Verifying Safety of Functional Programs with Rosette/Unbound

Dmitry Mordvinov[1] and Grigory Fedyukovich[2]

[1] Saint-Petersburg State University, Department of Software Engineering, Russia, dmitry.mordvinov@se.math.spbu.ru
[2] University of Washington Paul G. Allen School of Computer Science & Engineering, USA, grigory@cs.washington.edu

Abstract. The goal of unbounded program verification is to discover an inductive invariant that safely over-approximates all possible program behaviors. Functional languages featuring higher order and recursive functions become more popular due to the domain-specific needs of big data analytics, web, and security. We present Rosette/Unbound, the first program verifier for Racket exploiting the automated constrained Horn solver on its backend. One of the key features of Rosette/Unbound is the ability to *synchronize* recursive computations over the same inputs allowing to verify programs that iterate over unbounded data streams multiple times. Rosette/Unbound is successfully evaluated on a set of non-trivial recursive and higher order functional programs.

1 Introduction

Rapid growth in data sciences, web, and security opens the new dimensions for functional programming languages [5]. Due to streaming processing over "big data", applications to program synthesis and to the design of domain specific languages (DSLs) impose extra safety requirements in the unbounded setting. This makes the ability to discover inductive invariants crucial and necessitates the proper support by the verification tools. We present Rosette/Unbound, a new formal verifier for Racket[3], whose most distinguishing feature is the tight connection to the solver of constrained Horn clauses (CHC) offering an automated decision procedure aiming to synthesize safe inductive invariants.

Rosette/Unbound is available in the interactive mode of the DrRacket IDE[4], allowing the users to verify the programs without detracting from coding. It uses a symbolic execution engine of the Rosette [23, 24] tool (thus, sharing a part of the name with it), but it implements a conceptually different encoding strategy and relies on a conceptually different constraint solving paradigm. Furthermore, it features a support of unbounded symbolic data type of lists, which would be impossible in Rosette due to its bounded nature, but is still compatible with the existing bounded reasoning.

[3] https://docs.racket-lang.org/
[4] https://docs.racket-lang.org/drracket/

Yet another key feature of Rosette/Unbound is the ability to verify multiple-pass list-manipulating programs through automatic deriving so called *synchronous* iterators. While treating each element of a list nondeterministically, our tool ensures that it is accessed by all list iterators exactly once and exactly at the same time. We empirically show that this program transformation allows to effectively verify non-trivial safety properties (e.g., the head of a sorted list of integers is always equal to its minimal element). To the best of our knowledge, Rosette/Unbound is currently the only tool featuring such functionality.

The paper is structured as follows. Sect. 2 describes the workflow of Rosette/Unbound and lists the most important features of the tool, which allow it to verify a program in Sect. 3. Sect. 4 reports on the empirical evaluation of the tool, and finally Sect. 5 outlines the closest tools and concludes the paper.

2 Tool Overview

From a distance, Rosette/Unbound implements the workflow exploited by the SMT-based unbounded model checkers [10, 15, 8, 11, 6, 7, 14]. As an intermediate representation of the *verification condition*, it uses a system of constraints in second order logic which encode the functional program in Racket including the assertion specifying a safety property. While the functionality of solving the system of constraints is entirely due to the external solver, Rosette/Unbound contributes in an efficient way of constructing the solvable systems.

Like Rosette, the first steps of Rosette/Unbound are the symbolic execution of the program, symbolic merging of states in the points where branches of the control flow join [24], and transforming the user code by adding some system routines' calls for tracking its execution. This results in a compact symbolic encodings of separated acyclic parts of the program (thus acting close enough to the Large Block Encoding [2]) and ensures that no information about them is lost. Then, Rosette/Unbound uses the individual encodings for generating concise systems of constrained Horn clauses (CHCs). Unlike Rosette, our tool explicitly maintains the call graph and a set of uninterpreted relation symbols R, and for each function f that has not been encoded yet, it creates a fresh symbol f_r and inserts it to R.

