From Program Verification to Certified Binaries*

The Quest for the Holy Grail of Software Engineering

Angelos Manousaridis, Michalis A. Papakyriakou, and Nikolaos S. Papaspyrou

National Technical University of Athens
School of Electrical and Computer Engineering
Software Engineering Laboratory
Polytechnioupoli, 15780 Zografou, Athens, Greece
{amanous, mpapakyr, nickie}@softlab.ntua.gr

Abstract. The long tradition of formal program verification and the more recent frameworks for proof-carrying code share a common goal: the construction of certified software. In this paper, mainly through a simple motivating example, we describe our vision of a complete hybrid system that combines the two approaches. We discuss the feasibility of such an ambitious project and report on progress made so far.

Key words: Formal methods, type systems and type theory, certified code, proof-preserving compilation.

1 Introduction

Program verification aims at formally proving the correctness of a computer program, with respect to a certain formal specification or property. As a research field of computer science, program verification is well into the fourth decade of its existence. However, it can hardly be argued that it is often adopted in practice by software engineers, except for verifying mission-critical systems. For the vast majority of software systems, quality assurance amounts to dynamic testing, which unfortunately can produce no guarantees. In this respect, *software engineering*, defined as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" [1], is still very far from reaching the maturity of other branches of engineering.

Several formal logics, their majority greatly influenced by Hoare Logic [2], have been proposed in combination with programming languages as the vehicles for program verification [3]. Most of the proposed approaches advocate a clear separation between the language in which specifications are given (e.g. first-order predicate logic), the programming language, and the methodology and the tools—if any—that support the construction of proofs.

* This work has been funded by the research programme ΠΕΝΕΔ (grant number 03ΕΔ 330), cofinanced by public expenditure (75% by the European Social Fund and 25% by the Greek Ministry of Education, General Secretariat of Research and Technology) and by the private sector, under measure 8.3 of the Operational Programme "Competitiveness" in the European Union's 3rd Community Support Framework.

Proof-carrying code [4, 5] and *foundational proof-carrying code* [6] are general frameworks, expressing a relatively modern philosophy towards the verification of low-level (e.g. machine language) programs. A *certified binary* is a value (a function, a data structure, or a combination of both) together with a proof that the value satisfies a given specification. Certified binaries are essential in modern distributed computer systems, where executable code is transferred among computing devices that do not necessarily trust one another. The recipient of a certified binary does not need to trust the producer: the proof can be mechanically checked and, once found valid, it is known beyond doubt that the binary conforms to its specification. Existing compilers that produce certified binaries have mostly focused on simple memory and control-flow safety properties. Although the two frameworks are general enough to express arbitrary program properties, in the general case not much is known on how to construct certified binaries or how to automatically generate them from high-level source programs.

More recently, type-theoretic frameworks for constructing, composing and reasoning about certified software have been proposed [7, 8], based on the "formulae as types" principle [9]. The type-theoretic approach provides an embedding of logic in the type system of the programming language: program properties are encoded in types and proof checking is reduced to type checking. In analogy to a type-preserving compiler, which uses a typed intermediate [10] and a typed assembly language [11] and propagates type information from the source program down to the lower-level equivalent programs, a type-theoretic framework for certified binaries can support *proof-preserving compilation*. Provided that a common logic (type language) is used for expressing properties and proofs, from the source language to the target language, certified binaries can be generated by compiling previously verified source programs. This is important, because high-level programs are easier to reason about than low-level programs.

Still, in a type-theoretic framework such as that proposed by Shao *et al.* [7], constructing a proof of correctness even for a small program, written in an appropriate high-level source language, is far from simple. As the logic is part of the programming language (more accurately, part of the type language), the proof must be *embedded* in the code and has to be constructed at the same time with it. Although this has long been proposed as the "right way" to produce software [12, 13], it is not popular with programmers, who generally prefer to write down their algorithm first and then (if ever) prove its correctness. Furthermore, if something in the specification changes, the code has to change as well and, sometimes, the modifications can be substantial in size even if the code's operational behaviour does not change.

