Embedding Languages in Haskell

Gerrit van den Geest

Department of Information and Computing Sciences
Universiteit Utrecht
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands

ggeest@cs.uu.nl

Abstract

We present work of Hinze [2] and Okasaki [7] about a systematic approach to embed languages in Haskell [4] with a quote/antiquote mechanism. A quote/antiquote mechanism makes it possible to use the concrete syntax of a specific language inside a host language by enclosing it in quotes. Inside the quotation one can return to the host language by using an antiquotation.

This technique can be used to embed for instance Domain Specific Languages (DSLs) in Haskell. Domain Specific Languages have many benefits: programs are easier to understand and reason about. The additional start-up costs of implementing a lexer, parser, compiler, and pretty-printer for a DSL however, are often too high. We can lower those costs by embedding a DSL inside another language. Such a language is called a Domain Specific Embedded Language (DSEL).

In this paper we introduce an increasingly complex syntax for a combinator library expressing Financial Contracts [5]. We start with postfix notation, prefix notation, and a grammar in Greibach Normal Form (GNF) for this DSEL. Then we proceed by implementing a syntax given by an LL(1) grammar. We show how to translate those grammars to an implementation in Haskell. To accomplish this, we use: overloading, polymorphism, and higher order functions. Also techniques reminiscent of the continuation monad are used.

1. Introduction

We present techniques for embedding languages in Haskell [4]. These techniques are based on work of Hinze [2] and Okasaki [7]. As a first example we introduce a language for natural numbers.

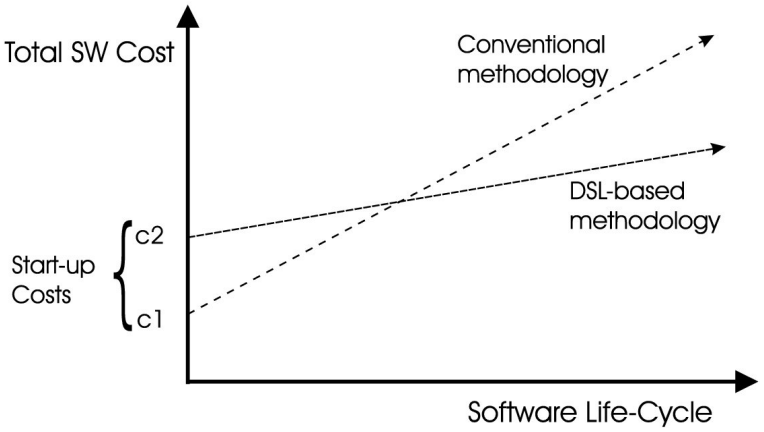
$$\begin{array}{l} quote\ tick\ tick\ tick\ end \\ \quad + \\ quote\ tick\ tick\ tick\ tick\ end \end{array}$$

Between the $quote$ and end keywords we can write concrete syntax, in this case just a sequence of n ticks representing the number n. So the embedding represents the expression $3 + 4$. The language above look like a

This paper is written as part of the Software Technology Colloquium course
of the master Software Technology at the Universiteit Utrecht.

sequence of terminals, but in fact the terminals are just functions and the spaces just function applications. In this paper we show how these functions can be implemented, and also how we can embed less trivial languages.

This technique can be used to implement Domain Specific Languages (DSLs). A DSL is a language tailored for a specific application domain. We all use Domain Specific Languages, such as SQL for querying databases and HTML for specifying the mark-up of a web-page. Writing specifications or programs in a DSL has many advantages: they are easier to write, reason about, and modify compared to programs written in general purpose languages. But the cost of designing and implementing a DSL from scratch is so high that we probably never break even [3].



There are existing efforts and approaches to lower the start-up costs of a DSL. Examples are Lisp macros, which have been used for years to develop embedded languages, or the software-generator approach, where a specification written in a DSL is used to generate a program. We try to embed a DSL in Haskell without preprocessing or macro-expanding, but by just using the Haskell syntax for functions and function application. The following features are crucial for the embedding of a DSL in Haskell: currying, higher-order functions, polymorphism, and type classes.

2. A Framework

2.1 Introduction

Let's take a closer look at the example shown in the introduction. This example looks like a sequence of instructions, but in fact the instructions are just functions and the spaces just function applications. Function application is left-associative in Haskel, so one should read the example as:

$$three = (((quote\ tick)\ tick)\ tick)\ end$$

Function application in Haskell is prefix, that is, a function precedes its arguments. If function application was both left-associative and postfix, that is, a function follows its arguments, we could simply implement the running example with the following functions:

$$\begin{aligned}
&quote :: && Int \\
&quote && = 0 \\
&tick :: Int \rightarrow && Int \\
&tick \quad i && = i + 1 \\
&end :: Int \rightarrow && Int \\
&end && = id
\end{aligned}$$

There is nothing special about these functions, except that they are applied in postfix style. We can see this very nicely if we show the evaluation steps. The part of the expression colored red is evaluated in the next step.

```
  quote tick tick tick end
0 tick  tick tick end
1 tick  tick end
2 tick  end
3 end
3
```

But the above functions will not work because Haskell doesn't have postfix function application. Ideally, we want to have the example work in Haskell and keep the simplicity of the postfix functions.

