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 Verified Programming

Software errors are estimated to cost the U.S. economy $60 billion a year, and they contribute to computer security vulnerabilities which end up costing U.S. companies a similar amount [1, 2]. Possibly buggy software cannot be used for safety critical systems like biomedical implants, nuclear reactors, airplanes, and utilities infrastructure, at least not without costly backup mechanisms to handle the case of software failure. These reasons alone are certainly enough to motivate our efforts to eliminate the possibility of bugs from our software.

But there is another reason to seek to create software that is absolutely guaranteed to be free from errors: the basic desire we have as computer scientists to create excellent software. How dissatisfying it is to write code that we know we cannot truly trust! Even if we test it heavily, it may still fail. It has famously been said that testing can establish the presence of bugs, but not their absence: we might always have missed that one input scenario that breaks the system. For anyone who loves the construction of elaborate virtual edifices and intricate logical structures, verification has to be an addicting activity.

Indeed it is. The approach we will follow in this book is to construct, along with our software, proofs that the software is correct. These proofs are formal artifacts, just like programs. The compiler checks that they are completely logically sound – no missing cases or incorrect inferences, for example – when it compiles our code. If the proofs check, then we can be much more confident that our software is correct. Of course, it is always possible there is a bug in the compiler (or in the operating system or standard libraries the compiler relies on), but assuming there is not, then we know our code truly has the properties we have proved it has. No matter what inputs we throw at it, it will always behave as our theorems promise us it will.

Constructing programs and proofs together is, quite possibly, the most complex engineering activity known to humankind. It can be quite challenging, and at times frustrating, for example when proofs fail to go through not because the code is buggy, but because the property one wishes to prove must be carefully rephrased. But building verified software is extremely rewarding. The mental effort required is very stimulating, even if we will never again write a line of machine-checked proof. Furthermore, even if we verify only fairly modest properties of a piece of code – and any verification is necessarily incomplete, since can never exhaust the things we might potentially wish to prove about a piece of code – it is my experience that even lightly verified code tends to work much, much better right from the start than unverified code.

1.2 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.

1.3 What is GURU?

GURU is a pure functional programming language, which is similar in some ways to Caml and Haskell. But GURU also contains a language for writing formal proofs demonstrating the properties of programs. So there are really two languages: the language of programs, and the language of proofs. When the compiler checks a program, it computes a type for it, just as compilers for other languages like JAVA do. But in GURU, such types can be significantly richer than in mainstream or even most research programming languages. These types are called *dependent types*, and they can express non-trivial semantic properties of data and functions. Analogously, when the compiler checks a proof, it computes a formula for it, namely the formula the proof proves. So we really have four kinds of expressions in GURU: programs (which we also call *terms*) and their types; proofs and their formulas.

GURU is inspired largely by the COQ theorem prover, used for formalized mathematics and theoretical computer science, as well as program verification [**?**]. Like COQ, GURU has syntax for both proofs and programs, and supports dependent types. GURU does not have as complex forms of polymorphism and dependent types as COQ does. But GURU supports some features that are difficult or impossible for COQ to support, which are useful for practical program verification. In COQ, the compiler must be able to confirm that all programs are *uniformly terminating*: they must terminate on all possible inputs. We know from basic recursion theory or theoretical computer science that this means there are some programs which really do terminate on all inputs that the compiler will not be able to confirm do so. Furthermore, some programs, like web servers or operating systems, are not intended to terminate. So that is a significant limitation. Other features GURU has that COQ lacks include support for functional modeling of non-functional constructs like destructive updates of data structures and arrays; and better support for proving properties of dependently typed functions.

So GURU is a verified programming language. 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.

1.4 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.

1.5 The Structure of This Book

We begin with *monomorphic* functional programming in GURU. Monomorphic means that code operates only over data of specific known types. We will see further how to write proofs demonstrating that such functions satisfy properties we might be interested in verifying. Next, we consider *polymorphic*, or generic, programming, where code may operate generically over data of any type, not known in advance by the code. We again see how to write proofs showing that such functions have the properties we might be interested in. The next step is *dependently typed* programming. Here, the types of data and functions themselves capture the properties we are interested in verifying. There is no separate proof to write for such properties, rather the program contains proofs to help the type checker check that the code really meets its specification. We will then see how to write additional proofs about dependently typed programs. Finally, we see how non-functional constructs like updatable arrays are handled in GURU via *functional modeling*.

1.6 Acknowledgments

The following people, listed alphabetically, have assisted me with with either the theory or implementation of GURU or its standard library: Morgan Deters, Henry Li, Todd Schiller, Timothy Simpson, Daniel Tratos, and Edwin Westbrook. This research has been partially supported by the National Science Foundation under grant CCF-0448275.

