PCC Framework for Program-Generators

Soonho Kong       Wontae Choi       Kwangkeun Yi

Programming Research Lab.
Seoul National University
{soon,wtchoi,kwang}@ropas.snu.ac.kr

15 August 2009
Workshop on Proof-Carrying Code and Software Certification
**Introduction**

- **Program-Generators**
  - Need to ensure the safe execution of *the generated programs* as well as the generator itself.
  
  ![Diagram](image)

  - Safety properties of the generated programs are efficiently expressed by *the grammar $G$*.  
    e.g. “*generated programs should not have nested loops*”

  - **Question:**  
    “Do the generated programs conform to the safety grammar $G$?”
Introduction

• Abstract Parsing
  - Powerful static string analysis technique presented by Doh, Kim, and Schmidt[1]
  - Determine whether the strings generated in the program conform to the given grammar $G$.
  - Use LR parser as a component
  - Formalized and parameterized in the abstract interpretation framework by Kong, Choi and Yi[2]

PCC Framework for Program-Generators

To ensure the termination of the analysis, we need to provide an abstraction for the infinite height domain $P$. Instead of using a particular abstract domain for $P$, we parameterize this abstract domain by providing conditions which an abstract domain $D$ needs to satisfy. $D$ should be a complete partial order (CPO), and $D$ is Galois connected with the set of parse stacks $P$. An abstracted parsing function $\text{Parse}_{\text{action}}$ is defined as a sound approximation of the parsing function $\text{Parse}_{\text{action}}$ which is defined by the LR parser generator with the safety grammar $G$.

Finally, we derive the abstract parsing semantics for $D$ as in Figure 2.

Given a program generator $e$ and an empty environment $\sigma$, the analysis computes $F(e) = D(\sigma)$, which is of type $D \rightarrow D$. To determine whether the programs generated by a program generator $e$ conform to the safety grammar, we check that the following equation holds:

$$F(\alpha[p_{\text{init}}]) = \alpha[p_{\text{acc}}]$$

where $p_{\text{init}}$ and $p_{\text{acc}}$ are the initial parse stack and accepting parse stack for the safety grammar $G$.

Figure 4 illustrates a PCC framework for program generators. An abstraction carrying code framework (A3) is specialized to program generators by means of abstract parsing. The code producer and code consumers share the safety grammar which specifies the safety properties of the generated programs.

The code producer proves the safety of the program generator by abstract parsing with the shared safety grammar. In a complex and iterative process, the analysis computes a fixed point solution. This solution is used as a certificate for the safety of the program generator. The code producer uploads or sends the program generator with the computed fixed point solution.

The code consumer downloads or receives the untrusted program generator and its attached fixed point solution. The code consumer validates that the received fixed point solution is indeed a fixed point solution of the received program generator. In contrast to the computing a fixed point solution on the code producer side, checking can be done in a single pass.

The safety grammar is shared between code producer and consumer.
To ensure the termination of the analysis, we need to provide an abstraction for the infinite height domain $P$. Instead of using a particular abstract domain for $P$, we parameterize this abstract domain by providing conditions which an abstract domain $D$ needs to satisfy.

$D$ should be a complete partial order (CPO), and $D$ is Galois connected with the set of parse stacks $P$.

An abstracted parsing function $\text{Parse action}$ is defined as a sound approximation of the parsing function $\text{Parse action}$ which is defined by the LR parser generator with the safety grammar $G$.

Finally, we derive the abstract parsing semantics for $D$ as in Figure 2.

Given a program generator $e$ and an empty environment $\sigma$, the analysis computes $F = [\{ e \}]_{D}$ which is of type $D \rightarrow D$. To determine whether the programs generated by a program generator $e$ conform to the safety grammar, we check that the following equation holds:

$$F(\alpha[P \rightarrow D]\{p_{\text{init}}\}) = \alpha[P \rightarrow D]\{p_{\text{acc}}\}$$

where $p_{\text{init}}$ and $p_{\text{acc}}$ are the initial parse stack and accepting parse stack for the safety grammar $G$.

Figure 1 illustrates a PCC framework for program generators, an abstraction carrying code framework (ACCF).

5] specialized to program generators by means of abstract parsing. The code producer and code consumer share the safety grammar which specifies the safety properties of the generated programs.

Program-Generator

Abstract Parser

Fixed Point

Send

Send

Code Producer

Receive

Receive

Code Consumer

Received Program-Generator

Fixed Point Checker

Safety Grammar

In code producer side, abstract parser computes fixed-point solution for the given program-generator.
To ensure the termination of the analysis, we need to provide an abstraction for the infinite height domain \( P \). Instead of using a particular abstract domain for \( P \), we parameterize this abstract domain by providing conditions which an abstract domain \( D \) needs to satisfy.

\( D \) should be a complete partial order (CPO), \( D \) is Galois connected with the set of parse stacks \( P \), an abstracted parsing function \( \text{Parse} \) is defined as a sound approximation of the parsing function \( \text{Parse} \) which is defined by the LR parser generator with the safety grammar \( G \).

Finally, we derive the abstract parsing semantics for \( D \) as in Figure 2 right.