Due to the complexity of the type languages used in the type theoretic frameworks that support proof-preserving compilation, type inference (or proof inference) is in general undecidable. The type system of the source language cannot do miracles. It can therefore be argued that, although the embedding of logic in the programming language is appropriate for the lower-level languages used by the compiler, it is not appropriate for the source language, in which the task of constructing the proof is—more or less—the programmers' responsibility.

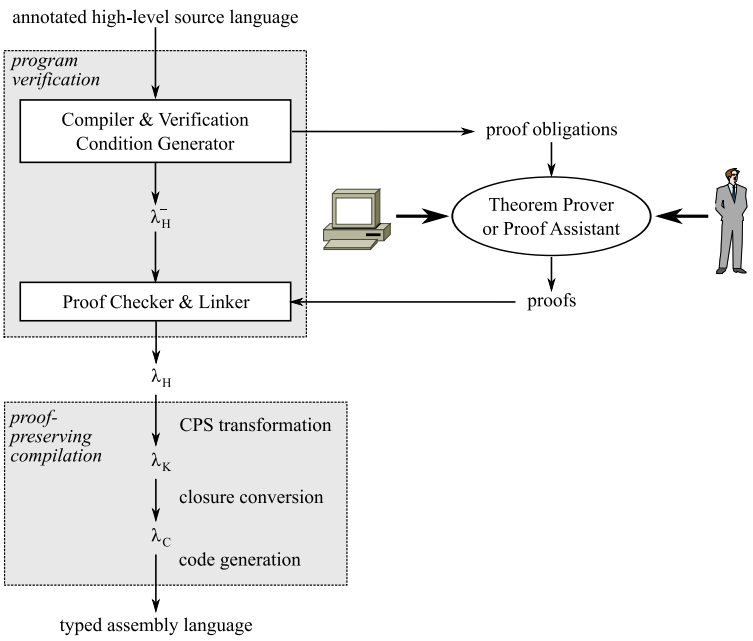

Fig. 1. Overview of a hybrid system for generating certified binaries.

In this paper, we register our dream of a hybrid system (half based on traditional program verification and half type-theoretic). Although similar dreams must be common among computer scientists who advocate program verification and proof-carrying code, it seems that they are still rather far from becoming reality. We outline our experience—limited, so far—in building such a system. If it turns out that, with the assistance of appropriate program verification tools, programmers are able to prove the correctness of their programs (we want to be optimistic and believe this hypothesis to be true), such a system can be thought of as the Holy Grail of software engineering.

2 A Hybrid System for Generating Certified Binaries

A hybrid system for generating certified binaries from annotated source programs can be structured in two layers, as depicted in Fig. 1. First, a "programmer-friendly" program verification layer assists programmers in constructing valid proofs for their source programs according to the specifications that they have set. In this layer, specifications and proofs are separate from the actual code and an ordinary, general-purpose programming language can be used.

The program verification layer can follow the methodology suggested in the work of Filliâtre *et al.* related to the *Why* software verification platform [14, 15]. The source code, written in any from a variety of languages, must be annotated

with specifications (preconditions, postconditions, invariants, etc.) in some appropriate logic. It is then given to a tool serving two purposes: (i) to compile the source code into a lower-level intermediate language λ_H^-; and (ii) to generate proof obligations that must be proved, in order to verify the correctness of the source code w.r.t. its specifications.

The language λ_H^- can be thought of as a typed intermediate language, such as λ_H in the paper by Shao *et al.* [7], with *some proofs missing*. The missing proofs are exactly those corresponding to the generated proof obligations. A variety of tools (from automatic theorem provers to human-driven proof assistants) can be used to discharge the proof obligations and generate the missing proofs. Subsequently, the intermediate program in λ_H^- can be automatically "linked" with the constructed proofs, resulting in a λ_H program. An additional type/proof checking step can be performed, to ensure the correctness of the "linked" program.

