Typed Quote/Antiquote

Embedding languages in Haskell

Gerrit van den Geest

Center for Software Technology, Universiteit Utrecht
http://www.cs.uu.nl/groups/ST/

Based on work of Ralf Hinze and Chris Okasaki

February 9, 2006

Domain Specific Languages

A Domain Specific Language (DSL) is:

- A language for a particular domain
- Counterpart of a general-purpose language
- Not always a programming language
- For example: SQL, shell scripting, HTML, Make

Domain Specific Languages

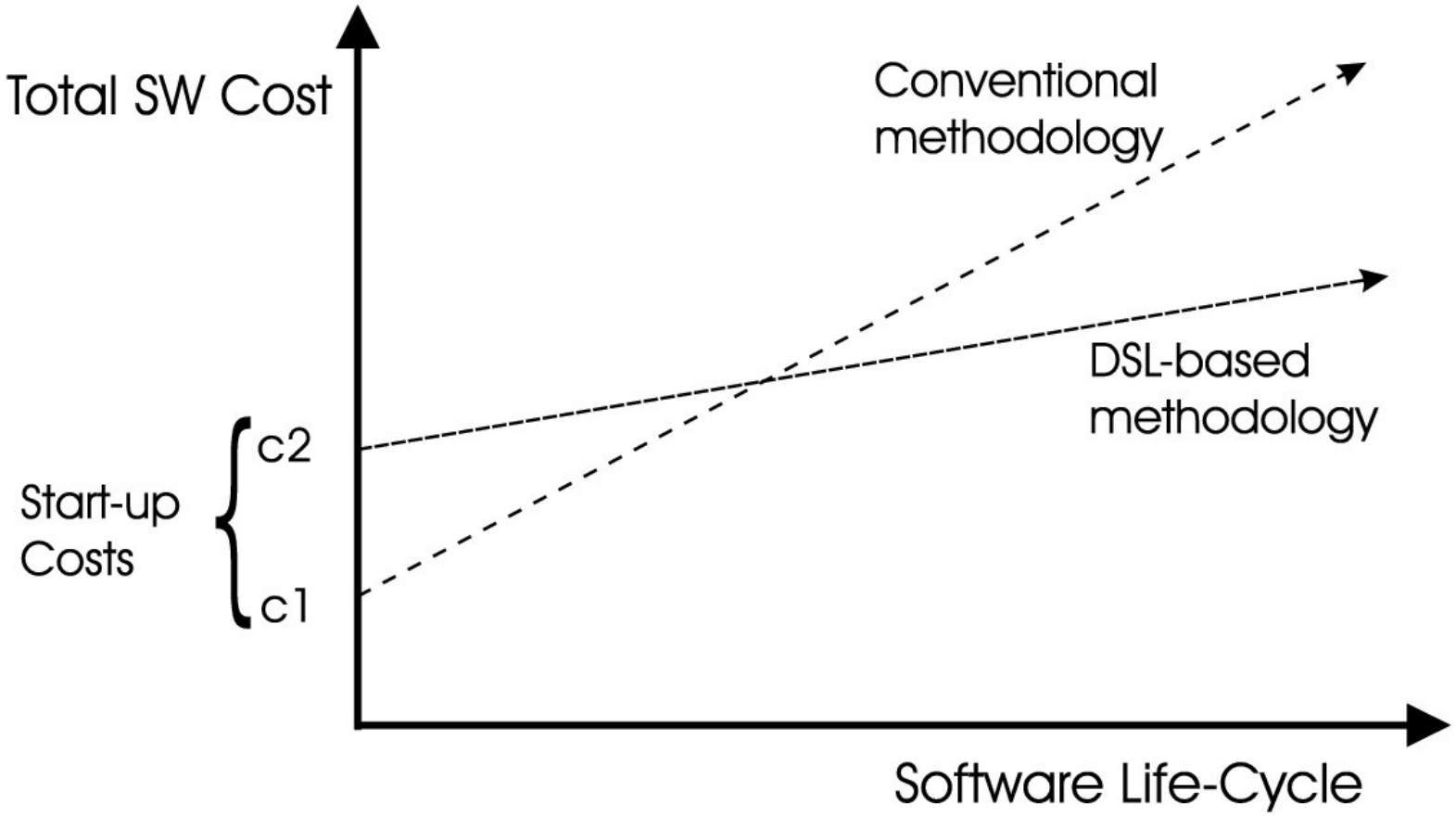
A Domain Specific Language (DSL) is:

- A language for a particular domain
- Counterpart of a general-purpose language
- Not always a programming language
- For example: SQL, shell scripting, HTML, Make

Programs written in a DSL are:

- Easier to write
- Easier to reason about
- Easier to learn for a domain expert
- Easier to modify

The payoff of DSL technology



Domain Specific Embedded Languages (DSELs)

Inherit the infrastructure of another language

- Macro expansion of DSL to another language
- Generating software from a specification in a DSL
- Embed a DSL in another language

Domain Specific Embedded Languages (DSELs)

Inherit the infrastructure of another language

- Macro expansion of DSL to another language
- Generating software from a specification in a DSL
- Embed a DSL in another language

Embed a DSL in Haskell

- Currying
- Higher-order functions
- Polymorphism
- Type classes (overloading)

Embedding concrete syntax in Haskell

Concrete syntax for natural number

$$
\begin{aligned}
test = \; & quote \\
& \quad tick \\
& \quad tick \\
& \quad tick \\
& end \\
& + \\
& quote \\
& \quad tick \\
& end
\end{aligned}
$$

How can we implement this?

How we going to implement this?

Function application is left-assiociative

$$(((quote\ tick)\ tick)\ tick)\ end$$

How we going to implement this?

Function application is left-assiociative

$$(((quote\ tick)\ tick)\ tick)\ end$$

If Haskell had postfix function application...

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

Postfix function application

Introduce an postfix operator

$$(\triangleleft) :: \alpha \to (\alpha \to r) \to r$$

$$x \triangleleft f = f\ x$$

$$test = quote \triangleleft tick \triangleleft tick \triangleleft tick \triangleleft end$$

Postfix function application

Introduce an postfix operator

$$(\triangleleft) :: \alpha \to (\alpha \to r) \to r$$
$$x \triangleleft f = f\ x$$
$$\mathit{test} = \mathit{quote} \triangleleft \mathit{tick} \triangleleft \mathit{tick} \triangleleft \mathit{tick} \triangleleft \mathit{end}$$

Pushing the operator into the terminals

$$\begin{aligned}
\mathit{quote} &:: \quad (\mathit{Int} \to r) \to r \\
\mathit{quote} &\quad = (\triangleleft)\ 0 \\
\mathit{tick} &:: \mathit{Int} \to (\mathit{Int} \to r) \to r \\
\mathit{tick}\ \ i &\quad = (\triangleleft)\ (i + 1) \\
\mathit{end} &:: \mathit{Int} \to \mathit{Int} \\
\mathit{end}\ \ i &\quad = i
\end{aligned}$$

Finishing touch

The CPS monad

$$
\begin{aligned}
&\textbf{type}\ \mathit{CPS}\ \alpha = \forall\ r.(\alpha \to r) \to r \\
&\mathit{lift} :: \alpha \to \quad \mathit{CPS}\ \alpha \\
&\mathit{lift}\quad a\ = \quad \lambda f \to f\ a \\
&\mathit{quote} :: \quad \mathit{CPS}\ \mathit{Int} \\
&\mathit{quote} \quad = \mathit{lift}\quad 0 \\
&\mathit{tick} :: \mathit{Int} \to \mathit{CPS}\ \mathit{Int} \\
&\mathit{tick}\quad i \quad = \mathit{lift}\quad (i+1) \\
&\mathit{end} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{end}\quad i \quad = i \\
&\mathit{test} = \mathit{quote}\ \mathit{tick}\ \mathit{tick}\ \mathit{tick}\ \mathit{end}
\end{aligned}
$$

