Verification for Program Manipulations using Iris

Sosuke Moriguchi*

Department of Computer Science, Institute of Science Tokyo, Tokyo, Japan,
chiguri@acm.org

Abstract. In this paper, we implement a framework for verifying programs targeting source code using the theorem prover Rocq and the program logic library Iris. As programming languages become more complex, it is becoming difficult to accurately describe their semantics. As a result, it is becoming increasingly difficult to predict the semantic issues that will arise when language extensions are added. In systems that perform static source code changes, such as macro and template metaprogramming, users can change syntax without affecting semantics. The proposed framework performs data-source mapping in the verification system, similar to metaprogramming, to extract another program from the data during program execution and verify its semantics. This paper verifies that the proposed method enables program verification of this kind.

Keywords: program verification, metaprogramming, program logic, Iris

1 INTRODUCTION

As programming languages become more complex, it is becoming increasingly difficult to accurately describe their semantics. Program verification ensures that a program behaves as described; however, the description of its meaning is not necessarily what the programmer intended. Similarly, the semantics underlying verification may not fully reflect the semantics described in natural language or in the implementation. In fact, CompCert, a formally verified C compiler, was released in 2009 [8]. However, bugs have been found in CompCert since its release. For example, [3] discovered an example where the semantics described in CompCert did not align with the C specification by making the semantics executable and testing them[1]. As such, constructing a consistent formal semantics is not straightforward, and as the semantics become more complex, the program verification system itself also grows in size. Achieving such consistency is not easy, and if modifications are made to the semantics or the core components of the system, verifying these changes requires significant effort.

Semantics-based program verification is not limited to compilers such as CompCert. Other examples include static program verification algorithms and

[1] In that example, CompCert returned an error at an intermediate compilation stage, and the verification was still correct.

© The Author(s) 2026
J. Caro et al. (eds.), *Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2025)*, Atlantis Highlights in Computer Sciences 24,
https://doi.org/10.2991/978-94-6239-638-8_2

program generation frameworks. Furthermore, static metaprogramming, such as macros, is a function that converts programs into programs, so it is reasonable to use semantics to express the programmers' intents.

In this study, we implement program verification using programs as input and output, minimizing the impact on the program verification system. As the program verification system, we use Iris [7], a library of higher-order program logic on Rocq, and as the programming language, we use HeapLang, which is implemented on Iris. HeapLang is a very simple language that uses recursion, higher-order functions, and references[2]. In order to perform meta-programming verification, the following two are required.

- A data structure representing a syntax tree can be constructed on the language, and
- A mechanism to convert the data structure representing a syntax tree into syntax that is handled by the verification system.

The former usually uses recursive data structures, but because HeapLang in this paper only has numeric values, Boolean values, tagged unions, and pairs, we use these to express syntax trees. This makes it more difficult to express syntax trees than in general-purpose languages; thus, the results of this study can be applied to other languages implemented on Iris, such as Rust [6] and WebAssembly [9].

The contribution of this study is the proposal and demonstration of a verification method for general-purpose static metaprogramming. Using HeapLang programs as an example, we demonstrate that verification is possible and show the practical feasibility of the proposed method. The source codes for this paper are available at https://github.com/chiguri/heaplang-meta.

The structure of this paper is as follows. Section 2 introduces HeapLang defined on Iris. Section 3 describes the data structure representing HeapLang programs and the implementation of metaprogramming that generates HeapLang programs. Section 4 describes the verification of metaprogramming that generates HeapLang programs. Section 5 describes how to implement macro-like functionality using this method. Section 6 discusses related research, and Section 7 summarizes this research and future prospects.

2 HEAPLANG

HeapLang is a functional programming language defined on Iris that supports references (heaps). This section provides an overview of HeapLang as used in this paper. For more details about the language, please refer to the Iris manual [5].

HeapLang has syntax for integer arithmetic, booleans, conditional branching, recursion, tagged unions and tuples, references, and assignments. Many language mechanisms are implemented as syntax sugar, e.g., there are no let expressions, algebraic data types, or pattern matching. Let expressions are implemented using

[2] Note that HeapLang also supports parallel processing, but it is not used in this paper.

recursive functions, and option types are implemented as syntax sugar for tagged unions.