Given a program generator \( e \) and an empty environment \( \sigma \), the analysis computes \( F = \left[ e \right] \left( D \right) \) which is of type \( D \rightarrow D \). To determine whether the programs generated by a program generator \( e \) conform to the safety grammar, we check that the following equation holds:

\[
\alpha \left[ P \rightarrow D \left( \{ \text{p init} \} \right) \right] = \alpha \left[ P \rightarrow D \left( \{ \text{p acc} \} \right) \right]
\]

where \( \text{p init} \) and \( \text{p acc} \) are the initial parse stack and accepting parse stack for the safety grammar \( G \).

**Big Picture**

PCC Framework for Program-Generators

The code producer proves the safety of the program generator by abstract parsing with the shared safety grammar. In a complex and iterative process, the analysis computes a fixed point solution. This solution is used as a certificate for the safety of the program generator. The code producer uploads or sends the program generator with the computed fixed point solution.

The code consumer downloads or receives the untrusted program generator and its attached fixed point solution. The code consumer validates that the received fixed point solution is indeed a fixed point solution of the received program generator. In contrast to the computing a fixed point solution on the code producer side, checking can be done in a single pass.

Code producer sends the program generator with the computed fixed point solution.
To ensure the termination of the analysis, we need to provide an abstraction for the infinite height domain $P$. Instead of using a particular abstract domain for $P$, we parameterize this abstract domain by providing conditions which an abstract domain $D$ needs to satisfy.

$D$ should be a complete partial order (CPO), $D$ is Galois connected with the set of parse stacks $P$, an abstracted parsing function $\text{Parse} \mapsto \text{action}$ is defined as a sound approximation of the parsing function $\text{Parse}$ which is defined by the LR parser generator with the safety grammar $G$.

Finally, we derive the abstract parsing semantics for $D$ as in Figure 2-right.

Given a program:generator $e$ and an empty environment $\sigma$, the analysis computes $F = \{ e \} (D)$ which is of type $D \rightarrow D$. To determine whether the programs generated by a program:generator $e$ conform to the safety grammar, we check that the following equation holds:

$$F(\alpha[P\rightarrow D](\{p\text{init}\})) = \alpha[P\rightarrow D](\{p\text{acc}\})$$

where $p\text{init}$ and $p\text{acc}$ are the initial parse stack and accepting parse stack for the safety grammar $G$.

Figure [ illustrates a PCC framework for program:generators, an abstraction:carrying code framework (1). Specialized to program:generators by means of abstract parsing, the code producer and code consumer share the safety grammar which specifies the safety properties of the generated programs.

The code producer proves the safety of the program:generator by abstract parsing with the shared safety grammar. In a complex and iterative process, the analysis computes a fixed point solution. This solution is used as a certificate for the safety of the program:generator. The code producer uploads or sends the program:generator with the computed fixed point solution.

The code consumer downloads or receives the untrusted program:generator and its attached fixed point solution. The code consumer validates that the received fixed point solution is indeed a fixed point solution of the received program:generator. In contrast to the computing a fixed point solution on the code producer side, checking can be done in a single pass.
To ensure the termination of the analysis, we need to provide an abstraction for the infinite height domain \( P \). Instead of using a particular abstract domain for \( P \), we parameterize this abstract domain by providing conditions which an abstract domain \( D \) needs to satisfy. 

\( D \) should be a complete partial order (CPO), \( D \) is Galois connected with the set of parse stacks \( P \), 

An abstracted parsing function \( \text{Parse} \) is defined as a sound approximation of the parsing function \( \text{Parse} \), which is defined by the LR parser generator with the safety grammar \( G \), 

Finally, we derive the abstract parsing semantics for \( D \) as in Figure 2.

Given a program generator \( e \) and an empty environment \( \sigma \), the analysis computes \( F(\{ p_{\text{init}} \}) = \alpha(\{ p_{\text{acc}} \}) \) where \( p_{\text{init}} \) and \( p_{\text{acc}} \) are the initial parse stack and accepting parse stack for the safety grammar \( G \).

In code consumer side, the checker validates that the received fixed-point is indeed the solution for the received program-generator.
To ensure the termination of the analysis, we need to provide an abstraction for the infinite height domain $P$. Instead of using a particular abstract domain for $P$, we parameterize this abstract domain by providing conditions which an abstract domain $D$ needs to satisfy.

$D$ should be a complete partial order (CPO), $D$ is Galois connected with the set of parse stacks $P$, An abstracted parsing function $\text{Parse}_{\text{action}}$ is defined as a sound approximation of the parsing function $\text{Parse}$ which is defined by the LR parser generator with the safety grammar $G$.

Finally, we derive the abstract parsing semantics for $D$ as in Figure 2.

Given a program generator $e$ and an empty environment $\sigma$, the analysis computes $F(e) = (\sigma)$ which is of type $D \rightarrow D$. To determine whether the programs generated by a program generator $e$ conform to the safety grammar, we check that the following equation holds:

$$F(\alpha[P \rightarrow D^\prime(\{p_{init}\})]) = \alpha[P \rightarrow D^\prime(\{p_{acc}\})]$$

where $p_{init}$ and $p_{acc}$ are the initial parse stack and accepting parse stack for the safety grammar $G$.

The code producer proves the safety of the program generator by abstract parsing with the shared safety grammar. In a complex and iterative process, the analysis computes a fixed point solution. This solution is used as a certificate for the safety of the program generator. The code producer uploads or sends the program generator with the computed fixed point solution.