The second layer of the hybrid system consists of a type/proof preserving compiler, which transforms the program in λ_H and produces a certified binary. This compiler performs type/proof preserving program transformations that produce progressively lower-level code. Shao *et al.* have shown how to perform type preserving CPS transformation and closure conversion on λ_H (and call the intermediate languages λ_K and λ_C respectively). One or more type-preserving "code generation" steps are required to obtain a certified binary in the form of a (machine dependent or independent) typed assembly language.

It should be noted that the *trusted computing base*, i.e. the piece of software that the recipient of a certified binary must blindly trust (not shown in Fig. 1), consists only of a type/proof checker for the typed assembly language and a (type/proof erasing) translator to native assembly language. Both pieces of software are of moderate size and relatively easy to build.

3 A Motivating Example

In this section we present a small case study: the construction of a certified binary from a C function annotated with its specification. The example is intentionally chosen to be very simple, so that self-contained equivalent programs in λ_H and λ_K can fit in this paper.

Consider a function `root` that calculates the *integer square root* of an integer number n, i.e. the greatest integer r with the property $r^2 \leq n$. A naïve C program that implements this function is the following:

```
int root (int n) {
  int y = 0;
  while ((y+1)*(y+1) <= n) y++;
  return y;
}
```

Following the notation used by the *Why* verification platform and the verification tool *Caduceus* for C programs [14, 15], the same program annotated with the function's pre- and postcondition and the loop invariant is given in Fig. 2.

```
//@ predicate  leRoot(int r, int x) { r >= 0 && r*r <= x }
//@ predicate  isRoot(int r, int x) { leRoot(r, x) && (r+1)*(r+1) > x }

/*@ requires  n >= 0
  @ ensures   isRoot(\result, n)
  @*/
int root (int n) {
  int y = 0;
  //@ invariant  leRoot(y, n)
  while ((y+1)*(y+1) <= n) y++;
  return y;
}
```

Fig. 2. The example program, annotated with its specification.

$$
\begin{array}{lcl}
\texttt{root} & \triangleright & \forall n{:}\mathsf{Z}.\,\forall n^*{:}(n \geq 0).\,\mathsf{sint}\,n \twoheadrightarrow \exists x{:}\mathsf{Z}.\,\exists x^*{:}\mathsf{isRoot}\,x\,n.\,\mathsf{sint}\,x \\
& = & \texttt{poly}\ n{:}\mathsf{Z}.\ \texttt{poly}\ n^*{:}(n \geq 0).\ \texttt{lambda}\ \texttt{n}{:}\mathsf{sint}\,n. \\
& & (\texttt{fix}\ \texttt{loop}{:}\forall y{:}\mathsf{Z}.\,\forall y^*{:}\mathsf{leRoot}\,y\,n.\,\mathsf{sint}\,y \twoheadrightarrow \exists x{:}\mathsf{Z}.\,\exists x^*{:}\mathsf{isRoot}\,x\,n.\,\mathsf{sint}\,x. \\
& & \quad \texttt{poly}\ y{:}\mathsf{Z}.\ \texttt{poly}\ y^*{:}\mathsf{leRoot}\,y\,n.\ \texttt{lambda}\ \texttt{y}{:}\mathsf{sint}\,y. \\
& & \quad\quad \texttt{if}\ [\clubsuit,\clubsuit]\ ((\texttt{y} + \texttt{cint}\,[1])^2 > \texttt{n}, \\
& & \quad\quad\quad p_1^*\ .\texttt{pack}\,(y, \texttt{pack}\,(\clubsuit, y)\ \texttt{as}\ \exists y^*{:}\mathsf{isRoot}\,y\,n.\,\mathsf{sint}\,y)\ \texttt{as} \\
& & \quad\quad\quad\quad\quad \exists x{:}\mathsf{Z}.\,\exists x^*{:}\mathsf{isRoot}\,x\,n.\,\mathsf{sint}\,x, \\
& & \quad\quad\quad p_2{}^*.\,\texttt{loop}\ [y+1]\ [\clubsuit]\ (\texttt{y} + \texttt{cint}\,[1]))) \\
& & \quad [0]\ [\clubsuit]\,\texttt{cint}\,[0]
\end{array}
$$

Fig. 3. The λ_H^- term with the missing proofs that correspond to proof obligations (♣).

