Logic-Enhanced Type Systems: Programming Language Support for Reasoning about Security and Other Program Properties

Nikolaos S. Papaspyrou Dimitrios Ch. Vytiniotis Vassilios M. Koutavas

National Technical University of Athens
School of Electrical and Computer Engineering
Software Engineering Laboratory
Polytechnioupoli, 15780 Zografou, Athens, Greece.
E-mail: {nickie, dvitin, vkoutav}@softlab.ntua.gr

Abstract

Existing compilers that produce certified code (e.g. the Java compiler) focus on relatively simple and informally specified safety properties that are enforced by an elaborate run-time system with a significant performance overhead. Recent research supports the opinion that it is possible to design and implement a general framework for formally reasoning about security and other interesting program properties using low-level languages with strong type systems. In such a language, a certified binary file is simply a program whose type provides a collection of properties that the program satisfies; the type checker can determine statically and with little cost whether a given certified binary is sound and, if it is, the program can be executed with no further performance overhead. In this paper, we exemplify how a formal logic can be embedded in the type system of a programming language. Without plunging into details, we discuss a number of interesting program properties that can be certified and what is to be expected in the foreseeable future.

1 Introduction

Modern software systems are organized as groups of small collaborating components, independently developed and deployed. These may be physically distributed over a network: executable code can be transferred from one computing device to another just as any form of data. In this setting, *platform independence* and *interoperability* are essential non-functional requirements. The former urges for a standardized intermediate form of code that will eventually be translated to native code. The latter is even more critical, as the transferred code is expected to integrate with the existing software in the target computing device.

In such distributed software systems, it is not reasonable to assume that executable code can be fully trusted. The transferred code may cause problems either on purpose (e.g. viruses, malicious attacks) or inadvertently, if it contains bugs that jeopardize the stability and security of the target computing device. As a variety of applications from different areas (e.g. telecommunications, electronic commerce, tele-medicine, distance education) largely depend on the quality of such code, it is evident that, to guarantee our safety in the information and knowledge society, we must assure the flawless behaviour of executable code.

In the mid 1990s, Sun's Java platform became the first widely known attempt to tackle the problems of portability and security of executable code [1]. A similar idea was later advocated by Microsoft's .NET framework [2], in which programs written in a variety of programming languages can be translated to a common intermediate language and then executed by a Common Language Runtime. On both platforms, the security of executable code is a responsibility of the virtual machine which surveys that executed software components satisfy a predetermined *safety policy*, hardwired in the implementation of the virtual machine and allowing for minor customization.

There are a few disadvantages in this approach. The virtual machine must scrutinize the execution of the distrusted code and this results in some inevitable performance loss. The safety guarantee is only valid as long as the virtual machine is well designed and correctly implemented and, given the size and complexity of its implementation, one can hardly be certain that it is. Furthermore, programmers are often forced to deceive the virtual machine or weaken the safety policy in order to program real applications in an efficient way. In response to this problem, both platforms suggest the use of cryptographic authentication techniques to guarantee that a possibly harmful component comes from a trusted source.

In parallel to the developments of the software industry, many researchers have dealt with the problem of security of executable code. Their goals have been: (i) to avoid performance loss by statically verifying software components and then translating to native code; (ii) to blindly trust very few and very simple things, including the safety policy and the implementation of the runtime environment; and (iii) to design programming languages expressive enough to allow the efficient implementation of components without sidestepping safety measures. Some of the most promising approaches are briefly presented in §2.

They all propose the use of a sound *formal logic* in combination with the programming language; the safety policy and other program properties are expressed as propositions provable in this logic.

In this paper we exemplify how a formal logic for reasoning about security and other interesting properties of executable code can be embedded in the type system of a programming language. We outline a framework for verifying arbitrary properties of software components with the following objectives in mind: (i) *Formality:* a formal logic for specifying properties of software components, visible and available as a reasoning tool inside the programming language; (ii) *Expressive power:* a logic powerful enough to express and verify arbitrary properties, even if programmers are required to provide assistance in proving these properties; (iii) *Flexibility:* source and target language independence, flexible safety policy; (iv) *Scalability:* the "logical" component must not increase in size and complexity disproportionately to the size of the code itself.