Evaluation steps

The quotation is evaluated like this

$$
\begin{aligned}
\mathit{quote} &= \mathit{lift}\ 0 \\
\mathit{tick}\ i &= \mathit{lift}\ (i+1) \\
\mathit{end}\ i &= i
\end{aligned}
$$

$$
\begin{array}{lll}
\mathit{quote} & & \mathit{tick}\ \mathit{tick}\ \mathit{tick}\ \mathit{end} \\
\mathit{lift}\ \ 0 & \mathit{tick} & \mathit{tick}\ \mathit{tick}\ \mathit{end} \\
\mathit{tick}\ 0 & & \mathit{tick}\ \mathit{tick}\ \mathit{end} \\
\mathit{lift}\ \ (0+1) & \mathit{tick} & \mathit{tick}\ \mathit{end} \\
\mathit{tick}\ 1 & & \mathit{tick}\ \mathit{end} \\
\mathit{lift}\ \ (1+1) & \mathit{tick} & \mathit{end} \\
\mathit{tick}\ 2 & & \mathit{end} \\
\mathit{lift}\ \ (2+1) & \mathit{end} & \\
\mathit{end}\ 3 & & \\
3 & &
\end{array}
$$

Ready for the next step

What we have

- Framework for postfix function application
- Threading a state through the terminals

Ready for the next step

What we have

- Framework for postfix function application
- Threading a state through the terminals

The next step

- Embed postfix notation
- Embed prefix notation

Postfix and Prefix notation

Postfix notation

- Functions follow their arguments
- 3 5 + 2 *
- Also known as Reverse Polish Notation (RPN)
- Stack-based implementation

Postfix and Prefix notation

Postfix notation

- Functions follow their arguments
- 3 5 + 2 *
- Also known as Reverse Polish Notation (RPN)
- Stack-based implementation

Prefix notation

- Functions precedes their arguments
- * + 3 5 2
- Also known as Polish notation
- Haskell data contructors

Example

Financial Combinators (S. Peyton-Jones et al.)

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

Example

Financial Combinators (S. Peyton-Jones et al.)

$$
\begin{array}{l}
\textbf{data}\ \mathit{Contract} = \mathit{Zero} \\
\qquad\qquad\qquad\quad \mid\ \mathit{One} \\
\qquad\qquad\qquad\quad \mid\ \mathit{Give}\ \mathit{Contract} \\
\qquad\qquad\qquad\quad \mid\ \mathit{Or}\quad \mathit{Contract}\ \mathit{Contract}
\end{array}
$$

Concrete syntax

$$
\begin{array}{l}
\mathit{contract} = \mathit{quote} \\
\qquad\qquad\qquad \mathit{one\ zero\ or\ give} \\
\qquad\qquad\quad \mathit{end}
\end{array}
$$

Should evaluate to: $\mathit{Give}\ (\mathit{Or}\ \mathit{One}\ \mathit{Zero})$

Systematic translation

Redefine quotation

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

Systematic translation

Redefine quotation

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

Translation of constructors

Introduce for each contructor $C :: \tau_1 \to \ldots \to \tau_n \to \tau$, a postfix function (terminal) c:

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

Example translation

Example constructors

$$
\begin{aligned}
&\mathit{Zero}, \mathit{One} &&:: \mathit{Contract} \\
&\mathit{Give} &&:: \mathit{Contract} \rightarrow \mathit{Contract} \\
&\mathit{Or} &&:: \mathit{Contract} \rightarrow \mathit{Contract} \rightarrow \mathit{Contract}
\end{aligned}
$$

Translated postfix functions (terminals)

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

Concrete example, types

The types correspond to the stack layout

$$
\begin{array}{lll}
\mathit{quote} & :: & \mathit{CPS}\ () \\
\mathit{zero}, \mathit{one} & :: \mathit{st} & \rightarrow \mathit{CPS}\ (\mathit{st}, \mathit{Contract}) \\
\mathit{give} & :: (\mathit{st}, \mathit{Contract}\ ) & \rightarrow \mathit{CPS}\ (\mathit{st}, \mathit{Contract}) \\
\mathit{or} & :: ((\mathit{st}, \mathit{Contract}), \mathit{Contract}) & \rightarrow \mathit{CPS}\ (\mathit{st}, \mathit{Contract}) \\
\mathit{end} & :: ((), \alpha) & \rightarrow \alpha
\end{array}
$$

