit{tick})\ \mathit{tick})\ \mathit{tick})\ \mathit{endquote} \quad = \quad \mathit{run}\ (\mathit{quote} \gg\!= \mathit{tick} \gg\!= \mathit{tick} \gg\!= \mathit{tick})$$

where *run* encapsulates a *CPS* computation:

[1] The instance declaration, which is not legal Haskell, serves only illustrative purposes. We shall only need *return* and only at this particular type.

$$\begin{aligned} &\mathit{run} \quad :: \mathit{CPS}\ \alpha \to \alpha \\ &\mathit{run}\ m = m\ \mathit{id} \end{aligned}$$

Generalising from the example, quotations are based on the identity

$$\mathit{quote}\ \mathit{act}_1\ \ldots\ \mathit{act}_n\ \mathit{endquote} \quad = \quad \mathit{run}\ (\mathit{quote} \gg\!= \mathit{act}_1 \gg\!= \cdots \gg\!= \mathit{act}_n)$$

where $\mathit{quote} :: \mathit{CPS}\ \tau_1$, $\mathit{act}_i :: \tau_i \nrightarrow \tau_{i+1}$, and $\mathit{endquote} = \mathit{id}$. It is useful to think of the τ_i as state types and the act_i as transitions: *quote* initialises the state; each active terminal act_i transforms the state. In our example, we had a single state type but this need not be the case in general. In fact, choosing precise state types is the key to 'typed quotes/antiquotes'.

Just in case you were wondering, none of the two *CPS* monads is a very exciting one in terms of expressiveness: they are both isomorphic to the *identity monad* with *return* and *run* converting between them. In other words, *CPS* offers no effects. Without loss of generality, we may therefore assume that *quote* and act_i are liftings: $\mathit{quote} = \mathit{return}\ a$ and $\mathit{act}_i = \mathit{lift}\ f_i$ for some a and suitable f_i. The following calculation summarises our findings:

$$\begin{aligned} & \mathit{quote}\ \mathit{act}_1\ \ldots\ \mathit{act}_n\ \mathit{endquote} \\ = \quad & \{\ \text{definition of '}\gg\!=\text{' and } \mathit{run}\text{, and } \mathit{endquote} = \mathit{id}\ \} \\ & \mathit{run}\ (\mathit{quote} \gg\!= \mathit{act}_1 \gg\!= \cdots \gg\!= \mathit{act}_n) \\ = \quad & \{\ \mathit{CPS}\ \text{is a pure monad: } \mathit{quote} = \mathit{return}\ a \text{ and } \mathit{act}_i = \mathit{lift}\ f_i\ \} \\ & \mathit{run}\ (\mathit{return}\ a \gg\!= \mathit{lift}\ f_1 \gg\!= \cdots \gg\!= \mathit{lift}\ f_n) \\ = \quad & \{\ \text{monad laws}\ \} \\ & \mathit{run}\ (\mathit{return}\ (f_n\ (\cdots\ (f_1\ a)\ \cdots))) \\ = \quad & \{\ \mathit{run} \cdot \mathit{return} = \mathit{id}\ \} \\ & f_n\ (\cdots\ (f_1\ a)\ \cdots) \end{aligned}$$

In the toy example and in the formal development above, *endquote* was always the identity. This is, however, not quite adequate, as the desired value of a quotation is not necessarily identical to the last state. Fortunately, *endquote* can be any function, since we can fuse a post-processing step with the final continuation: $\mathit{post}\ (\mathit{run}\ m) = m\ \mathit{post}$. This is an immediate consequence of the free theorem for the type $\mathit{CPS}\ \alpha$ (Wadler, 1989).

To summarise, a quotation of type τ is of the form

$$\mathit{quote}\ \mathit{act}_1\ \ldots\ \mathit{act}_n\ \mathit{endquote}$$

where $\mathit{quote} :: \mathit{CPS}\ \tau_1$, $\mathit{act}_i :: \tau_i \nrightarrow \tau_{i+1}$ and $\mathit{endquote} :: \tau_{n+1} \to \tau$.

Since the evaluation of a quotation is driven by the terminal symbols, the implementation of a quote/antiquote mechanism for a particular guest language goes hand in hand with the development of a parser for the concrete syntax. The following sections are ordered by the underlying parser's level of sophistication: Section 3 shows how to implement simple postfix and prefix parsers, Section 4 deals with predictive top-down parsers, and finally Section 5 introduces quotations that are based on bottom-up parsers.

3 Parsing data types

Continuing the example from the introduction, we show how to parse elements of data types in postfix and in prefix notation. Section 3.1 is an excerpt of Okasaki's extensive treatment of postfix languages (2002).

3.1 Postfix notation

In postfix notation, also known as reverse Polish notation, functions follow their arguments. Postfix notation dispenses with the need for parentheses, if the arity of functions is statically known. This is generally not the case in higher-order typed (HOT) languages, but it is true of **data** constructors (ignoring the fact that they are curried in Haskell).

