

An Introduction to Programming in Guru

Todd Schiller

December 6, 2007

Contents

1	Introduction	5
1.1	The Need for Program Verification	5
1.2	Functional vs. Imperative Programming Languages	5
1.3	Running the Guru Interpreter	6
2	The Guru Programming Language	7
2.1	Language Syntax	7
2.1.1	Command Syntax	7
2.1.2	Function Call Syntax	7
2.1.3	Comment Syntax	7
2.2	Defining Inductive Data Types	8
2.3	Functions	8
2.3.1	Defining Functions	8
2.3.2	Function Currying	9
3	Program Verification	11
3.1	Common Proof Directives	11
3.1.1	symm	11
3.1.2	trans	11
3.1.3	cong	11
3.1.4	join	12
3.1.5	cast	12
3.1.6	induction	12
3.2	Internal Program Verification	13
3.3	External Program Verification	13
4	Case Study: Sized Lists of Natural Numbers	15
4.1	Sized List Type Definition	15
4.2	Appending Two Sized Lists	15
A	Guru Command Listing	17
A.1	Compile	17
A.2	Define	17
A.3	Include	17
A.4	Inductive	17
A.5	Interpret	17
A.6	Set	18
A.7	Unset	18

Chapter 1

Introduction

1.1 The Need for Program Verification

Segmentation faults and memory access exceptions are no strangers to the average programmer. Most of the time, after much debugging, you can get your program to run. But, even if you get your program to run without crashing, can you be sure the program is running properly? How do you know your sort function always works? How about that binary search tree insert function you haven't written since freshman year in college?

For as long as people have been writing software, there has been a need to verify that the software works correctly. Recently, the need has manifested itself through the rise of unit testing, where parts of software are individually and "exhaustively" tested. However, these tests still cannot provide the level of certainty required by critical applications.

Guru, a programming language developed at Washington University in St. Louis, fills the need for software verification by enabling the user to prove their software's correctness. In Guru, you can write not only a function to sort a list, but also a proof that the list it returns will always be sorted. Guru eliminates the debugging headaches that most programming languages create.

1.2 Functional vs. Imperative Programming Languages

Most programmers have programmed in either C++ or Java, both of which are imperative languages. Imperative languages focus on creating pieces of data and then mutating them. For example, in C++ we may write:

```
vector<int> v(6,3);  
v.push_back(8);  
v.clear();
```

creating a vector of integers, adding another integer to the vector, and then removing all of the elements. We create and mutate.

While imperative programming languages are popular, there is another set of languages, called functional languages, which are also very powerful. In functional programming, we care less about mutating data and more about acting upon data. In Guru, we write statements such as

```
Define res := (plus three (multiply two four)).
```

and

```
Define res := (append list1 (sort list2)).
```

where we don't actually change any data – we create new data by combining functions in interesting ways. Functional programming languages, such as Haskell, LISP, and Scheme are commonly used in academia.

1.3 Running the Guru Interpreter

The easiest (and currently only) way to use Guru is by downloading the interpreter for Linux from <http://guru-lang.org>. To run a Guru program, type and run

```
java guru.Main /path/to/filename.g
```

in the Guru source directory. Guru will parse the file and interpret (evaluate) the last statement in the file unless told otherwise in the file. You can use any text editor to write Guru programs (**emacs** is a good choice).

Chapter 2

The Guru Programming Language

2.1 Language Syntax

2.1.1 Command Syntax

Each line in Guru contains either a command, a comment, or whitespace (nothing). The Guru interpreter parses each line in order. Commands each start with a directive and end with a period. Consider this simple Guru program:

```
Include "plus.g".
Set "print_parsed".
Define four := (plus two two).
Interpret four.
```

The first line tells Guru to include the definitions and commands contained within the `plus.g` file. The next line uses the `Set` command to set the `print_parsed` flag, which tells Guru to print each line it parses. Next, we define a function called `four` which always returns $2 + 2$. Finally, we tell Guru to evaluate `four` by evaluating the function and printing its value. The output of this program is:

```
Set "print_parsed".
Define four : nat := (plus two two).
(S (S (S (S Z))))
```

2.1.2 Function Call Syntax

Function calls in Guru are written differently than in languages such as C++ and Java. Instead of the function name being placed in front of the parenthesis, as in `plus(x, y)`, the name is placed inside the parenthesis with arguments separated by spaces: `(plus x y)`.

2.1.3 Comment Syntax

As in other languages, comments in Guru are ignored by the interpreter and compiler. A single line can be commented using the `%` symbol. For example,

```
%Define result := (append list1 (sort list2))
```

Segments of text can be commented in-line by using the pair `%-` and `-%`. For instance,

```
Define result := (append list1 (sort %-list1-% list2))
```

will use `list2` as an argument.