Chapter 2

Monomorphic Functional Programming

Like most other functional programming languages, the heart of the GURU's programming language is very compact and simple: we can define inductive datatype, write (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 inductive datatypes, then non-recursive functions, pattern matching, and finally recursive functions. When we turn to polymorphic and especially dependently typed programming in later chapters, we will have to revisit all these concepts (inductive types, recursive functions, pattern matching, and function applications), which become richer in those richer programming settings. So the syntax in this chapter will be enriched in later chapters.

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 a basic example.

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 declaration 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:

```
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 declaration 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` (zero) and `S` (successor). 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. There are no built-in datatypes in GURU: all data are inductive, constructed by applying constructors (like `S` and `Z`) to smaller data.

2.2 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 *body* of the `fun`-term. The body just gives the code to compute 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.

2.2.1 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))))`.

2.2.2 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.

2.2.3 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
```

2.2.4 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))).
```

2.2.5 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.

For another example of using functions as outputs, here is a function to compose two functions, each of type `Fun(x:nat).nat`:

```
fun(f g : Fun(x:nat).nat). fun(x:nat). (f (g x))
```

The inputs to this `fun`-term are functions `f` and `g`. The body, which computes the output value returned by the function, is

```
fun(x:nat). (f (g x))
```

This is, of course, a function that takes in input `x` of type `nat`, and returns `(f (g x))`. In GURU, what we have written as the definition of our composition function is equivalent to:

```
fun(f g : Fun(x:nat).nat)(x:nat). (f (g x))
```

That is, due to partial applications, we can write our composition function as a function with three arguments: `f`, `g`, and `x`. We can then just apply it to the first two, to get the composition.

2.2.6 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).
```

2.3 Pattern Matching

Like other functional languages that rely on inductive datatypes, GURU programs can use pattern matching to analyze data by taking it apart into its subdata. To demonstrate this, we will write a simple function to test whether a `nat` is zero (`Z`) or not. For this, we need the definition of booleans, provided in `guru-lang/lib/bool.g`. This file is included by `nat.g` (included by `plus.g`) is so we do not need to include `bool.g` explicitly. It is worth noting that it is not an error in GURU to include a file multiple times: GURU keeps track of which files have been included (by their full pathnames), and ignores requests after the first one to include the file. So suppose our `test.g` file in our home directory starts off as above:

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

This will pull in the declaration of the booleans, which is:

```
Inductive bool : type :=
  ff : bool
| tt : bool.
```

Just as for the declaration of `nat` above, this `Inductive`-command instructs GURU to add constructors `tt` (for true) and `ff` (for false), both of type `bool`. Now we can define the `iszero` function as follows:

```
Define iszero :=
  fun(x:nat).
    match x with
      Z => tt
    | S x' => ff
    end.
```

Let us walk through this definition. First, we see it is written across several lines, with changing indentation. Whitespace in GURU, as in most sensible languages, has no semantic impact. So the indentation and line breaks are just (intended) to make it easier to read the code. It would have the same meaning if we wrote it all on one line, like this:

```
Define iszero := fun(x:nat). match x with Z => tt | S x' => ff end.
```

To return to the code: we have a `Define`-command, just as we have seen above. We are defining `iszero` to be a certain `fun`-term. This `fun`-term takes in input `x` of type `nat`, and then it matches on `x`. Here is where the pattern matching comes into play.

We have "`match x with`". In this first part of the `match`-term, we are saying we want to pattern match on `x`. We are allowed to match on anything whose type is an inductive type (i.e., declared with an `Inductive`-command). We cannot match on functions, for example, because they have `Fun`-types, which are not inductive. The term we are matching on is called the *scrutinee* (because the `match`-term is scrutinizing – i.e., analyzing – it).

Next come the `match`-clauses, separated by a bar ("`|`"):

```
      Z => tt
    | S x' => ff
```

We have one clause for each constructor of the scrutinee's type. The scrutinee (`x` in "`match x with`") has type `nat`, which has constructors `Z` and `S`, so we have one clause for each of those constructors. It is required in GURU to list the clauses in the same order as the constructors were declared in the `Inductive`-command which declared the datatype. Our declaration of `nat` (back in Section 2.1.2) lists `Z` first and then `S`, so that explains the ordering of the `match`-clauses here.

Each `match`-case starts out with a pattern for the corresponding constructor. The pattern starts with the constructor, and then lists different variables for each of the constructor's arguments. So we have the patterns `Z` and `S x'`. The first pattern has no variables, since `Z` takes no arguments. The second pattern has the single variable `x'`, for the sole argument of `S`. These variables are called pattern variables. They are declared by the pattern, and their scope is the rest of the `match`-clause.

