Verified Programming in Guru

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

February 11, 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 [2, 4]. 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 [3, 1]. 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 2.2.2 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*.

Since this book is being used for a class, it contains a few references to matters of course organization. Anyone reading it who is not part of such a class can, of course, just ignore those references. Also, I will usually begin chapters with a **preview**, which gives an advance peek at the chapter's material; and end with a **summary**. Feel free to skip especially the previews, if you prefer not to see the material without a full explanation: all the material is explained in detail in the chapter.

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 datatypes, 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 Preview

For those who like an overview in advance, here briefly is the syntax for the programming features we will explore in this chapter. (For those who dislike reading things without a full explanation, just skip this section and you will see it all in great detail in the rest of the chapter.)

- Inductive datatypes are declared using a command like this one, for declaring the unary natural numbers:

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

- Applications of functions to arguments are written like the following, for calling the `plus` function (which is defined, not built-in) on `x` and `y`:

```
(plus x y)
```

- Non-recursive functions like this one to double an input `x` are written this way:

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

- Pattern matching on inductive data is written as follows, where we have one `match`-clause for when `x` is `Z`, and another for when it is `S x'` for some `x'`. This is returning boolean true (`tt`) if `x` is `Z`, and boolean false (`ff`) otherwise:

```
match x with
  Z => tt
| S x' => ff
end
```

- Recursive functions like `plus` can be written with this syntax:

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

2.2 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.

2.2.1 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 write `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.

2.2.2 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 create a subdirectory called `scratch` of your home directory where you will keep scratch GURU files (we will later use such a subdirectory for homework and the project, so we will start off that way for uniformity). Then start up a text editor, and create a new file in your scratch subdirectory 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 following command in your home directory

```
guru-lang/bin/guru scratch/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.

2.3 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 *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 scratch subdirectory (of your home directory), create a file `test.g`, and begin it with

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

As for the example in Section 2.2.2 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 scratch/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.

2.3.1 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))))`.

2.3.2 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.

2.3.3 Function types

You can see the classifier that GURU computes 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
```

2.3.4 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))).
```

2.3.5 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.

2.3.6 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).
```

Finally, it is legal to nest comments.

2.4 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`), 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 2.2.2) 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).
```

2.4.1 A note on parse errors

GURU generally tries to provide detailed error messages. One exception, unfortunately, is parse errors. These are errors in syntax, for example, writing something like "(plus Z Z" where the closing parenthesis is missing. Let us see one example of the kind of error message GURU will give for a parse error. Suppose we write our iszero function, but forget to put a period after the list of arguments:

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

GURU will print an error message like the following in this case:

```
"/home/stump/guru-lang/doc/test.g", line 5, column 4: parse error.
Expected "." parsing fun term
```

The error message begins with the location of the error, including the file where the error occurred, the line number and column within that line:

```
"/home/stump/guru-lang/doc/test.g", line 5, column 4
```

Next comes a very short statement of the rough kind of error in question. This is indeed a parse error, meaning that it is arose while trying to parse the text in test.g into a legal GURU expression. Then comes the more detailed error message, which in this case as for most parse errors is pretty short:

```
Expected "." parsing fun term
```

This happens to be somewhat informative, but regrettably, especially for parse errors, that is not often the case.

2.5 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:

$$
\begin{aligned}
0+m &= m && (2.1)\\
(1+n')+m &= 1+(n'+m) && (2.2)
\end{aligned}
$$

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:

$$
\begin{aligned}
\mathtt{Z}+m &= m\\
(\mathtt{S}\ n')+m &= (\mathtt{S}\ (n'+m))
\end{aligned}
$$

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

$$
\begin{aligned}
\mathtt{(plus\ Z\ m)} &= m\\
\mathtt{(plus\ (S\ n')\ m)} &= \mathtt{(S\ (plus\ n'\ m))}
\end{aligned}
$$

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.

2.6 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 functions, 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`).

2.7 Exercises

1. The standard library files in `guru-lang/lib/` define several other functions that operate on unary natural numbers. List at least three, and say what you think they do.

2. The `plus` function defined above (Section 2.5) analyzes its first argument. Write a similar function `plus'` that also adds two natural numbers, but analyzes its second argument. Test your function by adding 2 and 3 (in unary), using the appropriate `Interpret`-command and `plus'`.

3. Define a inductive datatype called `day`, with one constructor for each day of the week. Then define a function `next_day` which takes a `day` as input and returns a `day` as output. Your function should return the next day of the week. Test your function by getting the next day after Saturday (using an `Interpret`-command).

4. Using the function `next_day`, write a function `nth_day` of type `Fun(d:day)(n:nat).day`. Your function should return the `n`'th next day after the given day `d`. For example, if `d` is Monday and `n` is 2, you should return Wednesday. Test your function by getting the 2nd day after Monday.

5. Look at the function `mult` defined in `mult.g`. Write mathematical equations corresponding to the `fun`-term for `mult`, like those labeled (2.1) in Section 2.5 above. Give a brief informal explanation of why those equations are true mathematical facts.

6. The following equations return a `tt` or `ff` depending on whether or not two `nats` are in a certain relationship to each other. What is that relationship?

   ```
   (f Z Z) = ff
   (f (S x) Z) = tt
   (f Z (S y)) = tt
   (f (S x) (S y)) = (f x y)
   ```

   Define a function (in GURU) to implement these mathematical equations. Hint: because the equations analyze each argument, you will need to use nested pattern matching. Match first on one argument, and then in each resulting `match`-clause, match on the other. Test your function on 2 and 3.

7. The following mathematical equations define the `n`-fold iteration of a unary ("one argument") function `f` on an argument `a`:

   ```
   (iter Z f a) = a
   (iter (S n) f a) = f (iter n f a)
   ```

   First, write down the type (in GURU notation) that you expect `iter` to have. Next implement `iter`, and test your function with this testcase: `(iter (S (S (S Z))) double (S Z))`, where `double` is the doubling function of Section 2.3 above (before you run GURU on this: what do you think it will compute?).

8. Write a function `first` which, given a function `P` of type `Fun(x:nat).bool` returns the smallest natural number `n` such that `(P n)` evaluates to `tt`. Hint: you will probably need to write a second *helper* function which takes as an additional argument the next number to try (for whether `P` returns `tt` or `ff` for that number).

   Test your function with the following commands. Here, `eqnat` is a function, defined in `nat.g`, which takes two `nats` as input and returns `tt` if they are equal, and `ff` otherwise). Also, `nine` is defined in `nat.g` to be 9 in unary.

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

Interpret (first fun(x:nat). (eqnat (mult x x) nine)).
```

Give an informal description of the mathematical relationship between the value this returns and 9.

Chapter 3

Equational Monomorphic Proving

The material from the last chapter is probably not entirely alien to most readers, since, although the functional programming paradigm is quite a bit different from the iterative imperative programming which most computer scientists know best, it is, after all, still programming. In this chapter, we will move farther afield from what is most of our experience as programmers, and enter the world of formal, machine-checked proofs about programs. Proofs have a lot in common with typed programs. Both are written according to certain rules of syntax, and both have a rigid compile-time semantics: programs must type check, and proofs must proof check. In GURU, the compiler attempts to compute a formula for a proof in a very similar way as it computes a type for a program. The formula in question is the one proved by the proof.

Before we begin, it should be noted that the particular style of writing proofs used here is not the only one, and indeed, there are other styles which are more widely used. For an important example, tools like COQ are based not on proofs directly, but rather on *proof scripts*. These are higher level scripts that instruct COQ on how to build the actual proof. The level of indirection introduced by proof scripts can make life easier for us program provers, at least in the short run: there is less detail that needs to be written down in a proof script than in a proof. But in the long run, proof scripts have serious problems: because they are indirect, they are very hard or impossible to read; and they can be quite brittle, breaking badly under even minor changes to the program or proof in question. In contrast, fully detailed proofs make the proof information more explicit, and so are – while still quite difficult to read, usually – somewhat more readable than proof scripts. Also, minor changes do not so immediately lead to broken proofs.

The focus in this chapter is on equational reasoning. In Chapter 5 we will look at logical reasoning. The distinction I am drawing here is between reasoning which is primarily about the equational relationships between terms (that is equational reasoning); and reasoning which is primarily about the logical relationships between formulas. An example of equational reasoning is proving that for all `nats` `x`, `x` plus zero equals `x`. An example of logical reasoning is proving that if `x` and `y` are non-zero, then so is `(plus x y)`.

The most powerful and most difficult to master method of proof is proof by datatype induction, introduced in Chapter 4. Every program prover has to cope with this proof method, and learn to apply it effectively. We will begin in this chapter, however, with much more manageable forms of proof.

For the next several chapters, we will be using very simple examples of programs, like the addition program that adds two numbers. This is certainly not the most exciting program, but it seems to provide a good balance of simplicity and interesting theorems to prove. Please be assured that we will get to more complex and realistic programming examples after we get the basics of monomorphic programming and proving down.

3.1 Preview

We consider two of the five kinds of formulas in GURU (the rest are introduced in the next chapter):

- equations, like { `(plus Z Z) = Z` }. This one states that zero (`Z`) plus zero equals zero. There are also disequations $\{t_1$ `!=` $t_2\}$ stating that two entities t_1 and t_2 are not equal.

- `Forall`-formulas, like `Forall(x:nat).{ (plus Z x) = x}`. This one states that zero plus `x` equals `x`, for any `nat x`. This formula is provable in GURU, since indeed, adding zero to any number just returns that number.

The forms of proof covered in this chapter are:

- `join` t_1 t_2, where t_1 and t_2 are terms. This tries to prove the equation $\{t_1 = t_2\}$ just by evaluating t_1 and t_2 with the GURU interpreter, and seeing if the results are equal. We use partial evaluation to evaluate terms which contain variables.
- `foralli(x:nat).P`, where `P` is another proof, is a `Forall`-introduction: it lets us prove the formula `Forall(x:nat).F`, when `P` is a proof of `F` using an arbitrary `x`, about which nothing is known. If we have a proof `P` of a `Forall`-formula, we can instantiate the `Forall` quantifier, to replace the quantified variable with a value term t, using the syntax $[P\ t]$.
- `refl t`: this proves $\{t = t\}$.
- `symm P`: if `P` proves $\{t_1 = t_2\}$, then the `symm`-proof proves $\{t_2 = t_1\}$.
- `trans P1 P2`: if `P1` proves $\{t_1 = t_2\}$ and `P2` proves $\{t_2 = t_3\}$, then the `trans`-proof proves $\{t_1 = t_3\}$.
- `cong t* P`: if `P` proves $\{t_1 = t_2\}$, then the `cong`-proof proves $\{t * [t_1] = t * [t_2]\}$, where $t * [t_1]$ is our notation (not GURU's) for the result of substituting t_1 for a special variable $*$ occurring in *term context* $t*$.
- `case`-proofs, which are syntactically quite similar to `match`-terms, and allow us to prove a theorem by cases on the form of a value in an inductive datatype.