2.2 Developing a Framework

One idea is to introduce a postfix operator, which emulates postfix function application by first expecting an argument, and then a function.

```
infixl 0 ◁
(◁) :: α → (α → r) → r
x ◁ f = f x
three = quote ◁ tick ◁ tick ◁ tick ◁ end
```

Note that this operator can also be defined as $(\triangleleft) = flip\ (\$)$. If we now write this operator between the terminals of the concrete syntax for naturals, we almost have a concrete syntax embedding, and simple function definitions. After every terminal there is an operator, except for end. This is because we want to stop with parsing after the terminal end. Our last step is try to get rid of the operators between the terminals of the concrete syntax. This can be arranged by building the operator inside each terminal function, except for end. To reach this, we introduce the function $lift$, which has the same implementation as the postfix operator $(\triangleleft)$. One can read this function as: I expect another function that will be applied on the result of this function. We also introduce the type synonym CPS for the type of the lifted value. CPS is a monad, we will explore the relation between CPS and monads in the next subsection.

```
type CPS α = ∀ r.(α → r) → r
lift :: α →    CPS α
lift   a =     λf → f a
quote ::       CPS Int
quote      =   lift  0
tick :: Int → CPS Int
tick    i   =  lift  (i + 1)
end :: Int → Int
end     = id
three = quote tick tick tick end
```

Now we can redefine the terminals of the concrete syntax for naturals, so we get a language embedding with simple function definitions. If we take a closer look at the evaluation of the quotation we can see very clearly that the lift function models prefix function application.

```
quote              tick tick tick end
lift 0        tick tick tick end
tick 0             tick tick end
lift (0 + 1) tick tick end
tick 1             tick end
lift (1 + 1) tick end
```

$tick\ 2 \qquad\qquad end$
$lift\ (2+1)\ end$
$end\ 3$
3

The expressions colored red are evaluated in the next step. The second column is the state threaded through the terminals. Each terminal is applied to the state, just like postfix function application. The lift function is responsible for this kind of behavior.

2.3 Continuation Monads

The technique to more or less emulate left-associative postfix function application is reminiscent of Continuation Passing Style (CPS). CPS is a style of programming where one passes control to a function argument. We use the *lift* function to lift a value into a CPS type. As mentioned earlier, this type is also a monad. The following implementation is the default for $Monad\ CPS$.

$$\begin{aligned}&\textbf{instance}\ Monad\ CPS\ \textbf{where}\\&\quad return\ a = \lambda k \rightarrow k\ a\\&\quad c \gg\!= f \quad = \lambda k \rightarrow c\ (\lambda a \rightarrow f\ a\ k)\end{aligned}$$

Note that a type synonym cannot be an instance of a type class in Haskell, but this presentation makes the examples more readable and easier to explain. The $return$ function of the CPS monad has the same implementation as the *lift* function of the framework earlier defined. But the $\gg\!=$ operator seems unrelated, because in the CPS monad the continuation stands for the rest of the computation whereas in the running example it stands for the next parsing step. Fortunately, there is another implementation for the instance $Monad\ CPS$ and that is the monad for partial continuations.

$$\begin{aligned}&\textbf{instance}\ Monad\ CPS\ \textbf{where}\\&\quad return\ a = \lambda k \rightarrow k\ a\\&\quad c \gg\!= f \quad = c\ f\end{aligned}$$

The monad for partial continuations exactly captures our earlier defined framework. The implementation of $return$ is unchanged, but the implementation of the $\gg\!=$ operator is now just function application. If we now define a function to run a CPS computation, we can implement the running example in monadic style.

$$\begin{aligned}&run :: CPS\ \alpha \rightarrow \alpha\\&run\ m = m\ id\\&three = run\ (return\ 0 \gg\!= tick \gg\!= tick \gg\!= tick)\end{aligned}$$

The monadic style implementation of the running example is exactly the same as the implementation of the last subsection. We prove this for the general case:

$$quote\ term_1\ ...\ term_n\ end = run\ (return\ 0 \gg\!= term_1 \gg\!= ... \gg\!= term_n)$$

We use the earlier defined implementations of $quote$, end and run, and $return$ and $\gg\!=$ are the monad for partial continuations.

$quote\ term_1\ ...\ term_n\ end$

= {definition of end}

$quote\ term_1\ ...\ term_n\ id$

= {definition of run}

$run\ (quote\ term_1\ ...\ term_n)$

= {definition of $\gg\!=$}

$run\ (quote \gg\!= term_1 \gg\!= ... \gg\!= term_n)$

= {definition of $quote$}

$run\ (lift\ 0 \gg\!= term_1 \gg\!= ... \gg\!= term_n)$

= {definition of $lift$}

$run\ ((\lambda f \rightarrow f\ 0) \gg\!= term_1 \gg\!= ... \gg\!= term_n)$

= {definition of $return$}

$run\ (return\ 0 \gg\!= term_1 \gg\!= ... \gg\!= term_n)$

We have shown that the technique used to parse terminals is an instance of partial continuations. In the running example the state threaded to the terminals was just a single integer. In the next sections we only change the state to parse more advanced languages: we leave the parsing technique introduced in this section untouched.

3. Postfix notation

3.1 Introduction

In this section we show how to embed postfix notation in Haskell. Postfix notation, also known as Reverse Polish Notation (RPN), was invented by Australian philosopher and computer scientist Charles Hamblin. It is derived from the Polish notation, which was introduced by the Polish mathematician Jan Lukasiewicz. We will treat Polish or prefix notation in the next section.

In postfix notation, functions follow their arguments. So $3\ 5+2*$ is postfix notation for the expression $(3+5)*2$. In postfix notation there is no need for writing parentheses, because the arity of functions is statically known. Haskell data constructors can be written in postfix notation form, but Haskell functions not, because the arity of Haskell functions is not statically known.

