Higher-Order Program Verification and Language-Based Security
(Extended Abstract)

Naoki Kobayashi

Tohoku University

Abstract. Language-based security has been a hot research area of computer security in the last decade. It addresses various concerns about software security by using programming language techniques such as type systems and program analysis/transformation. Thus, advance in programming language research can also benefit language-based security. This paper reports some recent advance in verification techniques for higher-order programs, and discusses its applications to language-based security. More specifically, we summarize the recent result on model-checking of *higher-order recursion schemes*, and show how it may be applied to language-based security such as secure information flow and stack-based access control.

1 Recursion Schemes and Program Verification

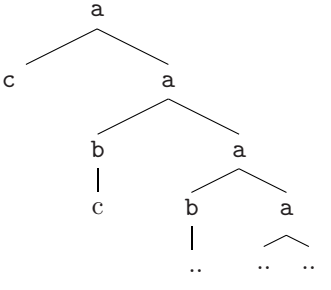
A higher-order recursion scheme (recursion scheme, for short) is a grammar for describing an infinite tree. Unlike regular tree grammars, non-terminal symbols can take trees or higher-order functions on trees as parameters. For example, the following is an order-1 recursion scheme, where the non-terminal F takes a tree as a parameter.

$$S \to F\,\mathtt{c}$$
$$F\,x \to \mathtt{a}\,x\,(F\,(\mathtt{b}\,x))$$

The start symbol S is reduced as follows:

$$S \longrightarrow F\,\mathtt{c} \longrightarrow \mathtt{a}\,\mathtt{c}\,(F\,(\mathtt{b}\,\mathtt{c})) \longrightarrow \mathtt{a}\,\mathtt{c}\,(\mathtt{a}\,(\mathtt{b}\,\mathtt{c})\,(F\,(\mathtt{b}(\mathtt{b}\,\mathtt{c}))))) \longrightarrow \cdots,$$

and the following infinite tree is generated.



A. Datta (Ed.): ASIAN 2009, LNCS 5913, pp. 17–23, 2009.

From a programming language point of view, a higher-order recursion scheme is a simply-typed call-by-name functional program with recursion and tree constructors (but without destructors).

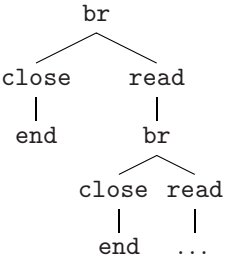
A major breakthrough was brought by Ong [1], who showed that the modal μ-calculus model checking of recursion schemes ("Given a recursion scheme $\mathcal{G}$ and a modal μ-calculus formula φ, does the tree generated by $\mathcal{G}$ satisfy φ?") is decidable.[1] The model-checking of recursion schemes subsumes finite-state model checking and pushdown model checking, as recursion schemes are at least as expressive as higher-order pushdown systems (for describing trees) [2]. We have recently applied the model checking of recursion schemes to various verification problems of higher-order programs, including reachability, flow analysis, resource usage verification [3], and exact type checking of XML-processing programs [4,5,6]. The idea behind those pieces of work is that, given a simply-typed functional program with recursion, one can transform it into a recursion scheme that generates a tree containing all information about interesting event sequences or program output. For example, consider the following program that accesses file "foo" [4]:

```
let rec f x = if * then close(x) else read(x); f x in
let fp = open_in "foo" in f(fp)
```

It can be translated into the following recursion scheme $\mathcal{G}$:

$$S \to F\,\mathtt{fp}\,\mathtt{end} \qquad F\,x\,k \to \mathtt{br}\,(\mathtt{close}\,k)\,(\mathtt{read}\,(F\,x\,k))$$

It generates the following tree:



Note that the tree above represents all the access sequences to file "foo", so that, by model-checking the recursion scheme, one can verify that the file is accessed in a valid manner.

