Typed Quotation / Antiquotation in Haskell
Or: Compile-time parsing

Ralf Hinze

Presented by Beerend Lauwers
Utrecht University, The Netherlands

January 9, 2013

Outline

Quotation: short introduction

- **Quotation** allows you to integrate a guest language into a host language:

```
Main⟩ ≪ fork fork leaf leaf leaf ≫
Fork (Fork Leaf Leaf) Leaf
Main⟩ size ≪ fork fork leaf leaf leaf ≫ + 1
4
```

- A quotation is surrounded by guillemets: ≪ ≫
- A quotation evaluates to an abstract syntax.
- Great for EDSLs!

Anti-quotation: short introduction

- With **anti-quotation**, we can use the host language in the guest language:

```
Main⟩ ≪ fork `(full 2) leaf ≫
Fork (Fork (Fork Leaf Leaf) (Fork Leaf Leaf))
    Leaf
```

- An anti-quotation begins with a backtick and usually evaluates to an abstract syntax.

Quotation/Anti-quotation: implementation in GHC

- The language needs to be extended to support quotation/anti-quotation.
- GHC provides the **Quasiquotation** extension.
- This extension uses Template Haskell, which suffers from typing issues.

Could we implement this functionality in plain Haskell?

Quotation/Anti-quotation: implementation in GHC

- The language needs to be extended to support quotation/anti-quotation.
- GHC provides the **Quasiquotation** extension.
- This extension uses Template Haskell, which suffers from typing issues.

Could we implement this functionality in plain Haskell?
Indeed, we can. That's what this paper's about.

Quotation/Anti-quotation: implementation in GHC

- The language needs to be extended to support quotation/anti-quotation.
- GHC provides the **Quasiquotation** extension.
- This extension uses Template Haskell, which suffers from typing issues.

Could we implement this functionality in plain Haskell?
Indeed, we can. That's what this paper's about.

Assumption: arbitrary terminal symbols in `typewriter` font are Haskell identifiers (!)

Example: DSL for natural numbers

- Simple example: a DSL for natural numbers:

```
Main⟩ (≪ | | | ≫,≪ | | | | | ≫ + 7)
(3,12)
```

- Let's define some aliases to help us decipher this:
  - $\ll$ = quote
  - $\gg$ = endquote
  - $|$ = tick
- Then,

```
≪ | | | ≫
```

becomes

```
(((quote tick) tick) tick) endquote
```

Example: DSL for natural numbers

```
(((quote tick) tick) tick) endquote
```

- How can we evaluate this?

```
quote f = f 0
tick n f = f (succ n)
endquote n = n
```

- Stepwise evaluation:

```
quote tick tick tick endquote
= tick 0 tick tick endquote
= tick 1 tick endquote
= tick 2 endquote
= endquote 3
= 3
```

- Evaluation is driven by terminal symbols = **active terminals**.

The CPS Monad

- This looks a lot like **continuation-passing style (CPS)**...
- Indeed, we can make it an instance of the CPS monad:

```
type CPS α = ∀ans.(α → ans) → ans
instance Monad CPS where
  return a = λκ → κ a
  m >>= k   = λκ → m (λa → k a κ)
```

- Then,

```
quote = return 0
= λκ → κ 0
-- Original: quote f = f 0)
tick = lift succ = λa → return (succ a)
= λa → λκ → κ (succ a)
-- tick n f = f (succ n))
```

The CPS Monad

- This looks a lot like **continuation-passing style (CPS)**...
- Indeed, we can make it an instance of the CPS monad:

```
type CPS α = ∀ans.(α →ans)→ans
instance Monad CPS where
   return a = λκ → κ a
  m ≫= k    = m k
```

- Then,

```
quote = return 0
= λκ → κ 0
-- Original: quote f = f 0)
tick = lift succ = λa→return (succ a)
= λa→ λκ → κ (succ a)
-- tick n f = f (succ n))
```

