Abstract Parsing for Two-staged Language

Soonho Kong
May 1, 2009

Contents

- Motivation
- Language - Syntax & Semantics
- LR Parsing
- Concrete Parsing
- Abstraction
- Instantiation

Motivation

Program generates another program.

Using eval() safely in python

The eval function lets a python program run python code within itself. It is used in Lybniz evaluate the functions input by the user.

PHP CORE - December 22, 2007

PHP eval is evil

JSON creator Douglas Crockford more than once pointed out that: in JavaScript, eval is evil. Well in PHP, eval is also almost always evil (though some people say it can be useful in a few limited cases). I personally have no guts to use it. Allowing any user-supplied data to go into an eval( ) call is asking to be hacked.

How can we make sure that

the generated program is **syntactically** correct?

Language: Syntax

We design a small & simple language
to focus on the property we want to verify

$$
\begin{array}{rcl}
e \in Exp & ::= & x \\
& | & \texttt{let}\ x\ e_1\ e_2 \\
& | & \texttt{or}\ e_1\ e_2 \\
& | & \texttt{re}\ x\ e_1\ e_2\ e_3 \\
& | & \texttt{`}f \\
\\
f \in Frag & ::= & x \\
& | & \texttt{let} \\
& | & \texttt{or} \\
& | & \texttt{re} \\
& | & f_1 . f_2 \\
& | & ,e
\end{array}
$$

Language: Example code

```
let x `a

let y `b

    or x y


=> `a

 | `b
```

```
let x = `a

let y = `b

    `x.y.,y


=> `x y b
```

```
re x `a `b x


=> `a

 | `b
```

```
re x `a `a.,x


=> `a

 | `a a

 | `a a a

 | `a a a a

 ...
```

```
re x `a `,x . ,x


=> `a

 | `a a

 | `a a a a

 | `a a a a a a a a

 | ...
```

Language: Collecting Semantics

$$
\begin{aligned}
\sigma \in Env &: Var \to Code \\
\mathcal{V}^0 &: Exp \to 2^{Env} \to 2^{Code} \\
\mathcal{V}^1 &: Frag \to 2^{Env} \to 2^{Code}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}^0\, x\, \Sigma &= \{\sigma(x) \mid \sigma \in \Sigma\} \\
\mathcal{V}^0(\texttt{let}\ x\ e_1\ e_2)\Sigma &= \bigcup_{c \in \mathcal{V}^0 e_1 \Sigma} \mathcal{V}^0 e_2 \{\sigma[x \mapsto c] \mid \sigma \in \Sigma\} \\
\mathcal{V}^0(\texttt{or}\ e_1\ e_2)\Sigma &= \mathcal{V}^0 e_1 \Sigma \cup \mathcal{V}^0 e_2 \Sigma \\
\mathcal{V}^0(\texttt{re}\ x\ e_1\ e_2\ e_3)\Sigma &= \bigcup_{\sigma \in \Sigma} \mathcal{V}^0 e_3 \{\sigma[x \mapsto c] \mid c \in fix\ \lambda C.\mathcal{V}^0 e_1\{\sigma\} \cup \mathcal{V}^0 e_2 \{\sigma[x \mapsto c'] \mid c' \in C\}\} \\
\mathcal{V}^0(\grave{}\, f)\Sigma &= \mathcal{V}^1 f \Sigma \\
\mathcal{V}^1(f_1.f_2)\Sigma &= \{xy \mid x \in \mathcal{V}^1 f_1 \Sigma,\ y \in \mathcal{V}^1 f_2 \Sigma\} \\
\mathcal{V}^1(,e)\Sigma &= \mathcal{V}^0 e \Sigma \\
\mathcal{V}^1\ \texttt{or}\ \Sigma &= \{\text{``}\texttt{or}\text{''}\} \\
\mathcal{V}^1\ \texttt{let}\ \Sigma &= \{\text{``}\texttt{let}\text{''}\} \\
\mathcal{V}^1\ \texttt{re}\ \Sigma &= \{\text{``}\texttt{re}\text{''}\} \\
\mathcal{V}^1 x \Sigma &= \{\text{``}\texttt{x}\text{''}\}
\end{aligned}
$$

Language: Abstract Collecting Semantics

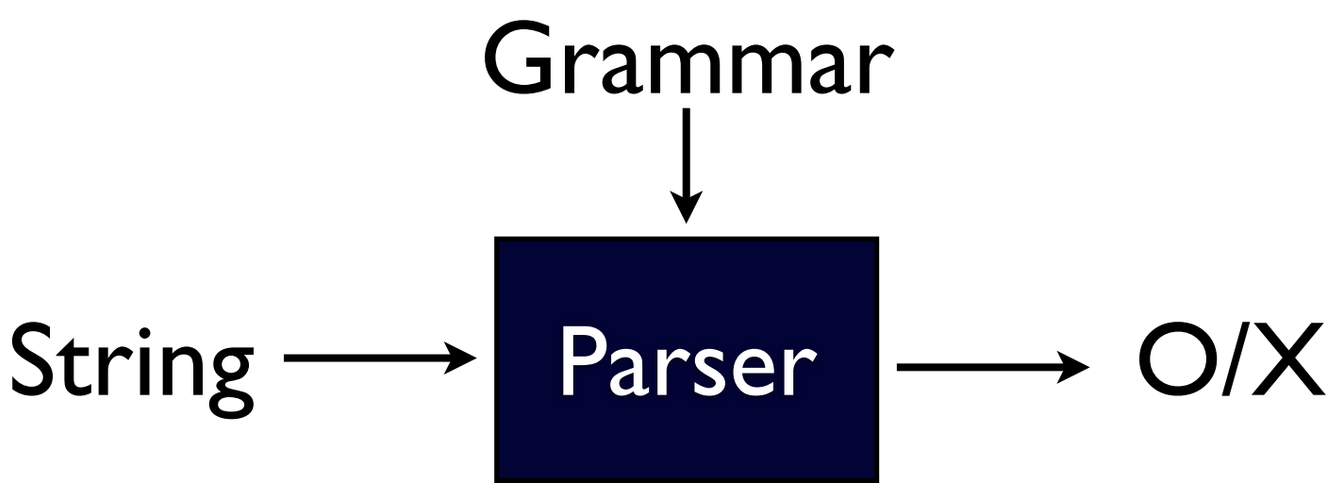
$$\alpha_{2^{Env} \to \widehat{Env}} = \lambda\Sigma.\lambda x.\{\sigma(x) \mid \sigma \in \Sigma\}$$