Postfix notation is evaluated using a stack. If we evaluate the expression $3\ 5\ +\ 2*$, 3 and 5 are pushed on the stack. Then the operator $+$ pops two values from the stack and pushes the result back. Next 3 is pushed on the stack, and, finally, the operator $*$ pops two values from the stack and pushes the result back.

In this section we first show a systematic translation of Haskell data constructors to a postfix notation embedding, then we show how to embed postfix notation for financial contracts [5].

3.2 Systematic translation

The state threaded through the terminals of concrete syntax for natural numbers was just a single integer. Because evaluation of postfix notation is stack-based, the state is now a stack of arguments. We represent the stack by nested tuples growing from left to right. The stack could also be represented by a list, but that is not the right choice. First, because nested tuples can contain elements of different types and a list not. Second, because the types of nested tuples correspond very closely to the actual values on the stack. This can be illustrated by the redefined $quote$ and end functions:

$$
\begin{aligned}
&quote :: \ CPS\ () \\
&quote = lift \quad () \\
&end :: ((), \alpha) \rightarrow \alpha \\
&end \quad ((), a) \ = \ a
\end{aligned}
$$

The $quote$ function initializes the state to the empty stack, and the end function expects a singleton stack and pops the value from the stack. The types correspond to the actual stack layout.

The $quote$ and end functions are independent from the actual data type we embed. To embed a data type in postfix notation we only have to introduce a function c for each data constructor $C :: \tau_1 \rightarrow ... \rightarrow \tau_n \rightarrow \tau$.

$$
\begin{aligned}
&c :: (((\alpha\,, \tau_1), ...), \tau_n) \to \mathit{CPS}\ (\alpha, \tau) \\
&c \quad (((st, t_1), ...), t_n) = \mathit{lift} \quad (st, C\ t_1\ ...\ t_n)
\end{aligned}
$$

The modification of the stack is visible in the type: the stack must contain at least n arguments of the correct type before we can apply c. The values are popped from the stack and combined with the constructor C, and the result of this is pushed back on the stack. The type variable α represents the type of the rest of the stack and the variable st represents the value of the rest of the stack.

3.3 Embedding financial contracts

As an example, we show an embedding for a combinator library expressing financial contracts [5]. Simple financial contracts can be represented by the following data type:

$$
\begin{aligned}
\mathbf{data}\ \mathit{Contract} = {} & \mathit{Zero} \\
\mid {} & \mathit{One} \\
\mid {} & \mathit{Give}\ \mathit{Contract} \\
\mid {} & \mathit{Or} \quad \mathit{Contract}\ \mathit{Contract}
\end{aligned}
$$

$Zero$ is a contract without rights and obligations. One is a contract that immediately pays the holder one unit. $Give\ c$ is a contract to acquire all the rights of c as obligations, and vice versa. If you acquire $Or\ c1\ c2$ then you acquire $c1$ or $c2$.

Postfix notation for financial contracts is not the most intuitive DSL for a domain expert, but for us it is a nice starting point.

$$
\mathit{contract} = \mathit{quote\ one\ zero\ or\ give\ end}
$$

The above embedding should evaluate to: $Give\ (Or\ One\ Zero)$. The holder of this contract has the obligation to give one unit or has no obligations. We can now introduce a postfix constructor for every data constructor:

$$
\begin{aligned}
&\mathit{zero} :: \alpha && \to \mathit{CPS}\ (\alpha, \mathit{Contract}) \\
&\mathit{zero} \quad st && = \mathit{lift} \quad (st, \mathit{Zero}) \\
&\mathit{one}\ :: \alpha && \to \mathit{CPS}\ (\alpha, \mathit{Contract}) \\
&\mathit{one} \quad st && = \mathit{lift} \quad (st, \mathit{One}) \\
&\mathit{give} :: (\alpha, \mathit{Contract}) && \to \mathit{CPS}\ (\alpha, \mathit{Contract}) \\
&\mathit{give} \quad (st, c\ \ ) && = \mathit{lift} \quad (st, \mathit{Give}\ c) \\
&\mathit{or} \quad :: ((\alpha, \mathit{Contract}), \mathit{Contract}) && \to \mathit{CPS}\ (\alpha, \mathit{Contract}) \\
&\mathit{or} \quad\ ((st, c1\ \ ), c2\ \ ) && = \mathit{lift} \quad (st, \mathit{Or}\ c1\ c2)
\end{aligned}
$$

The postfix constructors show very nicely, both in the implementation and the type, how the stack is manipulated. The functions $zero$ and one push a value on the stack, $give$ first pops a value and pushes back the result, and or first pops two values from the stack and pushes back the result.

Also the types of the subexpressions correspond very nicely to the stack layout. This ensures that we get a type error if we make an error in postfix notation for financial contracts.

$$
\begin{aligned}
&\mathit{quote} && :: \mathit{CPS}\ () \\
&\mathit{quote\ one} && :: \mathit{CPS}\ ((), \mathit{Contract}) \\
&\mathit{quote\ one\ zero} && :: \mathit{CPS}\ (((), \mathit{Contract}), \mathit{Contract}) \\
&\mathit{quote\ one\ zero\ or} && :: \mathit{CPS}\ ((), \mathit{Contract}) \\
&\mathit{quote\ one\ zero\ or\ give} && :: \mathit{CPS}\ ((), \mathit{Contract}) \\
&\mathit{quote\ one\ zero\ or\ give\ end} && :: \mathit{Contract}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{quote} && \mathit{one}\ \ \mathit{zero\ or} \quad \mathit{give\ end} \\
&\mathit{one}\ \ () && \mathit{zero\ or} \quad \mathit{give\ end}
\end{aligned}
$$

$zero\ ((), One)$ or $give\ end$
$or\ \ (((), One), Zero)$ $give\ end$
$give\ ((), Or\ One\ Zero)\ end$
$end\ \ ((), Give\ (Or\ One\ Zero))$
$Give\ (Or\ One\ Zero)$

In the evaluation steps of this example, the expressions colored red are evaluated in the next step. The state is visualized in the second column and each terminal is applied exactly once to the state, in order of appearance.

