A Programmer-Centric Approach to Program Verification in ATS*

Zhiqiang Ren[1] and Hongwei Xi[1]

[1] Boston University, Boston, Massachusetts, U.S.A.
aren@cs.bu.edu
[2] Boston University, Boston, Massachusetts, U.S.A.
hwxi@cs.bu.edu

Abstract

Formal specification is widely employed in the construction of high-quality software. However, there is often a huge gap between formal specification and actual implementation. While there is already a vast body of work on software testing and verification, the task to ensure that an implementation indeed meets its specification is still undeniably of great difficulty. ATS is a programming language equipped with a highly expressive type system that allows the programmer to specify and implement and then verify within the language itself that an implementation meets its specification. In this paper, we present largely through examples a programmer-centric style of program verification that puts emphasis on requesting the programmer to explain in a literate fashion why his or her code works. This is a solid step in the pursuit of software construction that is verifiably correct according to specification.

1 Introduction

In order to be precise in building software systems, we need to specify what such a system is expected to accomplish. In the current day and age, software specification, which we use in a rather loose sense, is often done in forms of varying degree of formalism, ranging from verbal discussions to pencil/paper drawings to various diagrams in modeling languages such as UML [11] to formal specifications in specification languages such as Z [12], etc. Often the main purpose of software specification is to establish a mutual understanding among a team of developers. After the specification for a software system is done, either formally or informally, we need to implement the specification in a programming language. In general, it is exceedingly difficult to be reasonably certain whether an implementation actually meets its specification. Even if the implementation coheres well with its specification initially, it nearly inevitably diverges from the specification as the software system evolves. The dreadful consequences of such a divergence are all too familiar; the specification becomes less and less reliable for understanding the behavior of the software system while the implementation gradually turns into its own specification; for the developers, it becomes increasingly difficult and risky to maintain and extend the software system; for the users, it requires extra amount of time and effort to learn and use the software system.

The design of ATS [15, 16] is partly inspired by Martin-Löf's constructive type theory [10], which was originally developed for the purpose of establishing a foundation for mathematics. Within ATS, there is a static component (statics) and a dynamic component (dynamics). Intuitively, the statics and dynamics are each for handling types and programs, respectively. In particular, specification is done in the statics and implementation in the dynamics. To verify

* This work is partially supported by NSF grants No. CCF-0702665.

$$
\begin{array}{llll}
\text{sorts} & \sigma & ::= & b \mid \sigma_1 \rightarrow \sigma_2 \\
\text{static terms} & s & ::= & a \mid sc[s_1, \ldots, s_n] \mid \lambda a : \sigma.s \mid s_1(s_2) \\
\text{static var. ctx.} & \Sigma & ::= & \emptyset \mid \Sigma, a : \sigma \\
\text{dyn. terms} & d & ::= & x \mid dc(d_1, \ldots, d_n) \mid \mathbf{lam}\, x.d \mid \mathbf{app}(d_1, d_2) \mid \ldots \\
\text{dyn. var. ctx.} & \Delta & ::= & \emptyset \mid \Delta, x : s
\end{array}
$$

Figure 1: Some formal syntax for statics and dynamics of ATS

that an implementation meets a given specification is to show that it is derivable that the program has certain type, which is stated as a guarantee based on types in this paper. In theorem-proving systems such as Coq [13] and NuPrl [3], a specification is encoded as a type; if a proof inhabiting the type is made available, then a program guaranteed to meet the specification can be automatically extracted out of the proof. While the very idea of program extraction is appealing, it is often difficult for the programmer to effectively control the efficiency (time-wise and memory-wise) of extracted programs.

On the other hand, the efficiency of a program written in ATS can rival that of its counterpart in C. Note that this is not achieved by performing aggressive compiler optimizations on the program. Instead, it is primarily due to the support in ATS that allows programs to be constructed by directly following typical C-like programming idioms such as native/unboxed data representation and explict pointer arithmetic. Unsurprisingly, automatically verifying such programs is beyond what we can really hope for at this moment. As an alternative, we expect that the programmer who does the implementation also constructs a proof in the theorem-proving subsystem of ATS to demonstrate the correctness of the implementation. In essence, we advocate a form of program verification that contains both automated and user-assisted components, and we refer to it as a programmer-centric approach to program verification. The primary contribution of the paper lies in our effort identifying such a style of program verification as well as putting it into practice based on ATS.

We organize the rest of the paper as follows. In Section 2, we give a brief overview of ATS. We then present in Section 3 a typical style of program verification in ATS that combines programming with theorem-proving. In Section 4, we employ some examples to illustrate that ATS is well-equipped with features to support program verification that is both flexible and effective for practical use. Lastly, we mention some related work in Section 5 and then conclude.

2 Overview of ATS

We give some formal syntax of ATS in Figure 1. The language ATS has a static component (statics) and a dynamic component (dynamics). The statics includes types, props, and type indexes while the dynamics includes programs and proofs. The statics itself is a simply typed language and a type in it is referred to as a *sort*. For instance, we have the following base sorts in ATS: *addr*, *bool*, *int*, *prop*, *type*, etc; we use L, B and I for static addresses, booleans, and integers of the sorts *addr*, *bool*, and *int*, respectively; we use T for static terms of the sort *type*, which are types assigned to programs; we use P for static terms of the sort *prop*, which are props assigned to proofs.