For every (possibly merged) execution branch of f, called with $\vec{in}$, the tool creates the following implication, the premise of which is the conjunction of first-order path conditions ϕ and the nested function calls f_i outputting the values $\vec{out}_i$ (i.e., return values and/or state mutations) produced in this branch for $\vec{in}_i$:

$$f_r(\vec{in}, \vec{out}), \leftarrow \phi, f_{r_1}(\vec{in}_1, \vec{out}_1), \ldots, f_{r_n}(\vec{in}_n, \vec{out}_n) \tag{1}$$

The verification condition with assumptions *pre* and requirements *post* for a piece of program calling functions $f_1, \ldots, f_m$, is translated into:

$$false \leftarrow pre, f_{r_1}(\vec{in}_1, \vec{out}_1), \ldots, f_{r_m}(\vec{in}_m, \vec{out}_m), \neg post \tag{2}$$

We refer to an implication of form either (1) or (2) as to constrained Horn clause. A system of CHCs serves as a specification for synthesis of inductive

invariants, and it is called solvable if there is an interpretation of symbols in R making all implications (1) or (2) true. Rosette/Unbound uses off-the-shelf Horn solver to obtain solutions for the system of CHCs it produces. The default Horn solver used by Rosette/Unbound is Spacer3 [15], but it is also possible to switch to Z3 [9].

In the rest of the section, we describe the implementation details and the outstanding features of Rosette/Unbound.

Multiple user-specified assertions. Each Racket function may contain numerous implicit and explicit assertions. The call of `verify/unbound` at some point of symbolic execution creates a system of CHCs by encoding all possible program executions up to that point as (1), and encodes so called *main assertions* (specified with arguments of `verify/unbound`) as (2). Rosette/Unbound gradually propagates assertions from the function bodies and conjoins them with the main assertions: thus, to find a bug, our tool needs to find a violation of at least one of all assertions.

Mutations and global variables. Global state accessed and mutated in the body of function f is added as an extra argument to relation f_r. Rosette/Unbound obtains a symbolic encoding of the mutations performed by arbitrary (possibly, mutually recursive) functions. One important feature is that unused global state is excluded from the encoding, keeping relations compact even for large programs.

Higher order functions. Racket allows manipulating functions as arguments to another functions, the feature common to functional programming languages. To encode a higher order function to CHCs, Rosette/Unbound needs a particular assignment A to its arguments and an assignment to the arguments of all other higher order function appearing among A. This way, Rosette/Unbound performs a so called *higher order inlining* whose goal is to instantiate all functional arguments with their particular meaning. Thus, each new application of a higher order function is treated as a unique new function, necessitating the creation of a fresh uninterpreted relation symbol in R and constructing a set of separate CHCs.

Unbounded symbolic lists. Rosette/Unbound introduces the unbounded symbolic data type of lists and supports built-in operations over lists (including the length, head, tail, iterators, mapping and appending functions whose outputs are the tailored symbolic constants). However, Rosette/Unbound also supports a partially specified lists (e.g., it allows to cons a symbolic list and a concrete head). The data type of symbolic lists is useful while verifying properties about list iterators (e.g, using a higher order function $fold$). That is, while an element is accessed in each iteration over the symbolic list, it is treated nondeterminstically. It has its negative side effect while dealing with multiple traversals over the same list. Indeed, nondeterminism relaxes the fact that both traversals are conducted over the same list.

Nondeterminism modulo synchronization. We compensate this weakness by merging the individual iterators over lists and producing a new CHC system

fold-map.rkt

```
#lang rosette/unbound
; functions over integers:
(define/typed (inc/typed x) (~> integer? integer?) (+ x 1))
(define/typed (+/typed x y) (~> integer? integer? integer?) (+ x y))
; nondeterministically treated list:
(define-symbolic xs (listof integer?))
; assertion requiring multiple list traversals:
(verify/unbound (assert (= (+ (foldl +/typed 0 xs) (length xs))
                           (foldl +/typed 0 (map inc/typed xs)))))
```

$$
\begin{cases}
\textsc{sum}(\ell_1, acc_1, res_1) \leftarrow \ell_1 = 0, res_1 = acc_1 \\
\textsc{sum}_{+1}(\ell_2, acc_2, res_2) \leftarrow \ell_2 = 0, res_2 = acc_2 \\
\textsc{sum}(\ell_1, acc_1, res_1) \leftarrow \textsc{sum}(\ell_1 - 1, acc_1, res_1'), \ell_1 > 0, res_1 = res_1' + hd_1 \\
\textsc{sum}_{+1}(\ell_2, acc_2, res_2) \leftarrow \textsc{sum}_{+1}(\ell_2 - 1, acc_2, res_2'), \ell_2 > 0, res_2 = res_2' + hd_2 + 1 \\
\bot \leftarrow \textsc{sum}(\ell, 0, res_1), \textsc{sum}_{+1}(\ell, 0, res_2), \ell \geqslant 0, res_1 + \ell \neq res_2
\end{cases}
$$