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

Concrete example, evaluation

Evaluation steps

$$
\begin{aligned}
&\mathit{quote} && = \mathit{lift}\ () \\
&\mathit{zero}\ \mathit{st} && = \mathit{lift}\ (\mathit{st}, \mathit{Zero}) \\
&\mathit{one}\ \ \mathit{st} && = \mathit{lift}\ (\mathit{st}, \mathit{One}) \\
&\mathit{give}\ (\mathit{st}, c) && = \mathit{lift}\ (\mathit{st}, \mathit{Give}\ c) \\
&\mathit{or}\ \ \ ((\mathit{st}, c1), c2) && = \mathit{lift}\ (\mathit{st}, \mathit{Or}\ c1\ c2) \\
&\mathit{end}\ \ ((), a) && = a
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{quote} && \mathit{one}\ \ \mathit{zero}\ \mathit{or}\ \ \ \mathit{give}\ \mathit{end} \\
&\mathit{one}\ \ () && \mathit{zero}\ \mathit{or}\ \ \ \mathit{give}\ \mathit{end} \\
&\mathit{zero}\ ((), \mathit{One}) && \mathit{or}\ \ \ \mathit{give}\ \mathit{end} \\
&\mathit{or}\ \ \ ((((), \mathit{One}), \mathit{Zero}) && \mathit{give}\ \mathit{end} \\
&\mathit{give}\ ((), \mathit{Or}\ \ \mathit{One}\ \ \mathit{Zero}) && \mathit{end} \\
&\mathit{end}\ \ ((), \mathit{Give}\ (\mathit{Or}\ \ \mathit{One}\ \ \mathit{Zero})) \\
&\mathit{Give}\ (\mathit{Or}\ \ \mathit{One}\ \ \mathit{Zero})
\end{aligned}
$$

Prefix notation

Embedding prefix notation

- Systematic translation for embedding data types
- State is a stack of pending arguments
- Stack is represented by a function
- Running example: Financial Combinators

Prefix notation

Embedding prefix notation

- Systematic translation for embedding data types
- State is a stack of pending arguments
- Stack is represented by a function
- Running example: Financial Combinators

Concrete syntax

$$\begin{aligned} \mathit{contract} = \mathit{quote} & \\ & \quad \mathit{give\ or\ zero\ one} \\ & \mathit{end} \end{aligned}$$

Should evaluate to: *Give* (*Or Zero One*)

Systematic translation

Redefine quotation

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

Systematic translation

Redefine quotation

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

Translation of constructors

Introduce for each contructor $C :: \tau_1 \to \ldots \to \tau_n \to \tau$, a prefix function (terminal) c:

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

Example translation

Example constructors

$$
\begin{aligned}
&\mathit{Zero}, \mathit{One} :: \mathit{Contract} \\
&\mathit{Give} \quad\quad :: \mathit{Contract} \to \mathit{Contract} \\
&\mathit{Or} \quad\quad\quad :: \mathit{Contract} \to \mathit{Contract} \to \mathit{Contract}
\end{aligned}
$$

Translated prefix functions (terminals)

$$
\begin{aligned}
&\mathit{zero} :: (\mathit{Contract} \to a) \to \mathit{CPS}\ a \\
&\mathit{zero} \quad \mathit{ctx} \quad\quad\quad\quad\quad = \mathit{lift} \quad (\mathit{ctx}\ \mathit{Zero}) \\
&\mathit{give} :: (\mathit{Contract} \to a) \to \mathit{CPS}\ (\mathit{Contract} \to a) \\
&\mathit{give} \quad \mathit{ctx} \quad\quad\quad\quad\quad = \mathit{lift} \quad (\lambda c \to \mathit{ctx}\ (\mathit{Give}\ c)) \\
&\mathit{or} \quad :: (\mathit{Contract} \to a) \to \mathit{CPS}\ (\mathit{Contract} \to \mathit{Contract} \to a) \\
&\mathit{or} \quad\quad \mathit{ctx} \quad\quad\quad\quad\quad = \mathit{lift} \quad (\lambda c1\ c2 \to \mathit{ctx}\ (\mathit{Or}\ c1\ c2))
\end{aligned}
$$