3.2 Proof by Evaluation

Probably the simplest form of proof in GURU, and other similar tools, is proof by evaluation. For example, we have seen above that `(plus (S (S Z)) (S (S Z)))` evaluates using an `Interpret`-command to `(S (S (S (S Z))))`. Let us write `two` for `(S (S Z))` and `four` for `(S (S (S (S Z))))` – in fact, `nat.g` makes such definitions. Then we can easily record this fact as a theorem, like this:

```
Define plus224 := join (plus two two) four.

Classify plus224.
```

This code defines `plus224` to be a certain proof. The proof is a `join`-proof. The syntax for such a proof is `join` t_1 t_2, where t_1 and t_2 are terms. Here, t_1 is `(plus two two)`, and t_2 is `four`. If you run GURU on this example, it will print, in response to the `Classify`-command, the following:

```
{ (plus two two) = four }
```

This is GURU syntax for an equation. An equation is provable in GURU only if the left and right hand sides both diverge (run forever), or both converge to a common value. A `join`-proof `join` t_1 t_2 attempts to prove the equation $\{t_1 = t_2\}$ by evaluating t_1 and t_2 (using the interpreter), and checking to see if the results are equal. In this case, they are, since `(plus two two)` evaluates to `four`, and of course, `four` also evaluates to `four`.

Based on this description of how `join`-proofs work, we can already see how to prove some slightly less trivial theorems: we do not have to put a value like `four` on the right hand side, but instead, we can put some other term that evaluates to the same value as the left hand side. So we could prove the formula

```
{ (plus two two) = (plus one three) }
```

using this `join`-proof:

```
join (plus two two) (plus one three)
```

Proof by evaluation may seem rather trivial, but since in GURU we are reasoning about programs based directly on their *operational* behavior – that is, on the behavior they exhibit when they are evaluated – it is in some sense the cornerstone of all other forms of proof we might want to use. Our reasoning about programs ultimately is based on running them.

3.3 `Foralli` and Proof by Partial Evaluation

Our next proof method is a slight extension of proof by evaluation, based on the following observation: we often do not need all the inputs to be known values in order to see how a program will run. Let us recall, for example, the `plus` function:

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

We can see here that it is not necessary to know what `m` is in order to evaluate `(plus n m)`. We do need to know what `n` is, because `plus` pattern-matches on it right away. But the code for `plus` does not inspect `m` at all: it never pattern-matches on `m`, and it does not call any other functions which might do so. That suggests that we should be able to prove theorems like

```
{ (plus Z m) = m }
```

just by evaluating the application (i.e., `(plus Z m)`). Since we usually think of evaluation as requiring all arguments to be known values, we call this proof by partial evaluation (as this is the name used in computer science for evaluating programs with some arguments left as unknowns).

To demonstrate proof by evaluation, we have to be able to introduce an unknown value `m` into our proof. One way to do this is with a `foralli`-proof. This `foralli` stands for "Forall-introduction", and it is a simple way to prove that some statement is true for every `m` of some type. For our example, we will prove:

```
Forall(m:nat). { (plus Z m) = m}
```

This is a `Forall`-formula. It says that for every `m` of type `nat`, `(plus Z m) = m`. Here is how we prove this formula in GURU, using `join` and `foralli`:

```
Define Zplus := foralli(m:nat). join (plus Z m) m.

Classify Zplus.
```

If you run GURU on this, it will indeed print out, in response to the `Classify`-command:

```
Forall(m : nat) . { (plus Z m) = m }
```

Let us look at our `Zplus` proof in more detail. The proof begins with "`foralli(m:nat)`". This is quite similar to a `fun`-term. Just the way a `fun`-term shows how to compute an output from any input `m`, in a similar way a `foralli`-proof like this one shows how to prove a formula for any `m`. Logically speaking, we are going to reason about an arbitrary `nat` `m`, about which we make no constraining assumptions other than that it is indeed a `nat`. Since our reasoning will make no assumptions about `m`, it would work for any `nat` we chose to substitute for `m`. It is in this way that it soundly proves a `Forall`-formula.

In this case, we are proving the formula {`(plus Z m) = m`}. That is done by the `join`-proof, which here is the body of the `foralli`-proof. As we noted above, we can evaluate `(plus Z m)` to `m` without knowing anything about `m`. This is because partial evaluation only needs to evaluate the pattern-match on the first argument (`Z`), and it can see that the first clause of the `match`-term is taken.

3.3.1 A note on classification errors

A `join`-proof works in the case we have just been considering, only because the first argument is a known value, and `plus` only inspects that first argument. If we try switching the arguments, we will get a classification error:

```
Define plusZa := foralli(m:nat). join (plus m Z) m.

Classify plusZ.
```

If you run this in GURU, you will get a pretty verbose error message (where I have truncated parts of it with "..."):

```
"/home/stump/guru-lang/doc/test.g", line 20, column 37: classification error.
Evaluation cannot join two terms in a join-proof.
1. normal form of first term: match m by n_eq n_Eq return ...
2. normal form of second term: m


These terms are not definitionally equal (causing the error above):
1. match m by n_eq n_Eq return ...
2. m
```

Because dealing with compile-time errors is a constant part of our work in typed programming and even more so in proving, it is worth stopping to take a look at this one. First, as for the parse error example in the previous chapter (Section 2.4.1), the error message begins with the location where the error occurred, and a brief description of the kind of error it is. This is a classification error, meaning that the expression in question is syntactically well-formed, but an error arose trying to compute a classifier for it. Then comes the more detailed error message:

```
Evaluation cannot join two terms in a join-proof.
1. normal form of first term: match m by n_eq n_Eq return ...
2. normal form of second term: m
```

This says that the two terms t_1 and t_2 given to `join` do not evaluate to the same *normal forms* – that is, final values that cannot be further evaluated. We use the terminology "normal form" here instead of "value", because in partial evaluation, we might be forced to stop (partially) evaluating before we get a value. This typically happens when we try to pattern-match on an unknown. Partial evaluation gets stuck in such a case, because it does not know what the unknown looks like, and so cannot proceed with the pattern-match. The error message here is telling us that the left hand side evaluated to `match m by ...`, while the right hand side evaluated to just `m`. Indeed, this makes sense: the `plus` function wants to pattern-match on its first argument, which here is `m`, and that is where partial evaluation got stuck, just as I was mentioning.

Finally, whenever an error is due to the failure of two expressions to be the same, we get a further piece of information:

```
These terms are not definitionally equal (causing the error above):
1. match m by n_eq n_Eq return ...
2. m
```

In this case, that does not shed much light on the problem, but in other cases, this information can be very useful. "Definitionally equal" is GURU's terminology for being the same expression, ignoring certain trivial syntactic differences. For example, `one` and `(S Z)` are definitionally equal, since `one` is defined to be `(S Z)`. Differences in folding or unfolding definitions (going from `(S Z)` to `one` is folding, and vice versa is unfolding) are considered trivial, and so fall under definitional equality.

3.3.2 Terms, types, formulas, and proofs

This is a good place to highlight briefly the fact mentioned earlier that GURU has four distinct classes of expression:

- terms: these constitute programs and data, as described in Chapter 3. An example is `(plus Z Z)`.
- types: these classify terms. Examples are `nat` and `Fun(x:nat).nat`.
- proofs: these prove formulas (and formulas classify proofs). We have just seen the examples of `join`-proofs for partial evaluation and `foralli` to prove a universal.
- formulas: these make statements about terms (and, we will see later, also about types). Examples we have seen so far are equations like `{ (plus two two) = four }`; and `Forall`-formulas (also called *universal quantifications* or universal formulas), like `Forall(m:nat). { (plus Z m) = m }`.

These classes use different syntax, except for a few commonalities like variables; and so we can generally tell just by looking at a GURU expression (and not needing to run GURU, for instance) what kind of expression it is: term, type, proof, or formula. Terms and proofs are similar, and types and formulas are similar: the latter pair classifies the former pair.

3.3.3 Instantiating `Forall`-formulas

To return to our methods of proof: we have just defined (in Section 3.3) `Zplus` to be a proof of the following formula:

```
Forall(m:nat). { (plus Z m) = m }
```

When we have a proof of a `Forall`-formula, we know that something is true for every value we can substitute for the quantified variable (`m` in this case). This substitution is called an *instantiation* of the `Forall`-formula. There is a form of proof for instantiating `Forall`-formulas. It is similar to application of a `fun`-term, but is written with square brackets. To instantiate the formula proved above by `Zplus` with, for example, `three`, we write:

```
[Zplus three]
```

So, our complete `test.g` file in our scratch subdirectory of our home directory – just to refresh this after all the previous discussion – can be written like this to demonstrate this instantiation:

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

Define Zplus := foralli(m:nat). join (plus Z m) m.