The CPS Monad

- So, the quotation

```
≪ | | | ≫
```

can be written as a monadic computation:

```
run (quote ≫= tick ≫= tick ≫= tick)
```

- (run encapsulates a CPS computation:)

```
run :: CPS α → α
run m = m id
```

The CPS Monad

- Generalizing:

```
(((quote act₁) ...) actₙ) endquote
=
run (quote >>= act₁ >>= ... >>= actₙ)
```

where

$$\begin{aligned}\mathtt{quote} &:: \mathrm{CPS}\tau_1\\ \mathtt{act}_i &:: \tau_i \to \mathrm{CPS}\tau_{i+1}\\ \mathtt{endquote} &= \mathbf{id}\end{aligned}$$

- In this example, there was only a single state type.
- Choosing your state types carefully is very important, as we shall see later on.
- Note: endquote can be any function, not just the identity:

```
postProcess (run m)
```

Next up: parsers in CPS monad

- So, evaluation of a quotation is driven by the terminals.
- For a specific guest language, we need a specific parser for the concrete syntax.
- We'll look at three types of parsers, ordered by complexity:
  - Simple postfix and prefix parsers
  - Predictive top-down parsers
  - Bottom-up parsers

Postfix: refresher course

- Postfix = Reverse Polish Notation (RPN)
- Arguments first, function call last.
- No need for parentheses if arity of functions is known.
- Not generally the case for higher-order languages, but it is for **data** constructors.
- Usually a stack-based implementation.

Postfix: implementation in Haskell

- We'll also use a stack.
- For each data constructor, we generate a **postfix constructor**.
- For example:

$$C \;::\; \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$$

generates

$$\begin{array}{lll} c & :: & (((\,st\,,\;\tau_1\,)\,,\;\ldots\,)\,,\;\tau_n\,) \rightarrow (\,st\,,\;\tau\;) \\ c & & (((\,st\,,\;t_1\,)\,,\;\ldots\,)\,,\;t_n\,) = (\,st\,,\;\mathbf{C}\;t_1\;\ldots\;t_n\,) \end{array}$$

- Stack is a nested pair, grows from left to right.
- We pop arguments from the stack and push the result back on.

Postfix: implementation in Haskell - Example

- Let's apply this to the Tree datatype.

```
leaf :: st → (st,Tree)
leaf st = (st, Leaf)

fork :: ((st,Tree),Tree) → (st,Tree)
fork (st,l),r) = (st, Fork l r)
```

- Now, we can use these definitions in the evaluation framework:

```
quote :: CPS ()
quote = return ()
leaf :: st → CPS (st,Tree)
leaf = lift leaf
fork :: ((st,Tree),Tree) → CPS (st,Tree)
fork = lift fork
endquote :: ((),Tree) → Tree
endquote ((),t) = t
```

Postfix: implementation in Haskell - Example

- Static type checking example:

```
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
```

- Note how the state type mirrors the stack layout.
- It's easy to see how the wrong amount of arguments will result in a type error.

Postfix: implementation in Haskell - Example

- Dynamic evaluation example:

```
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
```

Prefix: implementation in Haskell

- Function call first, then its arguments.
- We'll use a stack of pending arguments, growing from right to left.
- For each data constructor, we generate a **prefix constructor**.
- For example:

$$C \;::\; \tau_1 \to \ldots \to \tau_n \to \tau$$

generates

$$c^\circ \;::\; ((\tau, st) \to \alpha) \to ((\tau_1, (\ldots, (\tau_n, st))) \to \alpha)$$
$$c^\circ \; ctx = \lambda(t_1, (\ldots, (t_n, st))) \to ctx (C \; t_1 \; \ldots \; t_n, st)$$

- The first argument, **ctx**, can be seen as a request for a type τ.
- However, this request may generate new requests: those for its arguments.