$$\widehat{\sigma} \in \widehat{Env} : Var \to 2^{Code}$$

$$\begin{aligned}
\widehat{\mathcal{V}}^0 &: Exp \to \widehat{Env} \to 2^{Code} \\
\widehat{\mathcal{V}}^0 x \widehat{\sigma} &= \widehat{\sigma}(x) \\
\widehat{\mathcal{V}}^0(\texttt{or}\ e_1\ e_2)\widehat{\sigma} &= \widehat{\mathcal{V}}^0 e_1 \widehat{\sigma} \sqcup \widehat{\mathcal{V}}^0 e_2 \widehat{\sigma} \\
\widehat{\mathcal{V}}^0(\texttt{let}\ x\ e_1\ e_2)\widehat{\sigma} &= \widehat{\mathcal{V}}^0 e_2(\widehat{\sigma}[x \mapsto \widehat{\mathcal{V}}^0 e_1 \widehat{\sigma}]) \\
\widehat{\mathcal{V}}^0(\texttt{re}\ x\ e_1\ e_2\ e_3)\widehat{\sigma} &= \widehat{\mathcal{V}}^0 e_3(\widehat{\sigma}[x \mapsto fix\ \lambda k.\widehat{\mathcal{V}}^0 e_1 \widehat{\sigma} \sqcup \widehat{\mathcal{V}}^0 e_2(\widehat{\sigma}[x \mapsto k])]) \\
\widehat{\mathcal{V}}^0(\grave{}\, f)\widehat{\sigma} &= \widehat{\mathcal{V}}^1 f \widehat{\sigma} \\
\widehat{\mathcal{V}}^1(f_1.f_2)\widehat{\sigma} &= \{xy \mid x \in \widehat{\mathcal{V}}^1 f_1 \widehat{\sigma} \wedge y \in \widehat{\mathcal{V}}^1 f_2 \widehat{\sigma}\} \\
\widehat{\mathcal{V}}^1(,e)\widehat{\sigma} &= \widehat{\mathcal{V}}^0 e \widehat{\sigma} \\
\widehat{\mathcal{V}}^1\,\texttt{or}\,\widehat{\sigma} &= \{\text{``}\texttt{or}\text{''}\} \\
\widehat{\mathcal{V}}^1\,\texttt{let}\,\widehat{\sigma} &= \{\text{``}\texttt{let}\text{''}\} \\
\widehat{\mathcal{V}}^1\,\texttt{re}\,\widehat{\sigma} &= \{\text{``}\texttt{re}\text{''}\} \\
\widehat{\mathcal{V}}^1 x \widehat{\sigma} &= \{\text{``}\texttt{x}\text{''}\}
\end{aligned}$$

LR Parsing

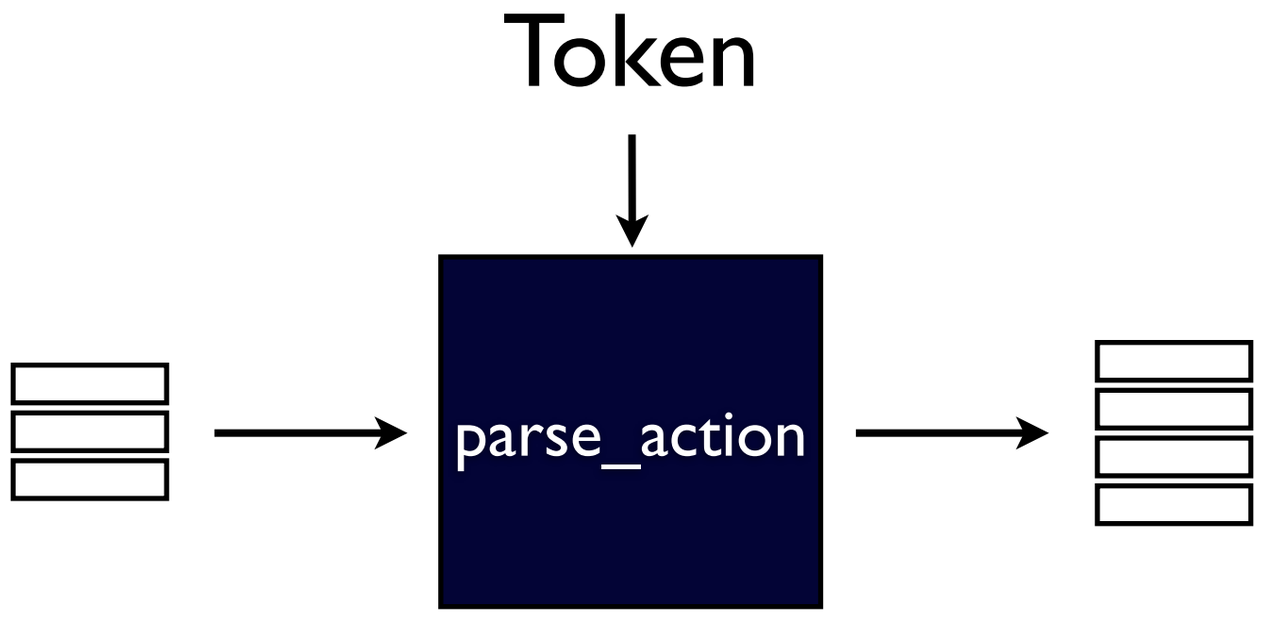
Parsing determines whether the string conforms to the grammar or not