Classify [Zplus three].
```

In response to the `Classify`-command, GURU will print:

```
{ (plus Z three) = three }
```

In this case, there is no need for instantiation, since we could have proved the same formula just as easily by `join (plus Z three) three`. Using instantiation is just for explanatory purposes. We will see a bit later a situation where using instantiation in a case like this can be necessary.

Now is not a bad time to see what classifies a formula:

```
Classify { (plus Z three) = three }.
```

GURU will print: `formula`. If you ask GURU what the classifier of `formula` is, it will say: `fkind`. There is no classifier of `fkind`, as it is not considered an expression. So we see that we have these classification relationships for proofs and formulas:

```
[Zplus three]  :  { (plus Z three) = three }  :  formula  :  fkind
```

This is similar to the classifications described in Section 2.3.3 above for terms and types:

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

We call `formula` and `type` *kinds* (the distinction between `tkind` and `fkind` is not important in the current version of GURU).

3.4 Reflexivity, Symmetry and Transitivity

The basic equivalence properties of equality are captured in the `refl`, `symm` and `trans` proof forms. Suppose we have these definitions, similar to one we had in Section 3.2 above:

```
Define plus224 := join (plus two two) four.
Define plus413 := join four (plus one three).
```

These proofs prove:

```
{ (plus two two) = four }
{ four = (plus one three) }
```

We can put these two proofs together using a `trans`-proof:

```
Classify trans plus224 plus413.
```

GURU will respond with:

```
{ (plus two two) = (plus one three) }
```

If we want to swap the left and right hand side of this equation, we put a `symm` around our existing proof:

```
Classify symm trans plus224 plus413.
```

GURU will respond with:

```
{ (plus one three) = (plus two two) }
```

Note that we do not use parentheses here. GURU uses parentheses exclusively for application terms. The parsing rules for `symm` and `trans` determine how things are grouped: the syntax is `symm P1` and `symm P1 P2`, where `P1` and `P2` are proofs. Judicious use of indentation is used to improve readability. These example show that there can be more than one way to prove something: we could have proved the theorems we just got using `trans` and `symm` a different way, namely with `join` directly.

Here is an example of a `refl`-proof:

```
Classify refl (fun loop(b:bool):bool. (loop b) tt).
```

This proves that

```
{  (fun loop(b : bool) : bool. (loop b) tt)
 = (fun loop(b : bool) : bool. (loop b) tt) }
```

This example is somewhat interesting, because the term `(fun loop(b : bool) : bool. (loop b) tt)` runs forever, as you will see if you run GURU with:

```
Interpret (fun loop(b : bool) : bool. (loop b) tt).
```

In most cases, the work of `refl t` can be done with `join t t`, but when `t` runs for a long time or does not terminate, `refl` is preferable or even necessary.

3.4.1 Error messages with **`trans`**-proofs

It is very easy to make a mistake trying to connect two equational subproofs using `trans`. Let us look at an example, so it is not shocking when such an error arises. Suppose we have these proofs:

```
Define plus224 := join (plus two two) four.
Define plus134 := join (plus one three) four.
```

We cannot, of course, glue them together with `trans`, because the right hand side of the equation proved by one must be the same as the left hand side of the equation proved by the other. If we try the following, we will get an error:

```
Classify trans plus224 plus134.
```

The error from GURU is:

```
"/home/stump/guru-lang/doc/test.g", line 12, column 14: classification error.
A trans-proof is attempting to go from a to b and then b' to c,
where b is not definitionally equal to b'.

1. First equation:  { (plus two two) = four }
2. Second equation: { (plus one three) = four }

These terms are not definitionally equal (causing the error above):
1. (S three)
2. (plus one three)
```

As above, we see the location of the error message first, and the fact that it is a classification error (i.e., the proof is in the correct syntax, but GURU encountered an error trying to compute a classifier for it). The error message states that the right hand side of equation 1 is not definitionally equal to the left hand side of equation 2. That is, they are not syntactically the same expression (ignoring certain minor syntactic differences). Then we see the last part of the error message:

```
These terms are not definitionally equal (causing the error above):
1. (S three)
2. (plus one three)
```

The first term listed is definitionally equal to `four`, the right hand side of equation 1. The second term is the left hand side of equation 2. GURU expects these to be definitionally equal, but they are not.

3.5 Congruence

Along with reflexivity, symmetry, and transitivity, the main equational reasoning inference is *congruence*. Consider again our simple proof `plus224` from above:

```
Define plus224 := join (plus two two) four.
```

As we have seen several times now, this proves:

```
{ (plus two two) = four }
```

From this, we can also prove:

```
{ (S (plus two two)) = (S four) }
```

That is, we can prove that the successor of two plus two is equal to the successor of four (namely five). What we are doing is substituting the left and right hand sides of our first equation into a pattern `(S *)` to get the second equation. The pattern is called a *term context*, and it uses the special symbol $*$ to indicate the position or positions where the substitution should take place. With these ideas, we can understand the `cong` form of proof in GURU which formalizes this congruence reasoning:

```
Classify cong (S *) plus224.
```

GURU will respond with the following, as expected:

```
{ (S (plus two two)) = (S four) }
```

As another demonstration of `cong`, try the following in GURU:

```
Classify cong (plus * *) plus224.
```

3.6 Reasoning by Cases

With the proof forms we have seen so far, we cannot prove very exciting theorems. For interesting theorems, we usually have to use induction. Induction involves a form of reasoning by cases. So as a warmup for induction, we will consider now a proof construct for reasoning by cases, without doing induction. This is the `case` proof construct.

To demonstrate `case`-proofs, let us look at a definition of boolean negation:

```
Define not :=
  fun(x:bool).
    match x with
      ff => tt
    | tt => ff
    end.
```

This `Define`-command defines `not` to be a function (i.e., a `fun`-term) that takes input `x` of type `bool` and pattern-matches on it. If `x` is `ff` (boolean true), then we return `tt` for its negation, and vice versa (if it is `tt`, we return `ff`). Notice that we have to list the `match`-clauses in this order, since that is the order in which the constructors for the `bool` datatype are declared, in `bool.g`:

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

We will now see how to prove the following slightly interesting theorem:

```
Forall(b:bool). { (not (not b)) = b }
```

Informally, the reasoning needed to prove this theorem is very simple. Suppose we have an arbitrary value `b` of type `bool`. Either `b` is `ff` or it is `tt`, given the declaration of the `bool` datatype. So suppose `b` is `ff`. Then `(not (not b))` is equal to `(not (not ff))`, which evaluates to `ff`, which is again equal to `b`. So by transitivity of equality, `(not (not b)) = b`. We can write this down (informally) with the following three equational steps:

```
(not (not b)) = (not (not ff)) = ff = b
```

Similar reasoning applies in the case where `b` is `tt`.

We can write this proof formally in GURU, as follows:

```
Define not_not : Forall(b:bool). { (not (not b)) = b } :=
  foralli(b:bool).
    case b with
      ff => trans cong (not (not *)) b_eq
            trans join (not (not ff)) ff
                  symm b_eq
    | tt => trans cong (not (not *)) b_eq
            trans join (not (not tt)) tt
                  symm b_eq
    end.
```

You can find this theorem in `guru-lang/lib/bool.g`. We will walk through this and see how it works. First, this is a `Define`-command, but it uses one feature of `Define` that we have not seen previously. We can list a classifier that the defined expression is supposed to have, and GURU will check for us that it does. So what we have written is of the form:

```
Define not_not : expected_classifier := proof.
```

GURU will compute a formula for the proof, and then make sure that that formula is definitionally equal (i.e., equal ignoring a few minor syntactic variations, like folding and unfolding defined symbols) to `expected_classifier`.

Looking now at the actual proof that is given in the definition, it is:

```
  foralli(b:bool).
    case b with
      ff => trans cong (not (not *)) b_eq
            trans join (not (not ff)) ff
                  symm b_eq
    | tt => trans cong (not (not *)) b_eq
            trans join (not (not tt)) tt
                  symm b_eq
    end.
```

This is a `foralli`-proof (see Section 3.3 above). We are assuming an arbitrary value `b` of type `bool`, just as in our informal proof above. The body of the `foralli`-proof is a `case`-proof, again corresponding to our informal case reasoning above. The syntax for a `case`-proof is very similar to the syntax for a `match`-term. We are performing a case analysis on the scrutinee `b`, and we have one clause for each form of `b`. The body of each `case`-clause gives the proof of the theorem in the case where `b` equals the pattern listed for the clause. To understand this better, let us look at the proof given as the body of the clause for `ff`:

```
trans cong (not (not *)) b_eq
trans join (not (not ff)) ff
      symm b_eq
```

This consists of the following three subproofs, which are glued together with `trans` (Section 3.4):

1. `cong (not (not *)) b_eq`
2. `join (not (not ff)) ff`
3. `symm b_eq`

Let us try to compute what theorem is proved by each of these subproofs. They all use familiar syntax, except that at this point, we have not seen what `b_eq` is. This is an *assumption variable* introduced by the `case`-proof. If the scrutinee is a symbol (as `b` is), then the `case`-proof introduces two assumption variables about `b`: `b_eq` and `b_Eq`. We will not use the second until quite a bit later. The variable `b_eq` can be used as a proof in the body of each `case`-clause that the scrutinee is equal to the pattern. For indeed, when this code is run, if we enter the body of the clause for `ff`, say, that can only be because `b` is, in fact, `ff`. So for the first of our three subproofs, let us determine what formula it proves. Our assumption variable `b_eq` proves

```
{ b = ff }
```

and we are applying `cong` to this proof. So the first subproof (i.e., "`cong (not (not *)) b_eq`") proves

```
{ (not (not b)) = (not (not ff)) }
```

The second subproof is a `join`-proof, proving

```
{ (not (not ff)) = ff }
```

Finally, the third subproof is `symm b_eq`. We know `symm P` just switches the left and right hand side of the equation proved by `P`. So here, our `symm`-proof proves

```
{ ff = b }
```

We can see that putting these three steps together with transitivity corresponds to the three informal equational reasoning steps we saw above:

```
(not (not b)) = (not (not ff)) = ff = b
```

This does indeed prove { `(not (not b)) = b` }, as required, and completes the proof in the `ff` `case`-clause. The proof in the `tt` `case`-clause is similar, except that there, our assumption variable `b_eq` proves