2 Programming language support for certified code: An overview

Typed Intermediate Language (TIL) [3] and *Typed Assembly Language* (TAL) [4] were two of the earliest general attempts to extend low-level languages with strong type systems that can be used for code certification. Appropriate type checkers can guarantee that some, usually simple, safety requirements are met by verifying the type correctness of programs before execution. Moreover, because the safety policy is embedded in the language, safety proofs can be constructed automatically by the compiler.

Proof-Carrying Code (PCC) is a general framework to tackle system integrity and security problems, using principles and techniques from logic and programming language semantics [5]. Certified binaries in the PCC framework are packages containing executable code and a representation of a proof that this code meets a given safety policy. By validating the attached proof, one can guarantee the safety of executing a certified binary. In summary, the PCC framework requires a relatively simple trusted infrastructure and does not impose run-time penalties.

In comparison with TAL, PCC uses a general-purpose first-order predicate logic and can express more elaborate security properties. On the other hand it uses explicit safety proofs, which in general cannot be produced automatically, and its underlying programming language is less expressive than TAL. Both approaches however share a basic principle: the logic in which safety proofs are expressed contains a built-in understanding of the underlying programming language and its type system.

Foundational Proof-Carrying Code (FPCC) intends to minimize the size of the components that must be blindly trusted (or proved in a meta-theory) in a PCC system [6]. FPCC uses a general-purpose higher-order predicate logic and a few axioms of arithmetic, which can be used as a foundation of modern mathematics. This logic is used to define the safety policy, the type system and the semantics of the underlying programming language. In the FPCC framework, certified binaries are packages containing executable code and a representation of a foundational safety proof, i.e. a proof which explicitly defines, down to the foundations of mathematics, all required concepts and explicitly proves all required properties of these concepts.

Compared with TAL and PCC, FPCC is more flexible. It is not restricted to a particular underlying programming language, nor to a given subset of logic or mathematics. Foundational proofs may contain the definition of a novel type system for the target language and new arguments to prove the safety policy. FPCC is also more secure, as fewer and simpler things need to be trusted. In contrast, foundational proofs are longer and harder to construct.

Recent attempts combine the benefits of TAL for automatically constructing certified code with the minimality and expressive power of FPCC. Shao *et al.* have proposed a type-theoretic framework for constructing, composing and reasoning about certified binaries [7]. Their work is based on the "formulae as types" principle [8]; propositions and proofs are expressed in a general and powerful type language encompassing higher-order predicate logic. The same type language is used in the underlying programming language; in this way, programs and proofs are elegantly integrated and formal reasoning can be performed inside the language itself. An approach similar in spirit but different in the details of the logic has been proposed by Crary and Vanderwaart [9]. The similarity of the two lies in the embedding of a logic capable of representing program properties in the language's type system.

3 An embedding of logic

Following the approach presented in [7], we split the language in which software components are written in two: the *type language* and the *computation language*.

The *type language* is the part of the language where all the logical reasoning takes place. It is a variant of the Calculus of Inductive Constructions [10], as implemented in the Coq proof assistant [11], which incorporates higher-order predicate logic. The type language serves as a uniform logical tool, the common denominator of all programming languages used for implementing software components.

A large body of *theories* useful for reasoning about software components can be defined in the type language. The Coq proof assistant has a vast library of theories including intuitionistic, classical and linear logic, Peano and binary arithmetic, rational and real numbers, sets, relations, abstract data structures, and many case studies coming from various areas of mathematics and computer science. These theories can be directly "translated" to our type language and, in similar fashion, new theories can be encoded.

The *computation language* is the part of the language in which all the computations are described. It is defined by: (i) *syntactic rules*, which specify the syntax of well-formed both computation terms and types; (ii) *typing rules*, which specify which well-formed computation terms are well-typed; and (iii) *semantic rules*, which specify how a software component that is represented by a well-formed and well-typed computation term behaves during execution. All these components can be formally defined in the type language itself. In this way, programmers are not restricted to a single computation language or even to a given set of possible computation languages.