The code consumer downloads or receives the untrusted program generator and its attached fixed point solution. The code consumer validates that the received fixed point solution is indeed a fixed point solution of the received program generator. In contrast to computing a fixed point solution on the code producer side, checking can be done in a single pass.

If the checker validates it successfully, the code consumer is ready to execute the received program generator.
To ensure the termination of the analysis, we need to provide an abstraction for the infinite height domain $P$. Instead of using a particular abstract domain for $P$, we parameterize this abstract domain by providing conditions which an abstract domain $D$ needs to satisfy. $D$ should be a complete partial order (CPO), $D$ is Galois connected with the set of parse stacks $P$, An abstracted parsing function $\text{Parse}_{\text{action}}$ is defined as a sound approximation of the parsing function $\text{Parse}$ which is defined by the LR parser generator with the safety grammar $G$.

Finally, we derive the abstract parsing semantics for $D$ as in Figure 2(right).

Given a program generator $e$ and an empty environment $\sigma$, the analysis computes $F(e)\sigma$ which is of type $D \rightarrow D$. To determine whether the programs generated by a program generator $e$ conform to the safety grammar, we check that the following equation holds:

$$F(\alpha \rightarrow D\{p_{\text{init}}\}) = \alpha \rightarrow D\{p_{\text{acc}}\}$$

where $p_{\text{init}}$ and $p_{\text{acc}}$ are the initial parse stack and accepting parse stack for the safety grammar $G$.
Language for Program-Generators

• Tow-staged language with concatenation

Syntax

\[
e \in \text{Exp} ::= x \mid \text{let } x \ e_1 \ e_2 \mid \text{or } e_1 \ e_2 \mid \text{re } x \ e_1 \ e_2 \ e_3 \mid 'f
\]

\[
f \in \text{Frag} ::= x \mid \text{let} \mid \text{or} \mid \text{re} \mid (\ ) \mid f_1.f_2 \mid ,e
\]
Language for Program-Generators

Example

```
re x `a

`(. ,x . )

let y

 `or . a

 `,y . ,x

=>
```
Language for Program-Generators

Example

```
x `a
`( . ,x . )
let y
`or . a
`y . ,x
```

- x is initialized with “a”
- y is initialized with “or a”
- Loop body is not executed.
Language for Program-Generators

Example

```plaintext
re x `a
  `( . ,x . )
let y
  `or . a
  `,y . ,x
=>
```

- x is initialized with “a”
- Loop body is not executed.
- y is initialized with “or a”
- Value is “or a a”
Language for Program-Generators

Example

re x `a
(` `,x . )

let y
`or . a
` , y . , x

=> or a a    (if loop body is not executed)
Language for Program-Generators

Example

```kotlin
re x `a
  `( . ,x . )
let y
  `or . a
  `\,y . ,x

=> or a a
```

x is initialized with “a”
Language for Program-Generators

Example

```plaintext
re x 'a
  `(. ,x .)
let y
  `or . a
  `,.y . ,x

=> or a a
```

x is initialized with “a”

x is “( a )”

Loop body is executed once
Language for Program-Generators

Example

```plaintext
re x `a
  `( . ,x . )
let y
  `or . a
  `,y . ,x

=> or a a
```

Loop body is executed once

- x is initialized with “a”
- x is “( a )”
- y is initialized with “or a”
Language for Program-Generators

Example

re x `a
` (. ,x . )

let y
`or . a
`.y . ,x

x is initialized with “a”
x is“( a )”
y is initialized with “or a”
Value is “or a (a)”

=> or a a
Language for Program-Generators

Example

re x 'a
  `( . ,x . )
let y
  `or . a
  `\,y . ,x

=> or a a
  or a (a)  (if loop body is executed once)
Language for Program-Generators

Example

```plaintext
re x `a
  `( . ,x . )
let y
  `or . a
  `,y . ,x

=> or a a
  or a (a)
```

x is initialized with “a”
Language for Program-Generators

Example

\[
\text{re } x \ `a \\
`(` ,x . `) \\
\text{let } y \\
`\text{or . } a \\
`\text{,y . ,x} \\
\Rightarrow \text{or } a \ a \\
\text{or } a \ (a)
\]

x is initialized with “a”

x is “( a )”

Loop body is executed once
Language for Program-Generators

Example

```plaintext
re x `a
  `( . ,x . )
let y
  `or . a
  `y . ,x

=> or a a
  or a (a)
```

Loop body is executed twice

x is initialized with “a”

x is “((a))”

Language for Program-Generators

Example

```plaintext
re x 'a
  `(. ,x .)
let y
  `or . a
  `x

=> or a a
  or a (a)
```

Loop body is executed twice

- x is initialized with “a”
- x is “(( a ))”
- y is initialized with “or a”
Language for Program-Generators

Example

```plaintext
re x `a
  `(. ,x . )
let y
  `or . a
  `,y . ,x
=> or a a
  or a (a)
```

- x is initialized with “a”
- x is “(( a ))”
- y is initialized with “or a”
- Value is “or a (( a ))”

Loop body is executed twice
Language for Program-Generators

Example

\[ \text{let } y \text{ = } \text{or a} \]
\[ \text{, } y . \text{, x} \]
\[ \Rightarrow \text{ or a a} \]
\[ \text{or a (a)} \]
\[ \text{or a ((a))} \]

(if loop body is executed twice)

\[ \text{x is initialized with “a”} \]
\[ \text{x is “((a))”} \]

\[ \text{Loop body is executed twice} \]

\[ \text{y is initialized with “or a”} \]

Value is “or a ((a))”
Language for Program-Generators

Example

re x `a
  `( , ,x . )
  let y
    `or . a
    `,y . ,x

=> or a a
  or a (a)
  or a ((a))
  or a (((a)))

Note that only one of them is taken as a value of this program in execution.
To ensure the termination of the analysis, we need to provide an abstraction for the infinite height domain \( P \). Instead of using a particular abstract domain for \( P \), we parameterize this abstract domain by providing conditions which an abstract domain \( D \) needs to satisfy.