```
{ b = tt }
```

and the rest of the proof uses `tt` instead of `ff` appropriately.

3.7 Summary

The forms of proof we have seen in this chapter are:

- proof by evaluation and proof by partial evaluation, both written in GURU using the syntax `join t1 t2`, which tries to prove { `t1 = t2` } by evaluating the two terms to a common normal form. A normal form is an expression which cannot evaluate further, either because it is a value like three or because evaluation is stuck trying to pattern match on a variable (during partial evaluation).

- `foralli`-proofs and instantiation proofs, the latter written like term applications except with square brackets instead of parentheses. These are for proving a `Forall`-formula, and for substituting a value for the quantified variable in a proven `Forall`-formula, respectively.

- equivalence reasoning and congruence reasoning, using `refl`, `symm`, `trans`, and `cong`.

- `case`-proofs for reasoning by cases on the form of a piece of inductive data.

3.8 Exercises

1. Include `guru-lang/lib/mult.g`, and prove the following theorems by evaluation. Here, `lt` is less-than and `le` is less-than-or-equal on `nats`, defined in `nat.g`:

   - `{ (mult zero three) = zero }`
   - `{ (lt zero three) = tt }`
   - `{ (le one three) = tt }`

2. Now prove the following, using `foralli` and `join`:

   ```
   Forall(x:nat). { (mult Z x) = Z }
   ```

3. Prove the following formula using `foralli` and `join`:

   ```
   Forall(x : nat)(y : nat) . { (lt Z (plus (S x) y)) = tt }
   ```

   Note that you can introduce multiple variables in a `foralli`-proof in a similar way as you accept multiple inputs in a `fun`-term.

4. The `and` function defined in `bool.g` computes the conjunction of two `bools`. Prove the following theorem about `and`:

   ```
   Forall(x:bool). { (and ff x) = ff }
   ```

5. Formulate and prove the theorem that `and`'ing any boolean with itself just returns that same value.

6. Prove the following formula using `foralli` and then a `case`-proof scrutinizing the universally quantified variable `x`:

   ```
   Forall(x : nat) . { (le Z x) = tt }
   ```

7. Consider the following datatype for buildings on The University of Iowa Pentacrest:

   ```
   Inductive penta : type :=
     MacBride : penta
   | MacLean : penta
   | Schaeffer : penta
   | Jessup : penta
   | OldCapitol : penta.
   ```

   - Define a function `clockwise` that takes a `penta` as input, and returns the next building in clockwise order (looking down on the Pentacrest) around the perimeter. We will consider the Old Capitol to be clockwise from itself.
   - Similarly, define a function `counter` that returns the next building in counter-clockwise order, again considering the Old Capitol to be counter-clockwise from itself.
   - Formulate and prove the theorem that going clockwise and then counter-clockwise gets you back to the same building.

Chapter 4

Inductive Equational Monomorphic Proving

In this chapter, we take our first look at proof by induction in GURU. We will use induction to prove equational theorems about monomorphic functions. In later chapters we will prove more complex theorems about polymorphic and dependently typed functions, but beginning with this simple setting will make induction in GURU easier to master. When we wish to prove properties of recursive functions – which are, of course, the most interesting functions and the ones we have to use to accomplish most non-trivial tasks – we generally need induction. Proof by induction and definition by recursion are very similar. Indeed, a deeper understanding of the connection helps in mastering induction, so we will start with that. Then we will see several examples of proof by induction in GURU.

4.1 Preview

The syntax for `induction`-proofs is demonstrated by this skeleton for induction on a `nat` `n`:

```
induction(n:nat) return F with
  Z => P1
| S n' => P2
end
```

This will prove `Forall(n:nat).F`, where `F` is a formula mentioning `n`; assuming that `P1` and `P2` are the base and step case proofs of `F`. In each of these (`P1` and `P2`), two special variables are available, which the `induction`-proof automatically declares:

- `n_eq`: in the body of each clause, this is an assumption that `n` equals the pattern of the clause (`Z` or `(S n')`, respectively).
- `n_IH`: in the step case (`P2`), this serves as a proof of the induction hypothesis. It proves `Forall(n:nat).F`, but may only be instantiated with `n'`, the subdatum (smaller piece of data) of `n`.

4.2 Induction and Terminating Recursion

In GURU, we are allowed to define functions by *general recursion*: we can make recursive calls on any inputs we want, even if that means the function might not terminate. For example, we saw the following simple example of a looping function in Chapter 3:

```
fun loop(b:bool):bool. (loop b)
```

This function calls itself recursively on the input it was given. Hence, when we call this function on an argument `b`, it will loop forever, as it tries again and again to evaluate the term `(loop b)`.

If we want to define a function that terminates on all inputs, however, we cannot use recursion in an unrestricted manner. A typical simple restriction to ensure (uniform) termination is the following:

- The function has a single input called the *parameter of recursion*.
- In every recursive call in the function's code, the argument passed for the parameter of recursion is smaller than the input parameter. In more detail, recursive calls can only be made on the parameter of recursion's subdata, obtained via pattern matching.

Functions that satisfy this requirement are called *structurally terminating*. For example, the `plus` function we saw is structurally terminating:

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

The parameter of recursion is input `n`. In the recursive call in the second `match`-clause, the argument given for the parameter of recursion is `n'`. This is indeed the subdatum of `n`, obtained by pattern-matching. So `plus` is structurally terminating. Functions like this are indeed guaranteed to terminate for all inputs (as long as any other functions they call are also terminating), because the argument given for the parameter of recursion cannot get smaller and smaller forever: eventually there are no more subdata to extract. In the case of `nat`, for example, we eventually reach `Z`, which has no subdata.

We will be interested later in proving termination of functions like `plus`. For now, though, the reason to consider structurally terminating functions is that they are very similar to induction proofs. Indeed, proof by induction can be thought of as the structurally terminating recursive construction of a proof. For example, for natural number induction, which the reader has probably seen in a discrete mathematics class, our goal is to prove that some formula $\phi(x)$ mentioning x is true for all natural numbers x. Proof by induction tells us that to do this, it is sufficient to prove:

- $\phi(Z)$
- $\phi(n)$ implies $\phi(S\ n)$.

The first case is called the base case, while the second is called the inductive (or step) case. Informally, proof by induction is sound for the following reason. Every natural number x is constructed by applying S some finite number of times (possibly zero) to Z. To prove $\phi(x)$ for a particular such x, we must merely use the second fact above n times, starting with the first fact. For example, if we want to prove $\phi(S\ (S\ (S\ Z)))$ (that is, $\phi(3)$), we reason like this:

- We have $\phi(Z)$ by the first fact above.
- We get $\phi(S\ Z)$ from $\phi(Z)$, which we just derived, using the second fact above.
- We get $\phi(S\ (S\ Z))$ from $\phi(S\ Z)$, which we just derived, using the second fact above.
- We get $\phi(S\ (S\ (S\ Z)))$ from $\phi(S\ (S\ Z))$, which we just derived, using the second fact above.

Another way to view what is happening with proof by induction is to think of the step case as making a recursive call to the proof. That is, we are trying to prove $\phi(S\ n)$, but we are allowed to use the assumption, usually called the *induction hypothesis* (IH), that $\phi(n)$ holds. Here we can see the structural decrease in the *parameter of induction* from $(S\ n)$ to n. This is similar to what we saw in the case of structural termination of recursive functions. When we appeal to the induction hypothesis, it is like we are making a structurally recursive call to the proof we are in the middle of writing. Even though this looks like circular reasoning, it is sound for the same reason that structurally terminating functions terminate: the argument given for the parameter of induction is getting structurally smaller. This cannot happen forever, so eventually the self-referential reasoning will "bottom out"; that is, will terminate in a base case.

Most students who have not studied induction previously find it takes a while to get used to. We will continue to try to provide intuition for why induction is sound, as we turn now to simple examples of induction proofs in GURU.

4.3 A First Example of Induction, Informally

In Section 3.3, we proved the following formula in GURU using partial evaluation and `foralli`:

```
Forall(m:nat). { (plus Z m) = m}
```

We also saw in Section 3.3.1 that a similar proof did not succeed in proving

```
Forall(m:nat). { (plus m Z) = m}
```

The reason is that as we have defined it, `plus` performs a pattern-match on its first argument. For the theorem we succeeded in proving, the first argument is `Z`, and so partial evaluation can evaluate the pattern-match, even though the second argument is just a variable `m`. For the theorem we failed to prove, partial evaluation gets stuck trying to pattern-match on the variable `m`, and so the proof cannot go through.

Here, we will see how to prove the second theorem by induction. Let us begin with a proof in English, and then see how this can be written in GURU. We wish to prove `Forall(m:nat).{ (plus m Z) = m}` by induction on `m`. For this, as described in Section 4.2, it suffices to prove the following base case and step case:

- `{(plus Z Z) = Z }`
- If `{(plus n Z) = n }`, then also `{(plus (S n) Z) = (S n) }`

The base case is easily proved by partial evaluation. For the step case, we first assume `{(plus n Z) = n }`. This is the induction hypothesis. Now we must prove, under this assumption, that `{(plus (S n) Z) = (S n) }`. We can prove by partial evaluation that

```
{ (plus (S n) Z) = (S (plus n Z)) }
```

This follows because, as we noted before, `plus` is pattern-matching on its first argument, so partial evaluation can proceed past that pattern-match, up to the recursive call. Now using our induction hypothesis and congruence, we can prove

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

Chaining the two equational steps we have done with transitivity, we conclude the desired formula:

```
{ (plus (S n) Z) = (S n) }
```

4.4 Example Induction in GURU

Now let us write the above proof in GURU. In fact, since the theorem we are proving, while simple, turns out to be rather important, we already have a proof of it in `guru-lang/lib/plus.g`:

```
Define plusZ : Forall(n:nat). { (plus n Z) = n } :=
  induction(n:nat) return { (plus n Z) = n } with
    Z => trans cong (plus * Z) n_eq
         trans join (plus Z Z) Z
               symm n_eq
  | S n' => trans cong (plus * Z) n_eq
            trans join (plus (S n') Z) (S (plus n' Z))
            trans cong (S *) [n_IH n']
                  symm n_eq
  end.
```

Let us walk through this. First, this is a `Define`-command, just like ones we have already seen. We are defining `plusZ`, and instructing GURU to confirm that what we are defining it to equal has the classifier listed between the colon and the colon-equals, namely `Forall(n:nat).{ (plus n Z) = n }`. Then, after the colon-equals, comes the proof:

```
induction(n:nat) return { (plus n Z) = n } with
  Z => trans cong (plus * Z) n_eq
       trans join (plus Z Z) Z
             symm n_eq
| S n' => trans cong (plus * Z) n_eq
          trans join (plus (S n') Z) (S (plus n' Z))
          trans cong (S *) [n_IH n']
                symm n_eq
end.
```

This begins with the `induction` keyword. Next comes the parameter of induction, with its type. Notice that this looks very similar to the argument list for a `fun`-term. We will see more complex versions of the argument list later, but this is typical for now. Then comes a `return`-clause, consisting of the `return` keyword, followed by the classifier `{ (plus n Z) = n }`. This classifier is the formula proved, for all `n`, by the induction proof. Each clause of the `induction`-proof must prove this formula. GURU requires a `return`-clause here for the same reason that it requires recursive `fun`-terms to specify their return type: it makes bottom-up type checking easy. Without this `return`-clause, GURU would have to infer the induction hypothesis. With the `return`-clause, however, the induction hypothesis can be easily computed.

After the `return`-clause, we have the keyword `with`, as for pattern-matching and `case`-proofs. Then come the `induction`-clauses, one for each constructor of the datatype, in the order the constructors are listed in the datatype's declaring `Inductive`-command. Let us look at the bodies of those `induction`-clauses.

4.4.1 The base case

The first subproof is for when `n` is zero:

```
trans cong (plus * Z) n_eq
trans join (plus Z Z) Z
      symm n_eq
```

This proof consists of three subproofs, glued together with `trans`:

- `cong (plus * Z) n_eq`
- `join (plus Z Z) Z`
- `symm n_eq`

Remember that we are obliged to prove `{ (plus n Z) = n }` in this clause (and in the clause for `S`). Just as in a `case`-proof (Section 3.6), we get an assumption variable `n_eq` that we can use in each clause as a proof that the parameter of induction (i.e., `n`) equals the pattern in the clause. So in the body of the clause for `Z`, we have

