Verified Programming in GURU

Aaron Stump
Computer Science
The University of Iowa
Iowa City, Iowa, USA

January 14, 2009

Contents

Chapter 1

Introduction

1.1 What is GURU?

GURU is a verified programming language, as described above. But in this book, we will also refer to the open-source project consisting of a compiler for GURU code, the standard library of GURU code, and other materials as "GURU" (or "the GURU project"). Finally, the compiler for GURU code, which includes a type- and proof-checker, as well as an interpreter, is called `guru`. We will work with version 1.0 of GURU.

1.2 Installing GURU

This book assumes you will be using GURU on a Linux computer, but it does not assume much familiarity with Linux. To install GURU, first start a shell. Then run the folllowing SUBVERSION command:

```
svn checkout http://guru-lang.googlecode.com/svn/branches/1.0 guru-lang
```

This will create a subdirectory called `guru-lang` of your home directory. This directory contains the JAVA source code for GURU version 1.0 itself (`guru-lang/guru`), the standard library written in GURU (`guru-lang/lib`), this book's source code (`guru-lang/doc`), and a number of tests written in GURU (`guru-lang/tests`). A few things in the distribution currently depend on its being called `guru-lang`, and residing in your home directory.

Before you can use GURU, you must compile it. To do this, in your shell, you should change to the `guru-lang` directory. Then run the command `make` from the shell. This will invoke the JAVA compiler to compile the JAVA source files in `guru-lang/guru`. After this is complete, you can run `guru-lang/bin/guru` from the shell to process GURU source files. This will be further explained in Section 2.1.2 below.

Chapter 2

Monomorphic Functional Programming

Mainstream programming languages like JAVA and C++, while powerful and effective for many applications, pose problems for program verification. This is for several reasons. First, these are large languages, with many different features. They also come with large standard libraries, which have to be accounted for in order to verify programs that use them. Also, they are based on programming paradigms for which practically effective formal reasoning principles are still being worked out. For example, reasoning about programs even with such a familiar and seemingly simple feature as *mutable state* is not at all trivial. Mutable state means that the value stored in a variable can be changed later. The reader perhaps has never even dreamed there could be languages where this is not the case (where once a variable is assigned a value, that value cannot be changed). We will study such a language in this chapter. Object-orientation of programs creates additional difficulties for formal reasoning.

Where object-oriented languages are designed around the idea of an object, functional programming languages are designed around the idea of a function. Modern examples with significant user communities and tool support include CAML (pronounced "camel", http://caml.inria.fr/) and HASKELL (http://www.haskell.org/). HASKELL is particularly interesting for our purposes, because the language is *pure*: there is no mutable state of any kind. Indeed, HASKELL programs have a remarkable property: any expression in a program is guaranteed to evaluate in exactly the same way every time it is evaluated. This property fails magnificently in mainstream languages, where expressions like "`gettimeofday()`" are, of course, intended to evaluate differently each time they are called. Reasoning about impure programs requires reasoning about the state they depend on. Reasoning about pure programs does not, and is thus simpler. Nevertheless, pure languages like HASKELL do have a way of providing functions like "`gettimeofday()`". We will consider ways to provide such functionality in a pure language in a later chapter.

GURU contains a pure functional programming language, which is similar in some ways to Caml and Haskell. In this chapter, we will learn the basics of this functional programming language. In the next chapter, we will see how to verify programs written in it. We will focus for now just on *monomorphic* programs. These are programs defined for specific datatypes. Generic, or *polymorphic*, programming will be considered a little later.

2.1 Inductive Datatypes

At the heart of functional programming languages like CAML and HASKELL – but not functional languages like LISP and its dialects (e.g., SCHEME) – are user-declared inductive datatypes. An inductive datatype consists of data which are incrementally and uniquely built up using a finite set of operations, called the *constructors* of the datatype. Incrementally built up means that bigger data are obtained by gradual augmentation from smaller data. Uniquely means that the same piece of data cannot be built up in two different ways. Let us consider several examples.

2.1.1 Unary natural numbers

The natural numbers are the numbers $0, 1, 2, \ldots$. We typically write numbers in decimal notation. Unary notation is much simpler. Essentially, a number like 5 is represented by making 5 marks, for example like this:

$$|||||$$

A few questions arise. How do we represent zero? By zero marks? It is then hard to tell if we have written zero or just not written anything at all. We will Z for zero. Also, how does this fit the pattern of an inductive datatype? That is, how are bigger pieces of data (i.e., bigger numbers) obtained incrementally and uniquely from smaller ones? One answer is that a number like five can be viewed as built up from its *predecessor* 4 by the *successor* operation, which we will write S. The successor operation just adds one to a natural number. In this book, we will write the *application* of a function f to an input argument x as $f\ x$ or $(f\ x)$. This is in contrast to other common mathematical notation, where we write $f(x)$ for function application. So the five-fold application of the successor operation to zero, representing the number 5, is written this way:

$$(S\ (S\ (S\ (S\ (S\ Z)))))$$

Every natural number is either Z or can be built from Z by applying the successor operation a finite number of times. Furthermore, every natural number is uniquely built that way. This would not be true if in addition to Z and S, we included an operation P for predecessor. In that case, there would be an infinite number of ways to build every number. For example, Z could be built using just Z, or also in these ways (and others):

$$\begin{array}{l}(S\ (P\ Z))\\(S\ (S\ (P\ (P\ Z))))\\(S\ (S\ (S\ (P\ (P\ (P\ Z))))))\\\ldots\end{array}$$

The operations Z and S are the *constructors* of the natural number datatype.

