Lightweight Formal Verification in Classroom Instruction of Reasoning about Functional Code

Andrei Lapets

November 6, 2009

Abstract

In college courses dealing with material that requires mathematical rigor, the adoption of a machine-readable representation for formal arguments can be advantageous. Students can focus on a specific collection of constructs that are represented consistently. Examples and counterexamples can be evaluated. Assignments can be assembled and checked with the help of an automated formal reasoning system. However, usability and accessibility do not have a high priority and are not addressed sufficiently well in the design of many existing machine-readable representations and corresponding formal reasoning systems. In earlier work [Lap09], we attempt to address this broad problem by proposing several specific design criteria organized around the notion of a natural context: the sphere of awareness a working human user maintains of the relevant constructs, arguments, experiences, and background materials necessary to accomplish the task at hand. We report on our attempt to evaluate our proposed design criteria by deploying within the classroom a lightweight formal verification system designed according to these criteria. The lightweight formal verification system was used within the instruction of a common application of formal reasoning: proving by induction formal propositions about functional code. We present all of the formal reasoning examples and assignments considered during this deployment, most of which are drawn directly from an introductory text on functional programming. We demonstrate how the design of the system improves the effectiveness and understandability of the examples, and how it aids in the instruction of basic formal reasoning techniques. We make brief remarks about the practical and administrative implications of the system's design from the perspectives of the student, the instructor, and the grader.

1 Introduction

In the classroom instruction of topics that require mathematical rigor, there exist many benefits to adopting a formal representation. These include reusability, automatic evaluation of examples, and the opportunity to employ formal reasoning systems. Such systems can offer anything from detection of basic errors, such as unbound variables and type mismatches, to full confidence in an argument because it is verifiably constructed using the fundamental principles of a mathematical logic. There exist a variety of such systems, and many of these have been surveyed and compared along a variety of dimensions [Wie03]. However, to date, broad accessibility and intuitive interface design has not been a priority in the design of formal reasoning systems that can provide these benefits. On the contrary, instructors hoping to employ a formal reasoning system in the classroom are presented with a variety of obstacles, both superficial and fundamental. Consequently, many instructors today continue to ignore such systems. In the literature in most domains of computer science and mathematics there are only isolated attempts to include machine-verified proofs of novel research results, and in only a few mathematics and computer science courses are such systems employed for presenting material or authoring solutions to assignments.

To address the aforementioned issues, we have proposed in earlier work [Lap09] principles for the design of formal reasoning systems capable of providing automatic verification and assistance to users authoring formal arguments. We argue that by following these principles it is possible to assemble formal reasoning systems that have better accessibility and useability. These principles require the adoption of what we consider *lightweight* design. In this context, lightweight design is not exclusively a passive principle that consists of loosening restrictions. A system that is lightweight from the user's perspective might also provide robust and natural interfaces for search and interaction, might give the user many degrees of freedom in deciding how precisely or extensively arguments are verified, and might actively anticipate and tighten what the user is trying to do using appropriate heuristics. This strategy is based on our own understanding and experience as well as the observations that motivated other recent projects with similar goals. In particular, the need for natural interfaces (both superficial and functional) and the value of appropriate heuristics have been recognized to varying degrees by the designers of the `Tutch` proof checker [ACP01], the Scunak mathematical assistant system [Bro06], the ForTheL language and SAD proof assistant [VLPA08], the EPGY Theorem-Proving Environment [MRS01], the ΩMEGA proof verifier [SBF$^+$02], and in the work of Sieg and Cittadini [SC05]. One author, commenting on designing an interface for representing proofs, opines that "we seem to be stuck at the assembly language level" [Wen99]. Both the need for a natural interface as well as the value of a lightweight design are also recognized by some in the model checking community, such as the creators of the Alloy modelling language [Jac02].

1.1 Natural Contexts in Lightweight Formal Verification

We focus our efforts by considering the notion of a *natural context*: the sphere of awareness a working human user maintains of the relevant constructs, arguments, experiences, and background materials necessary to accomplish the task at hand. The relevant characteristics of a natural context in such a scenario are (1) its size (encompassing a wide array of experiences, potentially inconsistent or unrelated), and (2) a flexible and powerful interface that allows easy exploration, querying, and short-term adjustments (using not just reference or index but also structure). In explicitly recognizing the possibility that a natural context can contain distinct, inconsistent collections of experiences (such as different logical systems), we are in part inspired by the notion of a *cognitive context* found in literature on formal ontologies [PB08].

This notion can be a useful guide within the realm of formal reasoning. We hypothesize that a lightweight formal reasoning system can be successful if it can efficiently *simulate* (i.e. maintain) an intuitive, rich, and relevant context for the user, and furthermore, if it allows the user to interact with this context in a familiar manner. A concrete example of a natural context familiar to those working in programming language design is a data structure that maintains type signatures and constraints. Its interface consists of the programming language syntax, the means for producing status or error messages, the type inference algorithm, and any constraint solver supporting that algorithm.