Types and props may depend on one or more type indexes of static sorts. Among such indexed types, singleton types, which are each a type for only one specific value, are of great use in practical programming. For instance, $\mathbf{bool}(B)$ is a singleton type for the boolean value equal to B, and $\mathbf{int}(I)$ is a singleton type for the integer equal to I, and $\mathbf{ptr}(L)$ is a singleton type for the pointer that points to the address (or location) L. Also, we can quantify over type index variables universally and existentially to form quantified types and props.

We use proving-types of the form $(P \mid T)$ for combining proofs with programs, where P and T stand for a prop and a type, respectively. One may think of the proving-type $(P \mid T)$ as a refinement of the type T because P often constrains some of the indexes appearing in T. For example, the following type:

$$(\mathbf{ADD}(m, n, p) \mid \mathbf{int}(m), \mathbf{int}(n), \mathbf{int}(p))$$

is a proving-type of the sort *type* for a tuple of integers (m, n, p) along with a proof of the prop $\mathbf{ADD}(m, n, p)$ which encodes $m + n = p$. Given a static boolean term B and a type T, we can form two special forms of types: guarded types of the form $B \supset T$ and asserting types of the form $B \wedge T$. Following is an example involving singleton, guarded and asserting types:

$$\forall a : \mathit{int}. a \geq \mathit{0} \supset (\mathbf{int}(a) \rightarrow \exists a' : \mathit{int}.(a' < \mathit{0} \wedge \mathbf{int}(a')))$$

The meaning of this type should be clear: Each value that can be assigned this type represents a function from nonnegative integers to negative integers.

ATS is also equipped with linear version of prop [17], where the word *linear* comes from *linear logic* [7]. Given a type T and a memory location L, a linear prop of the form $T@L$ can be formed to indicate a value of the type T being stored in the memory at the location L, where @ is a special infix operator. Following is a function declaration involving proving-type with linear prop.

```
fun ptr_set {i,j: int} {l:addr}
  (pf: (int i) @ l | p: ptr l, x: int j): ((int j) @ l | void)
```

The meaning of this function should be clear by its type. The implementation of the function should update the content at location l with the input value x.

3 Overview of Program Verification in ATS

We now use a simple example to illustrate the idea of programming with theorem proving. Suppose we want to compute Fibonacci numbers, which are defined inductively as follows:

$$\mathit{fib}(0) = 0 \qquad \mathit{fib}(1) = 1 \qquad \mathit{fib}(n+2) = \mathit{fib}(n) + \mathit{fib}(n+1) \;\text{ for } n >= 0$$

A implementation of *fib* in ATS with O(n) complexity can be done as follows:

```
fun fibats (n: int): int = let
  fun loop (r0: int, r1: int, ni: int): (int) =
    if ni > 0 then loop (r1, r0+r1, ni-1)
    else r0
in loop (0, 1, n) end // end of [fibats]
```

There is obviously a logic gap between the mathematical definition of *fib* and its implementation *fibats* in ATS.[1] In ATS, we can give another implementation (with different type) of *fib* that completely bridges this gap. First, we need a way to encode the definition of *fib* into ATS, which is fulfilled by the declaration of the following dataprop:

[1]We do not address the issue of possible arithmetic overflow here.

```
//
// the syntax [...] is for existential quantification
//
fun fibats2 {n:nat} (n: int n)
  : [r:int] (FIB (n, r) | int r) = let
  fun loop
    {n,i:nat | i <= n} {r0,r1:int} (
    pf0: FIB (i, r0), pf1: FIB (i+1, r1)
  | r0: int (r0), r1: int (r1), ni: int(n-i)
  ) : [r:int] (FIB (n, r) | int (r)) =
  if ni > 0 then
    loop {n,i+1} (pf1, FIB2 (pf0, pf1) | r1, r0+r1, ni-1)
  else (pf0 | r0)
in
  loop (FIB0(), FIB1() | 0, 1, n)
end // end of [fibats2]
```

Figure 2: A verified implementation of *fib* in ATS

```
dataprop FIB (int, int) =
  | FIB0 (0, 0) | FIB1 (1, 1)
  | {n:nat} {r0,r1:int}
    FIB2 (n+2, r0+r1) of (FIB (n, r0), FIB (n+1, r1))
// end of [FIB]
```

where the concrete syntax `{...}` is for universal quantification in ATS. This declaration introduces a type (or more precisely, a type constructor) **FIB** for proofs. Such a type is referred to as a prop (or prop-type) in ATS. Intuitively, if a proof can be assigned the type $\mathbf{FIB}(n, r)$ for some integers n and r, then $\mathit{fib}(n)$ equals r. In other words, $\mathbf{FIB}(n, r)$ encodes the relation $\mathit{fib}(n) = r$ inductively through *FIB0*, *FIB1*, and *FIB2*, three constructors associated with **FIB**. These constructors can be given the following types corresponding to the three equations in the definition of *fib*:

$$\begin{array}{lcl} \mathit{FIB0} & : & () \rightarrow \mathbf{FIB}(0,0) \\ \mathit{FIB1} & : & () \rightarrow \mathbf{FIB}(1,1) \\ \mathit{FIB2} & : & \forall n : \mathit{nat}.\forall r_0 : \mathit{int}.\forall r_1 : \mathit{int}. \\ & & \quad (\mathbf{FIB}(n, r_0), \mathbf{FIB}(n+1, r_1)) \rightarrow \mathbf{FIB}(n+2, r_0 + r_1) \end{array}$$