$$
\begin{cases}
\textsc{sums}(\ell_1, acc_1, res_1, \ell_2, acc_2, res_2) \leftarrow \ell_1 = 0, res_1 = acc_1, \ell_2 = 0, res_2 = acc_2 \\
\textsc{sums}(\ell_1, acc_1, res_1, \ell_2, acc_2, res_2) \leftarrow \textsc{sums}(\ell_1 - 1, acc_1, res_1', \ell_2 - 1, acc_2, res_2'), \boldsymbol{hd_1 = hd_2} \\
\qquad\qquad \ell_1 > 0, \ell_2 > 0, res_1 = res_1' + hd_1, res_2 = res_2' + hd_2 + 1 \\
\bot \leftarrow \textsc{sums}(\ell, 0, res_1, \ell, 0, res_2), \ell \geqslant 0, res_1 + \ell \neq res_2
\end{cases}
$$

Fig. 1: Higher order verification with Rosette/Unbound: a given program, the intermediate and ultimate encoding.

to be solved instead of the original one. In the lower level, it requires adding the extra synchronization constraints that equates the elements at the same position of the same list while being accessed by different iterators. The similar reasoning is exploited for proving relational properties over numeric recursive programs (e.g., monotonicity of the factorial). This transformation is proven sound in our previous work [17].

3 Running Example

We illustrate the Rosette/Unbound workflow on an example. Consider a higher order functional program in Fig. 1 (upper). It features two lists of integers: `xs` and `(map inc/typed xs)`. The latter is created from `xs` by adding `1` to each element of `xs`. Then, the program performs two traversals and sums all the elements of each list: `(foldl +/typed 0 ...)`. We want to verify that the difference between the results of those summing iterations is exactly equal to `(length xs)`.

Rosette/Unbound creates a verification condition as a system of CHCs over linear arithmetic. There are two main encoding stages:

1. Creating individual CHCs for all list iterators:
   - `(foldl +/typed 0 xs)` $\mapsto \text{SUM}(\ell_1, acc_1, res_1)$,
   - `(foldl +/typed 0 (map inc/typed xs))` $\mapsto \text{SUM}_{+1}(\ell_2, acc_2, res_2)$.

   Each call to a higher order function is "inlined" into a set of CHCs. Despite there are two calls of the same function `fold`, they are encoded separately as they get different functions as parameters. Note that the other higher order functions (e.g., `map`), if they are used only as parameters to some other higher order functions, are not encoded as separate CHCs. The system obtained as a result after this step is shown in Fig. 1 (central).
2. Synchronizing the CHCs using `xs`:
   - $\text{SUM}(\ell_1, acc_1, res_1) \bowtie \text{SUM}_{+1}(\ell_2, acc_2, res_2) \mapsto \text{SUMS}(\ell_1, acc_1, res_1, \ell_2, acc_2, res_2)$.

   Importantly, in this step, an equality over lists heads is implanted to the constructed system. The ultimate system (with the implanted equality in bold) is shown in Fig. 1 (lower). The key insight behind the system is that all reasoning is reduced to a single computation over a single nondeterministic list: each iteration accessing the head is shared among the original iterators.

Finally, Rosette/Unbound automatically passes the ultimate system of CHCs to the external solver and waits while the solving is delivered.

4 Evaluation

We evaluated our tool on a set of functional programs challenging for verification, i.e., (1) programs performing computations in non-linear arithmetics (e.g., monotonicity of powers, relational properties of `div`, `mod`, and `mult` operations, etc.) expressed as recursive functions using only linear arithmetics, (2) programs with recursive and mutually recursive functions performing mutations of variables, (3) programs with higher order functions, and (4) programs asserting complex properties of a nondeterministic symbolic list, due to its multiple (explicit or implicit) traversals.

Comparison with MoCHi. Since there is no unbounded verifier for Racket, we compared our tool with MoCHi [14], a state-of-art verifier of OCaml programs. Both tools also use constrained Horn solving in their verification backend. We translated the benchmark set of MoCHi to Racket and symmetrically translated our benchmark set to OCaml. The common features of Rosette/Unbound and MoCHi include the support of programs with recursion, linear arithmetics, lists. However, contrary to the current revision of Rosette/Unbound, MoCHi supports algebraic data types and exceptions handling. Thus, making Rosette/Unbound work for the remaining MoCHi's benchmarks is left for our future work.