1.2 Deployment in the Classroom

In this work, we report on our experience deploying within the classroom a variant of our formal reasoning system[1] designed according to our principles [Lap09]. This was done within a regular undergraduate curriculum course[2] on the principles of programming languages. Among the topics this course typically covers are pure functional programming and algebraic reasoning about functional programs, including the assembly of inductive proofs of propositions about programs. The formal reasoning system was used to present examples of proofs by induction, and was distributed to students within the framework of a homework assignment. Students used the verifier to assemble their proofs. Each student's solutions were graded according to a superficial inspection of the student's formal assumptions and an automated verification of the student's formal assertions. Shortly after the assignment, the students completed a classroom examination in which they had to assemble a formal proof using paper and pencil (without access to the formal reasoning system).

1.3 Overview

In Section 2 we present in their complete form the relevant examples and assigned problems. It is worth noting that the text of this section of the document can be supplied directly to the verification system we employed (including all the LaTeX markup and exposition). Readers may initially want to skip over this section and briefly go over the observations in Sections 3 and 4, as these will provide a context within which to examine the examples. In Section 3 we review the cases in which the formal reasoning system supplied to students was successful in supporting common formal reasoning techniques. We also discuss how its design and implementation contributed to this success, and where further improvements to the design are necessary. In Section 4 we make brief and general observations about our deployment of the formal reasoning system in the classroom. Finally, we review related work in Section 5 and present our conclusions and proposals for future work in Section 6.

2 Examples of Verifiable Formal Proofs

We list all of the examples presented to students, as well as all of the assigned problems and their solutions. Each example or problem involves a proof by induction of a formal proposition about a collection of one or more functions. Most examples are drawn directly from an introductory text on functional programming [Tho99]. In this article, we delimit the mathematical notation with pound signs (`# ... #`) to avoid invoking the LaTeX parser. We note that the contents of this section can be supplied directly to and verified by the formal reasoning system provided to students.

2.1 Examples with recursively-defined lists

The option to perform lightweight verification makes it possible to focus on certain aspects of the formal reasoning and proof authoring process without being bogged down in other aspects. For instance, the following examples make no explicit mention of the induction principle on lists, or of an explicit equality function on lists. In fact, there is no mention of types. Nevertheless, *some*

[1]The Ibis verifier, implemented in Haskell and distributed in the form of source code.

[2]The course in question was the fall 2009 iteration of "Concepts of Programming Languages", a required Computer Science curriculum course for advanced undergraduates within the Computer Science Department at the Boston University College of Arts and Sciences.

verification is possible: one can gain more confidence that a proof is correct by confirming that the proof can be verified automatically, and then briefly inspecting the assumptions and superficial structure of the proof. This brief manual inspection is significantly more simple than a manual proof verification.

The following introduces the two symbols corresponding to an empty list and a list head, as well as assumptions corresponding to several functions typically defined for within functional languages for recursively-defined lists.

```
\vbeg
 Introduce #cons, nil#.

 Introduce #length, append, map, foldr#.

 Assume #              length         nil = 0#.
 Assume #\forall x,xs. length (cons x xs) = 1 + length xs#.

 Assume #\forall ys.      append nil         ys = ys#.
 Assume #\forall x,xs,ys. append (cons x xs) ys = cons x (append xs ys)#.

 Assume #\forall f.      map f nil         = nil#.
 Assume #\forall f,x,xs. map f (cons x xs) = cons (f x) (map f xs)#.

 Assume #\forall f,b.      foldr f b nil         = b#.
 Assume #\forall f,b,x,xs. foldr f b (cons x xs) = f x (foldr f b xs)#.
\vend
```

These functions (sometimes under a different name) can be found in the Haskell [Je99] standard library.

2.1.1 Mapping an identity

Suppose that the identity function is introduced.

```
\vbeg
 Introduce #idf#.
 Assume #\forall x. idf x = x#.
\vend
```

We want to show that for all lists, applying `map idf` to the list will return an equivalent list. The base case is a direct application of the base case definition for `map`.

```
\vbeg
 Assert #map idf nil = nil#.
\vend
```

To argue that the inductive step is correct, we show that if we assume that our desired result holds for some list `xs`, it also holds for a list that has an additional element `x` added to the beginning of the list.

```
\vbeg
 Assert
   #
```

```
  \forall x,xs.
                        map idf xs  = xs
     \Rightarrow  cons x (map idf xs) = cons x xs
     \wedge       map idf (cons x xs) = cons (idf x) (map idf xs)
     \wedge       idf x = x
     \wedge       map idf (cons x xs) = cons x (map idf xs)
     \wedge       cons x (map idf xs) = cons x xs
     \wedge       map idf (cons x xs) = cons x xs
  #.
\vend
```

Notice that in both the base case assertion, and in the sequence of equations in the proof of the inductive step, it is never necessary to explicitly reference which property of equality is being applied, or which assumptions is being instantiated. The user is free to supply comments for steps that she feels might require explanation, but can safely omit this information in cases where this would only serve to clutter the argument.