For example, a program that calculates powers can be written in HeapLang as follows.

```
Definition example1 : expr :=
 let: "power" :=
   rec: "f" "x" "y" := if: "y" = #0 then #1
                                    else "x" * "f" "x" ("y" - #1) in
 "power" #2 #3
```

The first line is Rocq's command, which names the HeapLang program (`expr`) as `example1`. The second line binds the recursive function to the variable named `"power"` in HeapLang, and the third line calls it with 2 and 3 as arguments. Note that `#` is used to convert integers to values in HeapLang. The result of this program can be described as follows.

```
Lemma example1_spec : ⊢ WP example1 {{ v, ⌜ v = #8%V ⌝ }}.
```

This can be read as "If the result `v` of the program `example1` is `8` ($= 2^3$), then the weakest precondition can be derived without any assumptions." This can be easily verified using a tactic that proceeds with the execution of the program. Note that ⊢, ⌜, and ⌝ are notations for converting Iris's predicates and Rocq's predicates.

It is also possible to perform a more general verification of the program. For example, define only the function that calculates the power as follows.

```
Definition example2 : expr :=
 rec: "f" "x" "y" := if: "y" = #0 then #1
                                  else "x" * "f" "x" ("y" - #1).
```

This program itself is evaluated to a value (recursive function closure) immediately, so no further verification is possible. However, it is possible to verify this program by providing the arguments as follows.

```
Lemma example2_spec (m n : Z) :
  (0 ≤ n)%Z -> ⊢ WP (example2 #m%V #n%V) {{ v, ⌜ #(m ^ n)%Z = v ⌝ }}.
```

This proves that calling this function with any integer will calculate the power of that integer[3].

3 METAPROGRAMMING

As described in the previous section, HeapLang does not have language mechanisms that support metaprogramming, such as macro and syntax sugar definitions. To achieve these, all descriptions are made in Rocq. In addition, only primitive data types are provided, and although they have minimal functionality, detailed representation is not possible. Therefore, it is difficult to introduce them as language extensions.

[3] Strictly speaking, this weakest precondition is partial correctness, so execution termination is not guaranteed.

In the metaprogramming used in this paper, a program is represented as an existing HeapLang value. Here, integers are used as tags to represent syntax, and the nested tuples are used as the shape of the tree. In this paper, the tags correspond as follows.

0 represents values,
1 represents variables,
2 represents let expressions,
3 represents binary operations, and
4-10 represent if expressions, constructions of pairs, etc. (Omitted as not used in the paper)

For example, $(0, \text{true})$ represents a program consisting of the constant true. Similarly, $(3, (2, ((0, 4), (0, 5))))$ represents a binary expression (3) consisting of the values 4 and 5 connected by multiplication (the place of 2 is referred to as a binary operator, and 2 as a binary operator represents the multiplication).

Note that in Iris, when pairs are connected in a list (as in S-expressions), parentheses can be omitted and the pairs can be represented as tuples. For example, $(0, (1, 2))$ can be written as $(0, 1, 2)$, and the above example can be written as $(3, 2, (0, 4), 0, 5)$. However, for readability, nests are sometimes explicitly described to show the structure, like $(3, 2, (0, 4), (0, 5))$.

The correspondence between values and programs is defined as follows in Rocq. The full code for `reify` is included in Appendix A.

```
Inductive reify : val -> expr -> Prop :=
| rVal : forall v : val, reify (#0, v) (Val v)
| rVar : forall z : Z, reify (#1, #z) (Var (mk_varname z))
| rLet : forall (z : Z) v1 e1 v2 e2,
   reify v1 e1 -> reify v2 e2 ->
    reify (#2, #z, v1, v2) (let: (mk_varname z) := e1 in e2)
| rBinOp : forall (z : Z) b v1 e1 v2 e2,
   reify_binop #z b -> reify v1 e1 -> reify v2 e2 ->
    reify (#3, #z, v1, v2) (BinOp b e1 e2)
... (* other syntax *).
```

This correspondence has the following characteristics.

- The data is not total. Since the source data is all in pairs, integers or Boolean values will be recognized as data without a corresponding program. Similarly, if the data contains numbers outside the range (such as $(-1, 1)$, where the position of -1 corresponds only to values from 0 to 10), it will also be recognized as data without a corresponding program.
- Syntax sugar exists in the correspondence relationship. Additionally, there are syntactic elements that do not have corresponding representations. For example, as mentioned earlier, the let expression is not included in the original syntax, but in this correspondence relationship, the let expression is introduced into the corresponding program.

