MoCHi: Software Model Checker for a Higher-Order Functional Language

Ryosuke Sato
Tohoku University
ryosuke@kb.ecei.tohoku.ac.jp

Hiroshi Unno
University of Tsukuba
uhiro@cs.tsukuba.ac.jp

Naoki Kobayashi
University of Tokyo
koba@is.s.u-tokyo.ac.jp

Abstract

We will demonstrate MoCHi, a fully-automated program verification tool (so called a "software model checker") for a subset of OCaml, supporting integers, recursive data types (such as lists), exceptions, higher-order functions and recursion. MoCHi is based on higher-order model checking, and consists of three layers. The top layer transforms a source program into an intermediate program of a core language, which is a simply-typed call-by-value higher-order functional language with recursion, booleans and integers (a la "PCF"). The transformation is carried out by encoding exceptions and recursive data types using higher-order functions. The middle layer further transforms the intermediate program into a higher-order boolean functional program (a la "finitary PCF"), by using predicate abstraction and counterexample-guided abstraction refinement (CEGAR). Finally, on the bottom layer, the higher-order boolean functional program is verified by using a higher-order model checker.

In the presentation, we use MoCHi to verify several sample programs, and explain how it works internally. We will also discuss the current limitations and ongoing/future work.

1. Introduction

MoCHi is a software model checker for a subset of OCaml, constructed based on higher-order model checking [1–4], predicate abstraction, and counter-example-guided abstraction refinement (CEGAR) [5]. This can be viewed as a higher-order counterpart of previous software model checkers for imperative languages like BLAST [6] and SLAM [7]. MoCHi takes an OCaml program as an input, and statically checks lack of assertion failures, pattern match errors, and uncaught exceptions. The supported language features includes higher-order functions and recursion, base types, some recursive data types (such as lists and trees), and exceptions. Certain restrictions are imposed on recursive data types and exceptions, which will be explained later.

In the rest of this paper, we explain the features of MoCHi from a user's point of view in Section 2, and the internal structure of MoCHi in Section 3.

2. Overview of MoCHi

Given a program written in OCaml, MoCHi checks whether the program is safe or not. The input language supports the following features:

- base types (unit, booleans, and integers),
- tuples,
- recursion,
- higher-order functions,

- Input:

```
let rec mc91 x = if x > 100 then x - 10
                 else mc91 (mc91 (x + 11))
let main n = if n <= 101 then assert (mc91 n = 91)
```

- Output:

```
 ...
Abstraction Types::
  mc91 : x:int[x <= 111; x <= 101] ->
         x:int[x <= 101; x = 91]
  ...
Intersection Types::
  ...

Safe!

total: 0.229965 sec
```

Figure 1. Verification Example for Safe Program

- exceptions, and
- algebraic data types.

The following restrictions are imposed on exceptions and algebraic data types. As for exceptions, we do not allow those carrying function arguments. For algebraic data types, we do not support parametrized data types (such as `datatype 'a list = ...`), and more importantly, recursive type variables cannot occur under function constructors. Thus, $\mu\alpha.\,\mathbf{unit} + (\mathbf{int} \to \mathbf{int}) * \alpha$ (which corresponds to $(\mathbf{int} \to \mathbf{int})\ \mathbf{list}$) is OK, but neither $\mu\alpha.\,\alpha \to \mathbf{int}$ nor $\mu\alpha.\,(\mathbf{int} \to \alpha)$ is allowed.

MoCHi checks the following properties:

- The assertions in the program never fail.
- The pattern matches in the program are exhaustive.
- An uncaught exception does not occur.

MoCHi outputs "safe" with a certificate[1] when the program satisfies all the three properties above. Otherwise, MoCHi outputs "unsafe" with a counterexample that consists of an input of the main function and an execution sequence that reaches an error.

Figures 1 and 2 show verification examples of McCarthy 91 function using MoCHi. Programs assert that `mc91` $n = 91$ for all $n \leq 101$ and for all $n \leq 200$, respectively. MoCHi verifies that the first program is safe, i.e., the assertion never fails, and outputs

[1] A certificate consists of abstraction types, that represent how to abstract the program, and intersection types, that is a certificate for the safety of the abstracted program. See papers [2, 5] for details

- Input:

```
let rec mc91 x = if x > 100 then x - 10
                 else mc91 (mc91 (x + 11))
let main n = if n <= 200 then assert (mc91 n = 91)
```

- Output:

```
...
Unsafe!

Inputs:
  n = 102;
Error trace::
  main ... -->
 ...
  assert ... -->
  ERROR!

total: 0.104983 sec
```