The separation between the type language and the computation language allows us to preserve desired properties of the former, such as substitution of equals for equals and strong normalization, without sacrificing the expressive power of the latter. To allow reasoning about properties of software components at the level of the programming language, it is useful to incorporate (a part of) the type language in the type system of the computation language.

3.1 Overview of the type language

The abstract syntax of the type language is given by the following grammar:

$$
\begin{array}{lcl}
s & ::= & \mathsf{Set} \mid \mathsf{Type} \mid \mathsf{Ext} \\
A, B & ::= & s \mid X \mid \Pi X\!:\!A.\,B \mid \lambda X\!:\!A.\,B \mid A\,B \\
 & \mid & \mathsf{Ind}(X\!:\!A)\{\vec{A}\} \mid \mathsf{Constr}(n, A) \\
 & \mid & \mathsf{Elim}[A'](A\!:\!B\,\vec{B})\{\vec{A}\}
\end{array}
$$

where X denotes a variable, A and B denote terms, s denotes a subset of terms called *sorts*, n denotes a natural number and $\vec{A}$ denotes a sequence of terms.

Sorts are the constants of the type language. Apart from sorts and variables, a term can be a product $\Pi X\!:\!A.\,B$, an abstraction $\lambda X\!:\!A.\,B$, an application $A\,B$, an inductive type $\mathsf{Ind}(X\!:\!A)\{\vec{A}\}$, a constructor of an inductive type $\mathsf{Constr}(n, A)$ or an elimination of an inductive type $\mathsf{Elim}[A'](A\!:\!B\,\vec{B})\{\vec{A}\}$.

The *typing relation* in the type language determines the semantic validity of terms. For any pair of terms, $A : B$ is read "A has type B". Among sorts, we have $\mathsf{Set} : \mathsf{Type}$ and $\mathsf{Type} : \mathsf{Ext}$. Products are essentially dependent function types: an abstraction $\lambda X\!:\!A.\,B$ has type $\Pi X\!:\!A.\,B'$ provided that $B : B'$ and, if it is applied to a term of type A, it produces a term of type B'. We write $A \to B$ instead of $\Pi X\!:\!A.\,B$ if X does not occur free in B.

Inductive types can be defined by using Ind. As an example, the types Bool of Boolean values and Nat of natural numbers can be defined as follows:

$$
\begin{array}{lcllcl}
\mathsf{Bool} & \equiv & \mathsf{Ind}(X\!:\!\mathsf{Set})\{X; X\} & : & \mathsf{Set} \\
\mathsf{Nat} & \equiv & \mathsf{Ind}(X\!:\!\mathsf{Set})\{X; X \to X\} & : & \mathsf{Set}
\end{array}
$$

Inside Ind, the variable X is a synonym for the inductive type that is being defined. Bool and Nat have two constructors each; the types of the constructors are given inside the braces. $\mathsf{Constr}(n, A)$ provides access to the n-th constructor of an inductive type A but it is convenient to give constructors descriptive names:

$$
\begin{array}{lclcl}
\mathsf{true} & \equiv & \mathsf{Constr}(0, \mathsf{Bool}) & : & \mathsf{Bool} \\
\mathsf{false} & \equiv & \mathsf{Constr}(1, \mathsf{Bool}) & : & \mathsf{Bool} \\
\mathsf{zero} & \equiv & \mathsf{Constr}(0, \mathsf{Nat}) & : & \mathsf{Nat} \\
\mathsf{succ} & \equiv & \mathsf{Constr}(1, \mathsf{Nat}) & : & \mathsf{Nat} \to \mathsf{Nat}
\end{array}
$$

Although an abuse of notation, it is convenient to use decimal numerals for elements of Nat; we will write 2 instead of $\mathsf{succ}\,(\mathsf{succ}\,\mathsf{zero})$.