After the pattern, each `match`-clause has "`=>`", and then its *body*. This is similar to the body of a `fun`-term: it gives the code to compute the value returned by the function. For our `iszero` function, we return `tt` in the zero (`Z`) case, and `ff` in the successor (`S`) case. If we then run the following example, we will get the expected value of `tt`:

```
Interpret (iszero Z).
```

2.4 Recursive Functions

We are finally in a position now to see how to define recursive functions. GURU does not have iterative looping constructs like `while`- or `for`-loops. Instead, all looping is done by recursion. Here is the code for `plus`, taken from `guru-lang/lib/plus.g`:

```
  fun plus(n m : nat) : nat.
    match n with
      Z => m
    | S n' => (S (plus n' m))
    end
```

This is a recursive `fun`-term. There are two main differences from the non-recursive `fun`-terms we have seen above. First and foremost, the "`fun`" keyword is followed by a name for the recursive function. This name can be used in the body of the function to make a recursive call. We see it used in the second `match`-clause. We will walk through the `match`-clauses in just a moment, but before that we note the second distinctive feature of a recursive `fun`-term: after the argument list ("`(n m : nat)`"), there is colon and then the return type of the `fun`-term is listed ("` : nat`"). Since `plus` returns a `nat`, that is the return type. The reason GURU requires us to list the return type here for a recursive `fun`-term is that it makes it much easier to type check the term. Wherever `plus` is called in the body of the function, we know exactly what its input types and output type are. If GURU allowed us to omit the output type here at the start of the `fun`-term, then the type checker would not know the type of the value that is being computed by the recursive call to `plus` in the second `match`-clause.

Syntactically, there is nothing else new in the code. But let us try to understand how it manages to add two unary natural numbers. The code is based on the following two mathematical equations:

$$\begin{array}{rcl} 0+m & = & m \\ (1+n')+m & = & 1+(n'+m) \end{array}$$

These are certainly true statements about addition. But how do they relate to the `fun`-term written above? Let us see how to transform them step by step into that `fun`-term. First, we should recognize that 0 and $1+x$ are just different notation for zero and successor of x. If we use the notation we have used in GURU so far for these, the mathematical equations turn into:

$$\begin{array}{rcl} \texttt{Z}+m & = & m \\ (\texttt{S}\ n')+m & = & (\texttt{S}\ (n'+m)) \end{array}$$

Now, we do not have infix notation in GURU for functions, so let us replace the infix $+$ symbol with a prefix `plus`:

$$\begin{array}{rcl} \texttt{(plus Z m)} & = & m \\ \texttt{(plus (S n') m)} & = & \texttt{(S (plus n' m))} \end{array}$$

Now look at the right hand sides of the equations we have derived by this simple syntactic transformation. They are exactly the same as the bodies of the `match`-clauses for the recursive `fun`-term for `plus`. The final connection can be made between these equations and that `fun`-term by observing that the equations are performing a case split on the first argument (called `n` in the `fun`-term): either it is `Z`, or else it is `S n'` for some `n'`. This case split is done in the `fun`-term using pattern matching. The final point to observe is that where we use `plus` on the right hand side of the second equation, we are making a recursive call to `plus`. This corresponds to the recursive call in the `fun`-term. In fact, we can observe that with each recursive call, the first argument gets smaller. It is `(S n')` to start with, and then decreases to `n'`, which is *structurally smaller* than `(S n')`. Structurally smaller means that `n'` is actually subdata of `(S n')`. While we do not need this observation now, it will be critical when reasoning with `plus`, since it implies that `plus` is a *total* function. That is, `plus` is guaranteed to terminate with a value for all inputs we give it.

2.5 Summary

In this chapter, we have seen the four basic programming features of GURU, in the setting of monomorphic programming:

- inductive datatypes, like `nat` for unary natural numbers, which has *constructors* `Z` for zero and `S` for the successor of a number;
- applications like `(S Z)` of a function (which happens to be a constructor) `S` to argument `Z`, and like `(plus x y)` for applying the function `plus` to arguments `x` and `y`;
- non-recursive, like the doubling function `fun(x:nat).(plus x x)`, and recursive ones, like `plus`; and
- pattern matching, which allows us to analyze (i.e., take apart) a piece of data (the *scrutinee*) into its subdata.

We have also seen how to run GURU on simple examples, drawing on code from the GURU standard library (like the code for `plus`).

2.6 Exercises

Bibliography

[1] Research Triangle Institute. The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002. Sponsored by the Department of Commerce's National Institute of Standards and Technology.

[2] United States Federal Bureau of Investigation. 2005 FBI Computer Crime Survey.