For instance, $\mathit{FIB2}(\mathit{FIB0}(), \mathit{FIB1}())$ is a term of the type $\mathbf{FIB}(2, 1)$, attesting to $\mathit{fib}(2) = 1$. In Figure 2, the implemented function *fibats*2 is assigned the following type:

$$\mathit{fibats}2 \quad : \quad \forall n : \mathit{nat}.\ \ \mathbf{int}(n) \rightarrow \exists r : \mathit{int}.(\mathbf{FIB}(n, r) \mid \mathbf{int}(r))$$

where $\mid$ is just a separator (like a comma) for separating a proof from a value. For each integer value I, $\mathbf{int}(I)$ is a singleton type for the only integer whose value is I. When *fibats*2 is applied to an integer of value n, it returns a pair consisting of a proof and an integer of value r such that the proof, which is of the type $\mathbf{FIB}(n, r)$, asserts $\mathit{fib}(n) = r$. Therefore, *fibats*2 is a verified implementation of *fib*. We emphasize that proofs are completely erased after typechecking. In particular, there is no proof construction at run-time.

4 Programmer-Centric Verification

By programmer-centric verification, we mean a verification approach that puts the programmer at the center of the verification process. The programmer is expected to explain in a literate fashion why his or her implementation meets a given specification. The programmer may rely on external knowledge when doing verification, but such knowledge should be expressed in a format that is accessible to other programmers. We will employ some examples in this section to elaborate on programmer-centric verification.

```
//
// list(a, n) is the type for a list of length n
// in which each element is of the type a.
//
fun{a:type} insort {n:nat}
  (xs: list (a, n), lte: (a, a) -> bool): list (a, n) = let
  fun ins {n:nat}
    (x: a, xs: list (a, n), lte: (a, a) -> bool): list (a, n+1) =
    case xs of
    | list_cons (x1, xs1) =>
        if lte (x, x1) then
          list_cons (x, xs) else list_cons (x1, ins (x, xs1, lte))
        // end of [if]
    | list_nil () => list_cons (x, list_nil ())
  // end of [ins]
in
  case xs of
  | list_cons (x, xs1) => ins (x, insort (xs1, lte), lte)
  | list_nil () => list_nil ()
end // end of [insort]
```

Figure 3: A standard implementation of insertion sort

4.1 Example: Insertion Sort on Generic Lists

In Figure 3, we give a standard implementation of insertion sort written in ATS that takes a generic list and a comparison function and returns a generic list that is sorted according to the comparison function. Note that the use of generic lists clearly indicates our strive for practicality. In the literature, a similar presentation would often use integer lists (instead of generic lists), revealing the difficulty in handling polymorphism and thus weakening the argument for practical use of verification. We have no such difficulty. The implementation we present guarantees based on the types that the output list is of the same length as the input list. We also give a verified implementation of insertion sort in Figure 4 that guarantees based on the types that the output list is a sorted permutation of the input list. The fact that this verified implementation can be done in such a concise manner should yield strong support for the underlying verification approach. Note that we do not include the proof of termination, which can be verified in ATS [14], in Figure 4 for the sake of brevity,

Suppose that a programmer did the implementation in Figure 3. Obviously, the programmer did not do the implementation in a random fashion; he or she did it based on some kind of (informal) logic reasoning. We will see that ATS provides programming features such as abstract

```
fun{a:type} insort
  {xs:ilist} (xs: glist (a, xs), lte: lte(a))
  : [ys:ilist] (SORT (xs, ys) | glist (a, ys)) = let
  fun ins {x:int} {ys1:ilist} (
    pford: ORD (ys1) |
    x: E (a, x), ys1: glist (a, ys1), lte: lte(a)
  ) : [ys2:ilist] (SORT (cons (x, ys1), ys2) | glist (a, ys2)) =
    case ys1 of
    | glist_cons (y1, ys10) =>
        if lte (x, y1) then let
          prval pford = ORD_ins {x} (pford)
          prval pfperm = PERM_refl ()
          prval pfsrt = ORDPERM2SORT (pford, pfperm)
        in
          (pfsrt | cons (x, ys1))
        end else let
          prval pford1 = ORD_tail (pford)
          val (pfsrt1 | ys20) = ins (pford1 | x, ys10, lte)
          prval pfsrt2 = SORT_ins {x} (pford, pfsrt1)
        in
          (pfsrt2 | cons (y1, ys20))
        end // end of [if]
    | glist_nil () => (SORT_sing () | cons (x, nil ()))
  // end of [ins]
in
  case xs of
  | glist_cons (x, xs1) => let
      val (pfsrt1 | ys1) = insort (xs1, lte)
      prval pford1 = SORT2ORD (pfsrt1)
      prval pfperm1 = SORT2PERM (pfsrt1)
      prval pfperm1_cons = PERM_cons (pfperm1)
      val (pfsrt2 | ys2) = ins (pford1 | x, ys1, lte)
      prval pford2 = SORT2ORD (pfsrt2)
      prval pfperm2 = SORT2PERM (pfsrt2)
      prval pfperm3 = PERM_tran (pfperm1_cons, pfperm2)
      prval pfsrt3 = ORDPERM2SORT (pford2, pfperm3)
    in
      (pfsrt3 | ys2)
    end // end of [intlist_cons]
  | glist_nil () => (SORT_nil () | nil ())
end // end of [insort]
```