4. Prefix notation

4.1 Introduction

Embedding postfix notation in Haskell is possible, but can we do the same for prefix notation? Prefix notation was invented by Jan Lukasiewics, a Polish mathematician. Therefore, it is also known as Polish notation. In this notation a function precedes its arguments. For example, $3\ 5+2*$ is prefix notation for the expression $(3+5)*2$. Similar to postfix notation, it is not needed to write parenthesis in prefix notation, because the arity of functions is statically known.

Have a look at the contract from the last chapter in prefix notation:

$$contract = quote\ give\ or\ zero\ one\ end$$

This contract should evaluate to $Give\ (Or\ Zero\ One)$. In this section we first show a systematic translation of Haskell data constructors to a postfix notation embedding, then we show how the above example is implemented.

4.2 Systematic translation of data types

In postfix notation, a function follows its arguments, so a stack of arguments is a natural choice for the state. But in prefix notation a function precedes its arguments so the state must be a stack of argument requests. We represent a stack of argument requests by a function. To enable prefix notation we first have to redefine the $quote$ and end functions.

$$\begin{aligned} quote &:: CPS\ (\alpha \to \alpha) \\ quote &= lift\ (\lambda a \to a) \\ end &:: \alpha \to \alpha \\ end\ a &= a \end{aligned}$$

The $quote$ function pushes a request for one argument on the stack by lifting the identity function. The end function only stops the parsing process and returns the current state.

The $quote$ and end function are independent from the data type to embed. For writing a data type in prefix notation we only have to introduce a function c for each data constructor $C :: \tau_1 \to ... \to \tau_n \to \tau$.

$$\begin{aligned} c &:: (\tau \to \alpha) \to CPS\ (\tau_1 \to ... \to \tau_n \to \alpha) \\ c\ ctx &= lift\ (\lambda t_1\ ...\ t_n \to ctx\ (C\ t_1\ ...\ t_n)) \end{aligned}$$

The first argument of this prefix constructor is a function that requests something of type τ, also known as a context or expression with a hole. In order to deliver something of type τ, this function pushes argument request on the stack for t_1 till t_n, by extending the function. The type variable α represents the rest of the argument requests.

4.3 Embedding financial contracts

With the redefined quotation functions and the translation scheme for constructors we can embed prefix notation for the data type representing contracts.

$$\begin{aligned} zero &:: (Contract \to \alpha) \to CPS\ \alpha \\ zero\ ctx &= lift\ (ctx\ Zero) \end{aligned}$$

$$
\begin{array}{lll}
one & :: (Contract \to \alpha) \to CPS\ \alpha & \\
one & ctx & = lift\quad (ctx\ One) \\
give & :: (Contract \to \alpha) \to CPS\ (Contract \to \alpha) & \\
give & ctx & = lift\quad (\lambda c \to ctx\ (Give\ c)) \\
or & :: (Contract \to \alpha) \to CPS\ (Contract \to Contract \to \alpha) & \\
or & ctx & = lift\quad (\lambda c1\ c2 \to ctx\ (Or\ c1\ c2))
\end{array}
$$

As we can see from the types, $zero$ and one both reduce the number of pending arguments by one, $give$ leaves the number of pending arguments the same, and or increases the number of arguments by one.

The types of the different subexpressions of our example also show the stack layout. This statically guarantees that the syntax of our embedding is correct.

$$
\begin{array}{ll}
quote & :: CPS\ (\alpha \to \alpha) \\
quote\ give & :: CPS\ (Contract \to Contract) \\
quote\ give\ or & :: CPS\ (Contract \to Contract \to Contract) \\
quote\ give\ or\ zero & :: CPS\ (Contract \to Contract) \\
quote\ give\ or\ zero\ one & :: CPS\ (Contract) \\
quote\ give\ or\ zero\ one\ end & :: Contract
\end{array}
$$

$$
\begin{array}{ll}
quote & give\ or\ \ zero\ one\ end \\
give\ (\lambda c \to c) & or\ \ zero\ one\ end \\
or\ \ (\lambda c \to Give\ c) & zero\ one\ end \\
zero\ (\lambda c1\ c2 \to Give\ (Or\ c1\ c2)) & one\ \ end \\
one\ \ (\lambda c2 \to Give\ (Or\ Zero\ c2)) & end \\
end\ \ (Give\ (Or\ Zero\ One)) & \\
Give\ (Or\ Zero\ One) &
\end{array}
$$

Again the expression is evaluated in the same order as an expression in postfix notation. Only the state, and the functions manipulating the state have changed.

5. Greibach Normal Form

5.1 Introduction

In this section we extend the techniques used for embedding prefix notation to embed languages described by a grammar in Greibach Normal Form. A grammar is in Greibach Normal Form (GNF) if all the productions are of the form: $N \to \alpha B_1 \ldots B_n$, where N is a nonterminal and α is a terminal followed by a possibly empty sequence of nonterminals. A grammar in GNF cannot generate the empty word (ϵ). Every context-free grammar, which does not generate the null string, can be transformed to GNF.