\( D \) should be a complete partial order (CPO), and \( D \) is Galois connected with the set of parse stacks \( P \). An abstracted parsing function \( \text{Parse}_\text{action} \) is defined as a sound approximation of the parsing function \( \text{Parse} \) which is defined by the LR parser generator with the safety grammar \( G \).

Finally, we derive the abstract parsing semantics for \( D \) as in Figure 2.

Given a program generator \( e \) and an empty environment \( \sigma \), the analysis computes \( F = [ e ]_\sigma \) which is of type \( D \rightarrow D \). To determine whether the programs generated by a program generator \( e \) conform to the safety grammar, we check that the following equation holds:

\[
\alpha_{P \rightarrow D}([p_{\text{init}}]) = \alpha_{P \rightarrow D}([p_{\text{acc}}])
\]

where \( p_{\text{init}} \) and \( p_{\text{acc}} \) are the initial parse stack and accepting parse stack for the safety grammar \( G \).

The abstract parsing parsing framework is as follows:

- **Program-Generator**
  - Sends programs to the **Abstract Parser**
  - Receives the fixed point solution

- **Abstract Parser**
  - Generates the fixed point solution

- **Code Producer**
  - Provides the safety grammar
  - Sends the fixed point solution

- **Code Consumer**
  - Receives the program generator
  - Receives the fixed point solution
  - Validates the fixed point solution

- **Safety Grammar**
  - Specifies the safety properties of the generated programs

The code producer proves the safety of the program generator by abstract parsing with the shared safety grammar. In a complex and iterative process, the analysis computes a fixed point solution. This solution is used as a certificate for the safety of the program generator. The code producer uploads or sends the program generator with the computed fixed point solution. The code consumer downloads or receives the untrusted program generator and its attached fixed point solution. The code consumer validates that the received fixed point solution is indeed a fixed point solution of the received program generator. In contrast to computing a fixed point solution on the code producer side, checking can be done in a single pass.
Abstract Parsing

- Instead of executing the program and parsing the result,

\[ [e]^0 \Sigma = \{ c_1, c_2, \ldots, c_n \} \quad \text{parse}(c_i) = O/X \]

- Define abstract semantics using parse stack and execute the program on it.

\[ [\hat{e}]^0 \Sigma \{ p_{init} \} = \{ p_1, p_2, \ldots, p_n \} \]

Over-approximation of the parsing result of all the generated programs
Abstract Parsing

- Q: What should be the abstract value for Code c?
- A: Parse Stack Transition Function

\[
\begin{align*}
\text{or} & \quad \text{...} & \quad \text{a} & \quad \text{...} & \quad \text{or a} \\
\text{s}_1 & \mapsto \text{s}_1 \text{s}_5 & \quad & \quad \text{s}_1 \text{s}_5 & \mapsto \text{s}_1 \text{s}_5 \text{s}_{10} & \quad \text{s}_1 & \mapsto \text{s}_1 \text{s}_5 \text{s}_{10}
\end{align*}
\]

Code concatenation => Function Composition
Abstract Parsing

• Abstract parsing semantics of the program \( Pgm \) is used to determine whether generated programs conform to the grammar \( G \).

• If \( \text{AbstractParsing}(Pgm, G)(\{P_{\text{init}}\}) = \{P_{\text{acc}}\} \), then we can conclude that generated programs conform to the grammar \( G \). Otherwise not.
Abstract Parsing in PCC Framework

• Need *abstract parsing semantics* to certify the program.

• Semantic equations are derived from the program directly.

• Loop is the only component to require fixed-point computation.

• Certificate in our framework: *the fixed-point solution for every loop in the program.*
To ensure the termination of the analysis, we need to provide an abstraction for the infinite height domain $P$. Instead of using a particular abstract domain for $P$, we parameterize this abstract domain by providing conditions which an abstract domain $D$ needs to satisfy. $D$ should be a complete partial order (CPO) and $D$ is Galois connected with the set of parse stacks $P$. An abstracted parsing function $\text{Parse}_{\text{action}}$ is defined as a sound approximation of the parsing function $\text{Parse}$ which is defined by the LR parser generator with the safety grammar $G$.

Finally, we derive the abstract parsing semantics for $D$ as in Figure 2 (right).

Given a program generator $e$ and an empty environment $\sigma$, the analysis computes $F[e]$ (which is of type $D \rightarrow D$). To determine whether the programs generated by a program generator $e$ conform to the safety grammar, we check that the following equation holds:

$$F[\alpha[P \rightarrow D_{\sigma}(\{p_{\text{init}}\})] = \alpha[P \rightarrow D_{\sigma}(\{p_{\text{acc}}\})]$$

where $p_{\text{init}}$ and $p_{\text{acc}}$ are the initial parse stack and accepting parse stack for the safety grammar $G$.