Evaluation of postfix expressions is naturally stack-based: a function pops its arguments from the stack and pushes the result back onto it. To parse data types in postfix notation we introduce for each data constructor $C :: \tau_1 \to \cdots \to \tau_n \to \tau$ a *postfix constructor*:

$$\begin{aligned} &c :: (((st, \tau_1), \ldots), \tau_n) \to (st, \tau) \\ &c \quad (((st, t_1), \ldots), t_n) = (st, C\ t_1\ \ldots\ t_n) \end{aligned}$$

The stack, represented by a nested pair, grows from left to right. The modification of the stack is precisely captured in the type: c is only applicable if the stack contains at least n arguments and the topmost n have the correct types. For the *Tree* type this specialises to

$$\begin{aligned} &\mathit{leaf} && :: st \to (st, \mathit{Tree}) \\ &\mathit{leaf}\ st && = (st, \mathit{Leaf}) \\ &\mathit{fork} && :: ((st, \mathit{Tree}), \mathit{Tree}) \to (st, \mathit{Tree}) \\ &\mathit{fork}\ ((st, l), r) && = (st, \mathit{Fork}\ l\ r) \end{aligned}$$

Given these prerequisites, we can instantiate the framework of Section 2.

$$\begin{aligned} &\mathit{quote} && :: \mathit{CPS}\ () \\ &\mathit{quote} && = \mathit{return}\ () \\ &\texttt{leaf} && :: st \mathrel{-\!\!\triangleright} (st, \mathit{Tree}) \\ &\texttt{leaf} && = \mathit{lift}\ \mathit{leaf} \\ &\texttt{fork} && :: ((st, \mathit{Tree}), \mathit{Tree}) \mathrel{-\!\!\triangleright} (st, \mathit{Tree}) \\ &\texttt{fork} && = \mathit{lift}\ \mathit{fork} \\ &\mathit{endquote} && :: ((), \mathit{Tree}) \to \mathit{Tree} \\ &\mathit{endquote}\ ((), t) && = t \end{aligned}$$

The function *quote* initialises the state to the empty stack; *endquote* extracts the quoted tree from a singleton stack.

It is instructive to step through the static and dynamic elaboration of a quotation. Type checking statically guarantees that a quotation constitutes a well-formed postfix expression.

```
quote                                    :: CPS ()
quote leaf                               :: CPS ((), Tree)
quote leaf leaf                          :: CPS (((), Tree), Tree)
quote leaf leaf fork                     :: CPS ((), Tree)
quote leaf leaf fork leaf                :: CPS (((), Tree), Tree)
quote leaf leaf fork leaf fork           :: CPS ((), Tree)
quote leaf leaf fork leaf fork endquote  :: Tree
```

In each step, the state type precisely mirrors the stack layout. Consequently, pushing too few or too many or the wrong types of arguments results in a static type error. The dynamic evaluation shows how the state evolves.

```
   quote leaf leaf fork leaf fork endquote
= leaf () leaf fork leaf fork endquote
= leaf ((), Leaf) fork leaf fork endquote
= fork (((), Leaf), Leaf) leaf fork endquote
= leaf ((), Fork Leaf Leaf) fork endquote
= fork (((), Fork Leaf Leaf), Leaf) endquote
= endquote ((), Fork (Fork Leaf Leaf) Leaf)
= Fork (Fork Leaf Leaf) Leaf
```

The state is always passed as the first argument. This is something to bear in mind when implementing additional functionality such as an antiquote mechanism.

```
antiquote      :: st → Tree ↠ (st, Tree)
antiquote st t = return (st, t)
```

The tree is spliced into the current position simply by pushing it onto the stack.

3.2 Prefix notation

Postfix notation was easy; its dual, prefix notation, is slightly harder. Prefix notation was invented in 1920 by Jan Łukasiewicz, a Polish logician, mathematician, and philosopher. Because of its origin, prefix notation is also known as Polish notation.

In postfix notation a function follows its arguments; so a stack of arguments is a natural choice for the state. In prefix notation a function precedes its arguments. Consequently, the state becomes a stack of *pending* arguments. For each data constructor $C :: \tau_1 \to \cdots \to \tau_n \to \tau$ we introduce a *prefix constructor*:

```
c°  :: ((τ, st) → α) → ((τ1, (..., (τn, st))) → α)
c°  ctx              = λ(t1, (..., (tn, st))) → ctx (C t1 ... tn, st)
```

The stack, again represented by a nested pair, now grows from right to left. The first argument of c° can be seen as a request for a value of type τ (a request is also known as a context or as an expression with a hole). The prefix constructor can

satisfy this request but in turn generates requests for its arguments.[2] For the *Tree* type we obtain

$$
\begin{aligned}
&\mathit{leaf}^\circ && :: ((\mathit{Tree}, \mathit{st}) \to \alpha) \to (\mathit{st} \to \alpha)\\
&\mathit{leaf}^\circ\ \mathit{ctx} && = \lambda \mathit{st} \to \mathit{ctx}\ (\mathit{Leaf}, \mathit{st})\\
&\mathit{fork}^\circ && :: ((\mathit{Tree}, \mathit{st}) \to \alpha) \to ((\mathit{Tree}, (\mathit{Tree}, \mathit{st})) \to \alpha)\\
&\mathit{fork}^\circ\ \mathit{ctx} && = \lambda(t, (u, \mathit{st})) \to \mathit{ctx}\ (\mathit{Fork}\ t\ u, \mathit{st})
\end{aligned}
$$