The program in Fig. 2 is subsequently compiled to the λ_H^- program of Fig. 3. Readers not familiar with the syntax of λ_H will probably find it hard to decipher the code. However, two things are obvious. First, the specifications in Fig. 2 have been translated to the types that are embedded in the term of Fig. 3. For instance, the type of `root` itself contains a direct translation of the function's pre- and postconditions. Second, there are five parts of this code, marked with the symbol ♣, that are missing. The first of these five is the predicate associated with the condition of the `if` expression. The remaining four are proofs that have to be constructed externally. Four proof obligations are therefore produced by the verification condition generator.

The proof obligations must now be discharged, either by an automatic theorem prover or by a proof assistant. Suppose that the second alternative is used and the human prover provides the code given in Fig. 4 for the Coq[1] proof assistant [16]. The missing parts of Fig. 3 can then be filled in, resulting in the λ_H program of Fig. 5.

[1] Coq uses the *Calculus of Inductive Constructions* (CIC) as its type language and, for this reason, Coq proofs can be directly embedded in λ_H, which is also based on CIC. Other theorem provers or proof assistants can be used instead, but the resulting proofs would then have to be translated to CIC. It is worth mentioning that all proof obligations were easily proved automatically (`by auto`) in Isabelle/HOL.

```
Definition leRoot (r : Z) (x : Z) := (r >= 0 /\ r*r <= x)%Z.
Definition isRoot (r : Z) (x : Z) := leRoot r x /\ ((r+1)*(r+1) > x)%Z.

Definition decidable (P : Prop) (b : bool) := if b then P else ~P.

Lemma geDecidablePrf: forall n m : Z, decidable (n >= m)%Z (Zge_bool n m).
intros; unfold decidable, Zge, Zge_bool;
  case (Zcompare n m); [ discriminate | auto | discriminate ].

Lemma gtDecidablePrf: forall n m : Z, decidable (n > m)%Z (Zgt_bool n m).
intros; unfold decidable, Zgt, Zgt_bool;
  case (Zcompare n m); [ discriminate | discriminate | auto ].

Lemma Z_ge_refl: forall n : Z, (n >= n)%Z.
auto with zarith.

Lemma Zplus_ge_compat:
  forall n m p q : Z, (n >= m -> p >= q -> n + p >= m + q)%Z.
intros n m p q; intros H1 H2; apply Zle_ge; apply Zplus_le_compat;
  apply Zge_le; assumption.
```

Fig. 4. Coq code, useful in discharging the proof obligations.