The simplicity of unary natural numbers comes at a price. The representation of a number in unary is exponentially larger than its representation in decimal notation. For example, it takes very many slash marks or applications of S to write 100 (decimal notation) in unary. In contrast, it only takes 3 digits in decimal. On the other hand, it is much easier to reason about unary natural numbers than binary or decimal numbers, and also easier to write basic programs like addition. So we begin with unary natural numbers.

2.1.2 Unary natural numbers in GURU

GURU's standard library includes a definition of unary natural numbers, and definitions of standard arithmetic functions operating on them. To play with these, first start up a text editor, and create a new file called `test.g`. Start this file with the following text:

```
Include "guru-lang/lib/plus.g".
```

This `Include`-command will tell `guru` to include the file `plus.g` from the standard library. Then include the following additional command:

```
Interpret (plus (S (S Z)) (S (S Z))).
```

This `Interpret`-command tells GURU to run its interpreter on the given expression. The interpreter will evaluate the expression to a value, and then print the value. This expression is an application of the function `plus`, which we will see how to define shortly, to 2 and 2, written in unary. Naturally, we expect this will evaluate to 4, written in unary.

To run `guru` on your `test.g` file, first make sure you have saved your changes to it. Then, start a shell, and run the command

```
guru-lang/bin/guru test.g
```

This runs the `guru` tool on your file. You should see it print out the expected result of adding 2 and 2 in unary:

```
(S (S (S (S Z))))
```

The definition of the unary natural numbers is in `guru-lang/lib/nat.g`, which is included by the file `plus.g` which we have included here. If you look in `nat.g`, you will find at the top the following definition:

```
Inductive nat : type :=
  Z : nat
| S : Fun(x:nat).nat.
```

This is an `Inductive`-command. It instructs GURU to declare the new inductive datatype `nat`. The "`nat : type`" on the first line of the definition just tells GURU that `nat` is a type. We will see other examples later which use more complicated declarations than just "`: type`". In more detail, "`nat : type`" means that `type` is the *classifier* of `nat`. The concept of classifier is central to GURU. For example, the next two lines declare the classifiers for `Z` and `S`. So what is a classifier? In GURU, some expressions are classifiers for others. For example, `type` is the classifier for types. Following the processing of this `Inductive`-command, we will also have that `nat` is the classifier for unary natural numbers encoded with `Z` and `S`. The classifier for `S` states that it is a function (indicated with `Fun`) that takes in an input called `x` that is a `nat`, and then produces a `nat`. Generally speaking, classifiers partition expressions into sets of expressions that have certain similar properties. Every expression in GURU has exactly one classifier.

An additional simple piece of terminology is useful. The constructor `Z` returns a `nat` as output without being given any `nat` (or any other data) as input. In general, a constructor of a type `T` which has the property that it returns a `T` as output without requiring a `T` as input is called a *base* constructor. In contrast, `S` does require a `nat` as input. In general, a constructor of a type `T` which requires a `T` as input is called a *recursive* constructor.

We should note finally that GURU does not provide decimal notation for unary natural numbers. Indeed, GURU currently does not provide special syntax for describing any data. All data are inductive, and are constructed by applying constructors (like `S` and `Z`) to smaller data.

2.2 Programming Constructs

In this section, we will see the basic programming constructs of GURU. Like most other functional programming languages, the heart of the language is very compact and simple: we can define recursive functions, decompose inductive data using a simple pattern-matching construct, and apply (aka, call) functions. That is essentially it. Recursion is such a powerful idea that even with such a simple core, we can write arbitrarily rich and complex programs. We will consider first non-recursive functions, then pattern matching, and finally recursive functions.

2.2.1 Non-recursive functions

Suppose we want to define a doubling function, based on the `plus` function we used before. We have not seen how to define `plus` yet, since it requires recursion and pattern matching. But of course, we can write a function which calls `plus`, even if we do not know how `plus` is written. The doubling function can be written like this:

```
fun(x:nat).(plus x x)
```

Let us examine this piece of code. First, "`fun`" is the keyword which begins a function, also called a `fun`-term. After this keyword come the arguments to the function, in parentheses. In this case, there is just one argument, `x`. Arguments must be listed with their types (with a colon in between). In this case, the type is `nat`. After the arguments we have a period, and then the value returned by the function. In this case, the value returned is just the result of the application of `plus` to `x` and `x`, for which the notation, as we have already seen, is `(plus x x)`.