LR Parsing

Scan **L**eft-to-right, derive **R**ightmost derivation tree.

Uses parsing stack and parse_action function

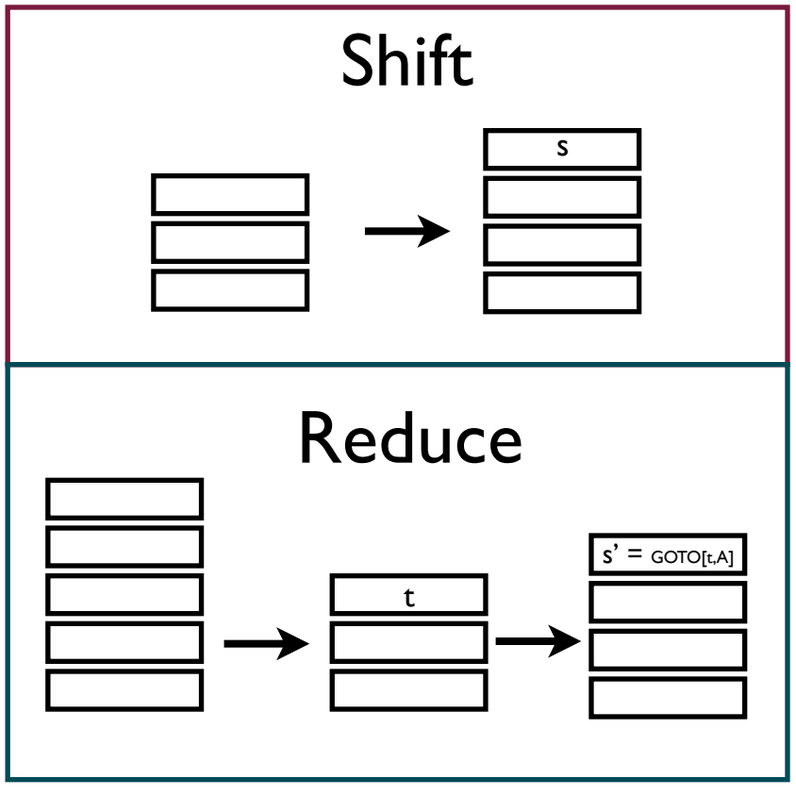


$$parse\_action : Token \rightarrow P \rightarrow P$$

LR Parsing

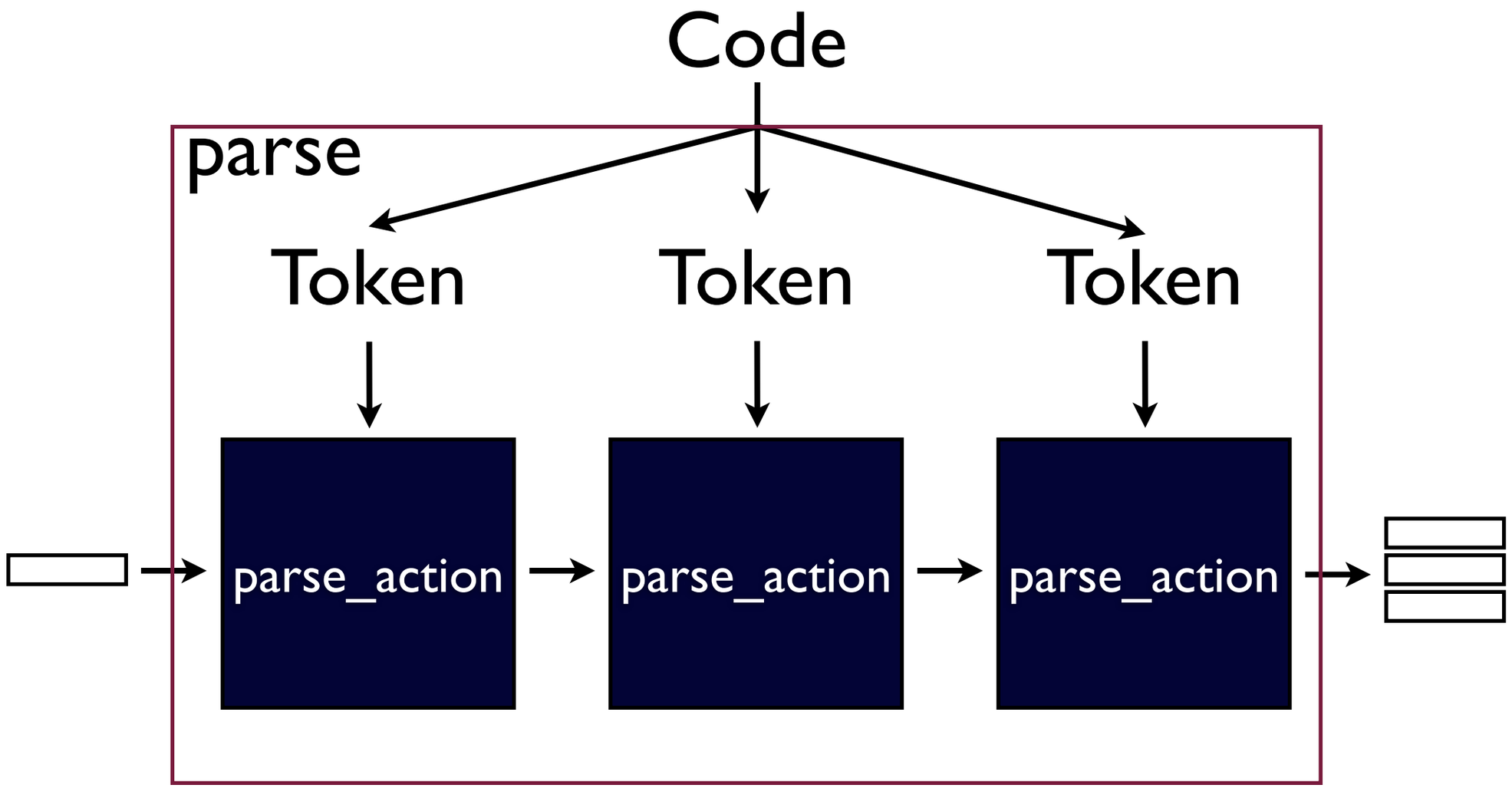
Algorithm 1 *parse_action* algorithm

```
1:  procedure parse_action(p, a)
2:      t ← the state on top of stack p
3:      if ACTION[t, a] = shift s then
4:          push s onto the stack p
5:          return p
6:      else if ACTION[t, a] = reduce A → β then
7:          pop | β | symbol off the stack p
8:          t ← the state on top of stack p
9:          push GOTO[t, A] onto the stack p
10:         return parse_action(p, a)
11:     end if
12: end procedure
```



