PCC Framework for Program-Generators *

Soonho Kong Wontae Choi Kwangkeun Yi
Seoul National University
{soon,wtchoi,kwang}@ropas.snu.ac.kr

Abstract

In this paper, we propose a proof-carrying code framework for program-generators. The enabling technique is abstract parsing, a static string analysis technique, which is used as a component for generating and validating certificates. Our framework provides an efficient solution for certifying program-generators whose safety properties are expressed in terms of the grammar representing the generated program. The fixed-point solution of the analysis is generated and attached with the program-generator on the code producer side. The consumer receives the code with a fixed-point solution and validates that the received fixed point is indeed a fixed point of the received code. This validation can be done in a single pass.

1 Introduction

To certify the safety of a mobile program-generator, we need to ensure not only the safe execution of the generator itself but also that of the generated programs. Safety properties of the generated programs are specified efficiently in terms of the grammar representing the generated programs. For instance, the safety property "generated programs should not have nested loops" can be specified and verified by the reference grammar for the generated programs.

Recently, Doh, Kim, and Schmidt presented a powerful static string analysis technique called abstract parsing [4]. Using LR parsing as a component, abstract parsing analyzes the program and determines whether the strings generated in the program conform to the given grammar or not.

In this paper, we propose a Proof-Carrying Code (PCC) framework [8, 9] for program-generators. We adapt abstract parsing to check the generated programs of the program-generators. With the grammar specifying the safety property of the generated programs, the code producer abstract-parses the program-generator and computes a fixed-point solution as a certificate. The code producer sends the program-generator with the computed fixed-point solution. The code consumer receives the program-generator accompanied with the fixed-point solution and validates that the received fixed point is indeed the solution for the received program-generator. Our framework can be seen as an abstraction-carrying code framework [1, 5] specialized to program-generators which is modeled by a two-staged language with concatenation.

This work is, to our knowledge, the first to present a proof-carrying code framework that certifies grammatical properties of the generated programs. Directly computing the parse stack information as a form of the fixed-point solution, abstract parsing provides an efficient way to validate the certificates on the code consumer side. In contrast to abstract parsing, the previous static string analysis techniques [3, 7, 2] approximate the possible values of a string expression of the program with a grammar and see whether the approximated grammar is included in the reference grammar. This grammar inclusion check takes too much time and makes those techniques difficult to be used as a validation component of a PCC framework.

*This work was supported by the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University in 2009 and the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF). (R11-2008-007-01002-0).

2 Language

For the further development of our idea, we consider a two-staged language with concatenation in which program-generators can be modeled. The language is an imaginary, first-order language whose only value is code. The language is minimal, so as not to distract our focus on static analysis. For example, loops and conditional jumps are without the condition expression, for which abstract interpretation anyway considers all iterations and all branches.

A program is an expression e:

$$e \in \mathit{Exp} ::= x \mid \mathtt{let}\, x\, e_1\, e_2 \mid \mathtt{or}\, e_1\, e_2 \mid \mathtt{re}\, x\, e_1\, e_2\, e_3 \mid \text{`}f$$

An expression can contain code fragments f:

$$f \in \mathit{Frag} ::= x \mid \mathtt{let} \mid \mathtt{or} \mid \mathtt{re} \mid \mathtt{(} \mid \mathtt{)} \mid f_1.f_2 \mid \mathtt{,}e$$

Operational semantics of the language is defined in Figure 3 (left).

Expression $\mathtt{or}\, e_1\, e_2$ is for branches. It could be the value of e_1 or the value of e_2. Expression $\mathtt{re}\, x\, e_1\, e_2\, e_3$ is for loops. Variable x has the value of e_1 as its initial value. Loop body e_2 is iterated ≥ 0 times. The result of each iteration e_2 will be bound to x in e_2 for next iteration or in e_3 for the result of the loop. Backquote form $\text{`}f$ is for code fragment $f$. We construct the fragment by using the following tokens: variables, `let`, `or`, `re`, `(`, and `)`. Compound fragment $f_1.f_2$ concatenates two code fragments $f_1$ and $f_2$. Comma fragment $\mathtt{,}e$ first evaluates $e$ then substitutes its result code value for itself. Note that the meaning of $\text{`}f$ and $\mathtt{,}e$ is the same as in LISP's quasi-quotation system.