```
n_eq : { n = Z }
```

The first step uses `n_eq` and congruence to prove:

```
{ (plus n Z) = (plus Z Z) }
```

The second step uses proof by evaluation (i.e., `join`) to prove:

```
{ (plus Z Z) = Z }
```

Finally, the third step proves

```
{ Z = n }
```

Chaining these steps together, we have this reasoning:

```
(plus n Z)   =   (plus Z Z)   =   Z   =   n
```

Notice that this is a bit more detailed than in the informal proof above, because we have to map from `n` to `Z` and back using `n_eq`.

4.4.2 The step case

The second subproof of our example `induction`-proof is for when `n` is `(S n')`:

```
trans cong (plus * Z) n_eq
trans join (plus (S n') Z) (S (plus n' Z))
trans cong (S *) [n_IH n']
      symm n_eq
```

Just as in the base case, we must map from `n` to `(S n')` and back using `n_eq`. That is what is happening in the first and last of the four subproofs glued together by `trans`. So let us look at the middle two:

- `join (plus (S n') Z) (S (plus n' Z))`
- `cong (S *) [n_IH n']`

The first is a proof by partial evaluation, corresponding to the first step we took above in our informal proof (Section 4.3). The second uses congruence and the induction hypothesis. The induction hypothesis is `n_IH`, whose name is automatically derived from the name of the parameter of induction, as for `n_eq`. In the GURU formalization of induction, the induction hypothesis proves exactly the same theorem as the proof. So in this case, we have

```
n_IH : Forall(n:nat) . { (plus n Z) = n }
```

But as discussed above, the use of the induction hypothesis is restricted. We can only instantiate the `Forall`-quantifier here with a strict subterm (subdatum) of the parameter of induction. The GURU compiler will ensure that this restriction is met, and report an error if the induction hypothesis is not instantiated accordingly. So in our subproof, we have `[n_IH n']` for the instantiation of the `Forall`-formula with `n'`. Since `n'` is indeed a strict subterm (from the pattern of the `induction`-clause for `n`), this is a legal use of the induction hypothesis. Finally, we use `cong` similarly to the way we used congruence in our informal proof above.

4.5 A Second Example Induction Proof in Guru

Let us look now at a second example `induction`-proof. The proof we will be constructing in this section can also be found as the lemma `plusS` in `guru-lang/lib/nat.g`. We wish to prove the formula

```
Forall(n m : nat). { (plus n (S m)) = (S (plus n m))}
```

Here we are faced with a small puzzle: we have two universally quantified variables `n` and `m`, so which one should be our parameter of induction? Furthermore, whichever variable we select for the parameter of induction, how do we handle the other variable? The answers to these questions are relatively easy to reach for this example, but for other more complicated ones can be trickier. The basic hint we should always keep in mind is:

Theorem Proving Hint 1 *As a first idea, we should choose our parameter of induction to be a variable which is used as the parameter of recursion (see Section 4.3) for one of the functions in our theorem.*

Of course, this hint only applies when a function has a (structurally decreasing) parameter of recursion. Not all interesting recursive functions do. Also, this hint does not tell us exactly what to do when there are multiple functions mentioned in the theorem, since then we may have several different variables all used as parameters of recursion. Nevertheless, induction and recursion do go hand in hand, and so a rough rule of thumb is to perform induction on a variable which is analyzed by recursion.

To return to our second example theorem: of our two variables, `n` and `m`, only one is used as a parameter of recursion by a call to `plus`: this is `n`. Our definition of `plus` analyzes its first argument, and we pass `n` as this argument in both recursive calls in the theorem (i.e., `(plus n (S m))` on the left hand side of the equation, and `(plus n m)` on the right). So following Theorem Proving Hint 1, we should try doing induction on `n`. So we start our proof with "`induction(n:nat).`". Now we must list the `return`-clause for our `induction`-proof, as described in our first example above. This `return`-clause must give the rest of the formula being proved. So our `induction`-proof starts with:

```
induction(n:nat)
  return Forall(m : nat). { (plus n (S m)) = (S (plus n m))}
```

The theorem we are proving, also called our *goal* formula, begins with "`Forall(n m:nat).`", which GURU views as definitionally equal to "`Forall(n :nat).  Forall(m:nat).`". That explains why, once we have started proving our goal formula with "`induction(n:nat)`", the `return`-clause starts with a `Forall`-quantification of the variable `m`.

There is really no choice what to write next; we have to have clauses for each way of constructing the `nat n` (after the keyword `with`):

```
induction(n:nat)
  return Forall(m : nat). { (plus n (S m)) = (S (plus n m))}
with
  Z => ...
| S n' => ...
end
```

We are not ready yet to fill in the bodies of the clauses, where I have written "..." (not GURU syntax). A good strategy for developing a proof like this is to put something – anything, or almost anything – in for those "...", so that GURU can parse our proof and start trying to classify it. I find this is more effective and less frustrating than writing a large proof and then trying to get it to go through the GURU compiler all at once. It is better to write the proof incrementally, and get each piece of it through GURU, since then the inevitable error messages you are dealing with are ones concerning the proof you are just focused on writing (not one you wrote twenty minutes ago when you started your proof). A good placeholder to put instead of "..." is `truei`. This proves the formula `True`. It is indeed a proof, so the GURU parser can parse it. Of course, it does not prove the right theorem yet, so we will definitely get a classification error. But that is alright, since we will gradually fill in more and more of the proof properly, and eventually eliminate all those errors. This gives rise to:

Theorem Proving Hint 2 *Write down a skeletal proof using `truei` as a placeholder for missing subproofs, and gradually refine it to a proof that can pass GURU's proof checker by replacing those uses of `truei` with the correct subproof.*

So in this example, we could write the following `Classify`-command:

```
Classify
induction(n:nat)
  return Forall(m : nat). { (plus n (S m)) = (S (plus n m))}
with
```

```
  Z => truei
| S n' => truei
end.
```

If we run this through GURU, as expected we will get this classification error:

```
"/home/stump/guru-lang/doc/ch4.g", line 7, column 2: classification error.
The classifier computed for the body of a case in an induction-proof
is different from the expected one.
1. computed classifier: True
2. expected classifier: Forall(m : nat) . { (plus n (S m)) = (S (plus n m)) }
3. the case: Z

These terms are not definitionally equal (causing the error above):
1. Forall(m : nat) . { (plus n (S m)) = (S (plus n m)) }
2. True
```

This exactly describes what we knew would happen: we have put a proof of `True` in each of the clauses of our `induction`-proof, where a proof of `Forall(m : nat) . { (plus n (S m)) = (S (plus n m)) }` was expected.

Now, let us start refining our proof by replacing some of these `truei`-proofs with the correct proofs for the cases. When `n` is `Z`, we know that `(plus n m)` equals `m`, and similarly `(plus n (S m))` equals `(S m)`. That is because of how `plus` partially evaluates when its first argument is `Z`. So our proof for the `Z` case is similar to proofs we did above. We start it with `foralli`, to introduce the universal variable `m`:

```
Classify
induction(n:nat)
  return Forall(m : nat). { (plus n (S m)) = (S (plus n m))}
with
  Z =>
  foralli(m:nat).
  trans cong (plus * (S m)) n_eq
       trans join (plus Z (S m)) (S (plus Z m))
              cong (S (plus * m)) symm n_eq
| S n' => truei
end.
```

When we run this proof through GURU, we get this error message:

```
"/home/stump/guru-lang/doc/ch4.g", line 25, column 2: classification error.
The classifier computed for the body of a case in an induction-proof
is different from the expected one.
1. computed classifier: True
2. expected classifier: Forall(m : nat) . { (plus n (S m)) = (S (plus n m)) }
3. the case: (S n')

These terms are not definitionally equal (causing the error above):
1. Forall(m : nat) . { (plus n (S m)) = (S (plus n m)) }
2. True
```

Notice that item (3) listed in the message has changed from our first error message. GURU proof-checks the `induction`-clauses in order starting with the one which is textually first. We have successfully gotten the proof for the `Z` case through the proof checker, since our error message now concerns the second case (the one for `(S n')`).

Now we are ready to tackle the S case. We can expect we will need to use our induction hypothesis, since we make a recursive call in the S case for plus, and uses of the induction hypothesis tend to mirror recursive calls. Let us see informally what our reasoning will be:

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

The first step is by partial evaluation. The second step uses the induction hypothesis to get:

```
{ (plus n' (S m)) = (S (plus n' m)) }
```

The second step then uses congruence. The third step is again by partial evaluation. Formalizing this reasoning in GURU, we get the following final proof, which successfully checks:

```
Classify
induction(n:nat)
  return Forall(m : nat). { (plus n (S m)) = (S (plus n m))}
with
  Z =>
  foralli(m:nat).
  trans cong (plus * (S m)) n_eq
       trans join (plus Z (S m)) (S (plus Z m))
             cong (S (plus * m)) symm n_eq
| S n' =>
  foralli(m : nat).
  trans cong (plus * (S m)) n_eq
  trans join (plus (S n') (S m)) (S (plus n' (S m)))
  trans cong (S *) [n_IH n' m]
  trans join (S (S (plus n' m))) (S (plus (S n') m))
        cong (S (plus * m)) symm n_eq
end.
```

We have a new subproof in the `S n'` clause, corresponding to the informal proof we just did above. We have to map from `n` to `(S n')` using the assumption variable `n_eq`, just as above. Then we do some partial evaluation (with `join`), then use the appropriately instantiated induction hypothesis (that is `[n_IH n' m]`), do some more partial evaluation, and then map back from `(S n')` to `n`.

4.6 Commutativity of Addition in GURU

As a final example, let us use the lemmas proved in the previous two sections to prove commutativity of addition:

```
Forall(n m:nat). { (plus n m) = (plus m n) }
```

The proof is in `guru-lang/lib/plus.g`, and it uses the following lemmas, which we proved above and which are also defined in `plus.g`:

```
plusZ : Forall(n:nat). { (plus n Z) = n }
plusS : Forall(n m : nat). { (plus n (S m)) = (S (plus n m))}
```

Indeed, we proved those lemmas just so we could prove commutativity of `plus`. The informal reasoning is as follows. We proceed by induction on `n`, and then in each case assume arbitrary `m`. So for the base case we must prove