2.1.2 Folding over an associative operator

Suppose we have an associative binary operator and corresponding identity.

```
\vbeg
 Introduce #op, id#.

 Assume #\forall x. op id x = x#.
 Assume #\forall x. op x id = x#.
 Assume #\forall x,y,z. op x (op y z) = op (op x y) z#.
\vend
```

We want to show that for any two lists, concatenating then folding with respect to the operator is the same as folding each of the two lists, and then concatenating the results. We first present the base case.

```
\vbeg
 Assert
   #
   \forall ys.
              append nil ys = ys
      \wedge   foldr op id ys  = foldr op id (append nil ys)
      \wedge   foldr op id nil = id
      \wedge   op id (foldr op id ys) = foldr op id ys

      \wedge   op id (foldr op id ys) = foldr op id (append nil ys)
      \wedge   op (foldr op id nil) (foldr op id ys) = foldr op id (append nil ys)
   #.
\vend
```

Next, we present the inductive step.

```
\vbeg
 Assert
   #
   \forall x, xs, ys.
```

```
          op (foldr op id xs) (foldr op id ys) = foldr op id (append xs ys)
   \Rightarrow
          op x (op (foldr op id xs) (foldr op id ys)) = op x (foldr op id (append xs ys))

  %-- right side
  \wedge  foldr op id (cons x (append xs ys)) = op x (foldr op id (append xs ys))
  \wedge  (append (cons x xs) ys) = cons x (append xs ys)
  \wedge  foldr op id (cons x (append xs ys)) = foldr op id (append (cons x xs) ys)

  %-- left side
  \wedge  op x (op (foldr op id xs) (foldr op id ys)) = op (op x (foldr op id xs)) (foldr op id ys)
  \wedge  foldr op id (cons x xs) = op x (foldr op id xs)
  \wedge  op x (op (foldr op id xs) (foldr op id ys)) = op (foldr op id (cons x xs)) (foldr op id ys)

  \wedge  op (foldr op id (cons x xs)) (foldr op id ys) = foldr op id (append (cons x xs) ys)
  #.
\vend
```

2.1.3 Identity of `append`

Suppose we want to show that the empty list is also an identity for the function that appends lists. The definition of the function already shows that it is a left identity, so all that remains is to show by induction that it is a right identity. The base case is trivial.

```
\vbeg
 Assert #append nil nil = nil#.
\vend
```

The inductive case is only slightly more complex.

```
\vbeg
 Assert
   #
   \forall x,xs.
                    append xs nil = xs
   \Rightarrow
            cons x (append xs nil) = cons x xs
   \wedge  (append (cons x xs) nil) = cons x (append xs nil)
   \wedge    append (cons x xs) nil = cons x xs
  #.
\vend
```

2.1.4 Associativity of `append`

Suppose we want to show that the function for appending lists is associative. We only need to perform induction over the first argument to the function because the second argument is never decomposed within the function's definition. We begin with the base case.

```
\vbeg
 Assert
   #
   \forall ys, zs.
            append ys zs = append ys zs
```

```
    \wedge  append nil ys = ys
    \wedge  append nil (append ys zs) = append ys zs
    \wedge  append (append nil ys) zs = append nil (append ys zs)
    #.
  \vend
```

We present the inductive step.

```
  \vbeg
   Assert
    #
    \forall x, xs, ys, zs.
            append (append xs ys) zs = append xs (append ys zs)
    \Rightarrow

            %-- put "x" on both sides
            cons x (append (append xs ys) zs) = cons x (append xs (append ys zs))

            %-- right side
    \wedge  append (cons x xs) (append ys zs) = cons x (append xs (append ys zs))

            %-- left side
    \wedge            (append (cons x xs) ys) = cons x (append xs ys)
    \wedge  append (append (cons x xs) ys) zs = append (cons x (append xs ys)) zs
    \wedge  append (cons x (append xs ys)) zs = cons x (append (append xs ys) zs)
    \wedge  append (append (cons x xs) ys) zs = append (cons x xs) (append ys zs)
   #.
  \vend
```

2.1.5 User-defined functions

We write a proof drawn directly from an example in an introductory text on functional programming [Tho99]. If our formal reasoning system were restricted to a straightforward search algorithm that verifies logical expressions (without native support for algebraic manipulation of equations), the following assumptions might be needed. However, *they are not necessary* for our formal reasoning system. We reproduce them here for the sake of completeness.

```
   Assume \forall x. 0 = 2 * 0
   Assume \forall x,y,z. (z*x) + (z*y) = z*(x+y)
```

We now present the example. Suppose we have defined the following two functions.

```
  \vbeg
   Introduce #sum, doubleAll#.

   Assume #              sum nil         = 0#.
   Assume #\forall x,xs. sum (cons x xs) = x + (sum xs)#.

   Assume #              doubleAll nil         = nil#.
   Assume #\forall z,zs. doubleAll (cons z zs) = cons (2 \cdot z) (doubleAll zs)#.
  \vend
```

We now want to prove that for any list of integers, it is the case that doubling the sum of its elements is equivalent to computing the sum of the doubled elements of the list. We start with the base case.

```
\vbeg
 Assert
   #
                             0 = 2 \cdot 0
   \wedge sum nil              = 2 \cdot (sum nil)
   \wedge sum (doubleAll nil) = 2 \cdot (sum nil)
   #.
\vend
```