The implementation of quotations and anti-quotations is again a straightforward application of the general framework:

$$
\begin{aligned}
&\mathit{quote} && :: \mathit{CPS}\ ((\mathit{Tree}, ()) \to \mathit{Tree})\\
&\mathit{quote} && = \mathit{return}\ (\lambda(t, ()) \to t)\\
&\texttt{leaf} && :: ((\mathit{Tree}, \mathit{st}) \to \alpha) \twoheadrightarrow (\mathit{st} \to \alpha)\\
&\texttt{leaf} && = \mathit{lift}\ \mathit{leaf}^\circ\\
&\texttt{fork} && :: ((\mathit{Tree}, \mathit{st}) \to \alpha) \twoheadrightarrow ((\mathit{Tree}, (\mathit{Tree}, \mathit{st})) \to \alpha)\\
&\texttt{fork} && = \mathit{lift}\ \mathit{fork}^\circ\\
&\mathit{endquote} && :: (() \to \mathit{Tree}) \to \mathit{Tree}\\
&\mathit{endquote}\ \mathit{ctx} && = \mathit{ctx}\ ()\\
&\mathit{antiquote} && :: ((\mathit{Tree}, \mathit{st}) \to \alpha) \to \mathit{Tree} \twoheadrightarrow (\mathit{st} \to \alpha)\\
&\mathit{antiquote}\ \mathit{ctx}\ t && = \mathit{return}\ (\lambda \mathit{st} \to \mathit{ctx}\ (t, \mathit{st}))
\end{aligned}
$$

The stack is initialised to one pending argument; we are done if there are no pending arguments left. Let us again step through an example.

$$
\begin{aligned}
&\mathit{quote} && :: \mathit{CPS}\ (((\mathit{Tree}, ()) \to \mathit{Tree})\\
&\mathit{quote}\ \texttt{fork} && :: \mathit{CPS}\ ((\mathit{Tree}, (\mathit{Tree}, ())) \to \mathit{Tree})\\
&\mathit{quote}\ \texttt{fork}\ \texttt{fork} && :: \mathit{CPS}\ ((\mathit{Tree}, (\mathit{Tree}, (\mathit{Tree}, ()))) \to \mathit{Tree})\\
&\mathit{quote}\ \texttt{fork}\ \texttt{fork}\ \texttt{leaf} && :: \mathit{CPS}\ ((\mathit{Tree}, (\mathit{Tree}, ())) \to \mathit{Tree})\\
&\mathit{quote}\ \texttt{fork}\ \texttt{fork}\ \texttt{leaf}\ \texttt{leaf} && :: \mathit{CPS}\ (((\mathit{Tree}, ()) \to \mathit{Tree})\\
&\mathit{quote}\ \texttt{fork}\ \texttt{fork}\ \texttt{leaf}\ \texttt{leaf}\ \texttt{leaf} && :: \mathit{CPS}\ (() \to \mathit{Tree})\\
&\mathit{quote}\ \texttt{fork}\ \texttt{fork}\ \texttt{leaf}\ \texttt{leaf}\ \texttt{leaf}\ \mathit{endquote} && :: \mathit{CPS}\ \mathit{Tree}
\end{aligned}
$$

The types show how the stack of pending arguments grows and shrinks. For instance, when the first two `fork`s have been processed, three further subtrees are required: the left and the right subtree of the second `fork` and the right subtree of the first `fork`. The stepwise evaluation makes this explicit:

[2] The type variable α that appears in the type signature of c° corresponds to the type of the entire quotation and can be safely instantiated to *Tree*. The polymorphic type is only vital if c° is used in quotations of different types.

$$
\begin{aligned}
&\mathit{quote}\ \texttt{fork}\ \texttt{fork}\ \texttt{leaf}\ \texttt{leaf}\ \texttt{leaf}\ \mathit{endquote}\\
=\ &\texttt{fork}\ (\lambda(t,())\to t)\ \texttt{fork}\ \texttt{leaf}\ \texttt{leaf}\ \texttt{leaf}\ \mathit{endquote}\\
=\ &\texttt{fork}\ (\lambda(t,(u,()))\to \mathit{Fork}\ t\ u)\ \texttt{leaf}\ \texttt{leaf}\ \texttt{leaf}\ \mathit{endquote}\\
=\ &\texttt{leaf}\ (\lambda(t',(u',(u,())))\to \mathit{Fork}\ (\mathit{Fork}\ t'\ u')\ u)\ \texttt{leaf}\ \texttt{leaf}\ \mathit{endquote}\\
=\ &\texttt{leaf}\ (\lambda(u',(u,()))\to \mathit{Fork}\ (\mathit{Fork}\ \mathit{Leaf}\ u')\ u)\ \texttt{leaf}\ \mathit{endquote}\\
=\ &\texttt{leaf}\ (\lambda(u,())\to \mathit{Fork}\ (\mathit{Fork}\ \mathit{Leaf}\ \mathit{Leaf})\ u)\ \mathit{endquote}\\
=\ &\mathit{endquote}\ (\lambda()\to \mathit{Fork}\ (\mathit{Fork}\ \mathit{Leaf}\ \mathit{Leaf})\ \mathit{Leaf})\\
=\ &\mathit{Fork}\ (\mathit{Fork}\ \mathit{Leaf}\ \mathit{Leaf})\ \mathit{Leaf}
\end{aligned}
$$

Again, if we pass too few or too many or the wrong types of arguments, then we get a static type error.