Figure 4: A verified implementation of insertion sort

props and external lemmas for turning such informal reasoning into formal verification. In particular, we can turn the implementation of insertion sort in Figure 3 into the verified one in Figure 4 by following a verification process.

```
abstype E (a:type, x:int) // abstract type constructor
datasort ilist = ilist_nil of () | ilist_cons of (int, ilist)
datatype glist (a:type, ilist) =
  | {x:int} {xs:ilist}
    glist_cons (a, cons (x, xs)) of (E (a, x), glist (a, xs))
  | glist_nil (a, nil) of ()
```

Figure 5: A generic list type indexed by the names of list elements

First, we need to map the elements in the dynamics to be sorted to appropriate terms in the statics, which can be reasoned about within the theorem-proving subsystem of ATS. This can be achieved by introducing an abstract type constructor E (in Figure 5). Given a type T and an integer I, $\mathbf{E}(T, I)$ is a *singleton* type for a value of the type T with an (imaginary) integer name I. Second, to reason about the ordering of integer sequence, we have to give its definition formally in the form of a new sort in the statics. In ATS, the user-defined sorts (datasorts) can be introduced in a manner similar to the introduction of user-defined types (datatypes) in a ML-like language. We introduce a datasort *ilist* for representing sequences of (static) integers. We may simply write *nil* and *cons* for *ilist_nil* and *ilist_cons*, respectively, if there is no potential confusion. Note that there is no mechanism for defining recursive functions in the statics, and this is a profound restriction that give rise to a unique style of verification in ATS. We lastly define a datatype **glist**: Given a list of values of types $\mathbf{E}(T, I_1), \ldots, \mathbf{E}(T, I_n)$, the type $\mathbf{glist}(T, \mathit{cons}(I_1, \ldots, \mathit{cons}(I_n, \mathit{nil})))$ can be assigned to this particular list. We may also simply write *nil* and *cons* for *glist_nil* and *glist_cons*, respectively, if there is no potential confusion. Please note that **glist** is in the dynamics while *ilist* is in the statics. With the aforementioned setting, we can verify the properties of an instance of **glist** by reasoning about its correspondence in the statics, which is of sort *ilist*.

To verify insertion sort, we first introduce an abstract prop as follows such that $\mathbf{SORT}(xs, ys)$ means that ys is a sorted permutation of xs:

```
absprop SORT (xs:ilist, ys:ilist)
```

Let $\mathbf{lte}(a)$ be a shorthand for the following type:

$$\forall a : \mathit{type}.\forall x_1 : \mathit{int}.\forall x_2 : \mathit{int}.(\mathbf{E}(a, x_1), \mathbf{E}(a, x_2)) \rightarrow \mathbf{bool}(x_1 \leq x_2)$$

If we can assign the following type to *insort*:

$$\begin{array}{l}\forall a : \mathit{type}.\forall xs : \mathit{ilist}.\\ \quad (\mathbf{glist}(a, xs), \mathbf{lte}(a)) \rightarrow \exists ys : \mathit{ilist}.(\mathbf{SORT}(xs, ys) \mid \mathbf{glist}(a, ys))\end{array}$$

then *insort* is verified as the type simply states that the output list is a sorted permutation of the input list.

For the purpose of verification, we also introduce the following two abstract props:

```
absprop ORD (xs:ilist)
absprop PERM (xs:ilist, ys:ilist)
```

SORT2ORD : $\forall xs : \mathit{ilist}. \forall ys : \mathit{ilist}.\ \mathbf{SORT}(xs, ys) \to \mathbf{ORD}(ys)$
- If ys is a sorted version of xs, then ys is ordered.

SORT2PERM : $\forall xs : \mathit{ilist}. \forall ys : \mathit{ilist}.\ \mathbf{SORT}(xs, ys) \to \mathbf{PERM}(xs, ys)$
- If ys is a sorted version of xs, then ys is a permutation of xs.

ORDPERM2SORT : $\forall xs : \mathit{ilist}. \forall ys : \mathit{ilist}.$
$(\mathbf{ORD}(ys), \mathbf{PERM}(xs, ys)) \to \mathbf{SORT}(xs, ys)$
- If ys is ordered and is also a permutation of xs, then ys is a sorted version of xs.

SORT_nil : $() \to \mathbf{SORT}(\mathit{nil}, \mathit{nil})$
- The empty list is a sorted version of itself.

SORT_sing : $\forall x : \mathit{int}.\ () \to \mathbf{SORT}(\mathit{cons}(x, \mathit{nil}), \mathit{cons}(x, \mathit{nil}))$
- A singleton list is a sorted version of itself.

ORD_tail : $\forall y : \mathit{int}. \forall ys : \mathit{ilist}.\ \mathbf{ORD}(\mathit{cons}(y, ys)) \to \mathbf{ORD}(ys)$
- If a non-empty list is ordered, then its tail is also ordered.

ORD_ins : $\forall x : \mathit{int}. \forall y : \mathit{int}. \forall ys : \mathit{ilist}.\ x \leq y \supset$
$\mathbf{ORD}(\mathit{cons}(y, ys)) \to \mathbf{ORD}(\mathit{cons}(x, \mathit{cons}(y, ys)))$
- If $x \leq y$ holds and $\mathit{cons}(y, ys)$ is ordered, then $\mathit{cons}(x, \mathit{cons}(y, ys))$ is also ordered.

PERM_refl : $\forall xs : \mathit{ilist}.\ () \to \mathbf{PERM}(xs, xs)$
- Each list is a permutation of itself.