LR Parsing

Parsing is composition of parse_action.



$$parse : Code \rightarrow P \rightarrow P$$

Abstract Parsing: Idea

Instead of executing the program and parsing the result,

$$\mathcal{V}_0 e \Sigma = \{c_1, c_2, \ldots, c_n\} \qquad parse(c_i) = O/X$$

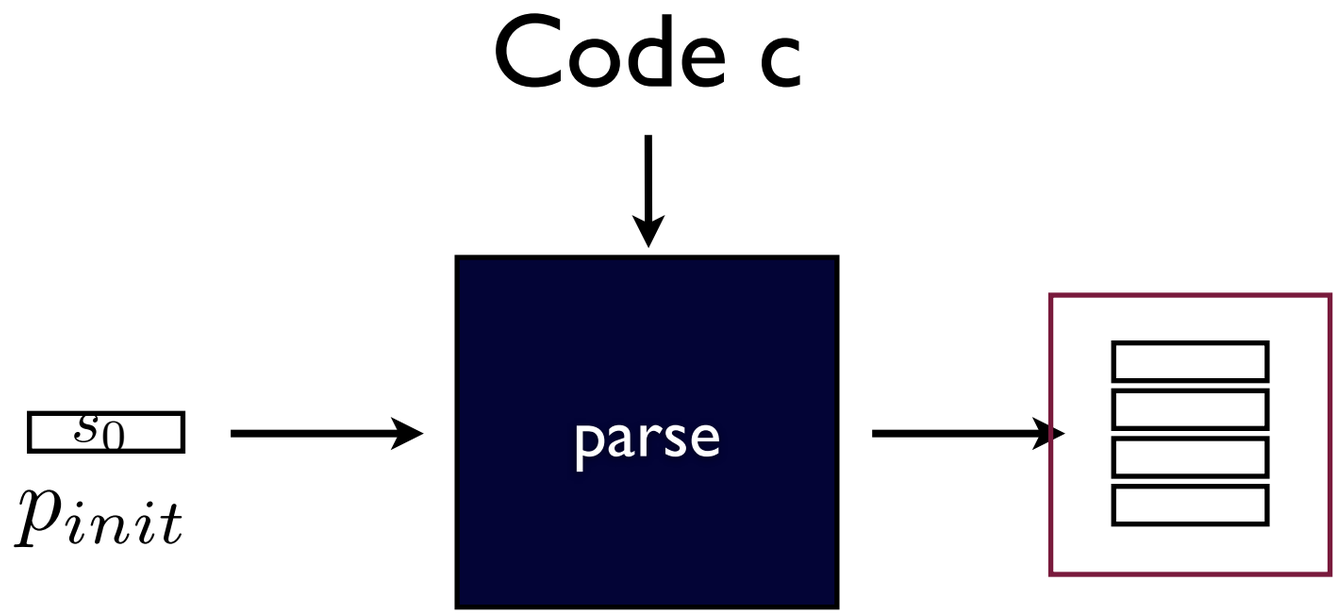
Execute the program on abstract semantics.

$$\widehat{\mathcal{V}}_0 e \widehat{\Sigma} = \{O, X\}$$

Concrete Parsing: Value

What should be the one corresponding with Code in parsing domain?