Remark 1
The deeply nested pairs can be avoided if we curry the prefix constructors:

$$
\begin{aligned}
&c^\circ :: (\tau\to\alpha)\to(\tau_1\ \to\cdots\to\tau_n\ \to\alpha)\\
&c^\circ\quad ctx\qquad =\ \lambda t_1\to\cdots\to\lambda t_n\to ctx\ (C\ t_1\ \ldots\ t_n)
\end{aligned}
$$

Additionally, we have generalised the result type of requests from $st\to\alpha$ to α. The adaptation of the remaining code is left as an exercise to the reader. □

4 Top-down parsing

The main reason for treating prefix parsers is that they pave the way for the more expressive class of LL(1) parsers. The basic setup remains unchanged; we need only one additional programming technique. To keep the learning curve smooth, however, we shall go through one intermediate step and treat grammars in Greibach normal form first.

4.1 Greibach normal form

A context-free grammar is in Greibach normal form (GNF) iff all productions are of the form $A\to a\omega$, where a is a terminal symbol and ω is a possibly empty sequence of non-terminal symbols. A grammar in GNF is (syntactically) *unambiguous* iff each pair of productions $A_1\to a\omega_1$ and $A_2\to a\omega_2$ satisfies $A_1=A_2\Longrightarrow\omega_1=\omega_2$. Unambiguous grammars in GNF generalise data type declarations, as a terminal (data constructor) may appear in different productions (data type declarations).

Here is an example grammar for a simple imperative language and an equivalent grammar in GNF.

$$
\begin{array}{lcl}
S & \to & \texttt{id}\ \texttt{:=}\ E\\
 & \mid & \texttt{if}\ E\ S\ S\\
 & \mid & \texttt{while}\ E\ S\\
 & \mid & \texttt{begin}\ B\ \texttt{end}\\
E & \to & \texttt{id}\\
B & \to & S\mid S\ \texttt{;}\ B
\end{array}
\qquad\Longrightarrow\qquad
\begin{array}{lcl}
S & \to & \texttt{id}\ C\ E\\
 & \mid & \texttt{if}\ E\ S\ S\\
 & \mid & \texttt{while}\ E\ S\\
 & \mid & \texttt{begin}\ S\ R\\
C & \to & \texttt{:=}\\
E & \to & \texttt{id}\\
R & \to & \texttt{end}\mid\texttt{;}\ S\ R
\end{array}
$$

Remark 2

If we add the production $S \to \texttt{if}\ E\ S$, then the grammar becomes ambiguous, an instance of the (in-) famous *dangling-else problem.* □

The abstract syntax of the imperative language is given by

$$\textbf{type}\ \mathit{Var} = \mathit{String}$$
$$\textbf{data}\ \mathit{Stat} = \mathit{Set}\ \mathit{Var}\ \mathit{Var} \mid \mathit{If}\ \mathit{Var}\ \mathit{Stat}\ \mathit{Stat} \mid \mathit{While}\ \mathit{Var}\ \mathit{Stat} \mid \mathit{Begin}\ [\mathit{Stat}]$$

As an example, the quotation

```
« begin
  x := y ;
  if x
     y := z
     z := y
  end »
```

evaluates to $\mathit{Begin}\ [\mathit{Set}\ \texttt{"x"}\ \texttt{"y"}, \mathit{If}\ \texttt{"x"}\ (\mathit{Set}\ \texttt{"y"}\ \texttt{"z"})\ (\mathit{Set}\ \texttt{"z"}\ \texttt{"y"})]$. The definition of the variables x, y and z poses a minor problem, which we shall discuss later.

A parser for a grammar in GNF is very similar to a prefix parser: the state is a stack of *pending* non-terminal symbols. An active terminal selects a production by looking at the topmost symbol on the stack. If the grammar is unambiguous, then there is at most one suitable production. The non-terminal is then replaced by the right-hand side of the production (omitting the leading terminal).

As before, we want to guarantee statically that a quotation is well-formed, so that its parse does not fail. To this end we represent non-terminals by types:

$$\begin{aligned}
&\textbf{newtype}\ S\ \alpha = S\ (\mathit{Stat} \to \alpha)\\
&\textbf{newtype}\ C\ \alpha = C\ (\phantom{\mathit{Stat}} \to \alpha)\\
&\textbf{newtype}\ E\ \alpha = E\ (\mathit{Var} \to \alpha)\\
&\textbf{newtype}\ R\ \alpha = R\ ([\mathit{Stat}] \to \alpha)
\end{aligned}$$

The declarations also list the types of the semantic values that are attached to the non-terminals. Using these type-level non-terminals we can program the type-checker to parse quotations: each production $A \to aB_1 \ldots B_n$ is mapped to a function a, the active terminal, of type $A\ \alpha \twoheadrightarrow B_1\ (\cdots\ (B_n\ \alpha)\ \cdots)$, that implements the expansion of A. There is one hitch, however: the terminal a may appear in different productions, so it cannot possibly translate to a single function. Rather, an active terminal stands for a family of functions, represented in Haskell by a multiple-parameter type class.