Note that variables are represented by numbers. This is because HeapLang does not hold strings as values, while variable names are represented as strings.

For a more rigorous mapping, it would be possible to construct a list of integers using tuples or references and interpret them as byte sequences in ASCII or UTF-8. In this paper, we will not consider anything other than the identity of variable names.

Consider the following program.

```
1 Definition gen1 : expr :=
2  let: "x" := genMult (genVar 1) (genVar 2) in
3  genLet 1 (genVal #2) (genLet 2 (genVal #3) "x"%E).
```

This program uses functions on Rocq for readability. These functions take numerical values representing variable names and actual values, and return data structures representing corresponding expressions. For instance, `gen1` is evaluated to the following value (representing a HeapLang expression) on Rocq.

```
let: "x" := (#3, #2, (#1, #1), (#1, #2))
 in (#2, #1, (#0, #2), (#2, #2, (#0, #3), "x"))
```

Note that `let:` is not from Rocq, but rather syntax from HeapLang. Therefore, `gen1` needs to be evaluated on HeapLang, and thus cannot be directly concretized using `reify`. The result of executing this HeapLang expression is the following value.

```
(#2, #1, (#0, #2), (#2, #2, (#0, #3), (#3, #2, (#1, #1), (#1, #2))))
```

Now, when this value is concretized using `reify`, we obtain an expression such as `let: "v1" := #2 in let: "v2" := #3 in "v1" * "v2"`. The result of executing this expression (on HeapLang) is 6.

As a more complex example, the following shows a program that generates a program that calculates powers.

```
1 Definition gen2 : expr :=
2  let: "unroll" := (rec: "f" "x" "acc" :=
3    if: (#1 ≤ "x") then "f" ("x" - #1) (genMult (genVar 1) "acc"%E)
4                   else "acc") in
5  genLet 1 (genVal #5) ("unroll" #3 (genVal #1%V))%E.
```

In recursion, the multiplication of `"v1"` is accumulated in `"acc"` `"x"`-times. As a result, the power is calculated by binding `"v1"` outside the generated program. In this program, a program that calculates 5 to the power of 3 (5 and 3 passed on line 5) is generated.

4 VERIFICATION

As introduced in the previous section, a program that generates programs is itself an expression in HeapLang. On the other hand, programs that have been converted into data are defined as values. Therefore, in order to refer to the generated program, it is necessary to convert it into a value as the semantics of the generating program. The specification of `gen1` is as follows.

```
1 Lemma gen1_spec (e : expr) :
2   ⊢ WP gen1 {{ v, ⌜ reify v e -> (⊢ WP e {{ x, ⌜ x = #6%V ⌝ }}) ⌝ }}.
```

This proof is not difficult.

1. By applying semantics to the expression `gen1`, we obtain the result as `v`.
2. The generated program is obtained by `reify`.
3. By applying semantics to the obtained program, its result (6) can be obtained as `x`.

1 and 3 can be easily demonstrated using Iris's built-in tactics, and 2 can be easily demonstrated using `inversion` tactic against reify.

Similarly, the following specification can be demonstrated for `gen2`.

```
Lemma gen2_spec (e : expr) :
  ⊢ WP gen2 {{ v, ⌜ reify v e
                   -> (⊢ WP e {{ x, ⌜ x = #((5 ^ 3)%Z)%V ⌝ }}) ⌝ }}.
```

Now, the above programs are closed on both the generation side and the generated side. Here, consider the following program, which extracts the generation part of `gen2`.

```
Definition gen3 : val :=
 rec: "unroll" "x" "acc" :=
  if: ("x" = #0) then "acc"
                 else "unroll" ("x" - #1) (genMult (genVar 1) "acc"%E).
```

To this, by giving parameters as in `example2`, the following specification can be proven.

```
Lemma gen3_spec :
 forall (n m : Z) res,
  (0 <= n)%Z -> ⊢ WP gen3 #n%V (genVal #1%V) {{ v, ⌜ reify v res ⌝  -∗
              WP (subst "v1" #m%V res) {{ v, ⌜ v = #(m ^ n)%Z ⌝ }} }}.
```

