Dependent Types, Theorem Proving, and Applications for a Verifying Compiler

Yves Bertot and Laurent Théry

INRIA Sophia Antipolis

1 Theorem proving and Program development

One approach to Prof. Hoare's challenge is to view the development of verified software from the perspective of interactive theorem provers. This idea is not new and many medium-scale software systems have been developed and verified in this manner. Developments based on HOL, ACL2, or PVS have already been described and advocated and our position stands on the same line: most powerful (higher-order) theorem proving systems already contain a programming language, programs can be developed and the correctness of these programs can be specified and verified, they can then be compiled into traditional executable code. In this sense, we already have a small scale example of a verification aware programming language.

We propose to take advantage of the notion of "dependent types" to ensure that this programming language combines powerful logical capabilities, reasonable expressive power, and practical linkage between computational content and logical annotations.

Almost all mathematic developments contain algorithms. This imposes that all theorem proving tools should contain a programming language. The usual approach is to restrict this programming language to make the verification of algorithms more regular. The most common kind of restriction is to consider a strongly typed side-effect free functional language, with restricted forms of recursion to ensure that all computations terminate (often boiling down to a language with algebraic data-types and a form of primitive recursion adapted to these datatypes). Apparently, this limitation on the programming language hinders the possibility to study realistic efficient programs, but this question can be circumvented to obtain the equivalent of full-fledged general purpose programming language.

The annotation mechanism found in the Floyd-Hoare approach is one of the best solutions to associate properties to programs and support mechanical verification of these properties. In a functional language programming with dependent types, this capability becomes naturally practical for functional programming languages.

2 Extension with dependent types

In the wide family of theorem provers for higher-order logic, we have experimented with provers based on type theory [5], where the distinguishing feature

is the use of dependent types and their possible interpretation as logical formulae. This feature adds expressive power to the programming language being studied inside the theorem prover. First, dependent types can be used to carry the annotations usually found in programs in the Floyd-Hoare approach. Second, the programs need not be total anymore, they can be partial because dependent types can be used to restrict the way functions are supposed to be used. Actually both points are the same: with dependent types we can assert that some function is to be used only on data that satisfies some input specification and we can also assert that the output data satisfies some property with respect to the input. In a sense, we consider that dependent types are the equivalent in the functional programming world to the Floyd-Hoare triples in the imperative programming world.

The treatment of partial functions is quite extensive. For instance, it is possible to describe functions whose termination is undecidable in general. Dependent types can be used to express that the value returned by these functions is valid only when one is able to prove that they do terminate. Dependent types are thus used to give a logical description of the domain of definition.

Translating the dependently typed programming language found in a type-theory based theorem prover to a conventional programming language is not direct. It is a more complex problem than for the recursive functional programming language found in other systems. It can be performed in a way that the computations that are relevant to the logical parts are discarded and only the relevant computations for an efficient construction of the result are present in the target program. This translation process, akin to separation between compile-time optimization run-time computation is usually known as *extraction* [9].

State-based programs from the conventional imperative programming world can also be described using monadic approaches (a big word to state that all functions take an extra argument representing the state and return a tuple as argument, where an extra component represents the modified state). Exceptions and other programming constructs can probably also be accommodated. The encoding of non-functional programming constructs needs to be taken into account in the extraction mechanism, so that efficient state-based programs are not translated to inefficient functional programming languages. This improvement of extraction is not done today, but there is no doubt it could be done satisfactorily in the foreseeable future.

The direct integration of dependent types into existing programming languages has also been studied for about 10 years. Notable results are exposed in [13] where the language ML is extended with a restricted form of dependent types, so that the conditions that need to be satisfied for type verification fall in the category of problems satisfied by some form of constraint programming. Another notable experiment is described in [1], where a subset of the Haskell language is extended with dependent types and verification relies on existence of proofs, which are simply other collections of well-typed programs. In both cases, the absence of restriction on recursion imply that well-typed programs may ex-

hibit a behavior that is not foreseen by their types, but dependent types make it possible to perform more accurate verification of the correctness of programs.

3 Example applications

In our team, we worked on a variety of examples, drawn from a variety of computer science domains, and often in collaboration with specialists from the addressed fields.