A grammar in GNF is syntactically unambiguous if every pair of productions $N_1 \to \alpha B_1$ and $N_2 \to \beta B_2$ satisfies $(N_1 = N_2 \wedge \alpha = \beta) \Rightarrow B_1 = B_2$. So for each production $N \to \alpha B_1 \ldots B_n$, N together with α must uniquely determine the following sequence of non-terminals ($B_1 \ldots B_n$). If you compare a nonterminal with a data type and a terminal with a constructor, an unambiguous grammar in GNF has the same expressive power as Haskell data type declarations. Unfortunately this is not completely true, because the same terminal may appear in different productions, and in Haskell the same constructor cannot appear in different data type declarations. It appears that we can nicely encode a terminal with multiple occurrences using type classes. We can enforce unambiguity with functional dependencies.

In this section we first show a systematic translation of grammars in GNF to an embedding in Haskell, then we show how an extended syntax for financial contracts is embedded.

$quote\ give\ one\ dollar$
$\quad or\ \ give\ one\ euro$
$\quad or\ \ give\ zero$
end

In the extended syntax, a currency has to be given for the one contract. Also the operator for combining contracts with or is now a left associative infix operator in the embedded language. Note that in the implementation this operator is an ordinary Haskell function.

5.2 Systematic translation

The embedding of a grammar in GNF is very similar to the embedding of prefix notation. The state is a stack of requests for nonterminals (the nonterminals we expect to parse). The stack is represented by nested data types, for each nonterminal: N, a data type: $\mathbf{newtype}\ N\ \alpha = N\ (S \to \alpha)$, is introduced. The type-variable α is the type of the rest of the stack, and S is the type of the requested argument, or in other words the type of the semantic value that parsing the nonterminal N yields.

Because a terminal can occur in multiple productions, a type class must be introduced for each terminal a.

$\mathbf{class}\ A\ old\ new \mid old \to new\ \mathbf{where}$
$\quad a :: old \to CPS\ new$

The functional dependency $old \to new$ is pronounced as: old uniquely determines new. This enforces that the embedded grammar must be unambiguous.

For each production $N \to aB_1 \ldots B_n$, an instance of the earlier introduced class must be defined:

$\mathbf{instance}\ A\ (N\ \alpha)\qquad (B1\ (\qquad ...\quad (Bn\ \alpha)...))\ \mathbf{where}$
$\qquad a\ (N\ ctx) = lift\ (B1\ (\lambda v1 \to ... \to Bn\ (\lambda vn \to ctx\ (f\ v1\ ...\ vn))))$

The meaning of this function is that we can parse the terminal a if non-terminal N is on top of the stack. In order to parse nonterminal N we expect to parse the nonterminals $B1\ ...\ Bn$ and combine the semantic values of parsing $B1\ ...\ Bn$ with the semantic function f. The semantic function f yields the semantic value expected by the context ctx.

If a terminal a occurs only once in the grammar, for each production $N \to aB_1 \ldots B_n$ a function a has to be introduced.

$a :: (N\ \alpha) \quad \to CPS\ (B1\ (\qquad ...\quad (Bn\ \alpha)...))$
$a\quad (N\ ctx) = lift\quad (B1\ (\lambda v1 \to ... \to Bn\ (\lambda vn \to ctx\ (f\ v1\ ...\ vn))))$

The implementation of the function is the same as the overloaded one. The function implicitly satisfies the functional dependency because we can only introduce one function with the name a.

5.3 Embedding financial contracts

The translation scheme for embedding languages described by a grammar in GNF can now be applied to the extended syntax for financial contracts. We first give the grammar of the extended syntax. Because this grammar is not in GNF, we also show the grammar after left-factoring the productions of nonterminal CS.

S	→	quote CS end		S	→	quote C CS
CS	→	C or CS		CS	→	or C CS
	|	C			|	end
C	→	give C	⇒	C	→	give C
	|	one CU			|	one CU
	|	zero			|	zero
CU	→	euro		CU	→	euro
	|	dollar			|	dollar

If we now add the production: C → give CU to the grammar in GNF, the grammar becomes ambiguous. The nonterminal C and the terminal give no longer uniquely determine the following nonterminal, it could either be C or CU. The functional dependency will statically prevent that we embed an ambiguous grammar.

The abstract syntax of the financial combinators is defined by the following data type declarations:

$$
\begin{aligned}
&\mathbf{type}\ \mathit{Contracts} = [\mathit{Contract}] \\
&\mathbf{data}\ \mathit{Contract}\ = \mathit{Give}\ \mathit{Contract} \\
&\qquad\qquad\quad \mid \mathit{One}\ \ \mathit{Currency} \\
&\qquad\qquad\quad \mid \mathit{Zero} \\
&\mathbf{data}\ \mathit{Currency}\ = \mathit{Euro} \mid \mathit{Dollar}
\end{aligned}
$$

The constructors will build the above abstract syntax, but of course other semantic functions can be used. To define more complex semantic functions, an Attributed Grammar (AG) system could be used. The first step we should do now is introduce data types for each nonterminal, and determine the type of the semantic functions:

$$
\begin{aligned}
&\mathbf{newtype}\ S\quad \alpha = S\quad (\mathit{Contracts} \to \alpha) \\
&\mathbf{newtype}\ \mathit{CS}\ \alpha = \mathit{CS}\ (\mathit{Contracts} \to \alpha) \\
&\mathbf{newtype}\ C\quad \alpha = C\quad (\mathit{Contract}\ \to \alpha) \\
&\mathbf{newtype}\ \mathit{CU}\ \alpha = \mathit{CU}\ (\mathit{Currency} \to \alpha)
\end{aligned}
$$

These types can, for example, be read as: in order to parse the nonterminal CU, a value of type $Currency$ is needed.