Concrete example, types

The types correspond to the stack layout

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

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

Concrete example, evaluation

Evaluation steps

$$
\begin{aligned}
\mathit{quote} &= \mathit{lift}\ \mathit{id} \\
\mathit{zero}\ \mathit{ctx} &= \mathit{lift}\ (\mathit{ctx}\ \mathit{Zero}) \\
\mathit{one}\ \ \mathit{ctx} &= \mathit{lift}\ (\mathit{ctx}\ \mathit{One}) \\
\mathit{give}\ \mathit{ctx} &= \mathit{lift}\ (\lambda c \rightarrow \mathit{ctx}\ (\mathit{Give}\ c)) \\
\mathit{or}\ \ \ \mathit{ctx} &= \mathit{lift}\ (\lambda c1\ c2 \rightarrow \mathit{ctx}\ (\mathit{Or}\ c1\ c2)) \\
\mathit{end}\ \ a &= a
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{quote} \\
&\mathit{give}\ (\lambda c \rightarrow c) \\
&\mathit{or}\ \ \ (\lambda c \rightarrow \mathit{Give}\ c) \\
&\mathit{zero}\ (\lambda c1\ c2 \rightarrow \mathit{Give}\ (\mathit{Or}\ c1\ c2)) \\
&\mathit{one}\ \ (\lambda c2 \rightarrow \mathit{Give}\ (\mathit{Or}\ \mathit{Zero}\ c2)) \\
&\mathit{end}\ \ (\mathit{Give}\ (\mathit{Or}\ \mathit{Zero}\ \mathit{One})) \\
&\mathit{Give}\ (\mathit{Or}\ \mathit{Zero}\ \mathit{One})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{give}\ \mathit{or}\ \ \ \mathit{zero}\ \mathit{one}\ \mathit{end} \\
&\mathit{or}\ \ \ \mathit{zero}\ \mathit{one}\ \ \mathit{end} \\
&\mathit{zero}\ \mathit{one}\ \ \mathit{end} \\
&\mathit{one}\ \ \mathit{end} \\
&\mathit{end}
\end{aligned}
$$

Ready for the real work

Embed languages described by context-free grammars

- Grammars in Greibach Normal Form (GNF)
- LL(1) grammars
- LR(0) grammars

Ready for the real work

Embed languages described by context-free grammars

- Grammars in Greibach Normal Form (GNF)
- LL(1) grammars
- LR(0) grammars

A grammars is in GNF if:

- All production are of the form $N \rightarrow aB_1 \ldots B_n$
- It cannot generate the empty word (ϵ).

Example syntax

New syntax for Financial Combinators

quote give one dollar or give one euro
or give zero
end

Abstract syntax

$$
\begin{array}{lll}
\textbf{type}\ \mathit{Contracts} & = & [\mathit{Contract}] \\
\textbf{data}\ \mathit{Contract} & = & \mathit{Give}\ \mathit{Contract} \\
 & \mid & \mathit{One}\ \mathit{Currency} \\
 & \mid & \mathit{Zero} \\
\textbf{data}\ \mathit{Currency} & = & \mathit{Euro} \mid \mathit{Dollar}
\end{array}
$$

Corresponding grammar

Grammar and equivalent grammar in GNF

S	$\longrightarrow$	quote CS end
CS	$\longrightarrow$	C or CS
	$\mid$	C
C	$\longrightarrow$	give C
	$\mid$	one CU
	$\mid$	zero
CU	$\longrightarrow$	euro
	$\mid$	dollar