2.2 Defining Inductive Data Types

Unlike some languages, Guru does not provide built-in types for integers, booleans, and characters. Instead, Guru allows the user to define their own data types using type constructors. For example, here is the definition of the boolean type found in the standard library:

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

In this definition, the type `bool` has two term constructors, `tt` and `ff`, which represent true and false. Both of these constructors are of arity zero, meaning that they take no arguments. We can also specify constructors which take in arguments. For instance, here is the definition of a natural number type:

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

The `nat` data type has a constructor for zero and a successor constructor which takes in a natural number. The successor constructor represents the next natural number after the supplied value. Using this definition, we can represent all of the natural numbers, such as the number three:

```
Define three := (S (S (S Z))).
```

Data type constructors can have parameters of any type, including proofs and the type `type`. This feature can be used to define polymorphic data types, such as a polymorphic list:

```
Inductive list : Fun(A:type).type :=
  nil : Fun(A:type).<list A>
| cons : Fun(A:type)(a:A)(l:<list A>). <list A>.
```

The type `list` has two constructors, a constructor called `nil` which creates an empty list of type `A` and a constructor called `cons` which adds an element of type `A` to the beginning of a list of type `A`. Notice that `list` is of type `Fun(A:type).type`, meaning that the type of `list` takes a `type` as a parameter. Using these definitions, we can create a list of booleans as follows:

```
Define mylist : <list bool> := (cons bool tt (cons bool ff (nil bool))).
```

2.3 Functions

2.3.1 Defining Functions

We use the `Define` directive to define functions instead of the `Inductive` directive we used to define data types. Functions in Guru closely follow the structure of the inductive types they act on. Here is the definition of the `or` function which takes as input two booleans and returns true if at least one of the inputs is true:

```
Define or :=
  fun(x y : bool).
  match x return bool with
  tt => tt
  | ff => y
  end.
```

The `fun` directive tells Guru that what follows is a function of the variables `x` and `y`, which are both booleans. The `match` directive, which comes next, allows us to consider a variable's cases individually. In this example, we are examining the first variable `x` and will return a `bool`. What comes next is a list of the corresponding constructors (in order that they were declared in the data type definition) and what to return in each case. If `x` was made using the `tt` constructor (`x` is true), we return true. Otherwise, we return the value of `y`, so the output will be true if and only if `y` is true.

At this point, you should be wondering how the `match` statement can deal with constructors that take in arguments. The `plus` function, which adds two natural numbers, demonstrates how to use constructor arguments:

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

As per the definition of `nat` in the previous section, the `S` constructor takes in a `nat`. In the `match` statement, we create a variable `n'` to hold the value that was passed to the `S` constructor. In the example, we also use the idea of recursion, which is when a function calls itself.

The `plus` function unrolls the first variable being passed to it, taking the successor of the number created by adding the smaller number `n'` and the number `m`. Guru would evaluate `(plus (S (S Z)) (S Z))` in the following order:

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

You can nest `match` statements to examine more variables or even expressions. Here is a function which computes the maximum of two natural numbers:

```
Define max : Fun(a b:nat).nat :=
  fun max(a b :nat) : nat.
    match a return nat with
      Z => b
    | S a' => match b return nat with
              Z => a
            | S b' => (S (max a' b'))
            end
    end.
```

Notice the inclusion of the type `Fun(a b:nat).nat`. This explicitly tells Guru that the function should have the type which takes in two natural numbers and returns a natural number. This type is checked against the function body you provide to make sure the body has the expected type. As seen in previous examples, this type statement can be excluded since the interpreter can automatically infer the type.

2.3.2 Function Currying

In Guru, you don't always have to pass a function all of its required arguments. Take the `plus` function, for example. In general, you want to add two numbers together, such as `(plus two two)`. But what if you only knew one number that you wanted to add? Guru allows you to create a new function by passing a limited number of arguments:

Define `plus2 := (plus two)`.

Now we have a new function called `plus2` that takes in a natural number and adds 2 to the number. This process is called function currying.

Chapter 3

Program Verification

The primary strength of the Guru language is that it allows users to verify that their programs work properly. Using Guru's internal and external program verification, you can prove statements such as:

- merge sort will always return a sorted list
- appending two lists will result in a list of size equal to the sizes of the two lists added together
- performing an infix transversal of a binary search tree will result in a sorted list

Using Guru's verification system and bit of elbow-grease, you can show that your program works as it should in all cases.

3.1 Common Proof Directives

Proofs in Guru are essentially puzzles, with pieces coming together to create a cohesive whole. The pieces of proof are the proof directives. This section describes the most common proof directives you'll need when programming in Guru.