Next, we present the inductive step.

```
 \vbeg
 Assert
   #
   \forall x,xs.
                         sum (doubleAll xs) =              2 \cdot (sum xs)
  \Rightarrow
         (2 \cdot x) + sum (doubleAll xs) = (2 \cdot x) + 2 \cdot (sum xs)

  %-- We first rewrite the left side.
  \wedge  (2 \cdot x) + (2 \cdot (sum xs)) = 2 \cdot (x + sum xs)
  \wedge  sum (cons x xs) = x + sum xs
  \wedge  (2 \cdot x) + (2 \cdot (sum xs)) = 2 \cdot (sum (cons x xs))

  %-- Next, we rewrite the right side.
  \wedge  sum (cons (2 \cdot x) (doubleAll xs)) = (2 \cdot x) + sum (doubleAll xs)
  \wedge  doubleAll (cons x xs) = cons (2 \cdot x) (doubleAll xs)
  \wedge  sum (doubleAll (cons x xs)) = (2 \cdot x) + sum (doubleAll xs)

  %-- Finally, we put the two sides back together.
  \wedge  sum (doubleAll (cons x xs)) = 2 \cdot (sum (cons x xs))
  #.
\vend
```

2.1.6 The `append` and `length` functions

For this next example, we again present a few assumptions that are not required by our formal reasoning system.

```
 Assume \forall x. 0 + x = x
 Assume \forall x,y,z. x+(y+z) = (x+y)+z
```

Suppose we want to prove that taking the lengths of two lists and adding them is equivalent to appending those same two lists and then computing the length of the result. We present the base case.

```
\vbeg
 Assert
   #
```

```
    \forall ys.
                     append nil ys = ys
    \wedge  length (append nil ys) = length ys
    \wedge  0 + length ys = length ys
    \wedge  length (append nil ys) = 0 + length ys
    \wedge  length (append nil ys) = length nil + length ys
    #.
  \vend
```

Next, we present the inductive step.

```
  \vbeg
   Assert
    #
    \forall x,xs,ys.
      length (append xs ys) = length xs + length ys
    \Rightarrow
      1 + length (append xs ys) = 1 + (length xs + length ys)

   %-- We first rewrite the left side.
   \wedge  length (cons x (append xs ys)) = 1 + length (append xs ys)
   \wedge  append (cons x xs) ys = cons x (append xs ys)
   \wedge  length (append (cons x xs) ys) = 1 + length (append xs ys)

   %-- Then we rewrite the right side.
   \wedge  1 + (length xs + length ys) = (1 + length xs) + length ys
   \wedge  length (cons x xs) = 1 + length xs
   \wedge  1 + (length xs + length ys) = length (cons x xs) + length ys

   %-- Finally, we put the two sides back together
   \wedge  length (append (cons x xs) ys) = length (cons x xs) + length ys
   #.
  \vend
```

2.2 Examples with other other datatypes

2.2.1 Naturals

Suppose we have the following data type definition (e.g. in Haskell), representing natural numbers. The `Z` constructor corresponds to 0, and the `S` constructor corresponds to (+1).

```
data Nat = Z | S Nat
```

Furthermore, suppose we defined addition on objects of type "Nat" using the following function:

```
plus Z     n = Z
plus (S m) n = S (plus m n)
```

If we want to model this situation mathematically, we would introduce all the defined entities (the constructors and function), and then write universally quantified equations to represent the function definition.

```
\vbeg
 Introduce #Z, S, plus#.

 Assume #\forall n.   plus Z     n = Z#.
 Assume #\forall m,n. plus (S m) n = S (plus m n)#.
\vend
```

Now, suppose we want to show that for any `x`, we also have that

```
plus x Z = x
```

The equations alone don't tell us this directly, so we need to prove it by induction. In other words, we have to show that the above statement holds for all possible `x` of type `Nat` by first showing that the `x = Z` case holds, and then showing that if it holds for some `x`, then it holds for `S x`.

First, we verify that the base case holds. The below assertion is verified automatically because it is merely an application of the first assumption above in which the `n` in `\forall n. ...` is instantiated to `Z`.

```
\vbeg
 Assert #plus Z Z = Z#.  %-- by definition of "plus"
\vend
```

Now, we review two operators from logic: implication (`\Rightarrow`) and conjunction (`\wedge`). These are understood by the verifier in the usual way. For example, we could assert that equality is transitive.

```
Assert #\forall x,y,z.
           x = y \wedge y = z  \Rightarrow x = z#.
```

The above should be read as "for any x,y,z, if x is equal to y and y is equal to z then x is equal to z". The `\wedge` operator has precedence over the `\Rightarrow` operator, so no parentheses are necessary. We can now use these operators to construct the inductive case of our proof.

```
\vbeg
 Assert
   #
   \forall x.
              plus x Z = x            %-- inductive hypothesis
   \Rightarrow                        %-- "implies that"
          S (plus x Z) = S x          %-- apply "S" to both sides
   \wedge  plus (S x) Z = S (plus x Z) %-- by definition of "plus"
   \wedge  plus (S x) Z = S x          %-- by transitivity of equality
   #.
\vend
```

The above can now be summarized for concision in another assertion.

```
\vbeg
 Assert #\forall x. plus x Z = x \Rightarrow plus (S x) Z = S x#.
\vend
```