Corresponding grammar

Grammar and equivalent grammar in GNF

S	$\rightarrow$	quote CS end		S	$\rightarrow$	quote C CS
CS	$\rightarrow$	C or CS		CS	$\rightarrow$	or C CS
	$\mid$	C			$\mid$	end
C	$\rightarrow$	give C	$\Rightarrow$	C	$\rightarrow$	give C
	$\mid$	one CU			$\mid$	one CU
	$\mid$	zero			$\mid$	zero
CU	$\rightarrow$	euro		CU	$\rightarrow$	euro
	$\mid$	dollar			$\mid$	dollar

Systematic translation, step 1

A datatype for each non-terminal: N

$$\textbf{newtype}\ N\ \alpha = N\ (S \to \alpha)$$

Where S is the type of the semantic value.

Systematic translation, step 1

A datatype for each non-terminal: N

$$\textbf{newtype}\ N\ \alpha = N\ (S \to \alpha)$$

Where S is the type of the semantic value.

Datatypes for the example grammar

$$\begin{array}{lll}
\textbf{newtype}\ S & a = S & (Contracts \to a) \\
\textbf{newtype}\ CS & a = CS & (Contracts \to a) \\
\textbf{newtype}\ C & a = C & (Contract \to a) \\
\textbf{newtype}\ CU & a = CU & (Currency \to a)
\end{array}$$

Systematic translation, step 2

For each production: $N \rightarrow aB_1 \ldots B_n$

We introduce the following function:

$$
\begin{array}{lll}
a :: (N\ \alpha) & \rightarrow CPS\ (B_1\ (& \ldots \quad (B_n\ \alpha)\ldots)) \\
a\ \ (N\ ctx) = lift & (B_1\ (\lambda v1 \rightarrow \ldots \rightarrow B_n\ (\lambda vn \rightarrow ctx\ (f\ v1\ \ldots\ vn)))) &
\end{array}
$$

Where f is the semantic function.

Example translation

Productions of the example grammar

S	$\rightarrow$	quote C CS
CS	$\rightarrow$	or C CS
	$\mid$	end
C	$\rightarrow$	give C
	$\mid$	zero

Translated parsing functions (terminals)

$$
\begin{aligned}
&\mathit{quote} && = \mathit{lift}\ (C\ (\lambda c \rightarrow CS\ (\lambda cs \rightarrow c : cs))) \\
&\mathit{or}\ \ (CS\ \mathit{ctx}) && = \mathit{lift}\ (C\ (\lambda c \rightarrow CS\ (\lambda cs \rightarrow \mathit{ctx}\ (c : cs)))) \\
&\mathit{end}\ \ (CS\ \mathit{ctx}) && = \mathit{ctx}\ [\,] \\
&\mathit{give}\ (C\ \mathit{ctx}) && = \mathit{lift}\ (C\ (\lambda c \rightarrow \mathit{ctx}\ (\mathit{Give}\ c))) \\
&\mathit{zero}\ (C\ \mathit{ctx}) && = \mathit{lift}\ (\mathit{ctx}\ \mathit{Zero})
\end{aligned}
$$

LL(1) grammars

Syntax we want to embed

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

Abstract syntax

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

Corresponding grammar

Grammar and equivalent LL(1) grammar

S	$\rightarrow$	AC or S
	|	AC
AC	$\rightarrow$	C and AC
	|	C
C	$\rightarrow$	(S)
	|	give C
	|	one CU
	|	zero
CU	$\rightarrow$	euro
	|	dollar

Corresponding grammar

Grammar and equivalent LL(1) grammar

S	$\rightarrow$	AC or S		S	$\rightarrow$	AC S'	
	|	AC		S'	$\rightarrow$	or AC S'	| ϵ
AC	$\rightarrow$	C and AC		AC	$\rightarrow$	C AC'	
	|	C	$\Rightarrow$	AC'	$\rightarrow$	and C AC'	| ϵ
C	$\rightarrow$	(S)		C	$\rightarrow$	(S)	
	|	give C				give C	
	|	one CU			$\rightarrow$	one CU	
	|	zero			|	zero	
CU	$\rightarrow$	euro		CU	$\rightarrow$	euro	
	|	dollar			|	dollar	