PERM_tran : $\forall xs : \mathit{ilist}. \forall ys : \mathit{ilist}. \forall zs : \mathit{ilist}.$
$(\mathbf{PERM}(xs, ys), \mathbf{PERM}(ys, zs)) \to \mathbf{PERM}(xs, zs)$
- The permutation relation is transitive.

PERM_cons : $\forall x : \mathit{int}. \forall xs_1 : \mathit{ilist}. \forall xs_2 : \mathit{ilist}.$
$\mathbf{PERM}(xs_1, xs_2) \to \mathbf{PERM}(\mathit{cons}(x, xs_1), \mathit{cons}(x, xs_2))$
- If xs_2 is a permutation of xs_1, then $\mathit{cons}(x, xs_2)$ is a permutation of $\mathit{cons}(x, xs_1)$.

SORT_ins : $\forall x : \mathit{int}. \forall y : \mathit{int}. \forall ys_1 : \mathit{ilist}. \forall ys_2 : \mathit{ilist}.\ x > y \supset$
$(\mathbf{ORD}(\mathit{cons}(y, ys_1)), \mathbf{SORT}(\mathit{cons}(x, ys_1), ys_2)) \to$
$\mathbf{SORT}(\mathit{cons}(x, \mathit{cons}(y, ys_1)), \mathit{cons}(y, ys_2))$
- If $x > y$ holds, $\mathit{cons}(y, ys_1)$ is ordered and ys_2 is a sorted version of $\mathit{cons}(x, ys_1)$, then $\mathit{cons}(y, ys_2)$ is a sorted version of $\mathit{cons}(x, \mathit{cons}(y, ys_1))$.

Figure 6: Some external lemmas needed for verifying insertion sort

Given xs and ys, $\mathbf{ORD}(xs)$ means that xs is ordered according to the ordering $\leq$ on integers and $\mathbf{PERM}(xs, ys)$ means that ys is a permutation of xs.

When verifying *insort*, we essentially try to justify each step in the code presented in Figure 3. This justification process may introduce various statements about the properties of those concepts **SORT**, **ORD**, and **PERM**. (We call such statements lemmas to indicate that their validity is now only justified by the programmers' reasoning informally.) For instance, the code presented in Figure 4 makes use of the lemmas listed in Figure 6.

For soundness, we need to define **SORT**, **ORD**, and **PERM** explicitly, based on which we can prove these lemmas formally. And we can indeed do this in the theorem-proving subsystem of ATS. However, this style of verifying everything from basic definitions can be too great a burden in practice. Suppose that we try to construct a mathematical proof and we need to make use of the proposition in the proof that the standard permutation relation is transitive. It is unlikely that we provide an explicit proof for this proposition as *it sounds so evident to us*. To put it from a different angle, if constructing mathematical proofs required that every single detail be presented explicitly, then studying mathematics would unlikely to be feasible. Therefore, we strongly advocate a style of theorem-proving in ATS that models the way we do mathematics.

The implementation of insertion sort on generic lists in Figure 3, which can be obtained from erasing proofs in Figure 4, is guaranteed to be correct if all of the lemmas in Figure 6 are true. It is probably fair to say that these lemmas are all evidently true except the last one: *SORT_ins*. If we are unsure whether the lemma *SORT_ins* is true or not, we can construct a proof in ATS or elsewhere to validate it. For instance, we can even give an informal proof as follows: Note that $\mathbf{PERM}(cons(x, ys_1), ys_2)$ holds as ys_2 is a sorted version of $cons(x, ys_1)$. Hence, $cons(y, ys_2)$ is a permutation of $cons(x, cons(y, ys_1))$. Since $cons(y, ys_1)$ is ordered, y is a lower bound for the elements in ys_1. Hence, y is a lower bound for elements in ys_2 as $x > y$ holds, and thus, $cons(y, ys_2)$ is ordered. Therefore, $cons(y, ys_2)$ is a sorted version of $cons(x, cons(y, ys_1))$.

What is of crucial importance is that *SORT_ins* is a lemma that is *manually* introduced and can be readily understood by any programmer with adequate training. This is a direct consequence of programmer-centric verification in which the programmer explains in a literate fashion why his or her implementation meets a given specification.

To sum up, we have proven that the integer sequence corresponding to the output generic list is a sorted permutation of the integer sequence corresponding to the input generic list. Besides, the type of the comparison function **lte** states that the mapping from element in the dynamics to integer in the statics preserves ordering relation. Therefore, it is verified that the implementation is correct given that **lte** is implemented correctly. The benefit that we only need to reason about integer sequence regardless of the real type of the list to be sorted comes from ATS' feature of separation of dynamics and statics. Going still further, we can map an array of elements in the dynamics to integer sequence in the statics via linear prop and *addr*. And there is really not much difference between lists and arrays as far as verification is concerned. The reason that we use lists instead of arrays is for simplifying the presentation and also that array based insertion sort has no advantage of efficiency.

4.2 Many Other Examples

In Appendix A, we give out an example of quicksort following the same paradigm of programmer-centric verification. Besides, there are also a variety of examples available on-line[2] which can further illustrate a style of programmer-centric verification in ATS that combines programming with theorem-proving cohesively. In particular, there are examples involving arrays, heaps, balanced trees, etc.

5 Related Work and Conclusion

Given the vastness of the field of program verification, we can only mention some closely related work in this section.