We introduce one class for each terminal symbol that appears more than once. In our case, the only such terminal is `id`, so we need just one class.

$$\begin{aligned}
&\textbf{class}\ \mathit{Id}\ \mathit{lhs}\ \mathit{rhs} \mid \mathit{lhs} \to \mathit{rhs}\ \textbf{where}\\
&\quad \texttt{id} :: \mathit{String} \to (\mathit{lhs} \twoheadrightarrow \mathit{rhs})
\end{aligned}$$

The functional dependency $\mathit{lhs} \to \mathit{rhs}$ avoids ambiguities during type-inference making use of the fact that the underlying grammar in GNF is unambiguous.

Each production is translated into an equation.

```
instance Id (S α) (C (E α)) where
  id l = lift (λ(S ctx) → C (E (λr → ctx (Set l r))))
if     = lift (λ(S ctx) → E (λc → S (λt → S (λe → ctx (If c t e)))))
while  = lift (λ(S ctx) → E (λc → S (λs → ctx (While c s))))
begin  = lift (λ(S ctx) → S (λs → R (λr → ctx (Begin (s : r)))))
:=     = lift (λ(C ctx) → ctx)
instance Id (E α) α where
  id i = lift (λ(E ctx) → ctx i)
end    = lift (λ(R ctx) → ctx [])
;      = lift (λ(R ctx) → S (λs → R (λr → ctx (s : r))))
quote       = return (S (λs → s))
endquote s = s
```

The *quote* function pushes the start symbol on the stack; *endquote* simply returns the final value of type *Stat*.

The terminal symbol `id` is a bit unusual in that it takes an additional argument, the string representation of the identifier. This has the unfortunate consequence that identifiers must be enclosed in parentheses as in « (`id "x"`) `:=` (`id "y"`) ». We can mitigate the unwelcome effect by introducing shortcuts

```
x, y :: (Id lhs rhs) ⇒ lhs ↠ rhs
x    = id "x"
y    = id "y"
```

so that the quotation becomes « *x* `:=` *y* ». Alternatively, we may swap the two arguments of `id`. In this case, the parentheses can, in fact, must be dropped, so that the quotation is written « `id "x" := id "y"` ».

It is instructive to walk through a derivation.

```
   « while x y := z »
= while (S (λs → s)) x y := z »
= x (E (λc → S (λs → While c s))) y := z »
= y (S (λs → While "x" s)) := z »
= := (C (E (λr → While "x" (Set "y" r)))) z »
= z (E (λr → While "x" (Set "y" r))) »
= » (While "x" (Set "y" "z"))
= While "x" (Set "y" "z")
```

To summarise, for each non-terminal symbol A we define a type: **newtype** $A\ \alpha = A\ (Val \to \alpha)$, where *Val* is the type of the semantic values attached to A. For each terminal symbol a we introduce a class: **class** $a\ lhs\ rhs \mid lhs \to rhs$ **where** $a :: lhs \twoheadrightarrow rhs$. Finally, each production $A \to aB_1 \ldots B_n$ gives rise to an instance declaration:

```
instance a (A α) (B1 (··· (Bn α) ···)) where
  a = lift (λ(A ctx) → B1 (λv1 → ··· → Bn (λvn → ctx (f v1 ... vn))))
```

where f is the semantic action associated with the production. Of course, if a terminal appears only once, then there is no need for overloading.

```
newtype I  α = I  (Expr           → α)  -- id
newtype A  α = A  (                 α)  -- +
newtype M  α = M  (                 α)  -- *
newtype O  α = O  (                 α)  -- (
newtype C  α = C  (                 α)  -- )

newtype E  α = E  (Expr           → α)
newtype E' α = E' ((Expr → Expr) → α)
newtype T  α = T  (Expr           → α)
newtype T' α = T' ((Expr → Expr) → α)
newtype F  α = F  (Expr           → α)

class Id     old new | old → new where id :: String → old ↠ new
class Add    old new | old → new where +  ::          old ↠ new
class Mul    old new | old → new where *  ::          old ↠ new
class Open   old new | old → new where (  ::          old ↠ new
class Close  old new | old → new where )  ::          old ↠ new
class Endquote old                   where »  ::          old → Expr
```

Fig. 1. The LL(1) parser for the expression grammar, part 1.