Prefix: implementation in Haskell - Example

- Let's apply this to the Tree datatype.

```
leaf° :: ((Tree, st)→ α) → (st→ α)
leaf° ctx = λst → ctx(Leaf, st)

fork° :: ((Tree, st)→ α) → ((Tree,(Tree, st))→ α)
fork° ctx = λ(t,(u, st)) → ctx(Fork t u, st)
```

- Now, we can use these definitions in the evaluation framework:

```
quote :: CPS (Tree,())→Tree)
quote = return (λ(t,())→t)
leaf :: ((Tree, st)→ α)→ CPS (st→ α)
leaf = lift leaf°
fork :: ((Tree, st)→ α)→ CPS ((Tree,(Tree, st))
    → α)
fork = lift fork°
endquote :: (()→Tree)→Tree
endquote ctx = ctx ()
```

Prefix: implementation in Haskell - Example

- Static type checking example:

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

- The stack is initialized to a single pending argument. If there are no arguments left, we're done.
- Note how, in the type, the stack of pending arguments grows and shrinks as the quotation is expanded.

Prefix: implementation in Haskell - Example

$$\begin{array}{l} \mathtt{fork}^{\circ}\ \mathtt{ctx} = \lambda(\mathtt{t},(\mathtt{u},\mathtt{st})) \rightarrow \mathtt{ctx}(\mathbf{Fork}\ \mathtt{t}\ \mathtt{u},\mathtt{st}) \\ \mathtt{leaf}^{\circ}\ \mathtt{ctx} = \lambda\mathtt{st} \rightarrow \mathtt{ctx}(\mathbf{Leaf},\mathtt{st}) \end{array}$$

- Dynamic evaluation example:

```
quote fork fork leaf leaf leaf endquote
= fork (λ(t,())→t) fork leaf leaf leaf endquote
= fork (λ(t,(u,()))→Fork t u) leaf leaf leaf endquote
= (λ(t',(u',st)) → (λ(t,(u,()))→Fork t u)(Fork t'
    u',st)) leaf leaf leaf endquote
= leaf (λ(t',(u',(u,()))))→Fork (Fork t' u') u) leaf
    leaf endquote
= leaf (λ(u',(u,()))→Fork (Fork Leaf u') u) leaf
    endquote
= leaf (λ(u,())→Fork (Fork Leaf Leaf) u) endquote
= endquote (λ()→Fork (Fork Leaf Leaf) Leaf)
= Fork (Fork Leaf Leaf) Leaf
```

But wait, there's more! Call now and receive an LL(1) and LR(0) parser free!

- Prefix parsers are a good prelude for LL(1) parsers.
- LL(1) parsers also use a stack of pending arguments.
- But first, we need to learn a bit more about Greibach Normal Form (GNF).

Greibach Normal Form

- A context-free grammar is in GNF iff all productions are of the form A $\rightarrow$ aω, where
  - a is a terminal symbol
  - ω consists of zero or more non-terminal symbols
- A GNF grammar is unambiguous iff each pair of productions of the form $A_1 \rightarrow a\omega_1$ and $A_2 \rightarrow a\omega_2$ satisfies $A_1 = A_2 \Rightarrow \omega_1 = \omega_2$.
- Unambiguous GNF grammars generalize data type declarations.
  - Terminals are data constructors.
  - Productions are data type declarations.
  - Terminals may occur in more than one production.

Greibach Normal Form - Example

- A grammar for a simple imperative language and its GNF equivalent on the right.

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{id} \; \texttt{:=} \; E \\
 & | & \texttt{if} \; E \; S \; S \\
 & | & \texttt{while} \; E \; S \\
 & | & \texttt{begin} \; B \; \texttt{end} \\
E & \rightarrow & \texttt{id} \\
B & \rightarrow & S \mid S \; ; \; B
\end{array}
\quad \Rightarrow \quad
\begin{array}{lcl}
S & \rightarrow & \texttt{id} \; C \; E \\
 & | & \texttt{if} \; E \; S \; S \\
 & | & \texttt{while} \; E \; S \\
 & | & \texttt{begin} \; S \; R \\
C & \rightarrow & \texttt{:=} \\
E & \rightarrow & \texttt{id} \\
R & \rightarrow & \texttt{end} \mid ; \; S \; R
\end{array}
$$