The second step is to introduce an active terminal function for every production.

$$
\begin{aligned}
&\mathit{quote}\qquad\quad\ \ = \mathit{lift}\ (C\ (\lambda c \to \mathit{CS}\ (\lambda cs \to c : cs))) \\
&\mathit{or}\quad\ (\mathit{CS}\ \mathit{ctx}) = \mathit{lift}\ (C\ (\lambda c \to \mathit{CS}\ (\lambda cs \to \mathit{ctx}\ (c : cs)))) \\
&\mathit{end}\quad (\mathit{CS}\ \mathit{ctx}) = \mathit{ctx}\ [\,] \\
&\mathit{zero}\quad (C\ \mathit{ctx})\ \ = \mathit{lift}\ (\mathit{ctx}\ \mathit{Zero}) \\
&\mathit{one}\quad\ (C\ \mathit{ctx})\ \ = \mathit{lift}\ (\mathit{CU}\ (\lambda \mathit{cur} \to \mathit{ctx}\ (\mathit{One}\ \mathit{cur}))) \\
&\mathit{give}\quad\ (C\ \mathit{ctx})\ \ = \mathit{lift}\ (C\ (\lambda c \to \mathit{ctx}\ (\mathit{Give}\ c))) \\
&\mathit{dollar}\ (\mathit{CU}\ \mathit{ctx}) = \mathit{lift}\ (\mathit{ctx}\ \mathit{Dollar}) \\
&\mathit{euro}\quad\ (\mathit{CU}\ \mathit{ctx}) = \mathit{lift}\ (\mathit{ctx}\ \mathit{Euro})
\end{aligned}
$$

The $quote$ function is the only function without the context parameter, because it both starts the quotation and parses the left-hand side of the S non-terminal. Note that $quote$ can be defined in terms of or. The end function is the only function which doesn't lift the computation into the CPS monad, because it serves as an end mark of the quotation.

6. LL(1) parsing

6.1 Introduction

The next step is to parse a language described by an LL(1) grammar. LL(1) grammars are the class of grammars that can be parsed by an LL(1) parser. An LL(1) parser is a top-down parser that parses the input from left to right and constructs the leftmost derivation. LL grammars are popular, because it is simple to create parser for them by hand. The 1 in LL(1) means that the parser uses one token of look-ahead when parsing a sentence. In this section we explain the architecture of an LL(1) parser, then we give a systematic translation for embedding languages described by an LL(1) grammar. Finally, we embed an even more advanced syntax for financial contracts.

$$
\begin{aligned}
&\mathit{quote} \\
&\quad \mathit{give}\ (\mathit{one}\ \mathit{dollar}\ \mathit{and}\ \mathit{one}\ \mathit{euro}) \\
&\quad \mathit{or}
\end{aligned}
$$

$give\ zero$
end

With this syntax we can combine contracts with left-associative infix and and or operators where and binds stronger then or. Also parentheses can be used to show which contracts should be combined first, or to make the existing priorities explicit.

6.2 Architecture of an LL(1) parser

Using the grammar describing the new syntax for financial contracts, we explain the architecture of an LL(1) parser. The following grammar describes the new syntax, and because this grammar is not LL(1) due to the productions of S and AC, we left factor S and AC. After this operation we get an equivalent grammar which is LL(1).

S	→	AC or S		S	→	AC S’		
	|	AC		S’	→	or AC S’	|	ϵ
AC	→	C and AC		AC	→	C AC’		
	|	C	⇒	AC’	→	and C AC’	|	ϵ
C	→	(S)		C	→	(S)		
	|	give C				give C		
	|	one CU			→	one CU		
	|	zero			|	zero		
CU	→	euro		CU	→	euro		
	|	dollar			|	dollar		

An LL(1) parser needs a parsing table to decide which production it should expand depending on the non-terminal on top of the stack and the next input symbol. To construct a parsing table, we have to know the set of first terminals for each non-terminal, including the empty derivation. If the empty derivation is in the first set of a non-terminal, we also need the follow set of the non-terminal to construct the parsing the table. The follow set is the set of terminals that can follow a non-terminal in any derivation from the start symbol.

The following first- and follow sets are calculated for the grammar of our running example:

first(S)	= {(, give, one, zero}	follow(S)	= {)}
first(S’)	= {or, ϵ}	follow(S’)	= {)}
first(AC)	= {(, give, one, zero}	follow(AC)	= {or,)}
first(AC’)	= {and, ϵ}	follow(AC’)	= {or,)}
first(C)	= {(, give, one, zero}	follow(C)	= {and, or,)}
first(CU)	= {euro, dollar}	follow(CU)	= {and, or,)}

To construct a parsing table for every terminal x in the first set of non-terminal A, a production of the form $A \rightarrow w$ must be filled in at table position $T[A, x]$. If the empty derivation is in the first set of A, also a production for each terminal in the follow-set must be filled in. Applying these rules on the first- and follow sets results in the following parsing table.