All benchmarks that we managed to translate for our tool were correctly solved. On the other hand, 6 of our benchmarks contain programs with mutations that not yet supported by MoCHi. For 9 of 12 supported programs with

	Rosette/Unbound's benchmarks	MoCHi's benchmarks
Rosette/Unbound	30/30, 0.087/0.115 sec	40/54, 0.075 sec
MoCHi	10/30, 0.475 sec	54/54, 0.227/0.590 sec

Table 1: The overall statistics. A ratio in the first position of each cell corresponds to the solved and the total numbers of benchmarks from the corresponding set. The first value in the second position stands for the average time of solving the opponent's benchmarks. The second value (if any) in the second position stands for the average time on all benchmarks.

bugs, MoCHi managed to find a counterexample. However, only for 1 of 12 safe programs, MoCHi managed to prove its correctness (which is significantly harder than searching for a finite counterexample). For 11 others, either a time-out was reached or an internal error occurred. Table 1 shows the brief statistics of our evaluation.[5]

5 Closing Remarks

Verification of functional languages is typically based on type inference [18, 12, 25, 22, 4]. Our tool implements an orthogonal, SMT-based approach which is native also for the Dafny [16], Leon [21], and Zeno [20] induction provers. None of those approaches has an completely automated CHC solver on its backend. In contrast, CHC solvers are exploited by the model checkers for imperative languages [10, 15, 8, 11], dataflow languages [6, 7], and functional programming languages [14]. Our work makes another functional programming language, rapidly becoming more popular, communicate with an external CHC solver.

One of the main contributions of our Rosette/Unbound is the support of new unbounded data type of symbolic lists. Furthermore, the ability to implicitly merge recursive functions over the same input data make Rosette/Unbound the first program verifier which is able to successfully deal with programs that iterate over unbounded data streams multiple times. To confirm this claim, Rosette/Unbound was successfully evaluated on a set of non-trivial recursive and higher order functional programs.

Applying Rosette/Unbound to functional program synthesis [1, 13, 19, 3] is the next step of our future work. Indeed, the support of multiple-pass list manipulating programs also enables writing "universally quantified" program specifications. It would be also interesting to see how the discovered inductive invariants could be encoded as the first class entities and be a part of the executable Racket code.

[5] The complete table can be found in Appending A.

References

1. A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *CAV*, volume 8044 of *LNCS*, pages 934–950. Springer, 2013.
2. D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software Model Checking via Large-Block Encoding. In *FMCAD*, pages 25–32. IEEE, 2009.
3. G. Fedyukovich and R. Bodík. Approaching Symbolic Parallelization by Synthesis of Recurrence Decompositions. In *SYNT*, EPTCS, pages 55–66, 2016.
4. P. Fu, E. Komendantskaya, T. Schrijvers, and A. Pond. Proof relevant corecursive resolution. In *FLOPS*, volume 9613 of *LNCS*, pages 126–143. Springer, 2016.
5. M. Gaboardi, S. Jagannathan, R. Jhala, and S. Weirich. Language based verification tools for functional programs (dagstuhl seminar 16131). *Dagstuhl Reports*, 6(3):59–77, 2016.
6. P. Garoche, A. Gurfinkel, and T. Kahsai. Synthesizing modular invariants for synchronous code. In *HCVS*, volume 169 of *EPTCS*, pages 19–30, 2014.
7. P. Garoche, T. Kahsai, and X. Thirioux. Hierarchical State Machines as Modular Horn Clauses. In *HCVS*, volume 219 of *EPTCS*, pages 15–28, 2016.
8. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn Verification Framework. In *CAV*, volume 9206 of *LNCS*, pages 343–361. Springer, 2015.
9. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, volume 7317, pages 157–171. Springer, 2012.
10. H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A verification toolkit for numerical transition systems - tool paper. In *FM*, volume 7436 of *LNCS*, pages 247–251. Springer, 2012.
11. T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. Jayhorn: A framework for verifying java programs. In *CAV, Part I*, volume 9779 of *LNCS*, pages 352–358. Springer, 2016.
12. G. Kaki and S. Jagannathan. A relational framework for higher-order shape analysis. In *ICFP*, pages 311–324. ACM, 2014.
13. E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In *CAV*, volume 9207 of *LNCS*, pages 217–233. Springer, 2015.
14. N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *ACM*, pages 222–233. ACM, 2011.
15. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-Based Model Checking for Recursive Programs. In *CAV*, volume 8559 of *LNCS*, pages 17–34, 2014. https://bitbucket.org/spacer/code/branch/spacer3.
16. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
17. D. Mordvinov and G. Fedyukovich. Synchronizing Constrained Horn Clauses. In *LPAR*, EPiC Series in Computing. EasyChair, 2017.
18. C. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL*, pages 587–598. ACM, 2011.
19. N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, pages 522–538. ACM, 2016.
20. W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *TACAS*, volume 7214 of *LNCS*, pages 407–421. Springer, 2012.
21. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS*, LNCS, pages 298–315. Springer, 2011.