Most interesting operations on elements of inductive types require the use of Elim, whose rôle is twofold. First of all, it provides access to the components of which an element of an inductive type was built, i.e. the constructor that was used and the parameters that were applied to it. Moreover, it allows primitive recursion on inductive types. As a simple example, consider the negation of a Boolean value:

$$
\begin{array}{lcl}
\mathsf{not} & : & \mathsf{Bool} \to \mathsf{Bool} \\
 & \equiv & \lambda b\!:\!\mathsf{Bool}.\,\mathsf{Elim}[\mathsf{Bool}](b\!:\!\mathsf{Bool})\{\mathsf{false}; \mathsf{true}\}
\end{array}
$$

The Boolean value b is inspected and the result is false if b is true, or true if b is false. The type of the result returned by Elim is given inside the square brackets.

The type language features three types of *reductions*: β, η and ι. The first two are well-known from the study of λ-calculus; the third is used in the elimination of inductive types. In addition, α-reduction allows the renaming of bound variables. When not is applied to a Boolean value, the result is computed with a sequence of reductions:

$$
\begin{array}{ll}
 & \mathsf{not}\,\mathsf{true} \\
\to_\beta & \mathsf{Elim}[\mathsf{Bool}](\mathsf{Constr}(0, \mathsf{Bool})\!:\!\mathsf{Bool})\{\mathsf{false}; \mathsf{true}\} \\
\to_\iota & \mathsf{false}
\end{array}
$$

As a more involved example, consider a function which tests whether a natural number is odd.

$$
\begin{array}{lcl}
\mathsf{odd} & : & \mathsf{Nat} \to \mathsf{Bool} \\
 & \equiv & \lambda n\!:\!\mathsf{Nat}.\,\mathsf{Elim}[\mathsf{Bool}](n\!:\!\mathsf{Nat})\{\mathsf{false}; \\
 & & \qquad\qquad \lambda m\!:\!\mathsf{Nat}.\,\lambda b\!:\!\mathsf{Bool}.\,\mathsf{not}\,b\}
\end{array}
$$

In this case, again, the number n is inspected. If it is zero, the result is false. Otherwise, if n is of the form $\mathsf{succ}\,m$ for some $m : \mathsf{Nat}$, the result depends on whether m is odd or not, i.e. odd is a primitive recursive function. In the second clause of Elim, the parameter b is the result of doing the same elimination on m instead of n. A longer sequence of reductions is required when odd is applied:

$$
\begin{array}{ll}
 & \mathsf{odd}\,1 \\
\to_\beta & \mathsf{Elim}[\mathsf{Bool}](\mathsf{succ}\,\mathsf{zero}\!:\!\mathsf{Nat})\{\ldots\} \\
\to_\iota & (\lambda m\!:\!\mathsf{Nat}.\,\lambda b\!:\!\mathsf{Bool}.\,\mathsf{not}\,b)\ \mathsf{zero} \\
 & \quad \mathsf{Elim}[\mathsf{Bool}](\mathsf{zero}\!:\!\mathsf{Nat})\{\ldots\} \\
\twoheadrightarrow_\beta & \mathsf{not}\,\mathsf{Elim}[\mathsf{Bool}](\mathsf{zero}\!:\!\mathsf{Nat})\{\ldots\} \\
\to_\iota & \mathsf{not}\,\mathsf{false} \ \twoheadrightarrow\ \mathsf{true}
\end{array}
$$

3.2 Propositions, proofs and theories

So far, it has been demonstrated how the type language can be used to define mathematical objects, such as sets.

Based on the "formulae as types" principle [8], it is possible to encode propositions and proofs as well. Propositions are terms of type Set. A proof of a proposition P is a term p^* such that $p^* : P$. Functions $P \to Q$ correspond to logical implication, whereas products $\Pi X\!:\!A.\,P$ correspond to universal quantification. Inductive types can be used to define specific propositions or operators; their constructors correspond to logical axioms. For example:

$$\begin{array}{lcllcl} \mathsf{True} & \equiv & \mathsf{Ind}(X\!:\!\mathsf{Set})\{X\} & : & \mathsf{Set} \\ \mathsf{False} & \equiv & \mathsf{Ind}(X\!:\!\mathsf{Set})\{\} & : & \mathsf{Set} \end{array}$$

The constructor of True is an axiom stating that True is a valid proposition. In contrast, False has no constructors. Negation and conjunction can be easily defined as follows.