3 Abstract Parsing

In our framework, we use abstract parsing [4] as a component to generate and validate the certificate. Abstract parsing derives data-flow equations from the program and solves them in the parsing domain. In [6], we formulated abstract parsing in the abstract interpretation framework.

The key idea of abstract parsing is an abstraction of code. Code c is abstracted into a parse-stack transition function $f = \lambda p.\mathit{parse}(p,c)$ where parse is a parsing function defined by an LR parser generator with the safety grammar G. This choice of abstraction is necessary to handle code concatenation $x.y$. If abstracted functions for the code fragments x and y are $f_x = \lambda p.\mathit{parse}(p,x)$ and $f_y = \lambda p.\mathit{parse}(p,y)$ respectively, an abstracted function for the code concatenation $x.y$ is constructed by function composition of f_x and f_y as $f_{x.y} = f_y \circ f_x$.

As illustrated in Figure 1, we take a series of abstraction steps for the value domain of the semantics.

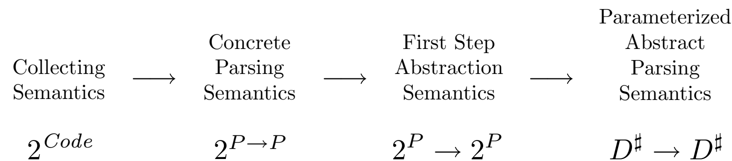

Figure 1: Series of abstraction steps for the value domain in semantics where P is the set of parse stacks.

Starting from the collecting semantics defined in Figure 3 (middle), each abstraction of the value domain derives new abstract semantics.

To ensure the termination of the analysis, we need to provide an abstraction for the infinite height domain 2^P. Instead of using a particular abstract domain for 2^P, we parameterize this abstract domain by providing conditions which an abstract domain $D^\sharp$ needs to satisfy.

1. $D^\sharp$ should be a complete partial order (CPO).
2. $D^\sharp$ is Galois connected with the set of parse stacks 2^P.
3. An abstracted parsing function $Parse_action^\sharp$ is defined as a sound approximation of the parsing function $Parse_action$ which is defined by the LR parser generator with the safety grammar G.

Finally, we derive the abstract parsing semantics for $D^\sharp$ as in Figure 3 (right).

Given a program-generator e and an empty environment σ_0, the analysis computes $F = [\![e]\!]^0_{D^\sharp}\sigma_0$ which is of type $D^\sharp \to D^\sharp$. To determine whether the programs generated by a program-generator e conform to the safety grammar, we check that the following equation holds:

$$F(\alpha_{2^P \to D^\sharp}(\{p_{init}\})) = \alpha_{2^P \to D^\sharp}(\{p_{acc}\})$$

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

4 PCC Framework for Program-Generators

Figure 2 illustrates a PCC framework for program-generators, an abstraction-carrying code framework [1, 5] 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.

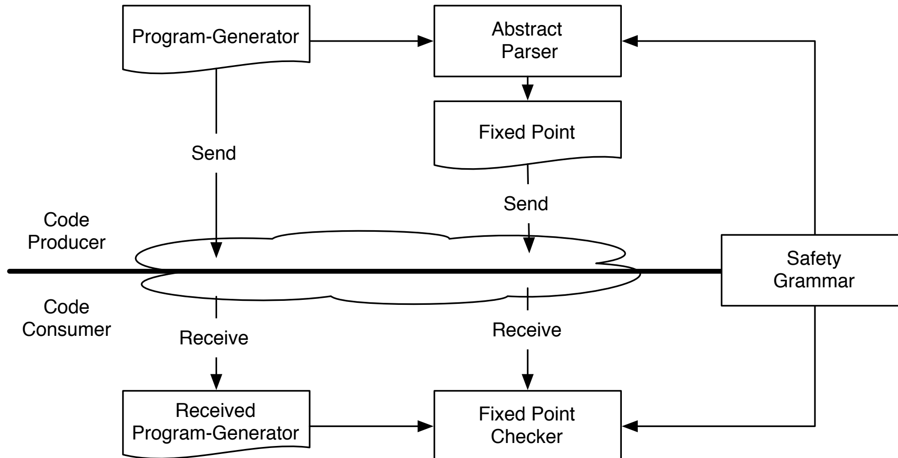

Figure 2: A proof-carrying code framework for program-generators.