- Computer algebra: Buchberger's algorithm to compute Gröbner bases [11],
- Decision procedures: Stålmarck's algorithm for propositional logic [8], Presburger's procedure.
- Computer arithmetics: libraries for correct rounding in floating points, algorithms on floating point expansions (based on IEEE754 operations) [7], efficient square root computation (taken from the GMP library) [6].
- Programming language technology: Java byte-code verifier [3], compiler for a non-trivial fragment of C.
- Computational geometry: convex-hulls [10].

While many of these algorithms have been described in a purely functional language, we studied state-based encodings in a variety of manners, through abstract data-types or encodings of imperative programming languages with Floyd-Hoare like assertions. In some cases, this made it possible to prove properties about memory usage. Most of these program studies were supported by the development of important bodies of mathematical theory concerning for instance polynomials, plane geometry, real or rational numbers, programming language semantics, lattice theory.

3.1 Detailed examples

In conventional functional programming, we write `a : b` to express that the value `a` has the type `b`. Types are constants like `int` or `bool`, sometimes parameterized constants like $\texttt{list}_\tau$, where τ is itself a type, and function types, written $\tau \to \tau'$. The use of parameterized constants is a precursor of using dependent types. For example the function that takes a list and returns its first element has the type $\texttt{list}_\tau \to \tau$, so the type of the result depends on the type of the input. So the extension is quite simple to express: with dependent types, parameterized type constants may be indexed by arbitrary values and the type of a function's result will be allowed to depend on the type of that function's argument. We need a new notation to represent this. We will use $\forall x : \tau, T\ x$ to represent the type of functions that take arguments in type τ and return a value in a type $T\ x$ that varies when the input x varies.

Meaningful dependent types and values could have the following form:

- `vect x`, the vectors of integers of length x, the value `vector_nil` could be a value in the type `vect 0` and the value

`vector_app : ∀ n m, vect n * vect m → vect (n+m)`
could be a dependently typed function that takes two vectors and builds an new vector by concatenation.

- `lt y x`, the witnesses (proofs) that $y < x$, The functions
  `lt_1: ∀ x, lt x (x+1)`, and
  `lt_m : ∀ x, lt x m → lt x (m+1)`
  could be used to construct these witnesses. The type family `lt` is not a real type of values, but rather a type of proofs. Types of this kind are mostly understood as predicates.
- Given an arbitrary predicate P on the type τ, the `{y | P y}` would be a special notation to represent the elements of τ that satisfy the predicate P. Actually the elements of this type are the pair of a value `y` in τ and an element in the type P `y`.
- `sorted` a predicate on lists of integers,
- `sorted_nil : sorted []`,
  `sorted_one : ∀ x, sorted [x]`, and
  `sorted_cons : ∀ x y l,`
  `lt x y → sorted (y::l) → sorted (x::y::l)`,
  theorems that can be used to build proofs that a list is sorted[1]
- `{l | sorted l}` could be the type of sorted list,
- `permutation` a 2-place predicate on lists of integers,
- `permutation_refl : ∀ l, permutation l l`,
  `permutation_swap : ∀ x y l, permutation (x::y::l) (y::x::l)`,
  `permutation_skip : ∀ x` l_1 l_2`,`
  `permutation` l_1 l_2 `→ permutation (x::`l_1`) (x::`l_2`)`, and
  `permutation_trans : ∀` l_1 l_2 l_3`, permutation` l_1 l_2 `→`
  `permutation` l_2 l_3 `→ permutation` l_1 l_3
  theorems that can be used to build proofs that a list is a permutation of another list.
- `∀ x, {y | sorted y /\ permutation x l}`
  could be the type of a sorting function.
- `mk_sorted_value : ∀ l x, sorted x →`
  `permutation l x → {y | sorted y /\ permutation l y}`
  could be the type of the function that takes a reference list as first argument and injects a list and proofs that this list is sorted and a permutation of the reference in the type of sorted permutations of the reference.
- `insert : ∀ x l, sorted l →`
  `{l' | sorted l' /\ permutation (x::l) l'}`
  could be a function that inserts an element in a sorted list, where the type ensures that the result list is sorted and contains the elements of the initial list plus the inserted element.