The worst-case time complexity of model-checking recursion schemes is n-EXPTIME (where n is the largest order of the type of a non-terminal: see [1,7]). Thus, one may think that the model-checking of recursion schemes is only of theoretical interest. A type-based model checking algorithm, however, has recently been developed that actually runs reasonably fast for recursion schemes obtained from various program verification problems [5,6].

[1] In prior to Ong's result, Knapik et al. [2] showed the decidability for a subclass of recursion schemes called *safe* recursion schemes. From the viewpoint of application to program verification, however, the safety restriction is rather restrictive.

2 Applications to Language-Based Security

Based on the results summarized above, it is natural to expect that one can also apply model-checking of recursion schemes to language-based security. We consider a simply-typed, call-by-value λ-calculus with recursion and booleans below, and sketch that non-interference and stack-based access control are decidable for that language.[2]

The syntax of the language is:

$$e ::= \mathbf{true} \mid \mathbf{false} \mid x \mid e_0 e_1 \mid \lambda x.e \mid \mathbf{fun}(f, x, e) \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$$

Here, $\mathbf{fun}(f, x, e)$ denotes a recursive function. The type system and the call-by-value operational semantics are defined as usual.

2.1 Non-interference

The non-interference property defined below is a formal criterion for secure information flow. It states that information about an input value does not flow to the output; see [8] for a nice survey of secure information flow.

Definition 1 (non-interference). *Let e be a term of type* $\mathbf{bool} \to \mathbf{bool}$. *The term e satisfies* non-interference *if, for every $b \in \{\mathbf{true}, \mathbf{false}\}$, $e(\mathbf{true})$ evaluates to b if and only if $e(\mathbf{false})$ evaluates to b.*

Given a term e' of type **bool**, one can easily transform it into a recursion scheme that generates a tree consisting of a single node b just if e' evaluates to b. For example, consider the following program:

```
let g x = if x then false else true in
let f x = if x then g x else false in
  f true
```

As usual, $\mathbf{let}\ f(x) = e_0\ \mathbf{in}\ e_1$ is an abbreviated form of $(\lambda f.e_1)\mathbf{fun}(f, x, e_0)$. It can be transformed into the following recursion scheme:

$$\begin{array}{l} S \to F\ \mathit{True}\ \texttt{true}\ \texttt{false} \\ G\ x\ t\ f \to x\ \mathit{False}\ \mathit{True}\ t\ f \\ F\ x\ t\ f \to x\ (G\ x)\ \mathit{False}\ t\ f \\ \mathit{True}\ t\ f \to t \\ \mathit{False}\ t\ f \to f \end{array}$$

Here, S corresponds to the main body of the program, and G and F correspond to functions g and f respectively. (The reason why G and F take additional parameters is that both sides of each rule of a recursion scheme must have a tree type.) Booleans **true** and **false** have been encoded into $\lambda t.\lambda f.t$ and $\lambda t.\lambda f.f$ respectively. The conditional $\mathbf{if}*\ e_0\ e_1 e_2$ has been encoded into $e_0\, e_1\, e_2$ as usual.

[2] Formal proofs of the decidability are however deferred to a longer version of this paper.

It is easy to see that the recursion scheme generates a tree consisting of a single node `false`, which is the same output as the original program.

Given a term e of type $\mathbf{bool} \to \mathbf{bool}$, one can transform $e(\mathbf{true})$ and $e(\mathbf{false})$ into recursion schemes, and then apply the model checking to decide whether $e(\mathbf{true})$ and $e(\mathbf{false})$ output b for each $b \in \{\mathbf{true}, \mathbf{false}\}$. Thus, we have:

Theorem 1. *The non-interference property is decidable (for the language above).*

Note that although e does not take a higher-order function, arbitrary higher-order functions may be used inside e. The same result holds for the call-by-name language.

The above decidability result does not extend to the higher-order case. Let us write $\approx_\tau$ for the standard observational equivalence at type τ.

Definition 2 (higher-order non-interference). *Let e be a term of type $\mathbf{bool} \to \tau$. The term e satisfies* non-interference *if $e(\mathbf{true}) \approx_\tau e(\mathbf{false})$.*

From Loader's result [9], it follows that $\approx_\tau$ is undecidable in general,[3] hence so is the higher-order non-interference.