We can also restate both cases if we want to do so.

```
\vbeg
 Assert
   #
          plus Z Z = Z
   \wedge \forall x. plus x Z = x \Rightarrow plus (S x) Z = S x
   #.
\vend
```

Thus, we've proven our result by induction over x. Note that the verifier supports "sequenced" equalities:

```
 Assert #\forall x,y,z. x = y = z \Rightarrow x = z#.
```

This might sometimes be a more concise way to do a proof. For example, the above proof of the inductive case could look a little shorter if it is written in the following way:

```
\vbeg
 Assert
  #
  \forall x.
        plus x Z = x
  \Rightarrow
          S (plus x Z) = S x
  \wedge  plus (S x) Z = S (plus x Z) = S x
  #.
\vend
```

3 Supporting Common Formal Reasoning Techniques

The formal manipulations and proof techniques required for the presented examples can be split into three categories: application of properties of basic logical operators (conjunction, implication, and universal quantification), application of properties of equality, and algebraic manipulations involving arithmetic operators. The design of our formal reasoning system provides relatively robust support (at least within the context of classroom instruction) for each kind of formal manipulation.

3.1 Logical Operators

The inference rules by which the formal reasoning system operates (and which define its behavior with respect to logical operators) are presented in full in earlier work on the formal reasoning system [Lap09]. We briefly review one inference rule in particular whose form is essential for the examples we present in this work.

3.1.1 Sequential Conjunction

As described in more detail in earlier work [Lap09], our formal reasoning system supports a sequential interpretation of the standard conjunction operator. This interpretation is consistent with straightforward formulations of the sequent calculus [Gen69]. The interpretation of conjunction is roughly as follows: if the left-hand argument is determined to be true, it is added to the context under which the right-hand argument is checked. For readers who are interested in a more precise reminder, we reproduce below the introduction rules for the two variants of conjunction. Let Φ by the context (containing information about existing assumptions and bound variables).

$$[\wedge\text{-}\textsc{Intro}]\ \frac{\Phi \vdash e_1 \quad \Phi \vdash e_2}{\Phi \vdash e_1 \wedge e_2}$$

$$[\wedge\text{-}\textsc{Seq-Intro}]\ \frac{\Phi \vdash e_1 \quad \Phi \cup \{e_1\} \vdash e_2}{\Phi \vdash e_1 \wedge e_2}$$

The second form of the rule is a consequence of the first when it combined with rules governing implication (the exact derivation can be found in earlier work [Lap09]). This sequential form of conjunction is essential to the usability of the interface provided by our formal reasoning system. It allows the user to author arguments consisting of a sequence of equations in which each equation is verified to be a consequence of one or more of the equations found earlier in the sequence. All of the formal arguments presented in this work are assembled in this manner.

3.2 Properties of Equality

The formal reasoning system we deployed is designed to simulate a natural context, and one essential characteristic of natural contexts is the presence of a reflexive, symmetric, and transitive equality relation. This relation is simulated using a component of the assumption context that maintains equivalence classes of expressions. This component is described in its basic form in the initial description of the formal reasoning system [Lap09] and developed in more detail in later related work [LH09].

This feature makes it possible to verify several very common algebraic manipulations involving equations. Any direct manipulation that takes advantage of these properties can be verified.

```
Introduce #x,y,z#.
Assert # x + y = x + y #.
Assert # ( x = y /\ y = z ) => x = z #.
Assert # x = y => y = x #.
```

Furthermore, some manipulations implied by these properties can also be verified. For example, it is possible to substitute a portion of an equation with an equivalent expression. In the example below, the asserted expression corresponds to the first assumption except that `(y + z)` is substituted with `(v + w)` and `c` is substituted with `u`.

```
Introduce #x,y,z,a,b,c,u,v,w#.
Assume # x + (y + z) = (a + b) + c #.
Assume # y + z = v + w #.
Assume # c = u #.
Assert # x + (v + w) = (a + b) + u #.
```

It is also possible to perform the same operation to both sides of an equation. In the example below, we add `z` to both sides of an equation.

```
Introduce #x,y,z,a,b#.
Assume # x + y = a + b #.
Assert # z + (x + y) = z + (a + b) #.
```

However, some intuitive manipulations are not yet possible with the current feature set of the formal reasoning system, and we mention these below in Section 3.4.

3.3 Algebraic Manipulations with Arithmetic Operators

The formal reasoning system we utilized is also capable of modelling algebraic manipulations involving the arithmetic operators corresponding to addition, substraction, multiplication, and division. In particular, the system can recognize applications of the associativity, commutativity, and identity laws of addition and multiplication, and the distributive law between addition and multiplication when the expression being distributed is a constant. This capability is described briefly in work describing the formal reasoning system [Lap09]. The examples in Sections 2.1.5 and 2.1.6 demonstrate how this capability allows users of the formal reasoning system to avoid explicitly stating (or retrieving from a library) a few of these common algebraic laws.

3.4 Missing Features

There were two noteworthy instances in which the intuitive expectations of users were not met by the formal reasoning system we deployed. This suggests that the natural context that most users possess has these capabilities, and thus, any formal reasoning system whose design is oriented around simulating a natural context should also have these capabilities.