It is easy to think that it is $(parse\ c\ p_{init}) : P$



Concrete Parsing: Value

Problem:

Code Concatenation : Easy

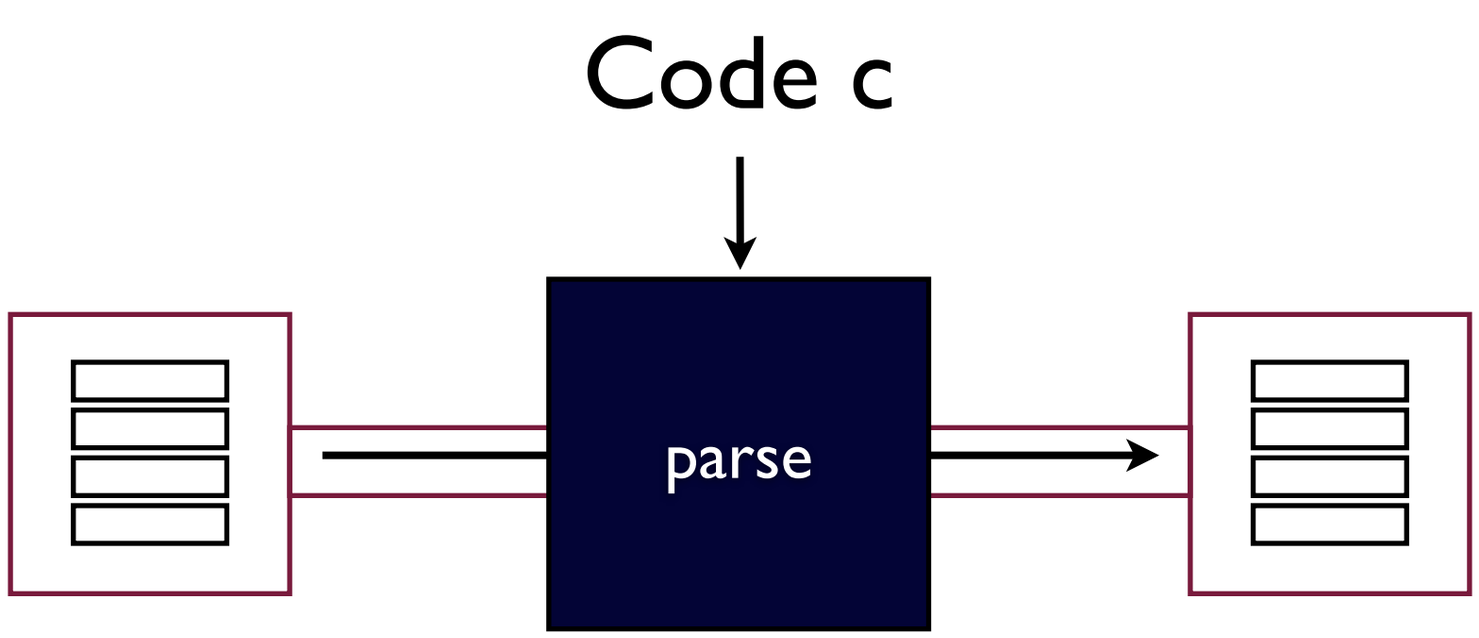
c1 + c2 = c1 c2

Parse Stack Concatenation : Hard

Concrete Parsing: Value

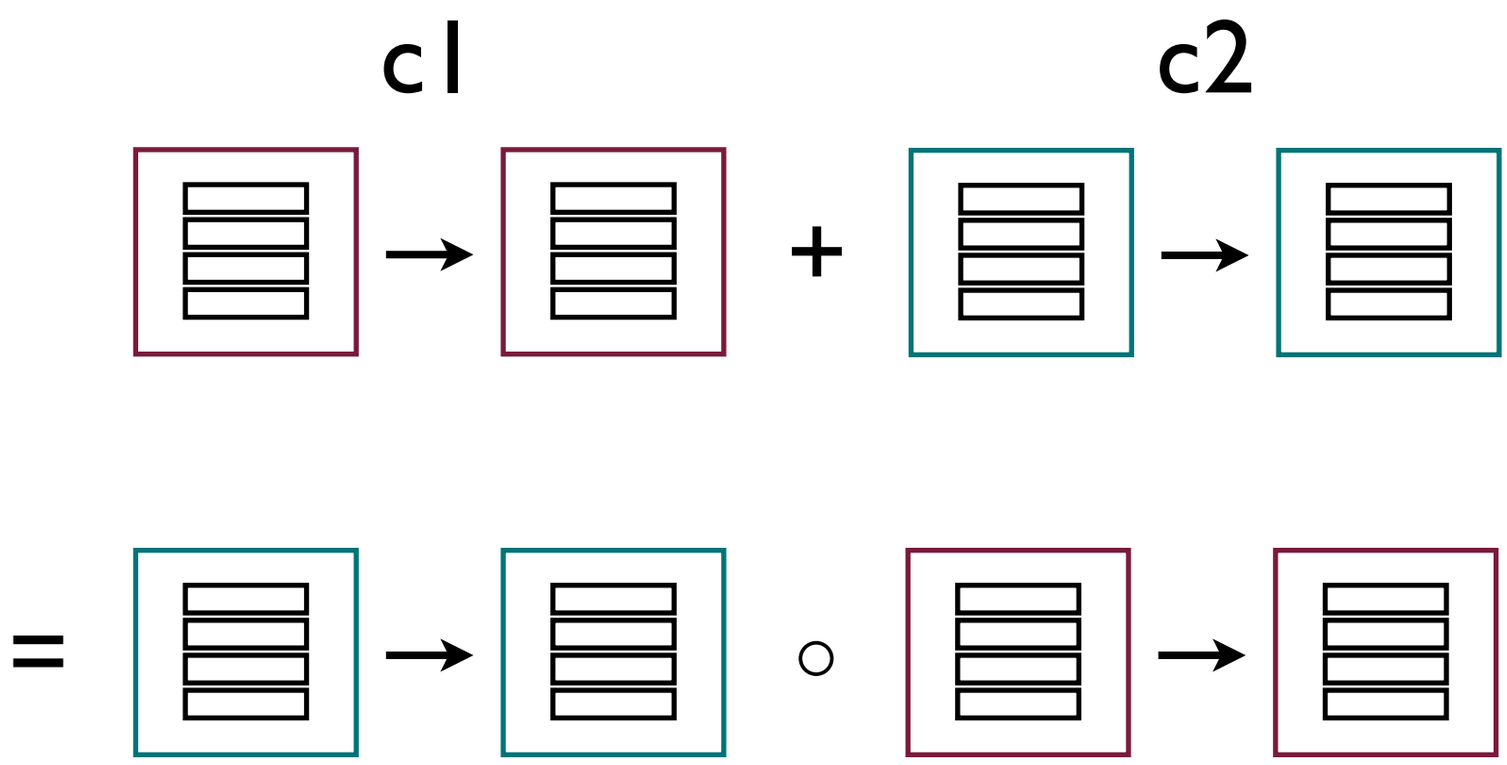
Solution

Let it be the function $(parse\ c) : P \rightarrow P$



Concrete Parsing: Value

Then the concatenation is handled as function composition



Concrete Parsing

Abstraction

$$\alpha_{2^{Code} \to 2^{P \to P}} = \lambda C.\{\lambda p.parse(p, c) \mid c \in C\}$$

Semantics

$$\begin{aligned}
\sigma_P \in Env_P &: Var \to 2^{P \to P} \\
\mathcal{V}_P^0 &: Exp \to Env_P \to 2^{P \to P} \\
\mathcal{V}_P^1 &: Frag \to Env_P \to 2^{P \to P}
\end{aligned}$$

$$\begin{aligned}
\mathcal{V}_P^0 x \sigma_P &= \sigma_P(x) \\
\mathcal{V}_P^0(\texttt{or}\ e_1\ e_2)\sigma_P &= V_P^0 e_1 \sigma_P \cup \mathcal{V}_P^0 e_2 \sigma_P \\
\mathcal{V}_P^0(\texttt{let}\ x\ e_1\ e_2)\sigma_P &= \mathcal{V}_P^0 e_2(\sigma_P[x \mapsto \mathcal{V}_P^0 e_1 \sigma_P]) \\
\mathcal{V}_P^0(\texttt{re}\ x\ e_1\ e_2\ e_3)\sigma_P &= \mathcal{V}_P^0 e_3(\sigma_P[x \mapsto fix\ \lambda k.\mathcal{V}_P^0 e_1 \sigma_P \sqcup \mathcal{V}_P^0 e_2(\sigma_P[x \mapsto k])]) \\
\mathcal{V}_P^0(\grave{}\ f)\sigma_P &= \mathcal{V}_P^1 f \sigma_P \\
\mathcal{V}_P^1(f_1.f_2)\sigma_P &= \{p_2 \circ p_1 \mid p_1 \in \mathcal{V}_P^1 f_1 \sigma_P \wedge p_2 \in \mathcal{V}_P^1 f_2 \sigma_P\} \\
\mathcal{V}_P^1(,e)\sigma_P &= \mathcal{V}_P^0 e \sigma_P \\
\mathcal{V}_P^1\ a\ \sigma_P &= \{\lambda p.parse_action\ a\ p\}
\end{aligned}$$

Abstraction

First, we abstract $2^{P \to P}$ to $2^P \to 2^P$ by,