Certificate Generation

Certificate Generation
Certificate Generation with Example

- **Safety Grammar** \( E \rightarrow id \mid (E) \mid \text{let } id E E \mid \text{or } E E \)
  1) syntactically correct, 2) contain no loops

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Example Program

```plaintext
re x `a
  `( . ,x . )
let y
  `or . a
  `,y . ,x
```

Semantic Equation
Certificate Generation with Example

Example Program

```plaintext
re x `a
  `( . ,x . )
let y
  `or . a
  `.,y . ,x
```

Semantic Equation

\[ P = X \circ Y \]
Certificate Generation with Example

Example Program

```
re x `a
  `( . ,x . )
let y
  `or . a
  ,y ,x
```

Semantic Equation

\[ P = X \circ Y \]
Certificate Generation with Example

Example Program

```
let y
  `or . a
` , y . , x
```

Semantic Equation

\[ P = X \circ Y \]

\[ Y = \lambda P. PA(PA(P, or), a) \]

ParseAction : Parse Stack x Token -> Parse Stack
Component of generated LR Parser
Certificate Generation with Example

Example Program

```
re x `a
  `( . ,x . )
let y
  `or . a
  `.y . ,x
```

Semantic Equation

\[ P = X \circ Y \]
\[ Y = \lambda P. PA(PA(P, or), a) \]
Certificate Generation
with Example

Example Program

```
re x `a
  `( . ,x . )
let y
  `or . a
  `,y ,x
```

Semantic Equation

\[
P = X \circ Y
\]

\[
Y = \lambda P. PA(PA(P, \text{or}), a)
\]

\[
T = \text{fix } \lambda T. \lambda s. (PA(s, a) \cup \lambda P. PA(P, \text{or})) \circ \lambda P. PA(P, (s)) \circ \lambda P. \text{reduce}(T(top(P)) \circ \text{tail}(P)) \circ \lambda P. PA(P, (s))
\]

\[
X = \lambda P. \text{reduce}(T(top(P)) \circ \text{tail}(P))
\]
Certificate Generation with Example

Example Program

```plaintext
re x `a
  `( . ,x . )
let y
  `or . a
  `y . ,x
```

Semantic Equation

\[ P = X \circ Y \]
\[ Y = \lambda P.\ PA(\ PA(P, \text{or}), \ a) \]
\[ T = \text{fix} \lambda T.\ \lambda s.\ \left( \PA(s, a) \cup \lambda P.\ PA(P, \ ) \right) \circ \lambda \ P.\ PA(\ P, ( ) s) \]
\[ X = \lambda P.\ \text{reduce}(T(top(P))@tail(P)) \]
Certificate Generation with Example

Example Program

```plaintext
re x `a
( , ,x , )
```

```plaintext
let y
`or . a
` , , y , , x
```

Semantic Equation

\[
P = X \circ Y
\]

\[
Y = \lambda P. PA(PA(P, or), a)
\]

\[
T = fix \lambda T. \lambda s. (PA(s, a) \cup \\
\lambda P. PA(P, ) \circ \lambda P. reduce(T(top(P))@tail(P)) \circ \lambda P. PA(P, ()s)
\]

\[
X = \lambda P. reduce(T(top(P))@tail(P))
\]
Certificate Generation with Example

Example Program

```
re x `a
   `(~) , x . )
let y
   `or . a
   `y , x
```

Semantic Equation

\[ P = X \circ Y \]
\[ Y = \lambda P. PA(PA(P, or), a) \]
\[ T = fix \lambda T. \lambda s. (PA(s, a) \cup \lambda P. PA(P, )) \circ \lambda P. \text{reduce} (T(top(P)) \circ\text{reduce} (T(top(P)) \circ\text{reduce} (T(top(P))) \circ T(top(P))) \]
\[ X = \lambda P. \text{reduce} (T(top(P)) \circ\text{reduce} (T(top(P)) \circ\text{reduce} (T(top(P))) \circ T(top(P))) \]
Certificate Generation with Example

Example Program

```plaintext
re x `a
  `( . ,x )
let y
  `or . a
  `,.y . ,x
```

Semantic Equation

\[ P = X \circ Y \]
\[ Y = \lambda P. PA(P, or), a \]
\[ T = \text{fix} \lambda T. \lambda s. (PA(s, a) \cup \lambda P. PA(P, )) \circ \lambda P. \text{reduce}(T(top(P))@tail(P)) \circ \lambda P. PA(P, ()s) \]
\[ X = \lambda P. \text{reduce}(T(top(P))@tail(P)) \]
Certificate Generation with Example

Example Program

re x `a
  `( . ,x )
let y
  `or . a
  `,y . ,x

Semantic Equation

\[ P = X \circ Y \]
\[ Y = \lambda P. PA( PA(P, or), a ) \]
\[ T = fix \lambda T. \lambda s. ( PA(s, a) \cup \lambda P . PA(P, ) ) \circ \lambda P . reduce( T(top(P))@tail(P)) \circ \lambda P . PA(P, ()s) \]
\[ X = \lambda P . reduce( T(top(P))@tail(P)) \]
Certificate Generation with Example

Example Program

```
re x `a
  `( . , x , )
let y
  `or . a
  `, , y , , , , x
```