To use this function in GURU, try the following. In your home directory, create a file `test.g`, and begin it with

```
Include "guru-lang/lib/plus.g".
```

As for the example in Section 2.1.2 above, this includes the definitions of `nat` and `plus`. Next write:

```
Interpret (fun(x:nat).(plus x x) (S (S Z))).
```

Save this file, and then from your home directory run GURU on your file:

```
guru-lang/bin/guru test.g
```

You should see it print out the expected result of doubling 2, in unary:

```
(S (S (S (S Z))))
```

This example illustrates the fact that `fun(x:nat).(plus x x)` is really a function, just like `plus`. Just as we can apply `plus` to arguments `x` and `y` by writing `(plus x x)`, we can also apply `fun(x:nat).(plus x x)` to an argument `(S (S Z))` by writing `(fun(x:nat).(plus x x) (S (S Z)))`, as we did in this example.

Definitions

Most often we write a function expecting it to be called in multiple places in our code. We would like to give the function a name, and then refer to it by that name later. In GURU, this can be done with a `Define`-command. To demonstrate this, add to the bottom of `test.g` the following:

```
Define double := fun(x:nat).(plus x x).

Interpret (double (S (S Z))).
```

The `Define`-command assigns name `double` to the `fun`-term. We can then refer to that function by the name `double`, as we do in the subsequent `Interpret`-command. If you run GURU on `test.g`, you will see the same result for this `Interpret`-command as we had previously: `(S (S (S (S Z))))`.

Multiple arguments

The syntax for functions with multiple arguments is demonstrated by this example:

```
Define double_plus := fun(x:nat)(y:nat). (plus (double x) (double y)).
```

This function is supposed to double each of its two arguments, and then add them. The nested application `(plus (double x) (double y))` does that. The `fun`-term is written with each argument and its type between parentheses, as this example shows. There is a more concise notation when consecutive arguments have the same type, demonstrated by:

```
Define double_plus_a := fun(x y:nat). (plus (double x) (double y)).
```

Multiple consecutive arguments can be listed in the same parenthetical group, followed by a colon, and then their type.

Function types

You can see the classifier that GURUcomputes for the `double` function as follows. In your `test.g` file (in your home directory, beginning with an `Include`-command to include `plus.g`, as above), write the following:

```
Define double := fun(x:nat).(plus x x).

Classify double.
```

If you (save your file and then) run GURU on `test.g`, it will print

```
Fun(x : nat). nat
```

This is a `Fun`-type. `Fun`-types classify `fun`-term by showing the input names and types, and the output type. We can see that GURU has computed the (correct) output type `nat` for our doubling function.

Earlier it was mentioned that every expression in GURU has a classifier. You may be curious to see what the classifier for `Fun(x : nat). nat` is. So add the following to your `test.g` and re-run GURU on it:

```
Classify Fun(x : nat). nat.
```

You will see the result `type`. If you ask GURU for the classifier of `type`, it will tell you `tkind`. If you ask for the classifier of `tkind`, GURU will report a parse error, because `tkind` is not an expression. So the classification hierarchy stops there. We have the following classifications (this is not valid GURU syntax, but nicely shows the classification relationships):

```
fun(x:nat).(plus x x)  :  Fun(x:nat).nat  :  type  :  tkind
```

Functions as inputs

Now that we have seen how to write function types, we can write a function that takes in a function `f` of type `Fun(x:nat).nat` and applies `f` twice to an argument `a`:

```
Define apply_twice := fun(f:Fun(x:nat).nat)(a:nat). (f (f a)).
```

There is no new syntax here: we are just writing another `fun`-term with arguments `f` and `a`. The difference from previous examples, of course, is that the type we list for `f` is a `Fun`-type. An argument to a `fun`-term (or listed in a `Fun`-type) can have any legal GURU type, including, as here, a `Fun`-type. You can test out this example like this (although before you run it, try to figure out what it will compute):

```
Interpret (apply_twice double (S (S Z))).
```

Functions as outputs

Functions can be returned as output from other functions. This is actually already possible with functions we have seen above. For example, consider the `plus` function. Its type, as revealed by a `Classify`-command, is

```
Fun(n : nat)(m : nat). nat
```

Now try the following:

```
Classify (plus (S (S Z))).
```

GURU will say that the classifier of this expression is:

```
Fun(m : nat). nat
```

This example shows that we can apply functions to fewer than all the arguments they accept. Such an application is called a *partial application* of the function. In this case, `plus` accepts two arguments, but we can apply it to just the first argument, in this case `(S (S Z))`. The result is a function that is waiting for the second argument `m`, and will then return the result of adding two to m. This point can be brought out with the following:

```
Define plus2 := (plus (S (S Z))).

Interpret (plus2 (S (S (S Z)))).
```

We define the `plus2` function to be the partial application of `plus` to `(S (S Z))`, and then interpret the application of `plus2` to three. GURU will print five (in unary), as expected.

Comments

This is not a bad place to describe the syntax for comments in GURU. To comment out all text to the end of the line, we use %. For example:

```
Define plus2 := (plus (S (S Z))).  % This text here is in a comment.
```

Comments can also be started and stopped by enclosing them betwee `%-` and `-%`, as in:

```
%- Comments can also be written using
   this syntax. -%
```

Comments can be placed anywhere in GURU input, including in the middle of expressions, like this:

```
Interpret (plus %- here is a comment -% Z).
```