$$
\begin{array}{lll}
\texttt{root} & \triangleright & \forall n{:}\mathsf{Z}.\,\forall n^*{:}(n \geq 0).\,\mathsf{sint}\,n \twoheadrightarrow \exists x{:}\mathsf{Z}.\,\exists x^*{:}\mathsf{isRoot}\,x\,n.\,\mathsf{sint}\,x \\
 & = & \texttt{poly}\ n{:}\mathsf{Z}.\ \texttt{poly}\ n^*{:}(n \geq 0).\ \texttt{lambda}\ \mathtt{n}{:}\mathsf{sint}\,n. \\
 & & (\texttt{fix loop}{:}\forall y{:}\mathsf{Z}.\,\forall y^*{:}\mathsf{leRoot}\,y\,n.\,\mathsf{sint}\,y \twoheadrightarrow \exists x{:}\mathsf{Z}.\,\exists x^*{:}\mathsf{isRoot}\,x\,n.\,\mathsf{sint}\,x. \\
 & & \quad \texttt{poly}\ y{:}\mathsf{Z}.\ \texttt{poly}\ y^*{:}\mathsf{leRoot}\,y\,n.\ \texttt{lambda}\ \mathtt{y}{:}\mathsf{sint}\,y. \\
 & & \quad\quad \texttt{if}\,[\mathsf{decidable}\,((y+1)^2 > n), \mathsf{gtDecidablePrf}\,(y+1)^2\,n]\,(\\
 & & \quad\quad\quad (\mathtt{y} + \texttt{cint}\,[1])^2 > \mathtt{n}, \\
 & & \quad\quad\quad p_1^*\ .\,\mathsf{pack}\,(y, \mathsf{pack}\,(\mathsf{conj}\,y^*\,p_1{}^*, y)\ \mathsf{as}\ \exists y^*{:}\mathsf{isRoot}\,y\,n.\,\mathsf{sint}\,y)\ \mathsf{as} \\
 & & \quad\quad\quad\quad\quad \exists x{:}\mathsf{Z}.\,\exists x^*{:}\mathsf{isRoot}\,x\,n.\,\mathsf{sint}\,x, \\
 & & \quad\quad\quad p_2{}^*.\,\texttt{loop}\ [y+1]\ [\mathsf{conj}\,(\mathsf{Zplus\_ge\_compat}\,y\,0\,1\,0 \\
 & & \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathsf{proj1}\,y^*) \\
 & & \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathsf{geDecidablePrf}\,1\,0)) \\
 & & \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathsf{Znot\_gt\_le}\,(y+1)^2\,n\,p_2{}^*)]\ \ (\mathtt{y} + \texttt{cint}\,[1]))) \\
 & & \quad [0]\ \ [\mathsf{conj}\,(\mathsf{Z\_ge\_refl}\,0)\,(\mathsf{Zge\_le}\,n\,0\,n^*)]\ \texttt{cint}\,[0]
\end{array}
$$

Fig. 5. The λ_H term with the "linked" proofs.

Proof-preserving compilation phases can now be applied to the λ_H program. However, after CPS transformation, the size and complexity of the resulting program are too much for the human reader. The λ_K program obtained by the CPS transformation of the λ_H program of Fig. 5 and after some simple optimizations (such as constant propagation and beta contraction) is given in Fig. 6 at the end of this paper. To increase readability, the types of continuation parameters have been omitted from the λ_K program. In subsequent phases, still lower-level programs are obtained. The corresponding λ_C program, after a naïve closure conversion, is a few hundred lines long when expressed in the same textual format. To obtain an efficient implementation, it is essential to invent and implement proof-preserving compiler optimizations and to find a compact representation for proofs.

4 Conclusion

The realization of a complete hybrid system for constructing certified binaries requires the implementation of the system's two main software layers. Both for program verification and for type-based proof-preserving compilation, there is still a long way to go. However, in order to exploit the feasibility of such a system, we have used existing techniques and tools. To verify high-level source programs and produce proof obligations, a platform such as *Why/Caduceus* can be used. The integration of such a platform with a compiler from the source language to λ_H^- and the implementation of a proof checker and linker are still future work.

So far, we have partially implemented a proof-preserving compiler in OCaml, manipulating programs in the set of languages described by Shao *et al.* [7]. As an implementation of the type language (CIC) we have used the Coq proof assistant, whose source code is freely available. In this way, we can build on Coq's rich set of proof libraries. Our system is incompetent with long and complex source programs. There is much to be done before such an approach to software verification can be applied to real software.