22. N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin. Dependent types and multi-monadic effects in F*. In *POPL*, pages 256–270. ACM, 2016.
23. E. Torlak and R. Bodík. Growing solver-aided languages with Rosette. In *Onward!*, pages 135–152. ACM, 2013.
24. E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, pages 530–541. ACM, 2014.
25. N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. P. Jones. Refinement types for haskell. In *ICFP*, pages 269–282. ACM, 2014.

A Raw Experimental Data

benchmark name	MoCHi	Rosette/ Unbound
ack	0.096	0.063
a-cppr	2.298	NI
a-init	3.441	NI
a-max-e	0.684	0.063
a-max	0.687	0.062
copy_intro	0.159	0.177
e-fact	0.096	0.058
e-simple	0.060	0.017
fact-not-pos-e	0.067	NI
fact-not-pos	0.110	NI
fold_div-e	0.084	0.118
fold_div	0.152	0.136
fold_fun_list	0.114	NI
fold_left	0.224	0.085
fold_right	0.160	0.085
forall_eq_pair	0.417	NI
forall_leq	0.398	0.091
harmonic-e	0.078	0.146
harmonic	0.144	0.141
hors	0.072	0.092
hrec	0.102	NI
intro1	0.059	0.031
intro2	0.060	0.031
intro3	0.058	0.021
isnil	0.121	0.054
iter	0.122	0.091
length	0.141	0.059
l-zipmap	0.104	0.118
l-zipunzip	0.119	NI
a-max	0.068	0.079
mc91-e	0.078	0.049
mc91	0.441	0.056
map_filter	10.951	NI
map_filter-e	0.934	NI
mem	0.191	0.090
mult-e	0.096	0.048
mult	0.147	0.055
neg	0.066	0.022
nth0	0.175	0.095
nth	0.534	0.115
repeat-e	0.065	0.050
reverse	0.281	0.102
risers	1.435	NI
r-file	1.476	0.135
r-lock-e	0.070	0.082
r-lock	0.062	0.089
search-e	0.385	0.028
search	0.659	0.032
sum-e	0.064	0.049
sum_intro	0.386	0.050
sum	0.083	0.049
tree_depth	0.164	NI
zip	1.601	NI
zip_unzip	1.023	NI

(a) on MoCHi benchmarks [14]

benchmark name	Rosette/ Unbound	MoCHi
div-jumps-e	0.100	1.443
div-jumps	0.103	EOT
fold-append-e	0.108	EOT
fold-append	0.120	EOT
fold-eq-e	0.113	EOT
fold-eq-minus-e	0.111	0.134
fold-eq-minus	0.117	EOT
fold-eq	0.106	EOT
fold-map-abs-e	0.130	0.749
fold-map-abs	0.121	EOT
fold-mutations-e	0.171	NI
fold-mutations	0.196	NI
heads-sum-e	0.072	EOT
heads-sum	0.076	EOT
length-append-e	0.036	0.401
length-append	0.019	EOT
length-append-simpl-e	0.035	0.978
lucas-vs-fib-e	0.134	0.073
lucas-vs-fib	0.160	EOT
map-fold	0.125	EOT
mod-div-mult-e	0.101	<1?[6]
mod-div-mult	0.160	EOT
mutual-recursion-e	0.064	NI
mutual-recursion	0.084	NI
power-monotone-e	0.122	0.125
power-monotone	0.282	EOT
single-fold-e	0.056	0.134
single-fold	0.063	0.239
sorted-e	0.163	NI
sorted	0.187	NI

(b) on Rosette/Unbound benchmarks

Fig. 2: Verification statistics

Tables 2b and 2a gather statistics on all benchmarks Rosette/Unbound was successfully evaluated. The benchmarks with the names ending with `-e` are buggy, so an error trace is expected from the verifier. All other benchmarks are

[6] The provided error trace was correct, but, due to an internal error, no execution time was reported.

safe. Two columns of numbers represent execution times (in seconds) in case when the verification process ended successfully by the two competing tools. `NI` means that some feature required for verification of the benchmark is not implemented in corresponding tool (see Sect. 4). `EOT` denotes that tool terminated with an error or timeout was reached.

Note that the timing were collected on the different machines. We ran Rosette/Unbound on Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz, 2 cores, 8 GB RAM. No desktop version of MoCHi is publicly available, and thus we we able to ran it only at the web interface (http://www.kb.is.s.u-tokyo.ac.jp/~ryosuke/mochi/), and unfortunately its characteristics are not known to us.

B Running Rosette/Unbound via DrRacket