$$\alpha_{2^{P \to P} \to (2^P \to 2^P)} = \lambda F \lambda P.\{f(p) \mid f \in F \wedge p \in P\}$$

Semantics is

$$\begin{aligned}
\sigma_{\widehat{P}} \in Env_{\widehat{P}} &: Var \to (2^P \to 2^P) \\
\mathcal{V}^0_{\widehat{P}} &: Exp \to Env_{\widehat{P}} \to (2^P \to 2^P) \\
\mathcal{V}^1_{\widehat{P}} &: Frag \to Env_{\widehat{P}} \to (2^P \to 2^P)
\end{aligned}$$

$$\begin{aligned}
\mathcal{V}^0_{\widehat{P}} x \sigma_{\widehat{P}} &= \sigma_{\widehat{P}}(x) \\
\mathcal{V}^0_{\widehat{P}}(\texttt{or}\ e_1\ e_2)\sigma_{\widehat{P}} &= \mathcal{V}^0_{\widehat{P}} e_1 \sigma_{\widehat{P}} \sqcup \mathcal{V}^0_{\widehat{P}} e_2 \sigma_{\widehat{P}} \\
\mathcal{V}^0_{\widehat{P}}(\texttt{let}\ x\ e_1\ e_2)\sigma_{\widehat{P}} &= \mathcal{V}^0_{\widehat{P}} e_2 (\sigma_{\widehat{P}}[x \mapsto \mathcal{V}^0_{\widehat{P}} e_1 \sigma_{\widehat{P}}]) \\
\mathcal{V}^0_{\widehat{P}}(\texttt{re}\ x\ e_1\ e_2\ e_3)\sigma_{\widehat{P}} &= \mathcal{V}^0_{\widehat{P}} e_3 (\sigma_{\widehat{P}}[x \mapsto fix\ \lambda k. \mathcal{V}^0_{\widehat{P}} e_1 \sigma_{\widehat{P}} \sqcup \mathcal{V}^0_{\widehat{P}} e_2 (\sigma_{\widehat{P}}[x \mapsto k])]) \\
\mathcal{V}^0_{\widehat{P}}(\grave{}\, f)\sigma_{\widehat{P}} &= \mathcal{V}^1_{\widehat{P}} f \sigma_{\widehat{P}} \\
\mathcal{V}^1_{\widehat{P}}(f_1.f_2)\sigma_{\widehat{P}} &= \mathcal{V}^1_{\widehat{P}} f_2 \sigma_{\widehat{P}} \circ \mathcal{V}^1_{\widehat{P}} f_1 \sigma_{\widehat{P}} \\
\mathcal{V}^1_{\widehat{P}}(,e)\sigma_{\widehat{P}} &= \mathcal{V}^0_{\widehat{P}} e \sigma_{\widehat{P}} \\
\mathcal{V}^1_{\widehat{P}}\ a\ \sigma_{\widehat{P}} &= \lambda P.\{parse_action\ a\ p \mid p \in P\}
\end{aligned}$$