2.2 Stack-Based Access Control

Stack inspection [10,11,12] is a mechanism for controlling resource accesses based on call sequences of functions or methods. Following Pottier et al. [11],[4] we extend the language as follows:

$$e ::= \cdots \mid R[e] \mid \mathbf{enable}\ R\ \mathbf{in}\ e \mid \mathbf{check}\ R\ \mathbf{in}\ e$$

Here, R represents a set of access permissions. The expression $R[e]$ updates the current permissions (say, R_0) with $R \cap R_0$, and executes e; after evaluating e, the permissions R_0 is restored. The expression $\mathbf{enable}\ R\ \mathbf{in}\ e$ adds R to the current permissions, and executes e, and $\mathbf{check}\ R\ \mathbf{in}\ e$ checks whether R is a subset of the current permissions, and if so, executes e; otherwise the program is aborted. Please consult [11,12] for the formal semantics of those primitives.

Consider the following program.

```
let f x = Trusted[check Trusted then x else fail] in
let g x = Untrusted[f x] in
  g true
```

Here, `Trusted` (`Untrusted`, resp.) is a set of permissions given to trusted principals. The program first invokes `g` (which is untrusted), which in turn invokes

[3] Loader's undecidability is for call-by-name, finitary PCF, but one can modify it to derive the undecidability of the observational equivalence for call-by-value, finitary PCF.

[4] Pottier et al. also allows permission testing $\mathbf{test}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$. We conjecture that the decidability result below holds also with permission testing.

trusted code `f`. The permission test in `f` fails, since it has been invoked through `g`, so that the current permission is the intersection of `Untrusted` and `Trusted`.

On the other hand, the permission test in the following program succeeds, since `f` is invoked only after the call of `h` has returned.

```
let f x = Trusted[check Trusted in x] in
let h x = Untrusted[x] in
  f(h true)
```

We consider the following stack-based access control problem (SBAC, for short):

> "Given a term e of type **bool**, will e be aborted (due to the failure of a `check` opertion)?"

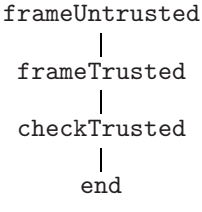
We show that SBAC can be encoded into a model-checking problem for a recursion scheme.

The idea of the encoding is to transform a term e into a recursion scheme that generates a tree consisting of sequences of framing ($R[\cdot]$), enable, and check operations.

For example, the first program above is transformed into the following recursion scheme:

$$\begin{aligned} &S \to G\ \mathit{True}\ (\lambda t.\lambda x.t)\\ &F\ x\ k \to k\ (\texttt{frameTrusted}\ (\texttt{checkTrusted}\ \texttt{end}))\ x\\ &G\ x\ k \to F\ x\ (\lambda t.\lambda x.k\ (\texttt{frameUntrusted}\ t)\ x) \end{aligned}$$

Here, we have used λ-abstractions for the sake of readability. The idea is similar to that of the CPS (continuation-passing-style) transformation. The function F takes an additional parameter k, which takes a tree representing the event sequences and a return value of F. The recursion scheme above generates the following tree:

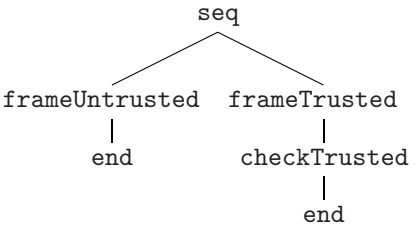


To check whether the program is aborted, it suffices to check whether `checkTrusted` occurs below `frameUntrusted`.