Semantic Equation

\[
P = X \circ Y
\]
\[
Y = \lambda P. PA(P, or), a)
\]
\[
T = \text{fix} \lambda T. \lambda s. (PA(s, a) \cup 
    \lambda P. PA(P, )) \circ \lambda P. reduce(T(top(P))@tail(P)) \circ \lambda P. PA(P, ()s)
\]
\[
X = \lambda P. reduce(T(top(P))@tail(P))
\]

T : Parse State -> Set of Parse Stack
Record the difference of the loop input and output.
Certificate Generation with Example

Example Program

```
re x `a
  `( . ,x )
let y
  `or . a
  `,y ,x
```

Semantic Equation

\[
\begin{align*}
P &= X \circ Y \\
Y &= \lambda P. PA(P, \text{or}, a) \\
T &= \text{fix} \lambda T. \lambda s. (PA(s, a) \cup \lambda P. PA(P, )) \circ \lambda P. \text{reduce}(T(top(P))@tail(P)) \circ \lambda P. PA(P, ()s) \\
X &= \lambda P. \text{reduce}(T(top(P))@tail(P))
\end{align*}
\]

Need to compute fixed-point \(T\)!
Certificate Generation with Example

Semantic Equation

\[ P = X \circ Y \]
\[ Y = \lambda P. PA(P, \text{or}, a) \]

\[ P(s_1) = X \circ Y(s_1) = X \circ PA(PA(s_1, \text{or}), a) \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ P = X \circ Y \]

\[ Y = \lambda P. PA(P, \text{or}), a \]

\[ P(s_1) = X \circ Y(s_1) \]

\[ = X \circ PA(PA(s_1, \text{or}), a) \]

\[ = X \circ PA(s_5s_1, a) \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ P = X \circ Y \]
\[ Y = \lambda P.PA(PA(P, or), a) \]

\[ P(s_1) = X \circ Y(s_1) \]
\[ = X \circ PA(PA(s_1, or), a) \]
\[ = X(s_8s_5s_1) \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ X = \lambda P. \text{reduce}(T(\text{top}(P))@\text{tail}(P)) \]

\[ X(s_8s_5s_1) = \text{reduce}(T(s_8)s_5s_1) \]

We need \( T(s_8) \).

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix} \lambda T. \lambda s. (PA(s, a) \cup \lambda P.PA(P, )) \circ \lambda P.\text{reduce}(T(top(P))@tail(P)) \circ \lambda P.PA(P, ()s) \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix} \lambda T. \lambda s. (PA(s, a) \cup \lambda P.PA(P, ()) \circ \lambda P.\text{reduce}(T(top(P))@tail(P)) \circ \lambda P.PA(P, ())) \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix}\lambda T.\lambda s.(PA(s, a) \cup \lambda P.PA(P, )) \circ \lambda P.\text{reduce}(T(top(P))@tail(P)) \circ \lambda P.PA(P, ())s \]

\[ s_8 \rightarrow s_4s_8 \]

\[ s_8 \leftrightarrow s_9s_8 \cup \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[
T = \text{fix } \lambda T. \lambda s. (PA(s, a) \cup \lambda P. PA(P, \epsilon)) \circ \lambda P. \text{reduce}(T(\text{top}(P)) \circ \text{tail}(P)) \circ \lambda P. PA(P, \epsilon)s)
\]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix} \lambda T. \lambda s. (PA(s, a) \cup \lambda P.PA(P,) \circ \lambda P.\text{reduce}(T(top(P))@\text{tail}(P)) \circ \lambda P.PA(P,())s) \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix } \lambda T. \lambda s. (PA(s, a) \cup \lambda P. PA(P, \)) \circ \lambda P. \text{reduce}(T(top(P)) \circ \text{tail}(P)) \circ \lambda P. PA(P, (s) \) \]

Part of the LR(0) parsing controller for the safety grammar

1st Iteration
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix} \lambda T. \lambda s. (PA(s, a)) \cup \lambda P. PA(P, ) \circ \lambda P. \text{reduce}(T(top(P))@\text{tail}(P)) \circ \lambda P. PA(P, ()\lambda P. PA(P, ()s) \to s_4 s_4 \]

\[ s_8 \to s_9 s_8 \]

\[ s_4 \to s_6 s_4 \cup \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix}_T \lambda s. (PA(s, a)) \cup \lambda P. PA(P, (\text{reduce}(T(top(P)) \circ \text{ail}(P)) \circ \lambda P. PA(P, (()))s) \circ T(s_4)? \]

\[
\begin{align*}
s_7 & : E \rightarrow (E) \\
s_6 & : E \rightarrow (E.) \\
s_2 & : S \rightarrow E \\
s_1 & : S \rightarrow .E \\
s_3 & : E \rightarrow id. \\
s_4 & : E \rightarrow (.E) \\
s_9 & : E \rightarrow or.E.E \\
s_5 & : E \rightarrow or.E.E \\
s_8 & : E \rightarrow or.E.E \\
s_8 & \rightarrow s_9 s_8 \\
s_4 & \rightarrow s_6 s_4
\end{align*}
\]

Part of the LR(0) parsing controller for the safety grammar

1st Iteration
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix } \lambda T. \lambda s. (PA(s, a) \cup \lambda P.PA(P, )) \circ \lambda P.\text{reduce}(T(top(P))@tail(P)) \circ \lambda P.PA(P, ()s) \]