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.

5 Issues

The proposed framework addresses two fundamental PCC issues.

1. The certificate, a fixed-point solution for the program-generator, is generated automatically by abstract parsing.

2. Checking procedure on the code consumer side is done efficiently by validating the received fixed-point solution.

However, we have several issues for further investigation.

1. Size of the certificate: We are not sure that the size of the fixed-point solution which our framework generates is small enough for the mobile platform. However, there are some ideas on reducing the size of certificates. First, the certificate can be compressed. Abstract parsing uses an abstract parse stack as a component of the value domain. Since a parse stack is a string of characters from a pre-defined finite alphabet, an appropriate compression algorithm can be used to reduce the size of fixed-point solution. Second, some parts of the certificate could be deleted as long as their recovery takes linear time to the size of the received code.

2. Size of the trust base: Similar to other abstraction-carrying code frameworks, the certificate checker of our framework is almost as complex as the certificate generator. It is essential to simplify the certificate checker to reduce the size of the trust base.

References

[1] E. Albert, G. Puebla, and M. Hermenegildo. Abstract interpretation-based approach to mobile code safety. In *Proceedings of Compiler Optimization meets Compiler Verification*, 2004.

[2] Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung-Goo Doh. A practical string analyzer by the widening approach. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, volume 4729 of *Lecture Notes in Computer Science*, pages 374–388, Sydney, Austrailia, November 2006. Springer-Verlag.

[3] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the Static Analysis Symposium*, pages 1–18. Springer-Verlag, 2003.

[4] Kyung-Goo Doh, Hyunha Kim, and David Schmidt. Abstract parsing: static analysis of dynamically generated string output using LR-parsing technology. In *Proceeeding of the International Static Analysis Symposium*, 2009. Available from `http://santos.cis.ksu.edu/schmidt/dohsas09.pdf`.

[5] Manuel V. Hermenegildo, Elvira Albert, Pedro López-García, and Germán Puebla. Abstraction carrying code and resource-awareness. In *Proceedings of the ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 1–11, New York, NY, USA, 2005. ACM.

[6] Soonho Kong, Wontae Choi, and Kwangkeun Yi. Abstract parsing for two-staged languages with concatenation. In *Proceeeding of the International Conference on Generative Programming and Component Engineering*, 2009. Available from `http://ropas.snu.ac.kr/~soon/paper/gpce09.pdf`.

[7] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the International Conference on World Wide Web*, pages 432–441, New York, NY, USA, 2005. ACM.

[8] George C. Necula. Proof-carrying code. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[9] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, New York, NY, USA, 1998. ACM.

$$\sigma \in Env = Var \to Code$$
$$v \in Code = Token\,sequence$$
$$e \in Exp$$
$$f \in Frag$$

$\boxed{\sigma \vdash^0 e \Rightarrow v}$

$$\frac{}{\sigma \vdash^0 x \Rightarrow \sigma(x)} \quad \text{(variable)}$$

$$\frac{\sigma \vdash^0 e_1 \Rightarrow v \qquad \sigma[x \mapsto v] \vdash^0 e_2 \Rightarrow v'}{\sigma \vdash^0 \texttt{let}\, x\, e_1\, e_2 \Rightarrow v'} \quad \text{(let binding)}$$

$$\frac{\sigma \vdash^0 e_1 \Rightarrow v}{\sigma \vdash^0 \texttt{or}\, e_1\, e_2 \Rightarrow v} \qquad \frac{\sigma \vdash^0 e_2 \Rightarrow v}{\sigma \vdash^0 \texttt{or}\, e_1\, e_2 \Rightarrow v} \quad \text{(branch)}$$

$$\frac{\sigma \vdash^0 e_1 \Rightarrow v \qquad \sigma[x \mapsto v] \vdash^0 loop\, x\, e_2\, e_3 \Rightarrow v'}{\sigma \vdash^0 \texttt{re}\, x\, e_1\, e_2\, e_3 \Rightarrow v'} \quad \text{(loop)}$$

$$\frac{\sigma \vdash^0 e_2 \Rightarrow v \qquad \sigma[x \mapsto v] \vdash^0 loop\, x\, e_2\, e_3 \Rightarrow v'}{\sigma \vdash^0 loop\, x\, e_2\, e_3 \Rightarrow v'} \qquad \frac{\sigma \vdash^0 e_3 \Rightarrow v}{\sigma \vdash^0 loop\, x\, e_2\, e_3 \Rightarrow v}$$

$$\frac{\sigma \vdash^1 f \Rightarrow v}{\sigma \vdash^0 \text{`} f \Rightarrow v} \quad \text{(back quote)}$$