3.1.1 `symm`

`symm` stands for symmetry. If you have a proof `p:{A = B}` and would like to get a proof of `B = A`, then you would use `symm p`. Similarly, if you have a proof `q:{A != B}`, `symm q` would give you a proof of `B != A`.

3.1.2 `trans`

The `trans` directive captures the transitive property. If you have proofs `p:{A = B}` and `q:{B = C}`, then `trans p q` would provide a proof of `A = C`. If instead of `q:{B = C}` you had a proof `q:{B != C}`, `trans p q` would be a proof of `A != C`.

3.1.3 `cong`

Congruence reasoning is done in Guru using the `cong` directive. Assume that we have a proof `p:{A = B}` and would like to prove that `(plus two A) = (plus two B)`. We can prove this by using the `cong` directive as follows:

```
cong (plus two *) p
```

We call the `(plus two *)` an active evaluation context because of the hole we create using the `*`.

3.1.4 join

The `join` directive provides a proof that two statements are equal if they evaluate to a common statement. At the most basic level, for example, `join tt tt` would provide a proof that `tt = tt`. More useful, however, are statements such as

```
join (S (plus a b)) (plus (S a) b)
```

which proves that $(S (plus a b)) = (plus (S a) b)$. You should note that the `join` statement is dumb – it doesn't do variable substitutions or try to apply formulas. Therefore, even if $c = (S a)$, you can't `join (S (plus a b)) = (plus c a)`. There is currently work being done to make a smarter version of `join` called `hjoin`, `hyperjoin`.

3.1.5 cast

The `cast` command allows you to change the type of an expression to an equivalent type. For example, suppose we have a list of natural numbers that keep track of length. A list, `mylist`, of size four would have the type `<list four>`. We could cast the `mylist` to have type `<list (plus two two)>` using the `cast` command and a proof `p:{four = (plus two two)}`:

```
cast mylist by
  cong <list *> p
```

As you'll learn, the `cast` command is essential when internally verifying programs.

3.1.6 induction

In our functions, we match variables and constructors to their underlying constructors. When reasoning about functions, we use `induction` to consider the different cases. Suppose, for example, we have a function `eqnat` that returns `tt` if its arguments are equal and we'd like to prove that `(eqnat x x)` is always true:

```
Define eqnat_refl : Forall(x:nat).{ (eqnat x x) = tt } :=
```

We'll need to case split on `x`, so we set up our induction as follows:

```
induction(x:nat) by a b IH return { (eqnat x x) = tt } with
  Z => ...
  | S x' => ...
end.
```

The `a`, `b`, and `IH` variables can have any name. For each case, the `a` variable contains a proof that $x =$ the case. For the second case for example, `a` is a proof of $x = (S x')$. Variable `b` holds a proof that the type of the case is equal to the type of `x`. For this example, `b` is a proof of `nat = nat` (which isn't very helpful). `IH` holds the inductive hypothesis. For the base case `Z` that has no arguments, this isn't useful. However, for the case `S x'`, `[IH x']` is a proof that `(eqnat x' x') = tt`. Here is the whole proof:

```
Define eqnat_refl : Forall(x:nat).{ (eqnat x x) = tt } :=
  induction(x:nat) by x1 x2 IH return { (eqnat x x) = tt } with
    Z => trans cong (eqnat * *) x1
          join (eqnat Z Z) tt
  | S x' => trans cong (eqnat * *) x1
            trans join (eqnat (S x') (S x'))
                      (eqnat x' x')
            [IH x']
end.
```

3.2 Internal Program Verification

In internal program verification, we insert properties into our data types. These properties could be things such as a size, in the case of sized lists, or a requirement that each element is greater than the next one, in the case of a sorted list. Let's consider the latter example:

```
Inductive slist : Fun(h:nat).type :=
  snil : <slist Z>
  | scon : Fun(h:nat)(o:nat)(l:<slist o>)(p:{ (le o h) = tt}).<slist h>.
```

This sorted list has type `Fun(h:nat).type`, meaning that the type of list is dependent on a natural number (which in this case is the first item in the list). To construct a list, we need a new element `h`, another (possibly empty) list `l`, and a proof that `h` is larger than the first element in `l` (and by extension, all the elements in `l`). By maintaining this property, we can be sure that all `slists` are sorted.

The case study in Chapter 4 provides a more in-depth example of internal program verification.

3.3 External Program Verification

External program verification is the process of verifying properties about our functions outside of the function. For instance, we may want to verify that the list merge sort returns is sorted

```
Define msort_lemma : Forall(l:nlist).{ (sorted (msort l)) = tt } :=
```

or, when we append two lists, that the length of the new list is equal to the sum of the lengths of the two lists being appended:

```
Define append_lemma : Forall(a:list)(b:list).
{ (length (append a b)) = (plus (length a) (length b)) } :=
```

We create external proofs by defining an expression that has the type of what we want to prove. Section 3.1.6 contains an example of an external proof using the `induction` construct.