Part of the LR(0) parsing controller for the safety grammar

\[ S \rightarrow s_1 \rightarrow \ldots \rightarrow s_8 \rightarrow s_9 s_8 \]

\[ E \rightarrow s_2 \rightarrow \ldots \rightarrow s_4 \rightarrow s_6 s_4 \]
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix}\lambda T.\lambda s.(PA(s, a) \cup \lambda P.PA(P, )) \circ \lambda P.\text{reduce}(T(top(P)) \circ \text{ail}(P)) \circ \lambda P.PA(P, ()s) \]

\[ T(s_4) = s_6 s_4 \quad s_8 \rightarrow s_4 s_8 \]

\[ s_8 \rightarrow s_9 s_8 \]

\[ s_4 \rightarrow s_6 s_4 \]

Part of the LR(0) parsing controller for the safety grammar

2nd Iteration
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix} \lambda T. \lambda s. (PA(s, a) \cup \lambda P.PA(P, \text{or}) \circ \lambda P.\text{reduce}(T(top(P)) \circ \lambda P.PA(P, \text{or}) \circ \lambda P.\text{reduce}(s6s4s8) = s6s4s8 \quad s8 \rightarrow s4s8} \]

\[ s8 \rightarrow s9s8 \]

\[ s4 \rightarrow s6s4 \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix}\lambda T.\lambda s.(PA(s, a) \cup \lambda P.PA(P, s)) \circ \lambda P.\text{reduce}(T(top(P))@tail(P)) \circ \lambda P.PA(P, (\cdot)s) \]

\[ s_4s_8 \rightarrow s_6s_4s_8 \quad s_8 \rightarrow s_4s_8 \]

\[ s_8 \mapsto s_9s_8 \]

\[ s_4 \mapsto s_6s_4 \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix} \lambda T. \lambda s. (PA(s, a)) \cup \lambda P. \text{PA}(P, ()) \circ \lambda P. \text{reduce}(T(\text{top}(P)) \circ \text{tail}(P)) \circ \lambda P. \text{PA}(P, ())s \]

\[ s_6 s_4 s_8 \rightarrow s_9 s_8 \]

\[ s_4 s_8 \rightarrow s_6 s_4 s_8 \]

\[ s_8 \rightarrow s_4 s_8 \]

\[ s_8 \mapsto s_9 s_8 \]

\[ s_4 \mapsto s_6 s_4 \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix } \lambda T. \lambda s. (PA(s, a)) \cup \lambda P.PA(P, ) \circ \lambda P.\text{reduce}(T(\text{top}(P))@\text{tail}(P)) \circ \lambda P.PA(P, ()s) \]

Part of the LR(0) parsing controller for the safety grammar

Not Changed!

\[ S_8 \mapsto S_9S_8 \]

\[ S_4 \mapsto S_6S_4 \]

2nd Iteration
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix} \lambda T. \lambda s. (PA(s, a) \cup \lambda P. PA(P, )) \circ \lambda P. \text{reduce}(T(top(P)) \circ \text{tail}(P)) \circ \lambda P. PA(P, ()s) \]

\[ T(s_4) = s_6 s_4 \quad s_4 \rightarrow s_4 s_4 \]

\[ s_8 \rightarrow s_9 s_8 \]

\[ s_4 \rightarrow s_6 s_4 \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix} \lambda T. \lambda s. (PA(s, a)) \cup \lambda P. PA(P, \lambda s. \text{reduce}(T(\text{top}(P)) \circ \text{tail}(P))) \circ \lambda P. PA(P, \lambda s. \text{reduce}(s_6 s_4 s_4)) = s_6 s_4 s_4 \quad s_4 \rightarrow s_4 s_4 \]

\[ s_8 \mapsto s_9 s_8 \]

\[ s_4 \mapsto s_6 s_4 \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix} \lambda T. \lambda s. (PA(s, a) \cup \lambda P.PA(P, ()) \circ \lambda P.\text{reduce}(T(top(P))@tail(P)) \circ \lambda P.PA(P, ()s) \]

Part of the LR(0) parsing controller for the safety grammar

2nd Iteration
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix} \lambda T. \lambda s. ((PA(s, a)) \cup \lambda P. PA(P, \ )) \circ \lambda P. \text{reduce}(T(\text{top}(P)) \@ \text{tail}(P)) \circ \lambda P. PA(P, \ )) s \]

Part of the LR(0) parsing controller for the safety grammar

2nd Iteration
Certificate Generation with Example

Semantic Equation

\[ T = \text{fix} \lambda T. \lambda s. (PA(s, a) \cup \lambda P. PA(P, \epsilon) \circ \lambda P. \text{reduce}(T(top(P)) \cdot \text{tail}(P)) \circ \lambda P. PA(P, \epsilon)s) \]

Part of the LR(0) parsing controller for the safety grammar
Certificate Generation with Example

- Code producer sends the program and computed fixed-point solution.