3.4.1 Composition of Two Kinds of Manipulation

The first is application of the symmetry of equality under a universal quantifier. Students expected (and rightly so, in our opinion) that proof scripts of the following form would be automatically verifiable:

$$\begin{array}{l}\texttt{Assume } \forall \overline{x}, e_1(\overline{x}) = e_2(\overline{x}) \\ \texttt{Assert } \forall \overline{y}, e_2(\overline{y}) = e_1(\overline{y})\end{array}$$

This demonstrates that when querying their own natural context, students are able to compose elimination of the universal quantifier with application of the symmetry of equality. Thus, it is necessary that a system that simulates a natural context be capable of inferring that this step has been performed.

3.4.2 Implicit Application of Induction Principles

The second instance was more substantial: the lightweight proof that the students were asked to construct did not allow the students to use the statement being proven in subsequent proofs. This was not necessary for the assignment, but it does illustrate a difference between the use of lightweight verification system by domain experts, and the use of such systems by students. An expert could decide whether he is satisfied with a lightweight proof, and introduce the theorem proven as an assumption. On the other hand, students must be given strict guidelines in this situation because they may not have the experience to make these decisions.

While we avoided this issue by assembling an assignment in which reuse of the proven statements was not needed, we still failed to meet the intuitive expectations of the students. One way to resolve this is to ask the students to write a more complete proof within our system (involving the explicit introduction and application of propositions corresponding to the induction principles over each of the data types involved) so that the proven statement *can* be used in subsequent arguments. However, we believe the approach that is more consistent with our general principles is to eventually introduce automated support for this capability. The students should indeed be able to use the statements proven, but should not need to do more work than was required of them in our version of the assignment.

3.4.3 On Lightweight Formal Verification and Limiting Formal Reasoning Systems

Some features of the formal reasoning system we deployed were never used in any examples or assigned problems, including those supporting manipulations involving disjunction and existential quantification. All of the examples presented in this report can be verified by a formal reasoning system that does not support manipulations involving disjunction and existential quantifiers. We accomplished this by requiring only lightweight verification of arguments, and by selecting only those examples that involve the proof of a simple universal proposition. We did not explicitly state or require the use of any induction principles within the exercises.

This approach had several advantages. Students were required to comprehend fewer formal concepts, and could thus use their time and effort to study the basic properties of implication, conjunction, and universal quantification (as well as the properties of equality). It also put less pressure on the verification system: providing support for a variety of manipulations involving the introduction and elimination of existential quantifiers (as well as disjunction) is a difficult design task. Users did not need to deal with any inadequacies the system might have in this regard. This demonstrates that while a limited verification system may not seem interesting from the perspective of an expert logician, it can seem quite reasonable to a domain expert unfamiliar with more general forms of logic. Our experience suggests that it is possible to have a limited but successful formal reasoning system that supports only a subset of the inference rules governing common logical operators.

4 Deployment and Evaluation

We briefly comment on how the design of the formal reasoning system affected administrative and practical issues from various perspectives.

4.1 Instruction

We observed a few pedagogical advantages to using our formal reasoning system within classroom instruction. The presence of an application that provides direct feedback about formal arguments makes it possible to easily present and reinforce a precise list of valid formal manipulations. The use of a parser combinator library in the implementation of the formal reasoning system's parser (as discussed briefly in previous work [Lap09]) makes it possible to support multiple parsing regimes, including one that corresponds almost exactly to the syntax of the particular programming language in which the formally analyzed code is written. Admittedly, the parser combinator library that we used [LM01] was designed with Haskell in mind, but we believe its flexibility would make it similarly easy to adjust the system for formal reasoning exercises involving other functional programming languages.

4.2 Usability from the Perspective of Students

One frequent concern about any sort of automated search or inference done by a formal reasoning system is that the user will become frustrated because she cannot predict what the system might be inferring, or what the system's limitations are. However, within this deployment we have found that no such frustration occurs when the system's capabilities are intuitive and can be described in a succinct and straightforward manner. In particular, the formal manipulations enabled by the automated context search capability, the sequential variant of the conjunction inference rule, and the automatic verification capability for common manipulations on equations were all easily and quickly understood by students.

4.3 Grading

It is important to observe that despite the use of a lightweight approach, the task of grading was simplified substantially. When reviewing a student's solutions to the assigned problems, the grader only needed to verify manually that the student's formal argument had the appropriate superficial form. In our particular examples, this amounted to confirming that the student's solutions did not introduce any assumptions beyond those which we provided, that the asserted arguments used only sequential conjunction (rather than, for example, implication), and that assertions corresponding to both the base case and the inductive case of an argument were present. The grader was not required to spend any effort on the more daunting and intellectually taxing task of actually verifying that each formal argument (which could vary from one student to the next) consisted of a sequence of valid manipulations.

5 Related Work

The topic of this report lies on the boundaries between multiple research areas. There exist some reports on attempts to utilize formal reasoning systems for classroom instruction. There also exists a great deal of work related to reasoning formally about both functional and imperative programs. These are both covered in Section 5.1. Our report also aims to demonstrate the benefits of a lightweight verification approach. This argument relates to the more general topic of usability and accessible design of formal reasoning systems, so we review related work in this area in Section 5.2.