```
(plus Z m) = (plus m Z)
```

The left hand side partial-evaluates to `m`, while the right hand side is equal to `m` by our `plusZ` lemma.

For the step case, we must prove

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

under the assumption (the induction hypothesis) that { `(plus n' m) = (plus m n')` }. Our equational reasoning is as follows:

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

The first step is by partial evaluation. The second is by the induction hypothesis (and congruence). The third is by our `plusS` lemma. That concludes our informal proof.

The proof in GURU mirrors this reasoning, although in a bit more detailed way:

```
induction (n : nat) return Forall(m : nat).{ (plus n m) = (plus m n) } with
  Z => foralli(m : nat).
       trans cong (plus * m) n_eq
       trans join (plus Z m) m
       trans cong * symm [plusZ m]
             cong (plus m *) symm n_eq
| S n' => foralli(m : nat).
          trans cong (plus * m) n_eq
          trans join (plus (S n') m) (S (plus n' m))
          trans cong (S *)  [n_IH n' m]
          trans cong * symm [plusS m n']
                cong (plus m *) symm n_eq
end
```

This is not terribly fun to read, but we can spot the uses of the induction hypothesis `[n_IH n' m]` in the `(S n')` case, and the uses of `plusZ` and `plusS`.

4.7 Summary

We have seen several examples of `induction`-proofs for proving equations about monomorphic programs like `plus`. `Induction`-proofs are similar to structurally terminating recursive functions: uses of the induction hypothesis are like recursive calls, which construct the desired proof for a structurally smaller piece of data. We have seen also several theorem proving hints, which can help make it easier to tackle a proof.

4.8 Exercises

As you browse through the GURU standard library, you will come across proof methods we have not seen yet, particularly `hypjoin`. For these exercises, you should use only the proof methods we have seen so far in this book.

1. Include `guru-lang/lib/mult.g` and prove by induction on `n`:

   ```
   Forall(n:nat).{ (mult n Z) = Z }
   ```

2. Including `guru-lang/lib/plus.g`, prove the following, but do not use induction. Just use existing theorems in `plus.g` (in particular, `plus_assoc` and `plus_comm`):

   ```
   Forall(x y z:nat). { (plus x (plus y z)) = (plus z (plus y x)) }
   ```

3. Again including `mult.g`, prove the following by induction, first determining which variable you should do induction on:

```
Forall(x y z :nat).{(mult (plus x y) z) = (plus (mult x z) (mult y z))}
```

Hint: my proof uses the lemma `plus_assoc` from `guru-lang/lib/plus.g` (and that is the only lemma I need).

4. The exclusive-or function is defined as `xor` in `guru-lang/lib/bool.g`. Prove the following (this does not need induction):

   ```
   Forall(x y : bool). { (xor (not x) y) = (not (xor x y)) }
   ```

5. The `mod2` function defined in `guru-lang/lib/pow.g` takes a `nat n` as input, and returns `ff` if `n` is even, and `tt` if `n` is odd. In this problem, we will prove the following non-trivial property of `mod2`:

   ```
   Forall(n m : nat). { (mod2 (plus n m)) = (xor (mod2 n) (mod2 m)) }
   ```

   An intuitive way to view this theorem is as saying how the parity of numbers is combined when the numbers are added. When we add an even number and an even number we get another even number; when we add odd and even we get odd; and when we add odd and odd we get even. With `ff` for even and `tt` for odd, we see that this description corresponds to exclusive-or: `ff` (even) and `ff` (even) gives `ff` (even); `ff` (even) and `tt` (odd) gives `tt` (odd); and `tt` (odd) and `tt` (odd) gives `tt` (odd). This is, of course, a fact about addition of numbers.

   To prove this theorem, first identify which variable you should most likely do induction on. During the course of the proof, I found I needed to use the lemma proved in the previous problem.

Chapter 5

Logical Monomorphic Proving

The last two chapters focused on equational proofs about monomorphic programs. That is, we were just trying to prove universally quantified equations, like `Forall(x y:nat).{(plus x y) = (plus y x)}`. Of course, there are other kinds of logical statements we would like to make. For one simple example, we might like to prove that if `x` plus `y` equals zero, then `x` must be zero, for `x` and `y` of type `nat` (of course, `y` must also be zero in this case). An "if-then" statement is called an *implication*. In GURU, implications are written with `Forall`, which turns out to make notation a bit more concise. So the statement would be written this way in GURU:

```
Forall(x y:nat)(u : { (plus x y) = Z }). { x = Z }
```

We need some other proof constructs to reason in the presence of implications. These will be introduced in this chapter. We will also see *conjunctions*, for "and" statements; and existential formulas, for saying that something exists with a certain property. As usual, we will try these out with several examples.

5.1 Preview

In this chapter we will see these additional kinds of formulas:

- Implications, which say that `F1` implies `F2`, are written as `Forall(u:F1).F2`. This can be thought of as saying that for any proof `u` of `F1`, `F2` is true.
- `Exists`-formulas, like `Exists(y:nat). { (plus y (S Z)) = Z }`. This one states that there is a `nat` `y` such that `y` plus one (that is, "`(S Z)`") equals zero. This is not provable in GURU, because for natural numbers, there is no number we can add to one to get zero. Of course, if we had negative numbers, we could prove this. But we are making a statement about `nats` `y`, not integers `y`.

The forms of proof covered in this chapter are:

- Implication-introduction and elimination are done using `Forall`-introduction and elimination.
- `existsi t F* P`, where `t` is a term, `F*` is a formula context, and `P` is a proof. This is to prove the formula `Exists(x:nat).F*[x]`. The situation is that we have a term `t` and a proof `P` that that term has a certain property. The property is described using a formula context, which is a formula containing the special symbol `*`. A shorthand for proving a conjunction (written as an `Exists`-formula) is `andi P1 P2`.
- `existse P1 P2`. If `P1` is a proof of the formula `Exists(x:nat).F`, and if `P2` is a proof of the formula `Forall(x:nat)(u:F).F2` for some `F2` not mentioning `x`, then the `existse`-proof also proves `F2`.
- `clash t1 t2`. If t1 and t2 are values built with different constructors, like `(S x)` and `Z`, this proves the disequation `{ t1 != t2 }`. We will also see how to use `symm` and `trans` with disequations.
- `contra P F`. If `P` proves `{ t != t }`, then this proof proves `F`. It is used to prove any formula `F` you happen to need in your proof, if you have derived a contradictory statement (i.e., `{ t != t }`).

5.2 Reasoning with Implication

An *implication* is an if-then formula. It says if formula `F1` is true, then so is `F2`. An example is, "x is zero, then x plus x equals zero." In GURU, implications are written using `Forall`. The example implication just mentioned is written

```
Forall(u : { x = Z }). { (plus x x) = Z}
```

You can think of this as saying, for all proofs `u` of { `x = Z` }, we have {`(plus x x) = Z`}. Using `Forall` for implications makes formulas a little more concise than they might otherwise be. For example, we can write:

```
Forall(x:nat)(u : { x = Z }). { (plus x x) = Z}
```

This quantifies over `x` of type `nat`, and then continues with the example implication. This idea of combining implication and universal quantification comes from other languages, for example COQ [3].

We reason with implications in exactly the same way as universal quantifications. To prove an implication, we use `foralli`. For example, here is the proof of our example formula:

```
Define plusZ' :=
foralli(x:nat)(u : { x = Z }).
  trans cong (plus * *) u
        join (plus Z Z) Z.
```

Here, `u` is an arbitrary proof of {`x = Z`}. So `u` acts as an assumption that {`x = Z`}. We use this assumption to transform `x` into `Z` in `(plus x x)`. This is done by the `cong`-proof. Then we can join `(plus Z Z)` with `Z`.

To use an implication, we instantiate it using the square brackets notation. This makes for a rather convenient notation for instantiating theorems. For example, to use this `plusZ'` theorem we have just proved, we can write:

```
[plusZ' Z refl Z]
```

Here, we are instiating `x` in the theorem with `Z` (the first argument), and `u` with `refl Z` (the second argument.

5.3 Existential Introduction

An existential formula is one that states that there is a value `x` of some type `T` which satisfies a stated property. Here is an example:

```
Forall(x:nat). Exists(y:nat). { (le x y) = (le y x) }
```

In English, this formula says, "for all `x` of type `nat`, there exists a `y` of type `nat` such that the (boolean) value returned by `(le x y)` is equal to that returned by `(le y x)`." In other words, for every `nat x`, there is a `nat y` such that `x` is less than `y` if and only if `y` is less than `x`. The only number with this property, in fact, is `x` itself. This uses the `le` function for less-than-or-equal-to on the unary natural numbers, which is defined in `guru-lang/lib/nat.g`.

To prove an existential, we must specify a value that has the property. That value is called the *witness* of the existential. So in this case, we will specify `x` as the witness, since {`(le x x) = (le x x)`}. Notice that this last formula has four occurrences of `x` in it. Two of these we wish to view as occurrences of our witness, and two are part of the property. This is indicated by using a *formula context*, which is a formula with a $*$ in it:

```
{ (le x *) = (le * x) }
```

To prove our existential, we will use an `existsi`-proof, to introduce the existential. The syntax for an `existsi`-proof is `existsi t F* P`, where `t` is the witness, `F*` is the formula context corresponding to the property the witness is supposed to have, and `P` is a proof that `t` has that property. In particular, `P` is a proof of the formula `F*[t]`, which is our notation (not GURU's) for the formula you get if you substitute the witness `t` for the $*$ in `F*`. Note that it is required that the witness term `t` be a value. So here, we will write:

```
existsi x { (le x *) = (le * x) } P
```

where P is a proof of

```
{ (le x x) = (le x x) }
```

The complete proof in GURU is:

```
Define ltcomm : Forall(x:nat).Exists(y:nat). { (le x y) = (le y x) } :=
  foralli(x:nat).
    existsi x { (le x *) = (le * x) } refl (le x x)
```

We start off with `foralli`, to introduce the variable `x` for an arbitrary `nat`. Then comes our `existsi`-proof. We can just use `refl (le x x)` as the proof `P` of {`(le x x) = (le x x)`} mentioned above. You can see the importance of the formula context in `existsi`-proofs by considering this modification of the proof:

```
foralli(x:nat).
  existsi x { (le x *) = (le * *) } refl (le x x)
```

The only change is that we are using a different formula context, one with three $*$s instead of two. The formula proved by this proof is

```
Forall(x:nat).Exists(y:nat). { (le x y) = (le y y) }
```

This says something quite different from the formula proved above.

5.3.1 Another example

Let us prove this formula:

```
Forall(x:nat). Exists(y:nat). { (le x y) = tt}
```

In English, this formula says, "for all `x` of type `nat`, there exists a `y` of type `nat` such that `x` is less than or equal to `y`." To prove this formula, we must just show how to find, for every `nat x`, a `nat y` such that `x` is less than or equal to y `y`. Of course, for any `x`, there are an infinite number of numbers that would serve for such a `y`: all the numbers greater than or equal to `x`. We must just pick one of them to serve as the witness of the existential quantification (i.e., the value that has the desired property). We will pick `x` as the witness, since there is a theorem `x_le_x` defined in `guru-lang/lib/nat.g` which proves:

```
Forall(a:nat).{ (le a a) = tt}
```

In GURU, our proof looks like this:

```
Define existsle : Forall(x:nat).Exists(y:nat). {(le x y) = tt } :=
  foralli(x:nat). existsi x { (le x *) = tt } [x_le_x x].
```

We are proving the theorem, which we call `existsle`, by first introducing the variable `x` for an arbitrary `nat` using `foralli`. Then we have our `existsi`-proof, with the witness `x`, the formula context {`(le x *) = tt`}, and the proof `[x_le_x x]`, which instantiates the `x_le_x` theorem with `x` to conclude {`(le x x) = tt` }.

5.4 Existential Elimination

If we have a proof of an `Exists`-formula, stating that there is a value which has a certain property, we can make use of that proof as follows. We may introduce a new variable `x` for the value that is stated to exists. We may also assume that this `x` has the stated property. In GURU, this is done using an `existse`-proof. The syntax is unfortunately a little cumbersome, although this is a problem with how existential elimination has been done in logic for around 80 years. We write `existse P1 P2`, where for any type `T`:

- `P1` proves `Exists(x:T).F`.
- `P2` proves `Forall(x:T)(u:F).F'`, where `x` may not be mentioned by the formula `F'`.

The role of `P1` is clear enough: this is our proof of the existential. The role of `P2` is a bit more puzzling. It proves some other formula `F'`, but the proof is allowed to make use of arbitrary `x` of type `T`, and an assumption `u` that `x` has property `F`. This corresponds to the informal intuition above: we introduce a variable `x` for the value that is stated to exist, along with an assumption that `x` has the stated property. The formula proven (`F'`) is not allowed to mention `x`, because the entire `existse`-proof then proves `F'` (and if `F'` mentioned `x`, that `x` would be used outside its scope, which is the `Forall`-formula).

Here is a simple example of existential elimination. In Section 5.3.1 just above, we proved:

```
Forall(x:nat). Exists(y:nat). { (le x y) = tt}
```

So for any value `x` of type `nat`, there is a value `y` of type `nat` such that `x` is less than or equal to `y`. Let us introduce variable `y` for this value, and assume that {`(le x y) = tt`}. Since `y` is less than `(S y)`, we can conclude that `(lt x (S y))`. Taking `(S y)` as our witness, we may conclude that there exists a `z` such that `(lt x z)`. This informal argument proves, in a somewhat roundabout way:

```
Forall(x:nat). Exists(z:nat). { (lt x z) = tt}
```

We may write this proof in GURU, making use of several lemmas from `guru-lang/lib/nat.g`:

```
lt_S : Forall(a:nat).{ (lt a (S a)) = tt}

lelt_trans : Forall(a b c:nat)(u:{ (le a b) = tt })(v:{ (lt b c) = tt }).
                { (lt a c) = tt }
```

The first lemma says `a` is less than `(S a)`. The second says that if

1. $a \leq b$, and
2. $b < c$,

then $a < c$. So this is a form of transitivity combining less-than-or-equals and less-then. The proof is then the following (the line numbers are not valid GURU syntax):

```
0. Define existslt : Forall(x:nat). Exists(z:nat). {(lt x z) = tt } :=
1.   foralli(x:nat).
2.   existse [existsle x]
3.     foralli(y:nat)(u:{(le x y) = tt}).
4.         existsi (S y) { (lt x *) = tt }
5.           [lelt_trans x y (S y) u [lt_S y]].
```

Let us walk through this line by line.

1. Introduce our arbitrary `x` of type `nat`.

2. Use existential elimination. The proof `[existslt x]` is our instantiation of the previously proved theorem. It proves `Exists(y:nat).{(le x y) = tt}`. This is the first proof that `existse` requires, namely, the proof that something exists which has a certain property.

3. The second proof `existse` requires begins here and stretches for the rest of the proof. This proof begins by assuming arbitrary `y` of type `nat`, along with an assumption `u` that `{(le x y) = tt}`.

4. Now we use `existsi` to prove the formula `Exists(z:nat).{(lt x z) = tt}`. There is no mention of the variable `z` in the proof itself. In fact, by default GURU names the variable `x`, keeping track of the fact that this `x` is different from other variables in scope which might have the same name.

5. Here we instantiate `lelt_trans`. We provide five arguments corresponding to the five inputs of `lelt_trans`:

   - `x` for `a:nat`
   - `y` for `b:nat`
   - `(S y)` for `c:nat`
   - `u` for `u:{ (le a b) = tt }`
   - `[lt_S y]` for `v:{ (lt b c) = tt }`

   The `lelt_trans`-proof then proves the desired `{(lt x (S y)) = tt}`.

5.5 Proving a Function Terminates

One of the most basic properties one might want to prove about a recursively defined function is that it terminates for all inputs. When the function is structurally terminating (see Section 4.2), this can be easily done by induction. To do this, we must first formalize the statement that the function terminates. In GURU, this is done by stating that for all inputs to the function, there exists an output of the function on those inputs. For example, here is the formalized statement that `plus` terminates on all inputs:

```
Forall(x y : nat). Exists(z:nat).{(plus x y) = z}
```

Quantifiers in GURU range over values of the given types. So this says that for all values `x` and `y` of type `nat`, there exists a value `z` such that `(plus x y)` equals `z`. As stated earlier, if an equality between terms is provable in GURU, it implies that the two terms either both diverge (run forever) or both converge to a common value. Since the variable `z` ranges over values, this implies that `(plus x y)` converges to `z` (since `z` evaluates just to itself).

The proof in `guru-lang/lib/plus.g` that `plus` is total is called `plus_total`:

```
induction (x : nat) return Forall(y:nat). Exists(z:nat).{(plus x y) = z} with
  Z => foralli(y:nat).
       existsi y {(plus x y) = *}
       trans cong (plus * y) x_eq
             join (plus Z y) y
| S x' => foralli(y:nat).
          existse [x_IH x' y] foralli(z':nat)(u:{(plus x' y) = z'}).
          existsi (S z') {(plus x y) = *}
            trans cong (plus * y) x_eq
            trans join (plus (S x') y) (S (plus x' y))
                  cong (S *) u
end.
```

We will not walk through this in all detail, but focus just on the clause for `(S x')`. Here, we use an instantiation of the induction hypothesis, `[x_IH x' y]`, to prove:

```
Exists(z:nat). {(plus x' y) = z}
```

The `existse`-proof's second subproof, which begins `foralli(z':nat)`, picks up from this existential formula. It introduces a variable `z'` for the value `z` such that {`(plus x' y) = z`}. It is fine to use a different name (here `z'`) for the variable introduced by `foralli` than for the variable mentioned by the `Exists`-formula (here `z`). The rest of the clause is:

```
existsi (S z') {(plus x y) = *}
trans cong (plus * y) x_eq
trans join (plus (S x') y) (S (plus x' y))
      cong (S *) u
```

The reasoning here is as follows. If {`(plus x' y) = z'`}, then `(plus (S x') y)` can be shown to be equal to `(S z')`; and so {`(plus x y) = (S z')`}. This reasoning is done by the last three lines of the subproof. So we will take `(S z')` as our witness for the existential statement that there exists `z` such that {`(plus x y) = z`}. That is why the `existsi`-proof begins with `(S z')`: that is the witness.

5.5.1 Registering a function as total

When a function has been proved total in the sense just discussed, we can register it as total with GURU, using a `Total`-command. For example, in `guru-lang/lib/plus.g`, this command is used to register `plus` as total, where `plus_total` is defined to be the proof discussed in the previous section:

```
Total plus plus_total.
```

The first expression is a symbol defined to be a function, and the second is a proof that for all inputs that may be given to the function, there exists an output produced by the function on those inputs. Why is it useful to register functions as total? Because of an important restriction on `Forall`-elimination and `Exists`-introduction which we have glossed over up to now. When instantiating a `Forall`-formula, the argument given must be a terminating term. Similarly, the witness used to prove an `Exists`-formula must also be a terminating term. The reason is simple. As remarked above, quantifiers in GURU range over values. So when we have a proof of a formula like `Forall(x:nat).{(plus x Z) = x}`, that `x` ranges over values. So it is not legal to instantiate it with a term which might not terminate in a value; i.e., a non-terminating term. Similarly, since existential quantifications range over values, it is not legal to offer as a witness a term which might fail to terminate. For this reason, proving termination of functions is quite important in GURU. When a function has been registered as total, it may then be used in terms which will instantiate universal quantifiers or witness existential ones. If we try to instantiate a quantifier with a term including a function that has not been registered as total, GURU will report an error. For example, suppose we run the following:

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

Define loop := fun f(x:nat):nat.(f x).

Classify [plusZ (loop Z)].
```

GURU will report:

```
Forall(x : nat)(u : { x = Z }) . { (plus x x) = Z }
"/home/stump/guru-lang/doc/test.g", line 37, column 17: classification error.
Checking termination, the head of an application is neither
declared total nor a term constructor.
1. the application in spine form: (loop Z)
2. the head: loop
```

We have defined `loop` as a looping function (since it just takes in `x` and immediately makes a recursive call on `x`). The GURU proof checker then reports an error when we attempt to instantiate the universal formula proved by `plusZ` with `(loop Z)`, since that term is not known to be terminating (in fact, it is non-terminating).