$$\begin{array}{lcl} \mathsf{propNot} & : & \mathsf{Set} \to \mathsf{Set} \\ & \equiv & \lambda P\!:\!\mathsf{Set}.\, P \to \mathsf{False} \\ \mathsf{propAnd} & : & \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set} \\ & \equiv & \lambda P\!:\!\mathsf{Set}.\, \lambda Q\!:\!\mathsf{Set}. \\ & & \quad \mathsf{Ind}(X\!:\!\mathsf{Set})\{P \to Q \to X\} \end{array}$$

For convenience, we write $\neg P$ and $P \wedge Q$ instead of $\mathsf{propNot}\, P$ and $\mathsf{propAnd}\, P\, Q$.

Similarly, we can define properties of mathematical objects, e.g. equality and inequality for numbers. We can then prove theorems about these properties, e.g. the transitivity and antisymmetry of $\leq$. The theories built in this way are the tool that helps us reason about programs.

4 Reasoning about programs

Using an appropriate type system for the computation language, the static type checking of computation terms can guarantee a number of desired execution properties. *Integrity* and *confidentiality* are such properties: executable code must not harm the target computing device by corrupting the contents of memory or user files and must not have access to private information. *Termination*, time/space *complexity* and *correctness* are even more important, but the construction of proofs for these properties is bound to require the programmer's assistance in general.

In the rest of this section, we assume that such a type system has been defined for an intermediate functional computation language. For lack of space, this language cannot be presented here in detail; we refer the reader to [7, 12]. We assume that the types of the computation language include $\mathsf{natural}$ and $\mathsf{integer}$ numbers; functions (written as $\tau \twoheadrightarrow \tau'$ to avoid confusion with functions of the type language); arrays of given length; universally and existentially *quantified types*, written as $\forall X\!:\!A.\,\tau$ and $\exists X\!:\!A.\,\tau$, where A is an object of the type language; and *singleton types*, such as $\mathsf{snat}\, n$, where $n : \mathsf{Nat}$ is an object of the type language. An expression of type $\mathsf{snat}\, n$ is simply a natural number whose value is known to be equal to n. As shown in the examples that follow, using quantified and singleton types it is possible to create a connection between the computation language and the type language, so as to use the logical reasoning power of the latter in the former.

(a) In imperative form	(b) In functional form

`i := 1; min := a[0];`
`while i < n do`
`begin`
`  if a[i] < min then`
`    min := a[i];`
`  i := i + 1`
`end`
	`letrec loop i min =`
	`  if i < n then`
	`    if a[i] < min then`
	`      loop (i + 1) a[i]`
	`    else`
	`      loop (i + 1) min`
	`  else min`
	`in loop 1 a[0]`

Figure 1: Program that finds the minimum of an array.

4.1 Array bounds and integrity issues

Consider a part of a program which finds the smallest element in an array of n integer numbers (Fig. 1). If we are concerned with security, we want to know that during its execution the program will respect the bounds of the array. This is a critical integrity issue: many security holes that currently exist in systems software are due to *buffer overrun*, which is essentially a violation of array bounds.

As a remedy to this problem, the computation language supports a selection operator for arrays that has the form $\mathtt{sel}\,[p^*]\,(\mathsf{a}, \mathtt{i})$. Suppose that a has type $\mathsf{array}\, n\, \tau$ for some natural number n and some type τ and that $\mathtt{i}$ has type $\mathsf{snat}\, i$, i.e. it is a natural number with value i. Then, for the expression $\mathtt{sel}\,[p^*]\,(\mathsf{a}, \mathtt{i})$ to type check, p^* must be a proof of the proposition $i < n$. In this way, to use the selection operator the programmer must provide a proof that array bounds are respected. Security is guaranteed at no performance loss. It remains to be seen how the programmer can provide such a proof.

Fig. 2 shows the complete program in the computation language. Compared to the functional version of Fig. 1, one can see that everything additional is written in the type language. Explicit types have been added and, as its type says, to call `findMin` one must provide a proof that $n > 0$. This proof is used in the last line, when the first element of the array $\mathsf{a}[0]$ is accessed.