Ynot [2] is an axiomatic extension of the Coq proof assistant for specifying and verifying properties of imperative programs. The programmer can encode a new domain by providing

[2]Please see `http://www.ats-lang.org/EXAMPLE/PCPV`

key lemmas in an ML-like embedded language. Relying on Coq to do theorem-proving, Ynot mixes the automated proof generation with manual proof construction, attempting to relieve the programmer from the heavy burden that would otherwise be necessary. The dependent types employed in Ynot is different from that of ATS. In particular, there is no separation of statics from dynamics in Ynot.

Krakatoa [9] is a front-end of the Why [1] platform for deductive program verification. It deals with Java programs annotated in a variant of the the Java Modeling Language (JML) called KML. An extension to KML is proposed in [6] to support the specification of generic Java code. It relies on parameterized theory for specification, and performes verification after the the theory is instantiated with specific types. If we think of types in ATS just as special annotations, then Krakatoa and ATS share a similar flavor of program verification. However, there is also fundamental difference between Krakatoa and ATS. In ATS, proofs are constructed during the same time when an implementation is written. In Krakatoa, an existing implementation needs to be properly annotated so that proof obligations generated during verification can be discharged (by Why).

The work on extended static checking (ESC) [4] also puts emphasis on employing formal annotations to capture program invariants. These invariants may be verified through (light-weighted) theorem proving. ESC/Java [5] generates verification-conditions based on annotated Java code and uses an automatic theorem-prover to reason about the semantics of the programs. It can catch many basic errors such as null dereferences, array bounds errors, type cast errors, etc. With more emphasis on usefulness, soundness is sacrificed in certain cases to reduce annotation cost or to improve checking speed.

VeriFast [8] is another system for verifying program properties through source code annotation. It supports direct insertion of simple proof steps into the source code while allowing rich and complex properties to be specified through inductive datatypes and fixed-point functions. VeriFast provides a program verifier for C and Java that supports interactive insertion of annotations into source code to alleviate the task of automatic reasoning about separation logic.

The paradigm of programming with theorem-proving as is supported in the ATS programming language system is fundamentally different from program extraction (from proofs) as is supported in theorem-proving systems such as Coq. Note that ATS is a full-fledged programming language that supports the construction of highly efficient programs (whose efficiency rivals that of their counterparts in C), and the presented approach to verification can also be applied to effectful programs written in imperative style (see some of the on-line examples).

In this paper, we have argued in support of a style of program verification that puts emphasis on requesting the programmer to formally explain in a literate fashion why the code he or she implements actually meets its specification. Though external lemmas introduced during a verification process can be discharged by formally proving them in ATS, doing so is often expensive in terms of effort and time. One possibility is to characterize such lemmas into different categories and then employ (external) specialized theorem-provers to prove them. Another possibility for discharging lemmas, which we strongly advocate, is to go through a peer-review process, which mimics the practice of (informally) verifying mathematical proofs. Obviously, the precondition for such an approach is that the lemmas to be verified can be expressed in a format that is easily accessible to a (trained) human being. This is where the programmer-centric verification as is presented in this paper can fit very well.

References

[1] The Why Verification Tool. `http://why.lri.fr/`.

[2] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective Interactive Proofs for Higher-Order Imperative Programs. *SIGPLAN Not.*, 44(9):79–90, 2009.

[3] Robert L. Constable, Stuart F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, Scott F. Smith, James T. Sasaki, and S. F. Smith. Implementing Mathematics with The Nuprl Proof Development System. 1986.

[4] David L. Detlefs. An Overview of the Extended Static Checking System. In *In Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, 1995.

[5] Cormac Flanagan, Rustan, K. Rustan, M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java, 2002.

[6] Alain Giorgetti, Claude Marché, Elena Tushkanova, and Olga Kouchnarenko. Specifying Generic Java Programs: Two Case Studies. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, LDTA '10, New York, NY, USA, 2010. ACM.

[7] Jean-Yves Girard, Yves Lafon, and Paul Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, April 1989.

[8] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: a powerful, sound, predictable, fast verifier for C and java. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] C. Marche, Paulin C. Mohring, and X. Urbain. The Krakatoa Tool for Certification of Java/JavaCard Programs Annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

[10] Per Martin-Löf. Constructive Mathematics and Computer Programming. In *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 167–184, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.

[11] Manny Rayner, Beth A. Hockey, Nikos Chatzichrisafis, and Kim Farrell. OMG Unified Modeling Language Specification. In *Version 1.3, © 1999 Object Management Group, Inc*, 2005.

[12] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, second edition, 1992.

[13] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2004. Available at `http://coq.inria.fr/refman/`.

[14] Hongwei Xi. Dependent Types for Program Termination Verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, pages 231–242, Boston, June 2001.

[15] Hongwei Xi. Applied Type System (extended abstract. In *In post-workshop Proceedings of TYPES 2003*, pages 394–408, 2004.

[16] Hongwei Xi. The ATS Programming Language System. Available at `http://www.ats-lang.org/`, 2008.

[17] Dengping Zhu and Hongwei Xi. Safe Programming with Pointers through Stateful Views. In *In Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, pages 83–97, 2005.