4.2 LL(1) *parsing*

We are well prepared by now to tackle the first major challenge: implementing quotations whose syntax is given by an LL(1) grammar. As before, we shall work through a manageable example. This time we implement arithmetic expressions given by the grammar on the left below; see page 193 of (Aho *et al.*, 2006).

$$
\begin{array}{lcl}
E & \to & E \texttt{+} T \mid T \\
T & \to & T \texttt{*} F \mid F \\
F & \to & \texttt{(}\ E\ \texttt{)} \mid \mathit{id}
\end{array}
\qquad \Longrightarrow \qquad
\begin{array}{lcl}
E & \to & T\ E' \\
E' & \to & \texttt{+}\ T\ E' \mid \epsilon \\
T & \to & F\ T' \\
T' & \to & \texttt{*}\ F\ T' \mid \epsilon \\
F & \to & \texttt{(}\ E\ \texttt{)} \mid \mathit{id}
\end{array}
$$

The expression grammar is *not* LL(1) due to the left recursion; eliminating the left recursion yields the equivalent LL(1) grammar on the right.

The abstract syntax of arithmetic expressions is given by

```
data Expr = Id String | Add Expr Expr | Mul Expr Expr
```

The semantic actions that construct values of type *Expr* are straightforward to define for the original expression grammar. They are slightly more involved for the LL(1) grammar: E' and T' yield expressions with a hole, where the hole stands for the missing left argument of the operator (see Figures 1 and 2).

The main ingredient of a predictive top-down parser is the *parsing table*. Here is the table for the LL(1) grammar above; see page 225 of (Aho *et al.*, 2006).

	id	+	*	(	)	»
E	$E \to T\ E'$			$E \to T\ E'$		
E'		$E' \to \texttt{+}\ T\ E'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to F\ T'$			$T \to F\ T'$		
T'		$T' \to \epsilon$	$T' \to \texttt{*}\ F\ T'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathit{id}$			$F \to \texttt{(}\ E\ \texttt{)}$		

```
instance Id (E α) (T' (E' α)) where                    -- expand: E → T E'
  id s (E  ctx) = id s (T (λt → E' (λe' → ctx (e' t))))
instance Id (T α) (T' α) where                         -- expand: T → F T'
  id s (T  ctx) = id s (F (λf → T' (λt' → ctx (t' f))))
instance Id (F α) α where                              -- expand: F → id
  id s (F  ctx) = id s (I (λv → ctx v))
instance Id (I α) α where                              -- pop
  id s (I  ctx) = return (ctx (Id s))
instance Add (E' α) (T (E' α)) where                   -- expand: E' → + T E'
  +  (E' ctx) = + (A (T (λt → E' (λe' → ctx (λl → e' (Add l t))))))
instance (Add α α') ⇒ Add (T' α) α' where              -- expand: T' → ϵ
  +  (T' ctx) = + (ctx (λe → e))
instance Add (A α) α where                             -- pop
  +  (A  ctx) = return ctx
instance Mul (T' α) (F (T' α)) where                   -- expand: T' → * F T'
  *  (T' ctx) = * (M (F (λf → T' (λt' → ctx (λl → t' (Mul l f))))))
instance Mul (M α) α where                             -- pop
  *  (M  ctx) = return ctx
instance Open (E α) (E (C (T' (E' α)))) where          -- expand: E → T E'
  (  (E  ctx) = ( (T (λt → E' (λe' → ctx (e' t))))
instance Open (T α) (E (C (T' α))) where               -- expand: T → F T'
  (  (T  ctx) = ( (F (λf → T' (λt' → ctx (t' f))))
instance Open (F α) (E (C α)) where                    -- expand: F → ( E )
  (  (F  ctx) = ( (O (E (λe → C (ctx e))))
instance Open (O α) α where                            -- pop
  (  (O  ctx) = return ctx
instance (Close α α') ⇒ Close (E' α) α' where          -- expand: E' → ϵ
  )  (E' ctx) = ) (ctx (λe → e))
instance (Close α α') ⇒ Close (T' α) α' where          -- expand: T' → ϵ
  )  (T' ctx) = ) (ctx (λe → e))
instance Close (C α) α where                           -- pop
  )  (C  ctx) = return ctx
instance (Endquote α) ⇒ Endquote (E' α) where          -- expand: E' → ϵ
  »  (E' ctx) = » (ctx (λe → e))
instance (Endquote α) ⇒ Endquote (T' α) where          -- expand: T' → ϵ
  »  (T' ctx) = » (ctx (λe → e))
instance Endquote Expr where                           -- pop
  »  e        = e
```

Fig. 2. The LL(1) parser for the expression grammar, part 2.

The table also includes a column for '»', which serves as an end marker.

The state is now a stack of pending symbols, both terminal and non-terminal. The symbol X on top of the stack determines the action of the current active terminal a. If $X = a$, then the terminal pops X from the stack and passes control to the next active terminal (pop action). If X is a non-terminal, then the terminal looks up the production indexed by X and a in the parsing table, replaces X by the right-hand side of the production, and remains active (expand action). Again,

we need not consider error conditions as the type checker will statically guarantee that parsing does not fail.