$\boxed{\sigma \vdash^1 f \Rightarrow v}$

$$\frac{}{\sigma \vdash^1 x \Rightarrow x} \qquad \frac{}{\sigma \vdash^1 \texttt{let} \Rightarrow \texttt{let}} \qquad \frac{}{\sigma \vdash^1 \texttt{or} \Rightarrow \texttt{or}} \quad \text{(token)}$$

$$\frac{}{\sigma \vdash^1 \texttt{re} \Rightarrow \texttt{re}} \qquad \frac{}{\sigma \vdash^1 \texttt{(} \Rightarrow \texttt{(}} \qquad \frac{}{\sigma \vdash^1 \texttt{)} \Rightarrow \texttt{)}}$$

$$\frac{\sigma \vdash^1 f_1 \Rightarrow v_1 \qquad \sigma \vdash^1 f_2 \Rightarrow v_2}{\sigma \vdash^1 f_1 . f_2 \Rightarrow v_1 v_2} \quad \text{(concatenation)}$$

$$\frac{\sigma \vdash^0 e \Rightarrow v}{\sigma \vdash^1 \texttt{,} e \Rightarrow v} \quad \text{(comma)}$$

Operational semantics of the language.

$$Code = Token\,sequence$$
$$\sigma \in Env = Var \to Code$$
$$[\![e]\!]^0 \in 2^{Env} \to 2^{Code} \qquad \text{(stage-0 semantics)}$$
$$[\![f]\!]^1 \in 2^{Env} \to 2^{Code} \qquad \text{(stage-1 semantics)}$$

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

Collecting semantics of the language.

$$\sigma \in Env_{D^\sharp} = Var \to V^\sharp$$
$$[\![e]\!]^0_{D^\sharp} \in Env_{D^\sharp} \to V^\sharp \qquad \text{(stage-0 abstract semantics)}$$
$$[\![f]\!]^1_{D^\sharp} \in Env_{D^\sharp} \to V^\sharp \qquad \text{(stage-1 abstract semantics)}$$

$$[\![x]\!]^0_{D^\sharp}\sigma = \sigma(x)$$
$$[\![\texttt{let}\, x\, e_1\, e_2]\!]^0_{D^\sharp}\sigma = [\![e_2]\!]^0_{D^\sharp}(\sigma[x \mapsto [\![e_1]\!]^0_{D^\sharp}\sigma])$$
$$[\![\texttt{or}\, e_1\, e_2]\!]^0_{D^\sharp}\sigma = [\![e_1]\!]^0_{D^\sharp}\sigma \sqcup [\![e_2]\!]^0_{D^\sharp}\sigma$$
$$[\![\texttt{re}\, x\, e_1\, e_2\, e_3]\!]^0_{D^\sharp}\sigma = [\![e_3]\!]^0_{D^\sharp}(\sigma[x \mapsto fix\,\lambda k.[\![e_1]\!]^0_{D^\sharp}\sigma \sqcup [\![e_2]\!]^0_{D^\sharp}(\sigma[x \mapsto k])])$$
$$[\![\text{`} f]\!]^0_{D^\sharp}\sigma = [\![f]\!]^1_{D^\sharp}\sigma$$
$$[\![t]\!]^1_{D^\sharp}\sigma = \lambda D.Parse_action^\sharp(D,t)$$
$$[\![f_1 . f_2]\!]^1_{D^\sharp}\sigma = [\![f_2]\!]^1_{D^\sharp}\sigma \circ [\![f_1]\!]^1_{D^\sharp}\sigma$$
$$[\![,e]\!]^1_{D^\sharp}\sigma = [\![e]\!]^0_{D^\sharp}\sigma$$

Abstract parsing semantics of the language.

Figure 3: Operational semantics, collecting semantics, and abstract parsing semantics of the language.