References

1. IEEE: Standard Glossary of Software Engineering Terminology. IEEE Standard 610.12-1990.
2. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (1969) 576–585
3. Cousot, P.: Methods and logics for proving programs. In van Leeuwen, J., ed.: Formal Models and Semantics. Volume B of Handbook of Theoretical Computer Science. Elsevier Science Publishers B.V., Amsterdam, The Netherlands (1990) 843–993
4. Necula, G.: Proof-carrying code. In: Proceedings of the 24th ACM Symposium on the Principles of Programming Languages. (1997) 106–119
5. Necula, G.: Compiling with Proofs. PhD thesis, Carnegie Mellon University (1998)
6. Appel, A.W.: Foundational proof-carrying code. In: Proceedings 16th IEEE Symposium on Logic in Computer Science. (2001) 247–258

```
(lambda k. k
 (poly n:Z. lambda k. k
  (poly n*:(n ≥ 0). lambda k. k
   (lambda x_arg:sint n × K_c(∃x:Z. ∃x*:isRoot x n. sint x).
    let n = sel [N_lt_prop 0 2] (x_arg, cnat [0]) in
    let k_0 = sel [N_lt_prop 1 2] (x_arg, cnat [1]) in
     (fix loop [y:Z] (k:K_c(∀x*:leRoot y n. sint y ↠ ∃x:Z. ∃x*:isRoot x n. sint x)). k
      (poly y*:leRoot y n. lambda k. k
       (lambda x_arg:sint y × K_c(∃x:Z. ∃x*:isRoot x n. sint x).
        let y = sel [N_lt_prop 0 2] (x_arg, cnat [0]) in
        let k_1 = sel [N_lt_prop 1 2] (x_arg, cnat [1]) in
        let z_1 = y + cint [1] in
        let z_2 = z_1 * z_1 in
        let z_3 = z_2 > n in
         if [decidable ((y + 1)^2 > n), gtDecidablePrf (y + 1)^2 n] (z_3,
          p_1*. k_1 (pack (y, pack (conj y* p_1*, y) as K(∃y*:isRoot y n. sint y)) as
                     K(∃x:Z. ∃x*:isRoot x n. sint x)),
          p_2*. loop [y + 1] (lambda k. k [conj (Zplus_ge_compat y 0 1 0 (proj1 y*))
                                                (geDecidablePrf 1 0))
                                          (Znot_gt_le (y + 1)^2 n p_2*)]
                (lambda k. let z_1 = y + cint [1] in k ⟨z_1, k_1⟩))))))) [0]
      (lambda k. k [conj (Z_ge_refl 0) (Zge_le n 0 n*)] (lambda k. k ⟨cint [0], k_0⟩)))))))
```

Fig. 6. The λ_K term, after the proof-preserving CPS transformation and some optimizations.

7. Shao, Z., Trifonov, V., Saha, B., Papaspyrou, N.: A type system for certified binaries. ACM Transactions on Programming Languages and Systems **27**(1) (2005) 1–45
8. Crary, K., Vanderwaart, J.C.: An expressive, scalable type theory for certified code. In: Proceedings of the 7th ACM International Conference on Functional Programming. (2002) 191–205
9. Howard, W.A.: The formulae-as-types notion of constructions. In Seldin, J.P., Hindley, J.R., eds.: To H. B. Curry: Essays on Computation Logic, Lambda Calculus and Formalism. Academic Press, Boston, MA (1980) 479–490
10. Harper, R., Morrisett, G.: Compiling polymorphism using intensional type analysis. In: Proc. 22nd ACM Symp. on Principles of Prog. Lang. (1995) 130–141
11. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. In: Proc. 25th ACM Symp. on Principles of Prog. Lang. (1998) 85–97
12. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)
13. Gries, D.: The Science of Programming. Springer-Verlag (1981)
14. Filliâtre, J.C.: Why: A multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud (March 2003)
15. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Computer Aided Verification. Volume 4590 of LNCS. Springer (2007) 173–177
16. The Coq Proof Assistant Reference Manual, URL: `http://coq.inria.fr/`