Since the state is a stack of symbols, we must introduce types both for terminal and non-terminal symbols (we only show some representative examples here, the complete code is listed in Figures 1 and 2):

```
newtype E α = E (Expr → α)   -- E
newtype I α = I (Expr → α)   -- id
```

Like non-terminals, terminal symbols may carry semantic information: the terminal *id s*, for instance, returns the semantic value *Id s* of type *Expr*. For each terminal symbol including '»' we introduce a class and an instance that implements the pop action:

```
class Add old new | old → new where
  + :: old ↠ new
instance Add (A α) α where
  + (A ctx) = return ctx
```

Finally, each entry of the parsing table gives rise to an instance declaration that implements the expand action. Here are the instances for '+':

```
instance Add (E' α) (T (E' α)) where
  + (E' ctx) = + (A (T (λt → E' (λe' → ctx (λl → e' (Add l t))))))
instance (Add α α') ⇒ Add (T' α) α' where
  + (T' ctx) = + (ctx (λe → e))
```

Since the look-ahead token is unchanged, the second instance requires an additional context, *Add* α α', which accounts for the 'recursive' call to '+'. The first instance also contains a call to '+' but this occurrence can be statically resolved: it refers to the 'pop instance'. In general, an instance for the production $A \to \omega$ requires a context iff $\epsilon \in \mathcal{L}(\omega)$, as in this case the stack shrinks.[3] The instance head always reflects the parsing state *after* the final pop action. Consider the *id* instance

```
instance Id (E α) (T' (E' α)) where
  id s (E ctx) = id s (T (λt → E' (λe' → ctx (e' t))))
```

The expansion phase proceeds $E \to T\ E' \to F\ T'\ E' \to id\ T'\ E'$. Consequently, the instance head records that the state changes from E to $T'\ E'$.

It remains to implement *quote* and *antiquote*.

```
quote              = return (E (λe → e))
antiquote          :: E st → Expr ↠ st
antiquote (E st) e = return (st e)
```

The type of *antiquote* dictates that we can only splice an expression into a position

[3] The standard construction of parsing tables using *First* and *Follow* sets already provides the necessary information: the parsing table contains the production $A \to \omega$ for look-ahead a if either $a \in \mathit{First}(\omega)$, or $\epsilon \in \mathcal{L}(\omega)$ and $a \in \mathit{Follow}(A)$. Only in the second case is a context required.

where the non-terminal E is expected. This can be achieved by enclosing the anti-quotation in '`(`' and '`)`'.

```
Main⟩   « x + ( ` (foldr1 Add [Id (show i) | i ← [1 .. 3]]) ) + y »
Add (Add (Id "x") (Add (Id "1") (Add (Id "2") (Id "3")))) (Id "y")
```

Remark 3
We have implemented the parsing actions in a rather ad-hoc way. Alternatively, they can be written using *lift* and monadic composition:

```
instance Id (E α) (T' (E' α)) where
  id s = id s ∘ lift (λ(E ctx) → T (λt → E' (λe' → ctx (e' t))))
(∘)     :: (b → CPS c) → (a → CPS b) → (a → CPS c)
q ∘ p   = λa → p a ≫= q
```

The rewrite nicely separates the expansion step from the 'recursive call'. □

5 Bottom-up parsing: LR(0) parsing

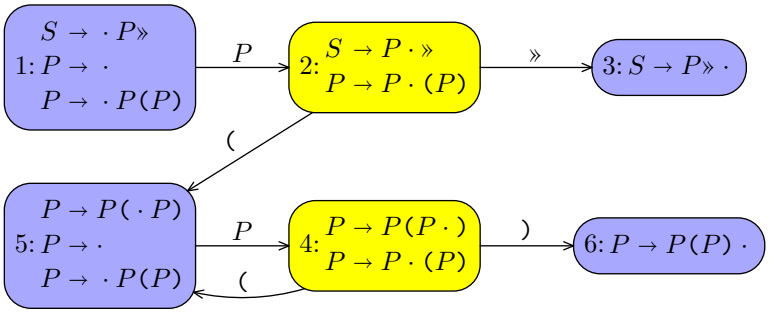
Let us move on to our final challenge: quotations whose concrete syntax is based on an LR(0) grammar—LR(1) grammars are also doable but we resist the temptation to spell out the details. Unlike LL parsing, the LR method is generally not suitable for constructing parsers by hand. For that reason, we shall base the treatment on a very simple example, the language of balanced parentheses.

$$P \quad \to \quad \epsilon \mid P\ (\ P\ )$$

The abstract syntax is given by the *Tree* data type: $P \to \epsilon$ constructs a *Leaf*, $P \to P\ (\ P\ )$ a *Fork*.

An LR parser is similar to a postfix parser in that the state is a stack of symbols the parser has already seen. In contrast, the state of an LL parser is a stack of symbols it expects to see.

For efficiency reasons, an LR parser maintains additional information that summarises the stack configuration in each step. This is accomplished by a finite-state machine, the so-called LR(0) *automaton*. Here is the automaton for the grammar above (S is a new start symbol, '»' serves as an end marker).