	S	S'	AC	AC'	C	CU
or		S'→or AC S'		AC'→ ϵ		
and				AC'→and C AC'		
(	S →AC S'		AC→C AC'		C→(S)	
)		S'→ ϵ		AC'→ ϵ		
give	S →AC S'		AC→C AC'		C→give C	
one	S →AC S'		AC→C AC'		C→one CU	
zero	S →AC S'		AC→C AC'		C→zero	
euro						CU→euro
dollar						CU→dollar
end		S'→ ϵ		AC'→ ϵ		

Before embedding the syntax in Haskell, we first show how an LL(1) parser uses the parsing table to parse a list of input symbols. An LL(1) parser consists of an input buffer, a stack on which it keeps symbols from the grammar, and a parsing table.

The parser starts with pushing the special terminal 'end' on the stack, which indicates the end of the input stream. Next it pushes the start symbol of the grammar on the stack:

Step	Stack	Input buffer
1	S, end	give one euro end
2	AC, S', end	give one euro end
3	C, AC', S', end	give one euro end
4	give, C, AC', S', end	give one euro end
5	C, AC', S', end	one euro end
n	end	end

Then only two operations take place. If the symbol on top of the stack is a nonterminal, an expansion takes place. If it is a terminal, a parse takes place. In the first step, the nonterminal S is on top of the stack, and the symbol 'give' is the next input symbol. The parser looks up the expansion rule in table position T[S, give], and expands S to AC S'. In the second step, AC expands in the same way to C AC', and in the third step C expands to give C. Now the terminal 'give' is popped from the top of the stack and is matched with the next input symbol, if they are the same, the input symbol is parsed. Finally, the string is successfully parsed if the parser pops 'end' from the stack and reads 'end' from the input buffer. If the terminal on top of the stack does not matches the terminal on the input buffer, the parser will report an error and stops. Also if there is no rule in the parsing table for a given symbol non-terminal pair, the parser will stop and report an error.

6.3 Systematic translation

This section describes the systematic translation of an LL(1) grammar to an embedding in Haskell. The state is a stack of pending symbols, both terminal and nonterminal symbols. The stack is again represented by nested data types. For this reason, a data type has to be introduced for each terminal and nonterminal. For each symbol A we introduce a data type: $\mathbf{newtype}\ A\ \alpha = A\ (S \to \alpha)$, where S is the type of the semantic value associated with the symbol. A symbol could have no semantic value if it just needs to be recognized, then the type becomes: $\mathbf{newtype}\ A\ \alpha = A\ \alpha$.

The stack is initialized by the $quote$ function. This function pushes the start symbol on the stack and expects a semantic value, which is left untouched.

$$quote = lift\ (S\ (\lambda x \to x))$$

The terminals must both encode the expansion rules of the parsing table, and the parsing of a terminal. Which action has to take place depends on the symbol on top of the stack. If the symbol on top of the stack is a nonterminal, an expansion has to be performed, otherwise a parsing step has to be taken place. To support

different operations depending on the top of the stack we use type classes. For each terminal we introduce a type-class, and for each operation the terminal must support an instance is introduced. For a terminal t we introduce the following type-class.

$$\textbf{class } CT\ old\ new \mid old \rightarrow new \textbf{ where}$$
$$\quad t :: old \rightarrow CPS\ new$$

The functional dependency $old \rightarrow new$ ensures that we cannot embed an ambiguous grammar. The functional dependency means that the symbol on top of the stack uniquely determines the expansion that has to be taken place.

First we have to encode the parsing of a terminal: we introduce an instance so for each terminal t.

$$\textbf{instance } CT\ (T\ \alpha)\ \alpha \textbf{ where}$$
$$\quad t\ (T\ ctx) = lift\ (ctx\ s)$$

CT is the class we have introduced for the terminal t, T is the data type introduced for terminal t, and s is the semantic value associated with parsing the terminal t. If s is omitted, the parser is just a recognizer.

The second action that must be encoded is the expansion rules of the parsing table. For each rule in the parsing table of the form $N \rightarrow B_1 \ldots B_n$, at table position $T[N, t]$, the following instance is introduced.

$$\textbf{instance } CT\ (N\ \alpha)\ ...\ \textbf{where}$$
$$\quad t\ (N\ ctx) = t\ (B1\ (\lambda v1 \rightarrow ... \rightarrow Bn\ (\lambda vn \rightarrow ctx\ (f\ v1\ ...\ vn))))$$

Note that $B_1 \ldots B_n$ can be both terminals and nonterminals. The meaning of this function is that if the nonterminal N is on top of the stack, the stack is extended with the symbols $B_1 \ldots B_n$ we expect to parse, and combine the values of parsing $B_1 \ldots B_n$ using the semantic function f. The most notable difference with the instance for parsing a symbol is that instead of *lift*, t is applied to the modified stack. This means that we do not parse the next symbol, but keep expanding till the correct terminal is on top of the stack. The type that must be filled in on the place of the dots is not trivial. It is the expansion of N on the stack, after parsing terminal t. This will become more clear if we show a concrete example.

For each rule in the parsing table of the form $N \rightarrow \epsilon$ at table position $T[N, t]$, the following instance is introduced.

$$\textbf{instance } (CT\ \alpha\ \beta) \Rightarrow CT\ (N\ \alpha)\ \beta \textbf{ where}$$
$$\quad t\ (N\ ctx) = t\ (ctx\ s)$$

The meaning of this function is that if nonterminal N is on top of the stack, we apply ctx on the semantic value s. Again, we don't parse the next symbol but keep expanding. Because in this case there are no additional terminals or nonterminals pushed on the stack, the stack layout is not known when applying the overloaded function t. This is the reason why we need to specify the context information $(CT\ \alpha\ \beta)$.