A Example: Quicksort on Generic Lists

We give a standard implementation of quicksort on generic lists in Figure 7. The reason that we use lists instead of arrays is solely for simplifying the presentation. As far as verification is concerned, there is really not much difference between lists and arrays. Note that we have already made various verification examples available on-line that involve arrays.

```
fun{a:type}
qsrt {n:nat}
  (xs: list (a, n), lte: lte a) : list (a, n) =
  case+ xs of  // case+ indicates to the compiler that the pattern match must be exhaustive.
  | list_cons (x, xs) => part (x, xs, lte, list_nil (), list_nil ())
  | list_nil () => list_nil ()
and part {p:nat} {q,r:nat} (
  x0: a, xs: list (a, p), lte: lte(a), ys: list (a, q), zs: list (a, r)
) : list (a, p+q+r+1) =
  case+ xs of
  | list_cons (x, xs) =>
      if lte (x, x0) then
        part (x0, xs, lte, list_cons (x, ys), zs)
      else
        part (x0, xs, lte, ys, list_cons (x, zs))
      // end of [if]
  | list_nil () => let
      val ys = qsrt (ys, lte) and zs = qsrt (zs, lte)
    in
      append (ys, list_cons (x0, zs))
    end // end of [list_nil]
```

Figure 7: A standard implementation of quicksort on generic list

The implementation in Figure 7 guarantees based on the types that the output list is of the same length as the input list. We also give a verified implementation of quicksort in Figure 8 that guarantees based on the types that the output list is a sorted permutation of the input list. The verified implementation is essentially obtained from the process to explain why the function *qsrt* in Figure 7 always returns a list that is the sorted version of the input list.

We now explain that the verified implementation of quicksort can be trusted. The function *append* in the implementation is given the following type:

$$\begin{array}{l}\forall a : \mathit{type}.\forall xs_1 : \mathit{ilist}.\forall xs_2 : \mathit{ilist}.\\ \ (\mathbf{glist}(a, xs_1), \mathbf{glist}(a, xs_2)) \rightarrow \\ \ \exists res : \mathit{ilist}.(\mathbf{APPEND}(xs_1, xs_2, res) \mid \mathbf{glist}(a, res))\end{array}$$

where **APPEND** is an abstract prop. Given lists xs_1, xs_2, and res, the intended meaning of $\mathbf{APPEND}(xs_1, xs_2, res)$ is obvious: it states that the concatenation of xs_1 and xs_2 is res. Both **LB** and **UB** are introduced as abstract props: $\mathbf{LB}(x, xs)/\mathbf{UB}(x, xs)$ means that x is a lower/upper bound for the elements in xs. Another introduced abstract prop is **UNION4**: Given x, xs, ys, zs, and res, $\mathbf{UNION4}(x, xs, ys, zs, res)$ means that the following equation holds

$$|res| = \{x\} \cup |xs| \cup |ys| \cup |zs|$$

where $|\cdot|$ turns an integer list into a multiset. The external lemmas used in Figure 8 are listed in Figure 9 with corresponding explanation.

```
fun{a:type}
qsrt {xs:ilist} (
  xs: glist (a, xs), lte: lte a
) : [ys:ilist] (SORT (xs, ys) | glist (a, ys)) =
  case+ xs of
  | glist_cons (x, xs) => let
      val (pford, pfuni | res) =
        part (UB_nil (), LB_nil () | x, xs, lte, nil (), nil ())
      prval pfperm = UNION4_perm (pfuni)
    in
      (ORDPERM2SORT (pford, pfperm) | res)
    end
  | glist_nil () => (SORT_nil () | nil ())

and part
  {x0:int} {xs:ilist} {ys,zs:ilist} (
  pf1: UB (x0, ys), pf2: LB (x0, zs)
| x0: E (a, x0), xs: glist (a, xs), lte: lte(a)
, ys: glist (a, ys), zs: glist (a, zs)
) : [res:ilist] (
  ORD (res), UNION4 (x0, xs, ys, zs, res) | glist (a, res)
) =
  case+ xs of
  | glist_cons (x, xs) =>
      if lte (x, x0) then let
        prval pf1 = UB_cons (pf1)
        val (pford, pfuni | res) =
          part (pf1, pf2 | x0, xs, lte, cons (x, ys), zs)
        prval pfuni = UNION4_mov1 (pfuni)
      in
        (pford, pfuni | res)
      end else let
        prval pf2 = LB_cons (pf2)
        val (pford, pfuni | res) =
          part (pf1, pf2 | x0, xs, lte, ys, cons (x, zs))
        prval pfuni = UNION4_mov2 (pfuni)
      in
        (pford, pfuni | res)
      end // end of [if]
  | glist_nil () => let
      val (pfsrt1 | ys) = qsrt (ys, lte)
      val (pfsrt2 | zs) = qsrt (zs, lte)
      val (pfapp | res) = append (ys, cons (x0, zs))
      prval pford1 = SORT2ORD (pfsrt1)
      prval pford2 = SORT2ORD (pfsrt2)
      prval pfperm1 = SORT2PERM (pfsrt1)
      prval pfperm2 = SORT2PERM (pfsrt2)
      prval pf1 = UB_perm (pfperm1, pf1)
      prval pf2 = LB_perm (pfperm2, pf2)
      prval pford = APPEND_ord (pf1, pf2, pford1, pford2, pfapp)
      prval pfuni = APPEND_union4 (pfperm1, pfperm2, pfapp)
    in
      (pford, pfuni | res)
    end // end of [glist_nil]
// end of [part]
```

Figure 8: A verified implementation of quicksort

LB_nil : $\forall x : \mathit{int}.\ () \rightarrow \mathbf{LB}(x, \mathit{nil})$
- Each integer is a lower bound for the empty list.