The automaton has six states; the production(s) contained in the states illustrate the progress of the parse with the dots marking the borderline between what we

have seen and what we expect to see. If the dot appears before a terminal symbol, we have a *shift state* (coloured in yellow/light grey). If the dot has reached the end in one of the productions, we have a *reduce state* (coloured in blue/dark grey). In a shift state, the parser consumes an input token and pushes in onto the stack. In a reduce state, the right-hand side of a production resides on top of the stack, which is then replaced by the left-hand side.

In our example, the parser first reduces $P \to \epsilon$ moving from the start state 1 to state 2. Then, it shifts either '»' or '`(`'. Each transition is recorded on the stack. This information is used during a reduction to determine the next state. Consider state 6; two sequences of moves end in this state: $1 \xrightarrow{P} 2 \xrightarrow{\texttt{(}} 5 \xrightarrow{P} 4 \xrightarrow{\texttt{)}} 6$ and $5 \xrightarrow{P} 4 \xrightarrow{\texttt{(}} 5 \xrightarrow{P} 4 \xrightarrow{\texttt{)}} 6$. Removing $P\texttt{(}P\texttt{)}$ from the stack means returning to either state 1 or state 5. Pushing P onto the stack we move forward to either state 2 or state 4. In short, reducing $P \to P\texttt{(}P\texttt{)}$ is accomplished by replacing the above transitions by either $1 \xrightarrow{P} 2$ or $5 \xrightarrow{P} 4$. The point is that in general there are several transitions for a single production.

Turning to the implementation, the parser's state is a stack of LR(0) states. Each LR(0) state carries the semantic value of the unique symbol that annotates the ingoing edges.

```
data S1    = S1            -- S
data S2 st = S2 Tree st    -- P
data S3 st = S3      st    -- »
data S4 st = S4 Tree st    -- P
data S5 st = S5      st    -- (
data S6 st = S6      st    -- )
```

Each state of the automaton is translated into a function that performs the corresponding action. A shift state simply delegates the control to the next active terminal. A reduce state pops the transitions corresponding to the right-hand side from the stack, and pushes a transition corresponding to the left-hand side. If there are several possible transitions, then a reduce action is given by a family of functions represented as a type class.

```
quote                 = state1 S1                                  -- start
state1 st             = state2 (S2 Leaf st)                        -- reduce
state2 st             = return st                                  -- shift
state3 (S3 (S2 t S1)) = t                                          -- accept
state4 st             = return st                                  -- shift
state5 st             = state4 (S4 Leaf st)                        -- reduce
class State6 old new | old → new where
  state6              :: old ↠ new
instance State6 (S6 (S4 (S5 (S2 S1)))) (S2 S1) where              -- reduce
  state6 (S6 (S4 u (S5 (S2 t S1))))     = state2 (S2 (Fork t u) S1)
instance State6 (S6 (S4 (S5 (S4 (S5 st))))) (S4 (S5 st)) where    -- reduce
  state6 (S6 (S4 u (S5 (S4 t (S5 st))))) = state4 (S4 (Fork t u) (S5 st))
```

The pattern $S_6\ (S_4\ u\ (S_5\ (S_4\ t\ (S_5\ st))))$ nicely shows the interleaving of states and semantic values. Since the stack is nested to the right, u is the topmost semantic value and consequently becomes the right subtree in $Fork\ t\ u$.

The active terminals implement the shift actions.

```
class Open old new | old → new where
  (                    :: old ↣ new
instance Open (S2 st) (S4 (S5 (S2 st))) where
  ( st@(S2 _ _)        = state5 (S5 st)
instance Open (S4 st) (S4 (S5 (S4 st))) where
  ( st@(S4 _ _)        = state5 (S5 st)
) st@(S4 _ _)          = state6 (S6 st)
endquote st@(S2 _ _) = state3 (S3 st)
```

We need a class if a terminal annotates more than one edge. Again, the instance types are not entirely straightforward as they reflect the stack modifications up to the next shift: for instance, '`(`' moves from S_2 to S_5 and then to S_4, which is again a shift state.

The implementation technique also works for LR(1) grammars. In this case, the active terminals implement both shift and reduce actions.

6 Conclusion

Quotations provide a new, amusing perspective on parsing: terminal symbols turn active and become the driving force of the parsing process. It is quite remarkable that all major syntax analysis techniques can be adapted to this technique.

Typed quotations provide static guarantees: using type-level representations of symbols Haskell's type checker is instrumented to scrutinise whether a quotation is syntactically correct. Of course, this means that syntax errors become type errors which are possibly difficult to decipher. Adding proper error handling is left as the obligatory 'instructive exercise to the reader'.

References

Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. (2006) *Compilers: Principles, Techniques, and Tools*. 2nd edn. Addison-Wesley Publishing Company.

Okasaki, C. (2002) Techniques for embedding postfix languages in haskell. Chakravarty, M. (ed), *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop* pp. 105–113. ACM Press.

Okasaki, C. (2003) Theoretical Pearls: flattening combinators: surviving without parentheses. *Journal of Functional Programming* **13**(4):815–822.

Peyton Jones, S. (2003) *Haskell 98 Language and Libraries*. Cambridge University Press.

Wadler, P. (1989) Theorems for free! *The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89), London, UK* pp. 347–359. Addison-Wesley Publishing Company.