Greibach Normal Form - Example

$$
\begin{array}{lll}
S & \to & \mathtt{id}\ C\ E \\
  & | & \mathtt{if}\ E\ S\ S \\
  & | & \mathtt{while}\ E\ S \\
  & | & \mathtt{begin}\ S\ R \\
C & \to & \mathtt{:=} \\
E & \to & \mathtt{id} \\
R & \to & \mathtt{end} \ |\ ;\ S\ R
\end{array}
$$

- Abstract syntax:

```haskell
type Var = String
data Stat = Set Var Var | If Var Stat Stat |
    While Var Stat | Begin [Stat]
```

Greibach Normal Form - Example

- Abstract syntax:

```
type Var = String
data Stat = Set Var Var | If Var Stat Stat |
    While Var Stat | Begin [Stat]
```

- Example quotation:

```
≪ begin
  x := y;
  if x
     y := z
     z := y
 end ≫
```

```
Begin [Set "x" "y", If "x" (
    Set "y" "z") (Set "z" "y")]
```

Greibach Normal Form - Parser

- Ok, so how do we parse this?
- State = 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, there is at most a single suitable production.
- Replace non-terminal with RHS of the production.

Greibach Normal Form - Parser

- We'd like static type checking of a quotation, as before.
- So, let's encode the non-terminals as types:

```
newtype S α = S(Stat→ α)
newtype C α = C(α)
newtype E α = E(Var→ α)
newtype R α = R([Stat]→ α)
```

- For each production A→a$B_1 ... B_n$, we create a function **a** of type A $\alpha \to$ CPS (B_1(... ($B_n \alpha$) ...), which implements the expansion of A.
- However, remember that a terminal can occur in more than one production!
- We'll need a multi-parameter type class for such terminals.

Greibach Normal Form - Parser - Example

- So, for each terminal that appears more than once, we make a class:

```
class Id lhs rhs | lhs → rhs
id :: String → (lhs → CPS rhs)
```

- *Id* appears more than once, so we make an instance for each production that uses it:

```
instance Id (S α) (C(E α)) where
  id l = lift (λ(S ctx)→C(E(λr→ctx(Set l r))))
instance Id (E α) α where
  id i = lift (λ(E ctx)→ctx i)
```

Greibach Normal Form - Parser - Example

- Non-overloaded terminals don't need the instance declaration:

```
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)
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
```

Greibach Normal Form - Parser - Example

- Assume x = id "x" and y = id "y"
- Example derivation:

```
≪while x y := z≫
= while (S(λs→s)) x y := z ≫
= (λ(S ctx)→E(λc→S(λs→ctx(While c s)))) (S(λs
    →s)) x y := z ≫
= E(λc→S(λs→(λs'→s')(While c s))) x y := z ≫
= x (E(λc→S(λs→While c s))) y := z ≫
-- id i = lift (λ(E ctx)→ctx i)
= y (S(λs→While "x" s)) := z ≫
-- id l = lift(λ(S ctx)→C(E(λr→ctx(Set l r))))
= := 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")
```

LL(1) parsing - Our example grammar

- Example grammar: arithmetic expressions.
- Left one is left-recursive, we'll use the rewrite on the right.

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \textbf{id} \end{array} \quad \Rightarrow \quad \begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (E) \mid \textbf{id} \end{array}$$

- Abstract syntax:

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

LL(1) parsing - Constructing the parser - Step 1

- Step 1: For each non-terminal, make a **newtype**:

```
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)
```

LL(1) parsing - Constructing the parser - Step 2

- Step 2: For each production A→a$B_1 \ldots B_n$, create a function which implements the expansion of A.
- The parsing table of our grammar will be of use here:

	id	$+$	$*$	$($	$)$	»
E	$E \to T\ E'$			$E \to T\ E'$		
E'		$E' \to +\ T\ E'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to F\ T'$			$T \to F\ T'$		
T'		$T' \to \epsilon$	$T' \to *\ F\ T'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (\ E\ )$		

- For each terminal (see first row), we need a stack pop action.
- For each non-terminal (see first column), we need one or more expand actions.