UB_nil : $\forall x : \mathit{int}.\ () \rightarrow \mathbf{UB}(x, \mathit{nil})$
- Each integer is an upper bound for the empty list.

LB_cons : $\forall x_0 : \mathit{int}.\forall x : \mathit{int}.\forall xs : \mathit{ilist}.\ x_0 \leq x \supset$
$\mathbf{LB}(x_0, xs) \rightarrow \mathbf{LB}(x_0, \mathit{cons}(x, xs))$
- If $x_0 \leq x$ holds and x_0 is a lower bound for xs, then x_0 is also a lower bound for $\mathit{cons}(x, xs)$.

UB_cons : $\forall x_0 : \mathit{int}.\forall x : \mathit{int}.\forall xs : \mathit{ilist}.\ x_0 \geq x \supset$
$\mathbf{UB}(x_0, xs) \rightarrow \mathbf{UB}(x_0, \mathit{cons}(x, xs))$
- If $x_0 \geq x$ holds and x_0 is an upper bound for xs, then x_0 is also an upper bound for $\mathit{cons}(x, xs)$.

LB_perm : $\forall x : \mathit{int}.\forall xs_1 : \mathit{ilist}.\forall xs_2 : \mathit{ilist}.$
$(\mathbf{PERM}(xs_1, xs_2), \mathbf{LB}(x, xs_1)) \rightarrow \mathbf{LB}(x, xs_2)$
- If x is a lower bound for xs_1 and xs_1 is a permutation of xs_2, then x is also a lower bound for xs_2.

UB_perm : $\forall x : \mathit{int}.\forall xs_1 : \mathit{ilist}.\forall xs_2 : \mathit{ilist}.$
$(\mathbf{PERM}(xs_1, xs_2), \mathbf{UB}(x, xs_1)) \rightarrow \mathbf{UB}(x, xs_2)$
- If x is an upper bound for xs_1 and xs_1 is a permutation of xs_2, then x is also a upper bound for xs_2.

UNION4_perm : $\forall x : \mathit{int}.\forall xs : \mathit{ilist}.\forall res : \mathit{ilist}.$
$\mathbf{UNION4}(x, xs, \mathit{nil}, \mathit{nil}, res) \rightarrow \mathbf{PERM}(\mathit{cons}(x, xs), res)$
- If $|res| = \{x\} \cup |xs|$, then res is a permutation of $\mathit{cons}(x, xs)$.

UNION4_mov1 : $\forall x_0 : \mathit{int}.\forall x : \mathit{int}.\forall xs : \mathit{ilist}.\forall ys : \mathit{ilist}.\forall zs : \mathit{ilist}.\forall res : \mathit{ilist}.$
$\mathbf{UNION4}(x0, xs, \mathit{cons}(x, ys), zs, res) \rightarrow$
$\mathbf{UNION4}(x0, \mathit{cons}(x, xs), ys, zs, res)$
- If $|res| = \{x_0\} \cup |xs| \cup |\mathit{cons}(x, ys)| \cup |zs|$, then $|res| = \{x_0\} \cup |\mathit{cons}(x, xs)| \cup |ys| \cup |zs|$.

UNION4_mov2 : $\forall x_0 : \mathit{int}.\forall x : \mathit{int}.\forall xs : \mathit{ilist}.\forall ys : \mathit{ilist}.\forall zs : \mathit{ilist}.\forall res : \mathit{ilist}.$
$\mathbf{UNION4}(x0, xs, ys, \mathit{cons}(x, zs), res) \rightarrow$
$\mathbf{UNION4}(x0, \mathit{cons}(x, xs), ys, zs, res)$
- If $|res| = \{x_0\} \cup |xs| \cup |ys| \cup |\mathit{cons}(x, zs)|$, then $|res| = \{x_0\} \cup |\mathit{cons}(x, xs)| \cup |ys| \cup |zs|$.

APPEND_ord : $\forall x : \mathit{int}.\forall ys : \mathit{ilist}.\forall zs : \mathit{ilist}.\forall res : \mathit{ilist}.$
$(\mathbf{UB}(x, ys), \mathbf{LB}(x, zs), \mathbf{ORD}(ys), \mathbf{ORD}(zs),$
$\mathbf{APPEND}(ys, \mathit{cons}(x, zs), res)) \rightarrow \mathbf{ORD}(res)$
- If x is an upper bound for ys and a lower bound for zs, both ys and zs are ordered and res is the concatenation of ys and $\mathit{cons}(x, zs)$, then res is ordered.

APPEND_union4 : $\forall x : \mathit{int}.\forall ys : \mathit{ilist}.\forall ys_1 : \mathit{ilist}.\forall zs : \mathit{ilist}.\forall zs_1 : \mathit{ilist}.\forall res : \mathit{ilist}.$
$(\mathbf{PERM}(ys, ys_1), \mathbf{PERM}(zs, zs_1),$
$\mathbf{APPEND}(ys_1, \mathit{cons}(x, zs_1), res)) \rightarrow$
$\mathbf{UNION4}(x, \mathit{nil}, ys, zs, res)$
- If ys_1 is a permutation of ys, zs_1 is a permutation of zs and res is the concatenation of ys_1 and $\mathit{cons}(x, zs_1)$, then $|res| = \{x\} \cup |ys| \cup |zs|$.

Figure 9: Some external lemmas needed for verifying quicksort