Architecture of an LL parser

The parsing table

	S	AC	C	CU
give	S →AC S'	AC→C AC'	C→give C	
one	S →AC S'	AC→C AC'	C→one CU	
euro				CU→euro

Architecture of an LL parser

The parsing table

	S	AC	C	CU
give	S →AC S'	AC→C AC'	C→give C	
one	S →AC S'	AC→C AC'	C→one CU	
euro				CU→euro

The parse process

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

Translation to Haskell

The state

- The state is a stack of pending symbols
- Represent a symbol by a data type
- *quote* pushes the start symbol on the stack
- $quote = lift\ (S\ (\lambda x \rightarrow x))$

Translation to Haskell

The state

- The state is a stack of pending symbols
- Represent a symbol by a data type
- *quote* pushes the start symbol on the stack
- $quote = lift\ (S\ (\lambda x \rightarrow x))$

Some of the data types

For the terminal *give*: **newtype** $G\ a = G\ a$
For the non-terminals S, AC, and C

$$\begin{array}{lll} \textbf{newtype}\ S & a = S & (Contract \rightarrow a) \\ \textbf{newtype}\ AC & a = AC & (Contract \rightarrow a) \\ \textbf{newtype}\ C & a = C & (Contract \rightarrow a) \end{array}$$

Translation to Haskell

Terminals must both

- Encode the parsing of a terminal
- Encode the expansion rules of the parsing table
- Overloading!

Translation to Haskell

Terminals must both

- Encode the parsing of a terminal
- Encode the expansion rules of the parsing table
- Overloading!

Recognition of *give*

Introduce a class for *give*

$$\textbf{class}\ \mathit{Give}\ \mathit{old}\ \mathit{new} \mid \mathit{old} \rightarrow \mathit{new}\ \textbf{where}$$
$$\quad \mathit{give} :: \mathit{old} \rightarrow \mathit{CPS}\ \mathit{new}$$

And an instance, parsing *give*

$$\textbf{instance}\ \mathit{Give}\ (\mathit{G}\ a)\ a\ \textbf{where}$$
$$\quad \mathit{give}\ (\mathit{G}\ \mathit{ctx}) = \mathit{lift}\ \mathit{ctx}$$

Encoding of the expansion rules

Entries in the parsing table

	S	AC	C
give	S →AC S'	AC→C AC'	C→give C

Translated instance declarations

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Give}\ (S\ a) \quad (C\ (AC'\ (S'\ a)))\ \textbf{where} \\
\quad \mathit{give}\ (S\ \mathit{ctx}) \quad = \mathit{give}\ (AC\ (\lambda t \rightarrow S'\ (\lambda e' \rightarrow \mathit{ctx}\ (e'\ t)))) \\
\textbf{instance}\ \mathit{Give}\ (AC\ a)\ (C\ (AC'\ a))\ \textbf{where} \\
\quad \mathit{give}\ (AC\ \mathit{ctx}) = \mathit{give}\ (C\ (\lambda f \rightarrow AC'\ (\lambda t' \rightarrow \mathit{ctx}\ (t'\ f)))) \\
\textbf{instance}\ \mathit{Give}\ (C\ a) \quad (C\ a)\ \textbf{where} \\
\quad \mathit{give}\ (C\ \mathit{ctx}) \quad = \mathit{give}\ (G\ (C\ (\lambda c \rightarrow \mathit{ctx}\ (\mathit{Give}\ c))))
\end{array}
$$

Conclusion

What we have done

- Developed a postfix framework
- Embedded postfix and prefix notation
- Embedded languages described by LL(1) and GNF grammars
- Type-checker guarantees correct syntax embedding

Conclusion

What we have done

- Developed a postfix framework
- Embedded postfix and prefix notation
- Embedded languages described by LL(1) and GNF grammars
- Type-checker guarantees correct syntax embedding

Practical concerns

- Type-error messages are unreadable
- Haskell compilers are not designed for this
- Lexical syntax of functions limits the concrete syntax