Abstraction

We can abstract $2^P \to 2^P$ to $D^\sharp \to D^\sharp$

if $D^\sharp$ satisfies the following conditions.

1. $(D^\sharp, \sqsubseteq, \sqcup, \bot_{D^\sharp})$ is CPO
2. 2^P and $D^\sharp$ are galois connected via $\alpha_{2^P \to D^\sharp}$ and $\gamma_{D^\sharp \to 2^P}$
3. $parse\_action^\sharp : Token \to D^\sharp \to D^\sharp$ is a sound abstraction of

   $parse\_action : Token \to 2^P \to 2^P$ .That is,

   $\forall a \in Token. \forall P \in 2^P.$

   $\alpha_{2^P \to D^\sharp}(\{parse\_action\ a\ p \mid p \in P\}) \sqsubseteq parse\_action^\sharp\ a\ \alpha_{2^P \to D^\sharp}(P)$

Abstraction

We define $\mathcal{V}^0_{D^\sharp}$ and $\mathcal{V}^1_{D^\sharp}$.

$$
\begin{array}{rcl}
\sigma_{D^\sharp} & \in & Env_{D^\sharp} \\
\mathcal{V}^0_{D^\sharp} & : & Exp \to Env_{D^\sharp} \to (2^{D^\sharp} \to 2^{D^\sharp}) \\
\mathcal{V}^1_{D^\sharp} & : & Frag \to Env_{D^\sharp} \to (2^{D^\sharp} \to 2^{D^\sharp})
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{V}^0_{D^\sharp} x \sigma_{D^\sharp} & = & \sigma_{D^\sharp}(x) \\
\mathcal{V}^0_{D^\sharp}(\texttt{or}\ e_1\ e_2)\sigma_{D^\sharp} & = & \mathcal{V}^0_{D^\sharp} e_1 \sigma_{D^\sharp} \sqcup \mathcal{V}^0_{D^\sharp} e_2 \sigma_{D^\sharp} \\
\mathcal{V}^0_{D^\sharp}(\texttt{let}\ x\ e_1\ e_2)\sigma_{D^\sharp} & = & \mathcal{V}^0_{D^\sharp} e_2 (\sigma_{D^\sharp}[x \mapsto \mathcal{V}^0_{D^\sharp} e_1 \sigma_{D^\sharp}]) \\
\mathcal{V}^0_{D^\sharp}(\texttt{re}\ x\ e_1\ e_2\ e_3)\sigma_{D^\sharp} & = & \mathcal{V}^0_{D^\sharp} e_3 (\sigma_{D^\sharp}[x \mapsto fix\ \lambda k. \mathcal{V}^0_{D^\sharp} e_1 \sigma_{D^\sharp} \sqcup \mathcal{V}^0_{D^\sharp} e_2 (\sigma_{D^\sharp}[x \mapsto k])]) \\
\mathcal{V}^0_{D^\sharp}\ `f \sigma_{D^\sharp} & = & \mathcal{V}^1_{D^\sharp} f \sigma_{D^\sharp} \\
\mathcal{V}^1_{D^\sharp}(f_1 \cdot f_2)\sigma_{D^\sharp} & = & \mathcal{V}^1_{D^\sharp} f_2 \sigma_{D^\sharp} \circ \mathcal{V}^1_{D^\sharp} f_1 \sigma_{D^\sharp} \\
\mathcal{V}^1_{D^\sharp}(, e)\sigma_{D^\sharp} & = & \mathcal{V}^0_{D^\sharp} e \sigma_{D^\sharp} \\
\mathcal{V}^1_{D^\sharp}\ a\ \sigma_{D^\sharp} & = & \lambda D. parse\_action^\sharp\ a\ D
\end{array}
$$

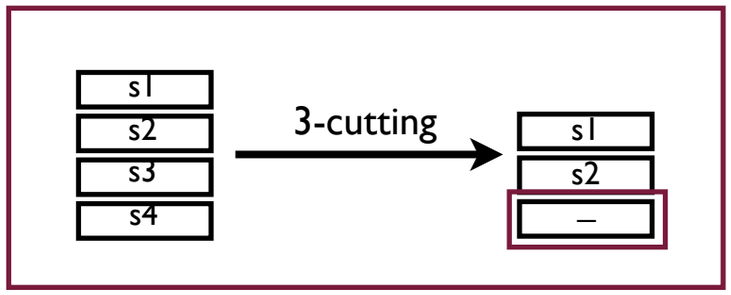
Then $\mathcal{V}^0_{D^\sharp}$ is a sound approximation of $\mathcal{V}^0_{\widehat{P}}$.