5.5.2 Aside: `show`-proofs

Sometimes while we are incrementally developing a proof, it is useful to see exactly what formula some subproof proves. There is a way to do that in GURU. You simply use a `show`-proof. The syntax is:

```
show P1 ... Pn end
```

where `P1` through `Pn` are proofs. GURU will compute the classifiers for those proofs and print them. It will then stop any other classification, as if we had a classification error. For example, to see the equational steps in the `S`-clause of the proof from the previous section that `plus` is a total function, we can use `show`:

```
induction (x : nat) return Forall(y:nat). Exists(z:nat).{(plus x y) = z} with
  Z => foralli(y:nat).
       existsi y {(plus x y) = *}
       trans cong (plus * y) x_eq
             join (plus Z y) y
| S x' => foralli(y:nat).
          existse [x_IH x' y] foralli(z':nat)(u:{(plus x' y) = z'}).
          existsi (S z') {(plus x y) = *}
            show
              trans cong (plus * y) x_eq
              trans join (plus (S x') y) (S (plus x' y))
                    cong (S *) u
            end
end.
```

GURU will then print:

```
"/home/stump/guru-lang/doc/test.g", line 14, column 20: classification error.
We have the following classifications:

1. (plus x y) =

2. (plus (S x') y) =

3. (S (plus x' y)) =

4. (S z')
```

GURU lists this as an error, but of course, it is really just informational. We see the four equational steps going into the `trans`-proof that is being displayed with `show`. GURU prints `trans`-proofs specially with `show`, by printing what is proved by its subproofs. For any other kind of proof, GURU will print just the formula proved by the entire proof.

5.6 Reasoning with Disequations

For some theorems, particularly implications, we need disequational reasoning: that is, we need to use disequalities between terms, which state that the terms do not either converge to different values (this is the case we are interested in) or do not both diverge (I have never had a case like this of interest). Here is a simple example. We would like to prove the following formula:

```
Forall(x:nat)(u:{(le x Z) = tt}). {x = Z}
```

This says that for all `x` of type `nat`, if `x` is less than or equal to zero, then `x` must equal zero. We certainly believe this to be true (for natural numbers), but how is it proved? Let us assume an arbitrary `x` of type `nat`, and let us assume that `x` is less than or equal to `Z`. Now let us do a case split on `x`. If `x` is zero, then we are done, since that is what we are supposed to prove. If `x` is `(S x')` for some `x'`, then our assumption that `x` is less than or equal to zero is contradicted. If we evaluate `(le (S x') Z)`, we will get `ff`. But our assumption says that `(le (S x') Z)` evaluates to `tt`. And `tt` is disequal to `ff`. So we reach a contradiction, because we have:

```
tt   =   (le (S x') Z)   =    ff
```

and also `{ tt != ff }`. From a contradiction we can conclude anything, since false implies anything. So in particular we can conclude `{ x = Z }`.

The two parts of reasoning used in this informal proof which we have not seen formalized in GURU are the use of the contradiction to prove any formula, and the proof of the disequation `{ff != tt}`.

- To prove a disequation like `{ff != tt}`, the syntax in GURU is

  ```
  clash ff tt
  ```

  A `clash`-proof takes any two values built with different constructors, and proves that they are disequal. So another example is `clash Z (S Z)`, which proves `{ Z != (S Z)}`.

- To derive a formula from a contradiction in GURU, we use a `contra`-proof. The syntax is `contra P F`, where `P` proves that `{ t != t }` for some term `t`. The `contra`-proof then proves `F`, which may be any formula we want.

Before we can formalize our proof of `Forall(x:nat)(u:{(le x Z) = tt}).{x = Z}` in GURU, we need one more ingredient, which is how to do equational reasoning for disequations. It works quite easily. The proof rules `symm` and `trans` work also with proofs of disequations. If

```
P  :  { t1 != t2 }
```

then we have:

```
symm P  :  { t2 != t1 }
```

And if we have

```
P1  :  { t1 = t2 }
P2  :  { t2 != t3 }
```

then we also have:

```
trans P1 P2 : { t1 != t3 }
```

So with `trans`, the first subproof must prove an equation, but the second one can prove an equation or a disequation. Notice that we cannot conclude anything about the relationship between `t1` and `t3` if we have two disequations `{ t1 != t2}` and `{t2 != t3}`. That is why `trans` requires the first proof to prove an equation. Now we have the tools we need to formalize our informal reasoning above in GURU:

```
Define le_Z1 : Forall(x:nat)(u:{(le x Z) = tt}). {x = Z} :=
  foralli(x:nat)(u:{(le x Z) = tt}).
  case x with
    Z => x_eq
  | S x' =>
    contra
```

```
      trans symm u
      trans cong (le * Z) x_eq
      trans join (le (S x') Z) ff
            clash ff tt
    { x = Z }
  end.
```

We start off by assuming arbitrary `x` of type `nat` such that {`(le x Z) = tt`}, using `foralli`. Now we case split on `x`, just as in our informal proof. The base case is really easy, since `x_eq` is a proof that {`x = Z`}, and that is what we are supposed to prove. For the step case, we have the following equational steps, which you can see by putting a `show` around the first argument to `contra` (i.e., from "`trans symm u`" to the end of the `clash`-proof):

1. `tt =`

2. `(le x Z) =`

3. `(le (S x') Z) =`

4. `ff !=`

5. `tt`

This chain of steps proves {`tt != tt`}, which is just the kind of contradictory equation that `contra` requires for its subproof. Then we give `contra` the formula {`x = Z`}, since that is what we wish to derive from our contradiction.

5.7 Case Splitting on Terminating Terms

For `case`-proofs (Section 3.6), the expression we are case splitting on must be a terminating term, like the instantiating and witnessing terms discussed in Section 5.5.1. If the term is something other than just a symbol, we need to use a feature of `case`-proofs we have not seen up until now, which is a `by`-clause. Suppose we are trying to prove the following:

```
Forall(x y:nat). {(eqnat x y) = (eqnat y x)}
```

Here, `eqnat` is a function testing whether or not `nat`s `x` and `y` are equal. We could prove this theorem by induction on `x`, but there is actually an easier proof using the following theorems in `nat.g`:

```
eqnatEq : Forall(n m:nat)(u:{(eqnat n m) = tt}). { n = m }

eqnatNeq : Forall(n m:nat)(u:{(eqnat n m) = ff}). { n != m }

neqEqnat : Forall(n m : nat)(u:{n != m}).{ (eqnat n m) = ff }
```

The idea of this easier proof is to case split on `(eqnat x y)`. This is allowed since `eqnat` is registered as a total function in `nat.g`. In the case where `(eqnat x y)` is `tt`, we can use the theorem `eqnatEq` to conclude that {`x = y`}. Using that fact, we can easily transform `(eqnat y x)` into `(eqnat x y)`. In the case where `(eqnat x y)` is `ff`, we can use `eqnatNeq` to conclude that {`x != y`}. From this we obtain {`y != x`} by symmetry, and from there, we get {`(eqnat y x) = ff`} by `neqEqnat`.

Here is the formalization of this proof in GURU, which I added to `nat.g` while writing this section. The new feature is the `by`-clause at the very start of the `case`-proof, which we will explain just below.

```
Define eqnat_symm : Forall(x y:nat). { (eqnat x y) = (eqnat y x) } :=
  foralli(x y:nat).
```

```
case (eqnat x y) by u ign with
  ff => trans u
              symm [neqEqnat y x symm [eqnatNeq x y u]]
| tt => trans cong (eqnat * y) [eqnatEq x y u]
              cong (eqnat y *) symm [eqnatEq x y u]
end.
```

Our `case`-proof begins with "`case (eqnat x y) by u ign with`". We have the `case` keyword, and then the terminating term (aka, the *scrutinee*) on which we are case splitting. Next comes the `by`-clause "`by u ign`", and then the `with` keyword. The `by`-clause is used when case splitting on a term which is not literally a symbol (like `x`). Here, we are splitting on `(eqnat x y)`, which is not a symbol; it is an application. GURU does not attempt to introduce a name automatically for the assumption variable relating the scrutinee with the pattern in each case, unless the scrutinee is a symbol, say `x`. In that case, we have seen that GURU automatically introduces this assumption variable, with the name `x_eq`. When splitting on a term that is not a symbol, it is up to us to choose the name of the assumption variable. There are actually two such variables introduced by a `case`-proof. The first one is the one we need here, and I have called it `u`. The second one, "`ign`" is ignored here. We will see what it is does, when we study dependently typed programming.

The clauses for the `case` proof are a bit dense, but they do follow the informal reasoning mentioned above. Let us just consider part of the `ff`-clause. The subproof `[eqnatNeq x y u]` proves that {`x != y`}. We use `symm` to reverse this. Call that proof `P`. It proves {`y != x`}. Then `[neqEqnat y x P]` proves {`(eqnat y x) = ff`}, as you can see if you instantiate the variables in the formula listed above for `neqEqnat` as `[neqEqnat y x P]` is doing.

5.8 Summary

We have seen how to reason with implications, existential formulas, and disequations. Implications are written using `Forall`, and then `Forall`-introduction and elimination are used for implications. To prove an existential, we must give to `existsi` a witness, which is a value that has the specified property. The property is specified to `existsi` with a formula context (a formula containing $*$). We have seen also how to state that a function terminates: for all possible inputs to the function, there exists an output such that the function applied to the inputs equals the output. We may instantiate universal quantifiers and witness existential ones only with values, which are terms guaranteed to terminate. Once we have proved a function terminates on all inputs, we can register it as total using a `Total`-command. This function may then be used in instantiating or witnessing terms.

5.9 Exercises

As usual, please use only the proof constructs we have seen so far in the book. You are free, however, to use any lemmas proved in the standard library (files in `guru-lang/lib/`).

1. Give an informal English translation of the following GURU formula:

   ```
   Forall(a b:nat)(u:{ (le (S a) b) = tt }).{ (le a b) = tt }
   ```

2. Prove the following formula:

   ```
   Forall(x:nat)(u:{(lt Z x) = tt}). Exists(x':nat). { x = (S x') }
   ```

   HINT: my proof does not require induction, just a case split on `x`, and then in the `Z` case, a proof using `contra` and `clash`.

3. Write a formula in GURU that says that raising natural number `x` to the power 1 gives you `x`. What is the name of that theorem in the standard library (where it is indeed proved)?

4. Prove

```
Forall(x y:nat)(u:{(mult y (S x)) = Z}). { y = Z }
```

HINT: this can be proved by induction without using any other lemmas, just reasoning directly about the behavior of `mult` (and `plus`).

5. Prove the following theorem about the exponentiation function `pow`, defined in `guru-lang/lib/pow.g`:

```
Forall(b e : nat)(u:{ b != Z }). { (le (S Z) (pow b e)) = tt }
```

HINT: my proof begins by case splitting on `(pow b e)` (see Section 5.7), so that in the base case I can use the lemma `pow_not_zero`, defined in `pow.g`. In the step case I made use of lemmas `S_le_S` and `leZ` from `nat.g`.

Bibliography

[1] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.

[2] 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.

[3] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version V8.0*, 2004. http://coq.inria.fr.

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