LL(1) parsing - Constructing the parser - Step 2

- We'll need a class for each terminal:

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

LL(1) parsing - Constructing the parser - Step 2

- Instances for pop actions:

```
instance Id (I α) α where
  id s (I ctx) = return (ctx (Id s))
instance Add (A α) α where
  + (A ctx) = return ctx
instance Mul (M α) α where
  ∗ (M ctx) = return ctx
instance Open (O α) α where
  ( (O ctx) = return ctx
instance Close (C α) α where
  ) (C ctx) = return ctx
instance Endquote Expr where
  ≫ e = e
```

LL(1) parsing - Constructing the parser - Step 2

- Instances for expand actions, taking Id as an example:

	id	+	*	(	)	»
E	$E \rightarrow T\ E'$			$E \rightarrow T\ E'$		
E'		$E' \rightarrow +\ T\ E'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow F\ T'$			$T \rightarrow F\ T'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *\ F\ T'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (\ E\ )$		

```
instance Id (E α) (T' (E' α)) where
  id s (E ctx) = id s (T (λt→E'(λe'→ctx(e' t))))
instance Id (T α) (T' α) where
  id s (T ctx) = id s (F (λf→T'(λt'→ctx(t' f))))
instance Id (F α) α where
  id s (F ctx) = id s (I (λv→ctx v))
```

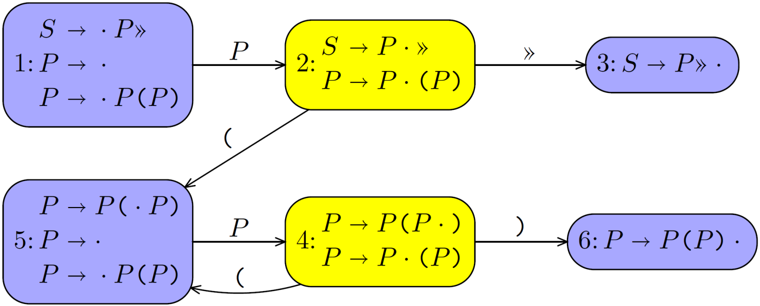
- Expansion phase is E → T E' → F T' E' → id T' E'.
- The instance head always reflects the parsing state after the final pop action (in this case, this is the *id* terminal).

LR(0) parsing - Our example grammar

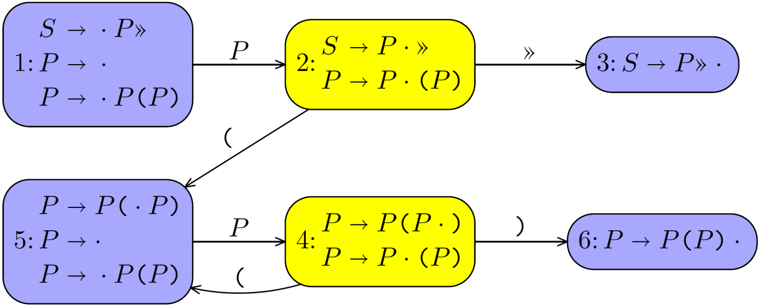
- LR(0) parsers are tricky to do by hand, so we'll use a very simple grammar: balanced parentheses.

$$P \rightarrow \epsilon \mid (P)$$

- Abstract syntax is our Tree datatype: P $\rightarrow \epsilon$ is a Leaf, P $\rightarrow$ P(P) is a Fork.
- The above grammar converts to the following LR(0) automaton:

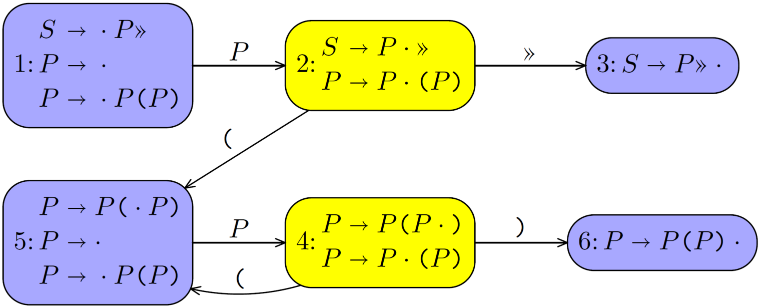


LR(0) parsing - LR(0) Automatons 101



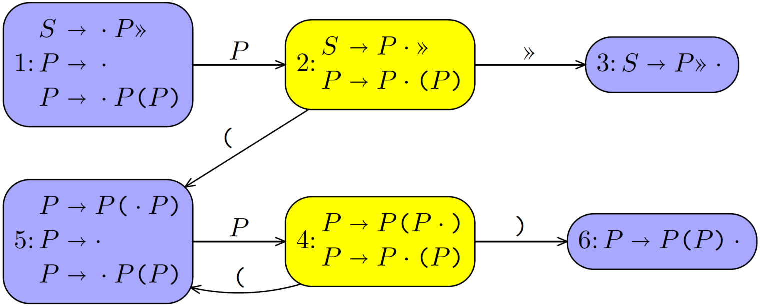
- An LR parser's stack contains what we've already seen (like postfix).
- Dots delineate what we've seen and expect to see.
  - Dot before a terminal: **shift** state, in yellow. Parser consumes the token and pushes it onto the stack.
  - Dot before a non-terminal: **reduce** state, in blue. RHS of a production is on the stack, we replace it with the LHS.

LR(0) parsing - LR(0) Automatons 101



- In our example, the first step is to reduce P $\rightarrow \epsilon$, moving from state 1 to state 2.
- In state 2, we can shift either '$\gg$' or '('.
- Each transition is recorded on the stack. We need this information during reductions.

LR(0) parsing - LR(0) Automatons 101



- Consider state 6: there are two sequences of moves that end in this state:
  - $1 \xrightarrow{P} 2 \xrightarrow{(} 5 \xrightarrow{P} 4 \xrightarrow{)} 6$
  - $5 \xrightarrow{P} 4 \xrightarrow{(} 5 \xrightarrow{P} 4 \xrightarrow{)} 6$
- We need to remove RHS ($P \to P(P)$) from the stack. We can end up in 1 or 5. Then, we need to add LHS (P). We can end up in 2 or 4. In general, there are several transitions for a single production.

LR(0) parsing - Constructing the parser - Data types

- Parser state = stack of LR(0) states.
- Each state carries the semantic value of the symbol that annotates the incoming edge(s) of that state.

```
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 -- )
```

LR(0) parsing - Constructing the parser - State transitions

- Each state is translated into a function:
  - Shift states delegate control to the next active terminal.
  - Reduce states pop the RHS transitions and push the LHS transitions. If there are several possible transitions, we use a type class.

```
quote = state1 S1 -- start
state1 st = state2 (S2 Leaf st) -- reduce
state2 = 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 →CPS new -- reduce
instance State6 (S6(S4(S5(S2 S1)))) (S2 S1) where
  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
  state6 (S6(S4 u(S5 (S4 t (S5 st))))) = state4 (S4 (Fork t u
      ) (S5 st))
```

LR(0) parsing - Constructing the parser - State transitions

- If a terminal annotates more than one edge, we also need a type family:

```
class Open old new | old → new where
  ( :: old → CPS 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)
```

- Instance types reflect the stack modifications up to the next shift state.
- For example, '(' moves from S_2 to S_5 and then to S_4.

Conclusion

- With quotations, terminal symbols turn active and become the driving force of the parsing process.
- We can construct different kinds of (anti-)quotation parsers in Haskell using the CPS monad.
- Different state types correspond to different kinds of parsers.
- Using type-level representations of symbols, we can provide statically-checked type safety.
- Caveat: Syntax errors become type errors, which may be hard to decipher.

Thanks for listening!

Questions?