The second program is transformed into the following recursion scheme.

$$\begin{aligned} &S \to H\ \mathit{True}\ (\lambda t_1.\lambda x_1.(F\ x_1\ (\lambda t_2.\lambda x_2.\texttt{seq}\ t_1\ t_2)))\\ &F\ x\ k \to k\ (\texttt{frameTrusted}\ (\texttt{checkTrusted}\ \texttt{end}))\ x\\ &H\ x\ k \to k\ (\texttt{frameUntrusted}\ \texttt{end})\ x \end{aligned}$$

The tree generated by the recursion scheme is:



Since `checkTrusted` does not occur below `frameUntrusted`, the second program is safe.

As sketched above, one can always transform a term of type **bool** into a recursion scheme that generates a tree representing sequences of framing, enabling, and checking of permissions. SBAC is therefore decidable. (Note that the actual transformation is a little more complicated than sketched above, because of the need to handle non-termination. More details will be described in a longer version.)

Remark 1. Another way for encoding SBAC into a model chekcing problem for a recursion scheme is to transform a program into a recursion scheme (extended with finite data domains [6]) that computes the current permission eagerly and passes it as an extra argument of each function, so that the recursion scheme generates a tree consisting of a single node `fail` if and only if the original program is aborted. This encoding is probably simpler, although the above encoding may be more intuitive.

2.3 Further Directions

In this paper, we discussed two applications of higher-order model checking to language-based security: secure information flow and stack-based access control. We think higher-order model checking is applicable to many other problems in language-based security. For example, we expect that our technique for SBAC is also applicable to history-based access control [13,14].

At this moment, it is not clear how much our verification method scales in practice. Even if automated verification does not scale, however, our method may be useful in the context of proof-carrying code [15]. Our model-checking algorithm for recursion schemes [5] is based on type inference. Therefore, if type information is given as a certificate of the correctness of a program, then the correctness can be checked efficiently by a type checking (instead of inference) algorithm.

We have considered a simply-typed language having only booleans as base values. For infinite data domains such as integers, predicate abstractions can be used to obtain a sound (but incomplete) verification method. For a language with more advanced types (such as recursive types), it is unclear whether a similar approach is applicable (even if we give up the completeness of a verification algorithm).

Acknowledgments

We would like to thank Luke Ong for pointing us to Loader's work on the undecidability of finitary PCF.

References

1. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS 2006, pp. 81–90. IEEE Computer Society Press, Los Alamitos (2006)
2. Knapik, T., Niwinski, D., Urzyczyn, P.: Higher-order pushdown trees are easy. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 205–222. Springer, Heidelberg (2002)
3. Igarashi, A., Kobayashi, N.: Resource usage analysis. ACM Transactions on Programming Languages and Systems 27(2), 264–313 (2005)
4. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages, pp. 416–428 (2009)
5. Kobayashi, N.: Model-checking higher-order functions. In: Proceedings of PPDP 2009 (to appear 2009)
6. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. Preprint (2009)
7. Kobayashi, N., Ong, C.H.L.: Complexity of model checking recursion schemes for fragments of the modal mu-calculus. In: Proceedings of ICALP 2009. LNCS. Springer, Heidelberg (2009)
8. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas in Communications 21(1), 5–19 (2003)
9. Loader, R.: Finitary pcf is not decidable. Theoretical Computer Science 266(1-2), 341–364 (2001)
10. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison Wesley, Reading (1999)
11. Pottier, F., Skalka, C., Smith, S.F.: A systematic approach to static access control. ACM Transactions on Programming Languages and Systems 27(2), 344–382 (2005)
12. Fournet, C., Gordon, A.D.: Stack inspection: Theory and variants. ACM Transactions on Programming Languages and Systems 25(3), 360–399 (2003)
13. Abadi, M., Fournet, C.: Access control based on execution history. In: Proceedings of the Network and Distributed System Security Symposium (NDSS 2003). The Internet Society, San Diego (2003)
14. Wang, J., Takata, Y., Seki, H.: HBAC: A model for history-based access control and its model checking. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 263–278. Springer, Heidelberg (2006)
15. Necula, G.C.: Proof-carrying code. In: Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages, pp. 106–119 (1997)