There are two more uses of the array selection operator in the `then` clause of the first `if`. Before $\mathsf{a}[\mathtt{i}]$ is accessed, a proof of $i < n$ must be provided. This is precisely the condition of the `if` and our intuition tells us that such a proof must exist, otherwise we wouldn't be in the `then` clause. The form of the conditional in the computation language follows our intuition: for $\mathtt{if}\,[F, p^*]\,(\mathsf{e}, p_1^*.\,\mathsf{e}_1, p_2^*.\,\mathsf{e}_2)$ to type check, e must be of the singleton type $\mathsf{sbool}\, b$ and F must be of type $\mathsf{Bool} \to \mathsf{Set}$, i.e. a function returning two propositions, one for each of the Boolean values true and false. Depending on the value of b, control passes either to the `then` or to the `else` clause. The two propositions returned by F provide information about what will hold if control passes to one of the two clauses; p^* must be a proof of the proposition $F\, b$. Inside the two clauses, p_1^* and p_2^* can be used as proofs of $F\, \mathsf{true}$ and $F\, \mathsf{false}$ respectively. In simple words, p_1^* is the proof that $i < n$. For a better understanding of `if` the reader is referred to [7, 12].

$$
\begin{array}{lll}
\texttt{findMin} & : & \forall n{:}\mathsf{Nat}.\,\forall n^*{:}(n>0).\,\mathsf{snat}\,n \twoheadrightarrow \mathsf{array}\,n\,\mathsf{integer} \twoheadrightarrow \mathsf{integer} \\
& = & \texttt{poly}\;n{:}\mathsf{Nat}.\;\texttt{poly}\;n^*{:}(n>0).\;\texttt{lambda}\;\texttt{n}{:}\mathsf{snat}\,n.\;\texttt{lambda}\;\texttt{a}{:}\mathsf{array}\,n\,\mathsf{integer}. \\
& & \texttt{letrec}\quad \texttt{loop}\quad : \quad \forall i{:}\mathsf{Nat}.\,\mathsf{snat}\,i \twoheadrightarrow \mathsf{integer} \twoheadrightarrow \mathsf{integer} \\
& & \qquad\qquad\qquad = \quad \texttt{poly}\;i{:}\mathsf{Nat}.\;\texttt{lambda}\;\texttt{i}{:}\mathsf{snat}\,i.\;\texttt{lambda}\;\texttt{min}{:}\mathsf{integer}. \\
& & \qquad\qquad\qquad\quad \texttt{if}\,[\mathsf{decidable}\,(i<n),\mathsf{ltDecidablePrf}\,i\,n]\,(\texttt{i}<\texttt{n}, \\
& & \qquad\qquad\qquad\qquad p_1^*.\,\texttt{if}\,[\lambda b{:}\mathsf{Bool}.\,\mathsf{propTrue},\mathsf{propTruePrf}]\,(\texttt{sel}\,[p_1^*]\,(\texttt{a},\texttt{i})<\texttt{min}, \\
& & \qquad\qquad\qquad\qquad\qquad q_1^*.\,\texttt{loop}\;[i+1]\;(\texttt{i}+1)\;\texttt{sel}\,[p_1^*]\,(\texttt{a},\texttt{i}), \\
& & \qquad\qquad\qquad\qquad\qquad q_2^*.\,\texttt{loop}\;[i+1]\;(\texttt{i}+1)\;\texttt{min}), \\
& & \qquad\qquad\qquad\qquad p_2^*.\,\texttt{min}) \\
& & \texttt{in}\qquad \texttt{loop}\;[1]\;\texttt{1}\;\texttt{sel}\,[\mathsf{natGtLtSymm}\,n\,0\,n^*]\,(\texttt{a},0)
\end{array}
$$

Figure 2: A secure version of the program in Fig. 1.

(a) In imperative form	(b) In functional form

`y := 0;`
`while (y + 1) *`
`      (y + 1) <= n do`
`  y := y + 1`
	`letrec loop y =`
	`  if (y+1)*(y+1) > n`
	`    then y`
	`    else loop (y + 1)`
	`in loop 0`

Figure 3: Program that finds the integer square root.