Using dependent types also imposes that one adapts the constructs in the programming language. For instance, the different values returned for different patterns in a pattern-matching construct may have different types and the pattern-

[1] the notation `a::l` represents the list whose first element and tail are `a` and `l`.

matching construct is adapted to let the programmer indicate how the return type varies depending on the matched pattern. The programming construct takes the following form:

```
match t1 return T t1 with
  p1 => e1
| p2 => e2
end
```

This expression is only well-typed if e_i has the type p_i for the two possible values of i. The text appearing after `return` serves as an annotation of the functional program. For instance, a sorting algorithm could have the following formulation:

```
let rec sort l :
     {l' | sorted l' /\ permutation l l'} :=
match l return sorted l /\ permutation l l' with
     [] => mk_sorted_value [] sorted0 (permutation_refl l [])
| x::l1 => let (l2,hyp_sorted_l2, hyp_perm_l1_l2) := sort l1 in
           let (l', hyp_sorted_l', hyp_perm_x_cons_l2_l') :=
                insert x l2 hyp_sorted_l2 in
              mk_sorted_value (x::l1) l' hyp_sorted_l'
                 (permutation_trans (x::l1) (x::l2) l'
                    (permutation_skip x l1 l2 hyp_perm_l1_l2)
                    hyp_perm_x_cons_l2_l')
end
```

This program contains computation information: the `sort` algorithm decomposes the argument list, sorts recursively the tail and then inserts the first element. The logical information given in the algorithm formulation shows how to combine hypotheses given by auxiliary functions and recursive calls to construct proofs that the result list is sorted and a permutation of the input.

This formulation of the algorithm contains both the description of what needs to be done to compute a sorted list and the justifications for these operation. If this is to be used as input to a compiler, one needs to get rid of all computations that are only relevant for the proofs. This operation is called extraction and it actually produces a new formulation of the algorithm where all logical information is discarded:

```
let rec sort l :=
match l with
       [] => []
| (x::l1) => let l2 := sort l1 in
             let l' := insert x l2 in
             l'
end
```

The corresponding program is close to a regular program in a functional programming language and can be further compiled using a conventional compiler.

It is a matter of taste to decide how much logical information should appear inside the algorithm formulation. A conservative approach is to first describe the algorithm without using dependent types and then to prove its correction. The approach outlined in this example was to show that the logical information can appear in selected places in the algorithm formulation, using auxiliary theorems to condense all reasoning steps. When the function to be considered is partial and checking if an element belongs to the domain of definition is undecidable, there may be no other possibilities than mixing logical information inside the program.

4 Research challenges

4.1 Libraries

The programming language supported by a verifying compiler is only likely to be adopted if it provides enough libraries so that programmers don't have to redo everything from scratch. Today, the programming languages for which complete program verification is possible suffer from the lack of companion libraries.

The work of adding libraries to a verifiable programming language can be done at two levels. The first level consists in weak specification and implementation, but no verification of the library itself. By weak specification, we mean that most of the characteristics of input and output for various procedures are left untold, but the specifications are enough to avoid most practical errors. This level is meaningful for libraries whose content is difficult to describe mathematically (think of the interface to windowing systems or networking tools, for instance).

In the second level for adding libraries to a verifiable programming language, the verifiable compiler itself is used to compile the libraries and ensure that they are correct. In this case, meaningful specifications of the procedures should be provided (for instance, the input to an integer square-root function should be a positive integer and the result is specified as the largest integer value whose square is smaller than the input).

Libraries of reusable components verified at the second level are being developed today, but true reusability has not been effectively assessed. Cases where a new software system is verified and relies on previously verified software units are rare. Intermediate objectives should be set for a variety of domains. In our case, we believe that libraries for continuous mathematics should be developed to make it possible to verify programs in domains that concern the measure and control of physical artifacts. More and more software is developed to control physical devices, in avionics, railroads, or automobiles. Some of this software may have a direct impact on human life, like for instance software used in medical robotics. We believe that this software should be verified with the utmost precision and should rely on libraries that are verified with the best available technology.

4.2 Algorithmic structures and programming constructs

In our study of a wide variety of algorithms, we have found that there are several kind of algorithmic structures: simple loops, structural recursion, well-founded recursion, lazy recursion on potentially infinite data-structures. We know techniques to encode most of these algorithmic structures inside the restricted language of theorem proving systems. However, some of these techniques are especially difficult to implement and sometimes require intensive work from an expert to achieve an encoding that is amenable both to compilation (respecting the intended efficiency) and to formal verification. We believe the required level of expertise can also be lowered by providing proof toolkits that are adapted to these algorithmic structures. For instance, we are currently working on lazy evaluation on infinite data-structures [4].