5.1 Classroom Instruction of Reasoning about Programs

There have been attempts to teach courses on the topic of program verification since the advent of Hoare logic [Hoa69]. Currently, there exist graduate-level, and some undergraduate-level, courses that cover the specific topic of rigorous machine-assisted formal reasoning and proof authoring using systems such as Coq [PV97]. There also exists a great deal of material demonstrating how systems such as Coq can be used to reason formally about both imperative programs [CMM$^+$09, BC04], and functional programs [Cha09, Mol97]. Other formal reasoning tools specifically designed for reasoning about functional code [Col96] exist, as well. Some have reported on their experience using similar systems in the classroom [WS04]. Our report is distinguished from this large body of work by our emphasis on a lightweight formal verification approach using a formal reasoning system whose design is oriented around simulating a natural context. There is also work in separate communities on using formal methods in the instruction of the more general task of software development [OP98], and systems such as Alloy [Jac02] have also been deployed in the classroom.[3] Our work is distinguished from this by our application of the lightweight approach for very local and specific results about

[3]A list of courses is found at `http://alloy.mit.edu/community/courses`.

individual functions within functional code usually addressed by heavyweight systems used in the first body of work.

The EPGY Theorem-Proving Environment [MRS01] is designed for teaching algebra within a classroom setting at the levels of middle and high school. Its design provides extensive support for algebraic manipulations and computations. However, its design does not directly support a lightweight approach to verification: manipulations are strictly enforced, and authoring arguments requires the use of an interactive interface.

5.2 Formal Verification Systems

5.2.1 Accessibility and a Lightweight Approach

Several formal reasoning systems exist whose development was motivated by observations and goals similar to those that motivated the development of our system, described in further detail in earlier work [Lap09]. Our system can be viewed in part as further development and integration of the various ideas incorporated into those systems, and furthermore as an example of which priorities and features of those systems we believe should be emphasized (and which are disadvantageous and may need to be removed or modified) if we hope to improve the practical usefulness and accessibility of verification systems.

In scenarios in which there are established conventions for notation, it is important that these conventions be exploited to the greatest extent possible. Our observations are shared by the designers of Scunak [Bro06], who refer to the need for "naturality" in a system's concrete representation. Our adoption of a notation that very closely resembles the functional programming language used in the course made the transition from programming to formal reasoning almost effortless for the students. Only a few new basic constructs needed to be introduced, and these were intuitive.

The designers of the ForTheL language and SAD proof assistant [VLPA08] share our motivations and suggest similar principles to ours. The design of the ForTheL language includes high-level constructs similar to the assumption and assertion statements in the formal reasoning system we deployed. However, the high-level constructs for managing proofs in ForTheL are reminiscent of proof scripts for interactive theorem provers and could potentially present an obstacle to new users. Enforcing high-level constructs like proofs and theorems inhibits the user from employing a lightweight approach. The examples in this report demonstrate that the option to leave out a high-level proof structure (such as entirely avoiding the use of an explicit inductive principle) makes a lightweight approach possible, and that a lightweight approach can still provide some advantages of formal verification. Furthermore, the usefulness of a superficial verification of individual symbolic manipulations (regardless of semantics or high-level structure) in turn demonstrates the importance of developing a wider spectrum of verification methods, both highly rigorous and lightweight.

In providing a familiar syntax and supporting verification of formal manipulations without explicit reference to them, the design of our system is similar to Scunak [Bro06]. However, it is worth noting that we disagree with the claim made by the designers of Scunak that proofs of fundamental concepts, such as algebraic distribution laws, are a good touchstone for measuring the success of proof verification systems. On the contrary, focusing on such proofs distracts from efforts directed towards recognizing the more essential reasoning activities we demonstrate in the examples presented in this report. For verification systems used at the college level and beyond, distributivity is one of many natural, implicit manipulations that should be supported without requiring any user effort. We believe that incongruously complex proofs of results whose importance is difficult to motivate at the college level may not constitute effective or compelling examples for classroom instruction of formal reasoning techniques.

Finally, our experience shows us that at least when coupled with a lightweight approach, a formal reasoning system that does not provide step-by-step interaction for proof construction can be effective within the classroom. As has been observed before [ACP01], interactive proof systems that give step-by-step feedback and direction to a user can actually be inconveniently rigid. The user is required to learn how to direct the interactive system, cannot easily "jump around" while constructing a proof, and cannot resort to a lightweight approach under which only some parts of the proof are correct.

5.2.2 Native Support

The formal reasoning system we used provides native support for verifying chains of algebraic manipulations on equations, and this feature is recognized by the authors of the `Tutch` system [ACP01] as essential to making further progress in the design of effective formal reasoning systems. While providing native support has drawbacks when it comes to guaranteeing consistency and makes it more difficult to show a system is fully trustworthy, it can also greatly improve the usability of a formal reasoning system because the system designer has greater flexibility in removing any obstacles a user might face. Making this tradeoff in favor of native support allowed us to design and deploy a system that natively supports equality, equational reasoning, sequential reasoning, and some algebraic properties of arithmetic operators. Because these correspond to intuitions already present in the a college student's own natural context, little effort was required on the part of a student to grasp and subsequently utilize these capabilities.