6.4 Embedding financial contracts

To make the translation more concrete we show some snippets of the code to implement the language for financial contracts. The complete implementation can be found on `http://www.cs.uu.nl/wiki/Stc/TypedQuoteAntiquote`. The language will be parsed to the abstract syntax, defined by the following data types:

$$\begin{aligned}\textbf{data } Contract\ = &\ Or\ \ Contract\ Contract\\ \mid &\ And\ \ Contract\ Contract\\ \mid &\ Give\ Contract\\ \mid &\ One\ \ Currency\\ \mid &\ Zero\end{aligned}$$

$$\begin{aligned}&\textbf{data}\ \mathit{Currency} = \mathit{Euro}\\ &\qquad\qquad\quad \mid\ \mathit{Dollar}\end{aligned}$$

The first step is to introduce a data type for each terminal and nonterminal. For the terminal $give$ we introduce the data type: $\textbf{newtype}\ G\ \alpha = G\ \alpha$. Parsing terminal $give$ has no semantic value. For the non-terminals S, AC, and C we introduce the following data types:

$$\begin{aligned}&\textbf{newtype}\ S\quad \alpha = S\quad (\mathit{Contract} \to \alpha)\\ &\textbf{newtype}\ AC\ \alpha = AC\ (\mathit{Contract} \to \alpha)\\ &\textbf{newtype}\ C\quad \alpha = C\quad (\mathit{Contract} \to \alpha)\end{aligned}$$

The parsing of each of the nonterminals yields a value of type $Contract$.

To support the different operations we introduce a type class for each terminal. For example, we introduce for the terminal $give$ the following type class:

$$\begin{aligned}&\textbf{class}\ \mathit{Give}\ \mathit{old}\ \mathit{new} \mid \mathit{old} \to \mathit{new}\ \textbf{where}\\ &\quad \mathit{give} :: \mathit{old} \to \mathit{CPS}\ \mathit{new}\end{aligned}$$

To implement the parse operation we introduce the following instance of the class:

$$\begin{aligned}&\textbf{instance}\ \mathit{Give}\ (G\ \alpha)\ \alpha\ \textbf{where}\\ &\quad \mathit{give}\ (G\ \mathit{ctx}) = \mathit{lift}\ \mathit{ctx}\end{aligned}$$

The instance is used if data type G, the representation of terminal $give$, is on top of the stack, G is popped from the stack, and we proceed by parsing the next terminal.

To explain how the expand action is implemented, recall the state of an LL(1) parser.

Step	Stack	Input buffer
1	S, end	give one euro end
2	AC, S', end	give one euro end
3	C, AC', S', end	give one euro end
4	give, C, AC', S', end	give one euro end
5	C, AC', S', end	one euro end

To implement the expand action, we introduce the following instance for the production S → AC S'.

$$\begin{aligned}&\textbf{instance}\ \mathit{Give}\ (S\ \alpha)\ (C\ (AC'\ (S'\ \alpha)))\ \textbf{where}\\ &\quad \mathit{give}\ (S\ \mathit{ctx}) = \mathit{give}\ (AC\ (\lambda t \to S'\ (\lambda e' \to \mathit{ctx}\ (e'\ t))))\end{aligned}$$

The function performs the expand action, and constructs the semantic value by applying the function that results from parsing S' to the result of parsing AC. The most interesting part of this instance is the type $(C\ (AC'\ (S'\ \alpha)))$: this type can be explained if we look at the stack layout of step 5 in the table above. The type is in fact the remaining expansion of S after parsing the terminal give.

The same holds for the implementation of the productions: $AC \to C\ AC'$ and $C \to give\ C$.

$$\begin{aligned}&\textbf{instance}\ \mathit{Give}\ (AC\ \alpha)\ (C\ (AC'\ \alpha))\ \textbf{where}\\ &\quad \mathit{give}\ (AC\ \mathit{ctx}) = \mathit{give}\ (C\ (\lambda f \to AC'\ (\lambda t' \to \mathit{ctx}\ (t'\ f))))\\ &\textbf{instance}\ \mathit{Give}\ (C\ \alpha)\quad (C\ \alpha)\ \textbf{where}\\ &\quad \mathit{give}\ (C\ \mathit{ctx})\quad = \mathit{give}\ (G\ (C\ (\lambda c \to \mathit{ctx}\ (\mathit{Give}\ c))))\end{aligned}$$

7. Conclusion

We have shown a systematic approach to embed languages in Haskell. All the systematic translations use the basic framework of modeling postfix function application. Although we have not: it is also possible to embed languages based on LR(0) and LR(1) grammars using the same framework. A nice feature of the embeddings is that type checking is used to guarantee that the embedding is syntactically correct. The downside is that

syntactic errors are given as type-errors: a solution would be to script specialized type error messages for an embedding [1]. Another maybe surprising result is that the industrial strength compiler GHC [6] has serious performance problems with compiling quotations which consists of more then 20 terminals.

References

[1] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.

[2] Ralf Hinze. Functional pearl: Typed quote/antiquote. Under consideration for publication in J. Functional Programming.

[3] Paul Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, pages 134–142, Washington, DC, USA, 1998. IEEE Computer Society Press.

[4] Simon L. Peyton Jones. *Haskell 98, Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[5] Simon L. Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: An adventure in financial engineering. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, page 435, London, UK, 2001. Springer-Verlag.

[6] Simon Marlow and Simon L. Peyton Jones. The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.

[7] Chris Okasaki. Techniques for embedding postfix languages in Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 105–113, New York, NY, USA, 2002. ACM Press.