$$
\begin{array}{l}
\forall e \in Exp. \forall \sigma_{\widehat{P}} \in Env_{\widehat{P}}. \\
\alpha_{(2^P \to 2^P) \to (D^\sharp \to D^\sharp)}(\mathcal{V}^0_{\widehat{P}}\ e\ \sigma_{\widehat{P}}) \sqsubseteq \mathcal{V}^0_{D^\sharp}\ e\ \alpha_{Env_{\widehat{P}} \to Env_{D^\sharp}}(\sigma_{\widehat{P}})
\end{array}
$$

Instantiation of $D^\sharp$

$\widehat{D}$: Abstract Parsing Stack with k-cutting

IDEA : limit the length of parsing stack with k



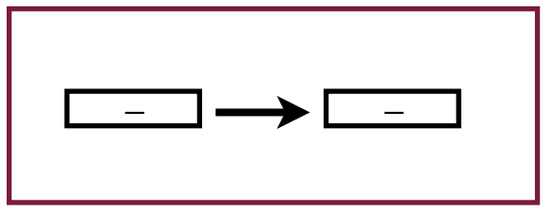
$$
\begin{aligned}
\widehat{D} &: 2^{\widehat{P}} \\
\widehat{P} &= P \cup \{\_ \cdot p \mid p \in P\} \\
\alpha_{2^P \to \widehat{D}} &= id \\
\gamma_{\widehat{D} \to 2^P} &= \lambda D. \bigcup_{d \in D} (if\ d \in P\ then\ \{d\}\ else \top)
\end{aligned}
$$

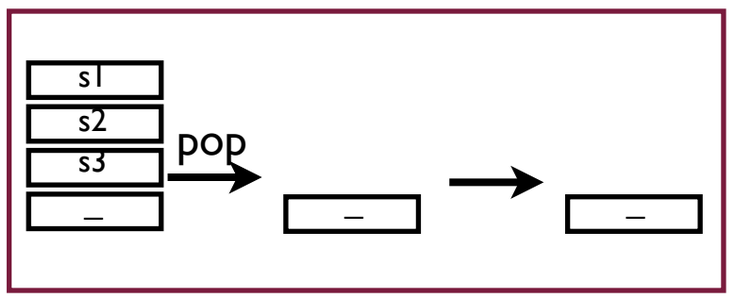
Instantiation of $D^\sharp$

$\widehat{D}$: Abstract Parsing Stack with k-cutting

Algorithm 2 $\widehat{parse\_action}$ algorithm

```
1:  procedure parse_action(p̂, a)
2:      if p̂ = _ then
3:          return p̂
4:      end if
5:      t ← the state on top of stack p̂
6:      if ACTION[t, a] = shift s then
7:          push s onto the stack p̂
8:          return p̂
9:      else if ACTION[t, a] = reduce A → β then
10:         pop | β | symbol off the stack p̂
11:         t ← the state on top of stack p̂
12:         if t = _ then
13:             return p̂
14:         end if
15:         push GOTO[t, A] onto the stack p̂
16:         return parse_action(a, p̂)
17:     end if
18: end procedure
```





Instantiation of $D^\sharp$

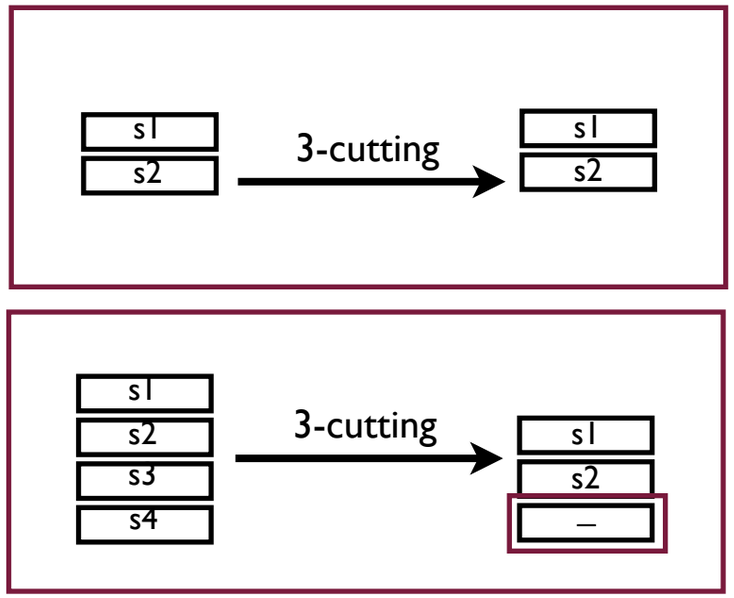
$\widehat{D}$: Abstract Parsing Stack with k-cutting

We only perform k-cutting at loop head
by defining widening operator $\nabla_{\widehat{D}}$.

$$
\begin{array}{rcl}
A \nabla_{\widehat{D}} B & : & \widehat{D} \to \widehat{D} \to \widehat{D} \\
A \nabla_{\widehat{D}} B & = & \{cut_k(\widehat{p}) \mid \widehat{p} \in A \cup B\}
\end{array}
$$

Algorithm 3 cut_k algorithm

1: **procedure** $cut_k(\widehat{p} = s_1 \ldots s_n)$
2: **if** $n \leq k$ **then**
3: **return** $\widehat{p}$
4: **else**
5: **return** $s_1 \ldots s_{k-1}$ __
6: **end if**
7: **end procedure**



Reference

Kyung-Goo Doh, Hyunha Kim and David Schmidt. *Abstract parsing: static analysis of dynamically generated string output using LR-parsing technology.* SAS 2009: The 16th International Static Analysis Symposium, LA, August 7-9, 2009. (to appear)

Thank you