6 Conclusions and Future Work

We have reported our observations about our deployment within the classroom of a lightweight formal reasoning system. We have identified a variety of specific ways in which a design in accordance with the principles of a natural context contributes to the usability of a formal reasoning system (at least in a narrow context restricted to classroom examples). Finally, we have discussed the practical implications of using such a formal reasoning system within the classroom.

Further work on this issue falls into one of two categories: improvements to the formal reasoning system, and further experimentation through deployment in the classroom. We have identified several specific capabilities that a formal reasoning system that simulates a natural context must have (such as automated recognition of the application of an induction principle), and these capabilities must be introduced into future versions of our formal reasoning system. Because one of our goals in developing design principles for formal reasoning systems is the creation of systems that are accessible to a variety of communities (students, instructors, and researchers), we must deploy the system in courses on other topics or in other areas (particularly in undergraduate-level mathematics and philosophy courses).

Acknowledgements

We would like to thank for their cooperation and feedback Professor Assaf Kfoury (course instructor), Michael Goldberg (course grader), and the students enrolled in the fall 2009 iteration of

the course “Concepts of Programming Languages”, a required curriculum course in the Computer Science Department at the Boston University College of Arts and Sciences.

References

[ACP01] A. Abel, B. Chang, and F. Pfenning. Human-readable machine-verifiable proofs for teaching constructive logic, 2001.

[BC04] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[Bro06] Chad E. Brown. Verifying and Invalidating Textbook Proofs using Scunak. In *Mathematical Knowledge Management, MKM 2006*, pages 110–123, Wokingham, England, 2006.

[Cha09] Arthur Charguéraud. Verification of call-by-value functional programs through a deep embedding. Unpublished. http://arthur.chargueraud.org/research/2009/deep/, March 2009.

[CMM$^+$09] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP ’09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 79–90, New York, NY, USA, 2009. ACM.

[Col96] Graham Collins. A proof tool for reasoning about functional programs. In *TPHOLs ’96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pages 109–124, London, UK, 1996. Springer-Verlag.

[Gen69] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen*. North-Holland Publishing Co., Amsterdam, 1969. Edited by M. E. Szabo.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[Je99] Simon Peyton Jones and John Hughes (editors). Haskell 98: A non-strict, purely functional language. Technical report, February 1999.

[Lap09] Andrei Lapets. Improving the accessibility of lightweight formal verification systems. Technical Report BUCS-TR-2009-015, Computer Science Department, Boston University, April 30 2009.

[LH09] Andrei Lapets and David House. Efficient Support for Common Relations in Lightweight Formal Reasoning Systems. Technical report, CS Dept., Boston University, November 2009.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical report, Departement of Computer Science, Universiteit Utrecht, 2001.

[Mol97] Maarten De Mol. Verified proofs concerning functional programs. In *Proceedings of the Nijmegen Student Conference on Computing Science*, pages 69–82, 1997.

[MRS01] David McMath, Marianna Rozenfeld, and Richard Sommer. A Computer Environment for Writing Ordinary Mathematical Proofs. In *LPAR '01: Proceedings of the Artificial Intelligence on Logic for Programming*, pages 507–516, London, UK, 2001. Springer-Verlag.

[OP98] Jonathan S. Ostroff and Richard F. Paige. Formal methods in the classroom: The logic of real-time software design. *Real-Time Systems Education Workshop, IEEE*, 0:63, 1998.

[PB08] Christiana Panayiotou and Brandon Bennett. Cognitive context and arguments from ontologies for learning. In *Proceeding of the 2008 conference on Formal Ontology in Information Systems*, pages 65–78, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press.

[PV97] Catherine Parent-Vigouroux. Verifying programs in the calculus of inductive constructions. *Formal Aspects of Computing*, 9(5-6):484–517, 1997.

[SBF$^+$02] Jörg H. Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, and Martin Pollet. Proof Development with OMEGA: sqrt(2) Is Irrational. In *LPAR*, pages 367–387, 2002.

[SC05] Wilfried Sieg and Saverio Cittadini. Normal Natural Deduction Proofs (in Non-classical Logics). In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 169–191. Springer, 2005.

[Tho99] Simon Thompson. *The Haskell: The Craft of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[VLPA08] Konstantin Verchinine, Alexander Lyaletski, Andrei Paskevich, and Anatoly Anisimov. On Correctness of Mathematical Texts from a Logical and Practical Point of View. In *Proceedings of the 9th AISC international conference, the 15th Calculemas symposium, and the 7th international MKM conference on Intelligent Computer Mathematics*, pages 583–598, Berlin, Heidelberg, 2008. Springer-Verlag.

[Wen99] Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *TPHOLs '99: Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 167–184, London, UK, 1999. Springer-Verlag.

[Wie03] Freek Wiedijk. Comparing mathematical provers. In *MKM '03: Proceedings of the Second International Conference on Mathematical Knowledge Management*, pages 188–202, London, UK, 2003. Springer-Verlag.

[WS04] Christoph Walther and Stephan Schweitzer. Verification in the classroom. *J. Autom. Reason.*, 32(1):35–73, 2004.