Chapter 4

Case Study: Sized Lists of Natural Numbers

4.1 Sized List Type Definition

We'd like our list to contain natural numbers and keep track of its size. Since we are using the natural numbers, we need to include the natural number library file:

```
Include "nat.g".
```

Next, we define our data type. Our data type will need two constructors, a base constructor for an empty list, and a constructor that can add a natural number to an already existing list.

```
Inductive nlist : Fun(size:nat).type :=  
  nnil : <nlist Z>  
| ncons : Fun(a:nat)(size':nat)(l':<nlist size'>). <nlist (S size')>.
```

The `nnil` constructor creates an empty list, and therefore has type `<nlist Z>` (where `Z` is the constructor for zero). The `ncons` constructor creates a list by adding a natural number to a `nlist` of size `size'`. Naturally, the size of this list is one larger than the size of the original list, `size'`. Therefore, the new list has size `(S size')`. We could create a list of the two numbers 2 and 1 with the definition:

```
Define mylist := (ncons two one (ncons one Z nnil)).
```

When Guru parses `mylist`, it correctly tells us that its type is `<nlist (S one)>`.

4.2 Appending Two Sized Lists

Suppose we'd like to append two of our lists together. Because of the size parameter in the data type, we are forced to write an internal proof that our new list will be the correct size. In this case, we'd like the size to be `(plus size1 size2)`, where `size1` and `size2` are the sizes of our list. The strategy for designing our function is as follows: If the first list is empty, we can just return the second list. Otherwise, we will add the first element in our list to the list created by recursively appending the rest of the first list to the second list. Here is the definition of our function, which we'll call `nappend`:

```
Define nappend :=  
fun nappend(s1:nat)(l1:<nlist s1>)(s2:nat)(l2:<nlist s2>) : <nlist (plus s1 s2)>.  
  match l1 by x y return <nlist (plus s1 s2)> with
```

```

nnil => cast l2 by cong <nlist *>
      symm trans
          cong (plus * s2) inj <nlist *> y
          join (plus Z s2) s2
| ncons a1' s1' l1' =>
  cast (ncons a1' (plus s1' s2) (nappend s1' l1' s2 l2)) by cong <nlist *>
      symm trans
          cong (plus * s2) inj <nlist *> y
          join (plus (S s1') s2) (S (plus s1' s2))
end.

```

Notice that we have to write `fun nappend(s1:nat)...` instead of `fun(s1:nat)...` since we recursively call the function. In order to get the type correct in the `nnil` case, we must cast the second list from type `<nlist s2>` to `<nlist (plus s1 s2)>`. This conversion follows easily from the fact that `s1 = 0` when `l1 = nnil`. For the other case, we must cast the newly created list from type `<nlist (S (plus s1' s2))>` to `<nlist s1 s2>`. This is also easy to prove since $(S (plus s1' s2)) = (plus (S s1') s2) = (plus s1 s2)$.

Appendix A

Guru Command Listing

A.1 Compile

Guru's interpreter is slow. For more intensive applications, Guru provides a compiler which compiles the Guru source code into Java which can then be compiled into machine code or another language. To compile an expression `exp` of zero arity to a file `filename`, use the command

```
Compile "exp" to "filename".
```

The path to the file name is relative to the file being parsed.

A.2 Define

The `Define` command is used to define a function or constant expression. Refer to section 2.3.1 for how to use this command.

A.3 Include

The `Include` command tells Guru to parse the specified file before continuing. For example,

```
Include "/lib/nat.g".
```

tells Guru to parse the file `nat.g` in the `/lib` directory. The path to the file name should be relative to the file you are currently parsing. Each file can be included only once.

A.4 Inductive

The `Inductive` command is used to define inductive data types. Refer to section 2.2 for how to use this command.

A.5 Interpret

When parsing a file, Guru does not actually evaluate any of the statements. Instead, Guru waits until you tell it to explicitly evaluate an expression using the `Interpret` command. You can only interpret expressions of arity zero (taking no argument). For example, you couldn't write the following:

```
Interpret plus.
```

since the plus function takes 2 arguments. But, you could write this:

```
Define four := (plus two two).  
Interpret four.
```

The `Interpret` command will print the result of the expression it evaluated.

A.6 Set

The `Set` command sets flags within the parser. One of the most useful flags is the `print_parsed` flag which tells Guru to print out each command it parses. The flag you are setting should be surrounded by quotation marks. For example:

```
Set "print_parsed".
```

A.7 Unset

The `Unset` command works like the `Set` command, except it turns off the flag. To turn off printing each command, for example, you would use the command:

```
Unset "print_parsed".
```