Figure 2. Verification Example for Unsafe Program

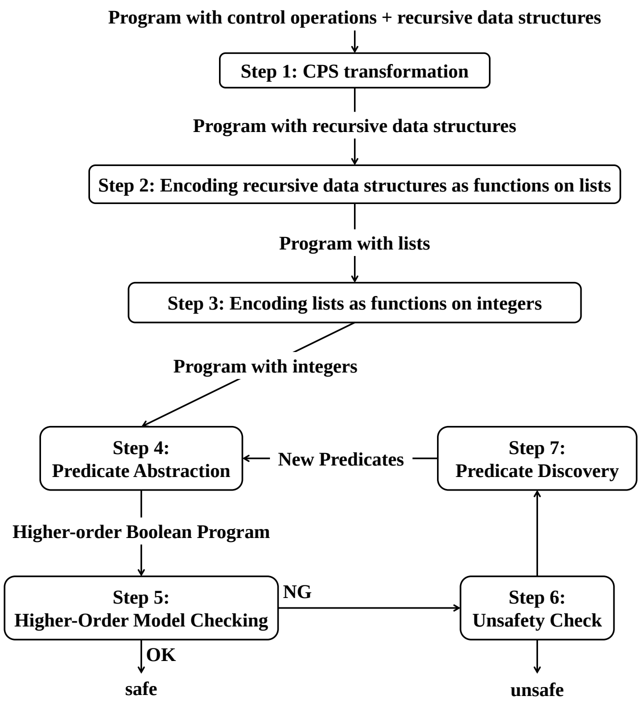
Program with control operations + recursive data structures
Step 1: CPS transformation
Program with recursive data structures
Step 2: Encoding recursive data structures as functions on lists
Program with lists
Step 3: Encoding lists as functions on integers
Program with integers
Step 4: Predicate Abstraction
New Predicates
Step 7: Predicate Discovery
Higher-order Boolean Program
Step 5: Higher-Order Model Checking
NG
Step 6: Unsafety Check
OK
safe
unsafe

Figure 3. Internals of MoCHi

a certificate. MoCHi rejects the second program, and show that the program fails when an input of the main function is 102.

MoCHi can be tested and example programs are available at `http://www.kb.ecei.tohoku.ac.jp/~ryosuke/mochi/`.

3. Internals of MoCHi

Figure 3 shows the internal structure of MoCHi.

First, an input program is translated to an equivalent higher-order program without exception and recursive data structures, by three transformations (Steps 1, 2, and 3). In Step 1, exceptions are eliminated by using CPS-transformation, where exception handlers are expressed as auxiliary continuations. We impose the restriction that exceptions should not carry functions, to ensure that the result of the transformation is well-typed. In Step 2, a recursive data structure is encoded as a function that maps paths of nodes to labels. Here, a path and a label are represented as a list of integers and an integer respectively. Consider binary trees defined as follows.

```
type btree = Leaf | Node of btree * btree
```

A binary tree is encoded as a term of the type $\textbf{int list} \to \textbf{int}$. For example, a binary tree $\mathbf{node}(\mathbf{leaf}, \mathbf{node}(\mathbf{leaf}, \mathbf{leaf}))$ is encoded as a function $\{[] \mapsto \mathbf{node}, [1] \mapsto \mathbf{leaf}, [2] \mapsto \mathbf{node}, [2,1] \mapsto \mathbf{leaf}, [2,2] \mapsto \mathbf{leaf}\}$ where $\mathbf{leaf}$ and $\mathbf{node}$ are defined as some integers. Note that the restriction on recursive types mentioned in Section 2 is necessary to make this encoding work. In Step 3, a list is encoded as a pair of its length and a function that maps indices to the elements of the list. For example, the list $[2; 3; 5]$ is encoded as the pair $(3, f)$ where $f(0) = 2$, $f(1) = 3$, and $f(2) = 5$. See [8] for more details.

The translated program with integers is verified by higher-order model checking with predicate abstraction and CEGAR. In Step 4, the program with integers is abstracted to a higher-order boolean program, that is a higher-order program only with boolean, by using given predicates for abstraction. The abstracted program is verified by a higher-order model checker (that is sound and complete) in Step 5. If the abstracted program is safe, the original functional program is also safe. If not, we check whether the original program is in fact unsafe or the abstraction is too coarse in Step 6. If the latter, we discover new predicates to refine abstraction in Step 7. We repeat these steps until we find whether the program is safe or not. See [5] for more details.

MoCHi uses TRecS [2, 4] as the underlying higher-order model checker (for Step 5 in Figure 3), and uses CSIsat [9] for predicate discovery (for Step 7). CVC3 [10] is used for unsafety check (for Step 6) and predicate abstraction (for Step 4).

4. Conclusion

We have implemented MoCHi, a verifier for a subset of OCaml with some base types (unit, booleans, and integers), higher-order functions, recursions, recursive data structures, and exceptions.

Future work includes:

- Supporting a larger subset of OCaml (references, modules, etc.)
- Supporting a larger class of properties such as resource usage safety [2].
- Making the implementation scalable for larger programs.
- Improving the usability of reports of verification results. For example, we can actually recover dependent types from the current output of MoCHi and show them as a certificate.

References

[1] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS 2006)*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[2] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*, pages 416–428, 2009.

[3] Naoki Kobayashi. Higher-order model checking: From theory to practice. In *Proceedings of the 26st Annual IEEE Symposium on Logic in Computer Science (LICS 2011)*, pages 219–224, 2011.

[4] Naoki Kobayashi. Model-checking higher-order functions. In *The 11th International Symposium on Principles and Practice of Declarative Programming (PPDP 2009)*, pages 25–36, 2009.

[5] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the 2001 ACM SIGPLAN conference on Programminglanguage design and implementation (PLDI 2001)*, pages 222–233, 2011.

[6] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 58–70, 2002.

[7] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[8] Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. Towards a scalable software model checker for higher-order programs. Submitted.

[9] Dirk Beyer, Damien Zufferey, and Rupak Majumdar. CSIsat: Interpolation for LA+EUF. In *CAV 2008*, pages 304–308, 2008.

[10] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.