Program

\[
\text{re } x \ `a \\
\text{ `( . ,x . )} \\
\text{let y} \\
\text{ `or . a} \\
\text{` ,y . ,x}
\]

Fixed-point Solution

\[
\begin{align*}
\text{s}_8 & \mapsto s_9 s_8 \\
\text{s}_4 & \mapsto s_6 s_4
\end{align*}
\]
To ensure the termination of the analysis, we need to provide an abstraction for the infinite height domain $P$. Instead of using a particular abstract domain for $P$, we parameterize this abstract domain by providing conditions which an abstract domain $D$ needs to satisfy.

$D$ should be a complete partial order (CPO), and $D$ is Galois connected with the set of parse stacks $P$. An abstracted parsing function $\text{Parse action}$ is defined as a sound approximation of the parsing function $\text{Parse action}$ which is defined by the LR parser generator with the safety grammar $G$.

Figure 1 illustrates a PCC framework for program generators. An abstraction carrying code framework (ACCF) is specialized to program generators by means of abstract parsing. The code producer and code consumers share the safety grammar which specifies the safety properties of the generated programs.

The code producer proves the safety of the program generator by abstract parsing with the shared safety grammar. In a complex and iterative process, the analysis computes a fixed-point solution. This solution is used as a certificate for the safety of the program generator. The code producer uploads or sends the program generator with the computed fixed-point solution.

The code consumer downloads or receives the untrusted program generator and its attached fixed-point solution. The code consumer validates that the received fixed-point solution is indeed a fixed-point solution of the received program generator. In contrast to computing a fixed-point solution on the code producer side, checking can be done in a single pass.
Certificate Check with Example

Received Program

```plaintext
re x `a
  `( . ,x . )
let y
  `or . a
  `,y . ,x
```

Semantic Equation

1. From the received program, derive semantic equations.

\[
P = X \circ Y
\]

\[
Y = \lambda P. PA(P, \text{or}) \circ \text{a}
\]

\[
T = \text{fix} \lambda T. \lambda s. (PA(s, \text{a}) \cup \lambda P. PA(P, \text{or})) \circ \lambda P. \text{reduce}(T(top(P))@tail(P)) \circ \lambda P. PA(P, \epsilon)s)
\]

\[
X = \lambda P. \text{reduce}(T(top(P))@tail(P))
\]
Certificate Check with Example

Received Fixed-point solution

\[ s_8 \mapsto s_9 s_8 \]
\[ s_4 \mapsto s_6 s_4 \]

2. Check that the received solution is indeed the fixed-point for the program.
(one iteration is enough)

Semantic Equation

\[
P = X \circ Y
\]
\[
Y = \lambda P. PA(PA(P, \text{or}), a)
\]
\[
T = \text{fix} \lambda T. \lambda s. (PA(s, a) \cup \lambda P. PA(P, \text{or}) \circ \lambda P. PA(P, \text{and})) \circ \lambda P. PA(P, \text{and})) \circ \lambda P. PA(P, \text{and}))
\]
\[
X = \lambda P. \text{reduce}(T(\text{top}(P))@\text{tail}(P))
\]
Certificate Check with Example

Received Fixed-point solution

\[ s_8 \mapsto s_9 s_8 \]

\[ s_4 \mapsto s_6 s_4 \]

Semantic Equation

\[ P = X \circ Y \]

\[ Y = \lambda P. PA(P, \text{or}, a) \]

\[ T = \text{fix} \lambda T. \lambda s. (PA(s, a) \cup \lambda P. PA(P, s)) \circ \lambda P. PA(P, ()s) \]

\[ X = \lambda P. \text{reduce}(T(\text{top}(P)) \circ \text{tail}(P)) \]

3. Using the fixed-point solution, construct abstract parsing semantics of the program.

\[ P = \{ s_1 \mapsto s_1 s_2 \} \]
Certificate Check with Example

Received Fixed-point solution

\[ s_8 \mapsto s_9 s_8 \]
\[ s_4 \mapsto s_6 s_4 \]

Semantic Equation

\[ P = X \circ Y \]
\[ Y = \lambda P. PA(P, \text{or}), a \]
\[ T = \text{fix} \lambda T. \lambda s. (PA(s, a) \cup \lambda P. PA(P, \text{or})) \circ \lambda P. PA(P, (\text{)}s) \]
\[ X = \lambda P. \text{reduce}(T(top(P))@\text{tail}(P)) \circ \lambda P. PA(P, (\text{)}s) \]

3. Using the fixed-point solution, construct abstract parsing semantics of the program.

\[ P = \{ s_1 \mapsto s_1 s_2 \} \]
Summary

- Our framework addresses two fundamental PCC issues.

  1. The certificate, a fixed-point solution, is generated automatically by abstract parser.

  2. Checking procedure on the code consumer side is done efficiently by validating the received fixed-point solution.
Two issues need further investigation.

1. Size of the certificate:

   Certificate in our framework: 
   \textit{the fixed-point solution for every loop in the program}.

   - $O(# \text{ of loops})$ : linear to the program size
Issues

• Two issues need further investigation.

1. Size of the certificate:
   - $O(\# \text{ of parse states})$
   # of parse states is fixed with the given grammar.

\[
\text{ParseState} \rightarrow 2^{\text{ParseStack}}
\]
Issues

• Two issues need further investigation.

2. Complexity of the checker:
   - As complex as the certificate generator
     Need to derive the same semantic equations.
     Need to implement all the abstract operators.
   - Shared problem with other abstract-carrying code frameworks.
Future Work

• Work is in progress
  - Implement the abstract parser and do the experiment.
Thank You