4.2 A simple correctness proof

The computation language can also be used for reasoning about the (partial) correctness of programs. Consider a function calculating the integer square root of n (Fig. 3). The function's type can contain a complete specification:

$$
\begin{array}{l}
\forall n{:}\mathsf{Int}.\,\forall n^*{:}(n\geq 0).\,\mathsf{sint}\,n \twoheadrightarrow \\
\quad \exists x{:}\mathsf{Int}.\,\exists x^*{:}(x\geq 0 \wedge x^2\leq n \wedge (x+1)^2>n).\,\mathsf{sint}\,x
\end{array}
$$

i.e. given an integer whose value is $n \geq 0$, there exists a result value $x \geq 0$ such that $x^2 \leq n$ and $(x+1)^2 > n$.

In constructing a complete correctness proof for this function, the programmer must provide the *loop invariant*, which is again encoded in the type of function `loop`:

$$
\begin{array}{l}
\forall y{:}\mathsf{Int}.\,\forall y^*{:}(y\geq 0 \wedge y^2\leq n).\,\mathsf{sint}\,y \twoheadrightarrow \\
\quad \exists x{:}\mathsf{Int}.\,\exists x^*{:}(x\geq 0 \wedge x^2\leq n \wedge (x+1)^2>n).\,\mathsf{sint}\,x
\end{array}
$$

The complete program, including the proof of partial correctness, is 16 lines long. It is omitted for lack of space.

5 Conclusion

In this paper, we have presented a type system enhanced with higher-order predicate logic and shown how it can be used for reasoning about program properties. This formal reasoning can be expressed inside the programming language, thus laying the foundations for principled development of software with explicit and verifiable properties.

Our research in this area is at a preliminary stage and the current state-of-the-art is still far from reaching maturity. This approach builds upon a large body of work in logic and theorem-proving and attempts to incorporate this work in industrial-strength compilers. Although many and difficult problems, both theoretical and practical, must be solved before programmers harvest the crop of this effort, we believe that this approach is a significant step towards building and using efficient executable code which provably satisfies arbitrary desired properties.

Acknowledgement. This work builds on a past collaboration with Zhong Shao, Valery Trifonov, Bratin Saha and other members of the FLINT group at Yale University, to whom the original vision described in this paper should be attributed.

References

[1] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison-Wesley, 1996.

[2] Microsoft .NET home page, URL: `http://msdn.microsoft.com/net/`.

[3] R. Harper and G. Morrisett, "Compiling polymorphism using intensional type analysis," in *Proc. 22nd ACM Symp. on Principles of Prog. Lang.*, pp. 130–141, 1995.

[4] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From System F to typed assembly language," in *Proc. 25th ACM Symp. on Principles of Prog. Lang.*, pp. 85–97, 1998.

[5] G. Necula, *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Sept. 1998.

[6] A. W. Appel, "Foundational proof-carrying code," in *Proc. 16th IEEE Symp. on Logic in Computer Science*, pp. 247–258, 2001.

[7] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou, "A type system for certified binaries," in *Proc. 29th ACM Symp. on Principles of Prog. Lang*, pp. 217–232, 2002.

[8] W. A. Howard, "The formulae-as-types notion of constructions," in *To H. B. Curry: Essays on Computation Logic, Lambda Calculus and Formalism*, Academic Press, 1980.

[9] K. Crary and J. C. Vanderwaart, "An expressive, scalable type theory for certified code," in *Proc. 7th ACM Intl. Conf. on Functional Prog.*, pp. 191–205, 2002.

[10] B. Werner, *Une Théorie des Constructions Inductives*. Thèse de doctorat, Université Paris VII, May 1994.

[11] The Coq Proof Assistant Reference Manual, URL: `http://coq.inria.fr/`.

[12] V. M. Koutavas, "Calvin to NFlint compiler," Tech. Rep. CSD-SW-TR-2-02, National Technical Univ. of Athens, Software Engineering Laboratory, Oct. 2002. In Greek.