However, as with other complex theorems on Rocq, it is somewhat difficult to prove this theorem directly. Instead, it can be proven by demonstrating the following lemma.

```
Inductive gen3_expr (acc : expr) : Z -> expr -> Prop :=
| gen3_0 : gen3_expr acc 0%Z acc
| gen3_succ :
   forall z e, gen3_expr acc z e ->
    gen3_expr acc (Z.succ z) ("v1" * e)%E.

Lemma gen3_expr_step :
 forall (acc : expr) (z : Z) (res : expr),
  gen3_expr ("v1" * acc) (z - 1)%Z res -> gen3_expr acc z res.

Lemma gen3_spec' (n : Z) (acc : val) (e acce: expr) :
 (0 <= n)%Z ->
  ⊢ WP gen3 #n%V acc
       {{ v, ⌜ reify v e -> reify acc acce -> gen3_expr acce n e ⌝ }}.

Lemma gen3_expr_spec :
 forall (n m : Z) res,
  gen3_expr #1%V n res ->
   ⊢ WP (subst "v1" #m%V res) {{ v, ⌜ v = #(m ^ n)%Z ⌝ }}.
```

First, we define `gen3_expr` as a predicate representing the generated program. This predicate enables us to describe the properties of the generating program (`gen3_spec`). Additionally, we use this predicate to separately describe the properties of the generated program and prove those properties (`gen3_expr_spec`). In other words, we prove the program on the generating side and the generated program in two separate stages.

As shown above, although each program is not particularly difficult, the overall structure of the proof using this method becomes complicated because it involves multiple stages. However, there are no mixed expressions such as those that occur in multi-staged programming [1], where it is fundamentally impossible to use direct meta values as values, and therefore it is possible to focus on verifying the properties of the program.

5 ADVANCED FEATURES: MACRO

As mentioned earlier, HeapLang does not have a macro function, so here we describe the implementation of macro-like functionality. Here, a macro is a functionality that transforms a program into another program by rewriting its syntax. Also, macro calls are written in the same way as function calls.

Macro processing is as follows.

- Macros are implemented as functions in HeapLang. The arguments are data structures that represent the syntax at the time of invocation.
- Macro processing is performed on the entire expression. The definition list of the macros is used as the environment, and whether it is a macro is determined from the list.
- If the calling function is a macro, the argument part is converted to data using `reify`, and the function that is the actual macro is called with that data as the argument. The result is then converted back to an expression using `reify`.

The core of macro processing is `reify`. Note that, unlike the `reify` in Section 3, we define `reify` as covering all expressions in HeapLang.

The specifications of macros implemented in this way can be described using the same method shown in Section 4. However, the code handled by macros is generally not closed, and may be affected by outer variable bindings, e.g., variables given as arguments. Therefore, the specifications for macros should require specific (or sometimes abstract) information about the contexts of their invocations. Information about contexts is also required for code generation and code analysis, but our method does not target such information. Therefore, establishing the method and constructing a library through larger verification examples is a future task.

Furthermore, it is not easy to discuss the functionality achieved by multiple macros in our method. Our method allows describing the specifications of the program received or generated by a single macro, but the interaction between multiple macros is controlled by the verification system that performs macro

processing. Therefore, we would verify such a specification as the overall property of macro processing. Verification can be achieved by separating the specifications of data and those of programs, as is done in the case of `gen3_spec`, such that specifications of macros treat arguments as data and the property of the overall processing treats them as programs.

6 RELATED WORKS

There is research for JavaScript covering `eval` using strings in a verification system [4]. This research mainly discusses techniques for making parsing of code unique, and `eval` itself is semantically embedded. Unlike our method, this research deals with dynamic code generation, and it is difficult to receive code as a parameter and analyze it.

A method similar to the one used in this study has been implemented on miniKanren to derive Quine programs in Scheme [2]. In our method, when expressing Quine programs in the same way, its specification would be as follows.

```
1 Definition quine_spec (quine : expr) : Prop :=
2   ⊢ WP quine {{ v, ⌜ reify v quine ⌝ }}.
```

miniKanren performs checking while constructing Scheme programs, so there has been discussion about efficiency for constructions. However, the method used in our study is only a framework, and it is not possible to construct programs. It is possible to implement such a method as a tactic on Rocq, but since the exploration does not perform weighting as in miniKanren, it is not practical. On the other hand, our method can handle more general properties, such as universal properties for functions.

7 CONCLUDING REMARKS

In this paper, we proposed a verification method for programs that generate programs, targeting HeapLang language implemented on Iris. Although the examples are simple, we showed that verification is possible in practice. We also explained how to implement macro-like functionality using the method.