In general, we believe that each specific domain requires specific methodologies. While an expert can encode most programs in a theorem prover's language by coming back to first principles, it is relevant to capitalize the expertise in a domain specific programming language. In this sense, verifying compilers are already provided in today's world, in the form of specialized programming languages with a well known formal semantics. Examples that come to the authors' mind (because of geographical proximity) are Esterel and Scade [2].

4.3 Software reuse and collaborative work

Specifying the requirements and guarantees for a software component should obviously contribute to better re-usability of compiled code and promote contract-based collaboration on software development. However, our direct experience has shown that some problems need to be overcome to make software verification more productive.

The first point is that the specification of a software fragment should be interchangeable (for the same algorithmic content). Some characteristics of a given algorithm are usually left untold when writing its specification to make it easier to replace this algorithm with another. However, some untold characteristics may have a crucial importance for some larger programs. For instance, sorting algorithms work quite well if the relation with respect to which sorting occurs is not anti-symmetric. However, some algorithms are better than others (even for the same algorithmic complexity), because they may or may not provide the guarantee that an already sorted input will be left unchanged. This shows that a piece of software may have to be recompiled with respect to a modified specification. Verification re-use should be optimized in this case.

A second point concerns maintainability. When new improvements to the algorithms are designed, there is no certainty that verifying the improvement will incur only a marginal cost of verification. Systematic approaches to re-use the proof-work that was performed for a previous version of an algorithm need to be proposed.

5 The wider perspective

The challenge of providing a verifying compiler implies the challenge of designing a new programming language, because verification is more likely to succeed if the semantics of the language are suited for the purpose. Previous successful languages also came with a new field of application: C for system programming, Java for Internet applications, etc. It is sensible that a new important concept like verified software should justify investing in a new language. Past experiments where legacy languages were instrumented to support verification did clarify the situation, but the post-hoc verification of all legacy software is probably out of reach, and future programs should be written in a language that encourages programmers to insert meaningful logical information.

Software formal verification is gradually accepted in the industry today. If a verifying compiler has to become widely accepted, it has to prove its own relevance by bootstrapping. Verifying the compiler itself is also a keypoint for acceptance in the production of safety critical software. In this sense, it is a suitable landmark to provide a formally verified compiler. We have also worked on this topic and we believe this landmark can be reached in a few years.

Another general question is whether one should impose complete verification, as often performed in the theorem proving community, or be content with only partial verification of well-known kinds of properties, hoping to stay in a decidable logic. For instance, array bounds correctness problems are often solved easily with decision procedures for Presburger's arithmetic, and many programs will be verified completely automatically when they do not rely on advanced number theory. This characteristic will be an important one for the acceptance of this approach in a general setting. But seemingly simple programs may sometimes fall outside the area of decidable (or tractable) logic. In the example we studied in [12], array bound correctness actually relies on a complex theorem, stating that there always exists a prime number between n and $2n$. Algorithms that cannot be verified automatically should always be programmable in the language. Fully automatic and partial verification must be accommodated, but it should rather be at the choice of the programmer (for instance by choosing to admit unproven facts) than by restricting the logic.

References

1. Lennart Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.
2. Gerard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
3. Yves Bertot. Formalizing a jvml verifier for initialization in a theorem prover. In *Computer Aided Verification (CAV'2001)*, volume 2102 of *LNCS*, pages 14–24. Springer-Verlag, 2001.
4. Yves Bertot. Filters on coinductive streams, an application to Eratosthenes' sieve. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications, TLCA 2005*, pages 102–115. Springer-Verlag, 2005.

5. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.
6. Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 22(3–4):225–252, 2002.
7. Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *LNCS*, pages 169–184. Springer-Verlag, 2001.
8. Pierre Letouzey and Laurent Théry. Formalizing Stålmarck's algorithm in Coq. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 387–404. Springer-Verlag, 2000.
9. Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5-6):607–640, 1993.
10. David Pichardie and Yves Bertot. Formalizing convex hull algorithms. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *LNCS*, pages 346–361. Springer-Verlag, 2001.
11. Laurent Théry. A machine-checked implementation of Buchberger's algorithm. *Journal of Automated Reasoning*, 26:107–137, 2001.
12. Laurent Théry. Proving pearl: Knuth's algorithm for prime numbers. In David Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *LNCS*. Springer-Verlag, 2003.
13. Howgwei Xi and Frank Pfenning. Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.