We used axiomatic semantics with the proposed method in this paper, but the method can be combined with other kinds of semantics, such as operational semantics and denotational semantics. Also, we used the same language for the base programs and the generated programs, but we can use a different language for those programs.

Future challenges include applying this method to more realistic languages and verifying tools such as compiler generators. In addition, we will develop more practical libraries through these efforts.

Acknowledgements

This work was supported in part by JSPS KAKENHI Grant Number JP22K11967.

References

1. Berger, M., Tratt, L.: Program logics for homogeneous meta-programming. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 64–81. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
2. Byrd, W.E., Holk, E., Friedman, D.P.: Minikanren, live and untagged: Quine generation via relational interpreters (programming pearl). In: Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming. pp. 8–29. Scheme '12, Association for Computing Machinery, New York, NY, USA (2012), https://doi.org/10.1145/2661103.2661105
3. Campbell, B.: An executable semantics for CompCert C. In: Hawblitzel, C., Miller, D. (eds.) Certified Programs and Proofs. pp. 60–75. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
4. Charlton, N.: Reasoning about string-based runtime code generation. Unpublished (Oct 2011), http://www.billiejoecharlton.com/wp-content/uploads/2012/02/rasbrcg.pdf
5. The Iris 4.3 reference. https://plv.mpi-sws.org/iris/appendix-4.3.pdf [Accessed: 2025-05-31] (2024)
6. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: securing the foundations of the Rust programming language. Proc. ACM Program. Lang. 2(POPL) (Dec 2017), https://doi.org/10.1145/3158154
7. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming 28, e20 (2018), https://doi.org/10.1017/S0956796818000151
8. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (Jul 2009), https://doi.org/10.1145/1538788.1538814
9. Rao, X., Georges, A.L., Legoupil, M., Watt, C., Pichon-Pharabod, J., Gardner, P., Birkedal, L.: Iris-Wasm: Robust and modular verification of WebAssembly programs. Proc. ACM Program. Lang. 7(PLDI) (Jun 2023), https://doi.org/10.1145/3591265

A FULL CODES OF `reify`

The following code defines data representing operators for binary expression operations.

```
Inductive reify_binop : val -> bin_op -> Prop :=
| rbPlusOp : reify_binop #0 PlusOp
| rbMinusOp : reify_binop #1 MinusOp
| rbMultOp : reify_binop #2 MultOp
| rbQuotOp : reify_binop #3 QuotOp
| rbRemOp : reify_binop #4 RemOp
| rbAndOp : reify_binop #5 AndOp
| rbOrOp : reify_binop #6 OrOp
| rbLeOp : reify_binop #7 LeOp
| rbLtOp : reify_binop #8 LtOp
| rbEqOp : reify_binop #9 EqOp.
```

```
Inductive reify : val -> expr -> Prop :=
| rVal : forall v : val, reify (#0, v) (Val v)
| rVar : forall z : Z, reify (#1, #z) (Var (mk_varname z))
| rLet : forall (z : Z) v1 e1 v2 e2,
   reify v1 e1 -> reify v2 e2 ->
    reify (#2, #z, v1, v2) (let: (mk_varname z) := e1 in e2)
| rBinOp : forall (z : Z) b v1 e1 v2 e2,
   reify_binop #z b -> reify v1 e1 -> reify v2 e2 ->
    reify (#3, #z, v1, v2) (BinOp b e1 e2)
| rIf : forall v e vt et ve ee,
   reify v e -> reify vt et -> reify ve ee ->
    reify (#4, v, vt, ve) (If e et ee)
| rPair : forall v1 e1 v2 e2,
   reify v1 e1 -> reify v2 e2 ->
    reify (#5, v1, v2) (Pair e1 e2)
| rFst : forall v e, reify v e -> reify (#6, v) (Fst e)
| rSnd : forall v e, reify v e -> reify (#7, v) (Snd e)
| rInjL : forall v e, reify v e -> reify (#8, v) (InjL e)
| rInjR : forall v e, reify v e -> reify (#9, v) (InjR e)
| rMatch : forall v e (x : Z) vl el vr er,
   reify v e -> reify vl el -> reify vr er ->
    reify (#10, v, #x, vl, vr)
          (Match e (mk_varname x) el (mk_varname x) er).
```

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (http://creativecommons.org/licenses/by-nc/4.0/), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.