Verified Programming in GURU

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

March 26, 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 [11, 14]. 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 [13, 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:

$$0 + m = m \tag{2.1}$$

$$(1 + n') + m = 1 + (n' + m) \tag{2.2}$$

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:

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

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

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

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 `nat`s, 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 `bool`s. 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 `ff` (even). 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 `nat`s `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 [13].

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 `nats` `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`.

Chapter 6

Polymorphic Programming and Proving

Up until now, we have limited ourselves to monomorphic programming, where programs operate just on particular specified types of data. With polymorphic programming, also known as generic programming, we can write functions that operate polymorphically (or generically) for any type of data. This kind of polymorphism, the only kind we will study in this book, is crucial for implementing generic data structures, which can hold any type of data. Such data structures are, of course, of central importance for achieving code reuse. We do not wish to code up a different list datatype for every new type of data we might wish to store in a list. With a generic list datatype, we design one datatype for lists of any single type of data, and write functions operating on such lists. With verified programming, we of course also have the burden of writing proofs about those programs. Naturally, those proofs are also generic, proving theorems for any kind of data stored in the generic data structure.

6.1 Preview

The main differences we find moving to generic programming are:

- polymorphic datatypes are described using datatype constructors, also just called type constructors. A type constructor constructs a type when applied, at the type level, to some arguments (here, just other types). The notation for type-level application uses angle brackets. So for example, `<list nat>` is GURU notation for applying the type constructor `list` to the type `nat` to get the type of lists which store `nat`s.
- pattern-matching will automatically equate type variables A in the type of the scrutinee with corresponding variables A' in the type of the pattern.
- the class of examples we can consider becomes much richer, because we are now working with data structures – i.e., structures designed to hold other data – rather than basic data like `nat`s and `bool`s.

6.2 Polymorphic Datatypes

In GURU, we use polymorphic datatypes to implement generic data structures. Polymorphic datatypes are declared like monomorphic datatypes, using an `Inductive`-command. The difference is that they use type variables in several places, instead of particular types. We will begin by looking at a representative and frequently used polymorphic datatype, the type for polymorphic lists. We are all familiar with various list datatypes in object-oriented programming languages. Lists in functional languages are implemented a bit differently, because they must be described as inductive datatypes, where bigger data are incrementally and uniquely built from smaller data (see Section 2.2).

Before we look at the polymorphic type, let us consider how we would implement a monomorphic datatype `nlist` for lists of `nat`s. As an inductive datatype, lists are usually considered to be built up from the empty list by gradually adding elements (`nat`s for `nlist`s) one at a time to the front of the list. The constructor to do this is traditionally

called `cons` (going back to LISP). The constructor for the empty list is `nil`. Here is the declaration in GURU for this:

```
Inductive nlist : type :=
  nil : nlist
| cons : Fun(n:nat)(l:nlist).nlist.
```

The value in GURU for the list "1,2,3" would then be `(cons one (cons two (cons three nil)))`. We apply `cons` three times to incrementally grow our final list from the starting list `nil`, the empty list. Now let us see how to declare the polymorphic `list` datatype:

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

This is similar to the definition for `nlist`: we still have constructors `nil` and `cons`, for example. But obviously, there are some significant differences. The first line of the `Inductive`-command declares `list` to have classifier `Fun(A:type).type`. This makes `list` a *type constructor*. We can think of it as a function that takes an input `A` which is a type, and returns a type as output. The input type `A` is the type for elements of the list, and the output type is they type for lists of `A`s.

Let us look now at the types of the constructors. The type for `nil` says that `nil` takes an input `A` which is a type, and then returns an output of type `<list A>`. This notation, with the angle brackets, is for applying a type constructor to arguments. Such an application is called a *type-level* application, since it takes place at the level of types; in particular, it produces a type as output. Here, `list` is the type constructor, and the argument is `A`. Since `A` has classifier `type`, and since `list` expects its argument to have classifier `type`, this application (i.e., `<list A>`) is allowed, and has classifier `type`. Sometime classifiers like `type` and `Fun(A:type).type`, which classify types and type constructors, are called *kinds* (see also Section 2.3.3, where the classification hierarchy for terms is mentioned).

Finally, we have the type for `cons`, which is `Fun(A:type)(a:A)(l:<list A>).<list A>`. This type says that `cons` takes three inputs: a `type` `A`, a value `a` of type `A`, and a list `l` of `A`s. Then `cons` produces as output another list of `A`s. Here we see how type variables like `A` enable us to describe generic datatypes. For any type `A` we like, we can pass a piece of data "`a`" of type `A` to `cons`, to be stored in the list. Notice that a single list can hold data of just a single type `A`. Such lists are sometimes called *homogeneous*. We cannot store a `nat` and a `bool` in the same list. Lists that allow different types of data in the same list are sometimes called *heterogeneous*. As an example of how the datatype works, here is the value that we will write for the list "1,2,3":

```
(cons nat one (cons nat two (cons nat three (nil nat))))
```

The sole argument to `nil` is a type, and the first argument to `cons` is a type. Here, the type is `nat`. It is a bit annoying that `nat` is repeated all these times, and it would be an improvement to GURU if it could be instead inferred from the types of the data values like `one` that are stored in the list. Functional programming languages like OCAML and HASKELL provide powerful type inference mechanisms to support polymorphic programming without type annotations in code such as these uses of `nat` as arguments here. Unfortunately, in the presence of the kind of polymorphism supported by GURU, the problem of inferring a type for a term with no type annotations is provably unsolvable (similarly to the way the more familiar *halting problem*, of whether or not a Turing machine halts, is unsolvable). So type inference is necessarily limited or approximate in some way for such languages. At the moment, GURU does not attempt to provide such type inference, and we must live with writing annotations like type arguments to polymorphic functions.

6.3 Polymorphic Functions

Programming with polymorphic data is similar to programming with monomorphic data, except that we use type variables. For example, let us write an `append` function, for concatenating two polymorphic lists. As a warmup, here

is code for appending monomorphic nlists, where we will rename the constructors slightly so that they do not clash with those for lists:

```
Inductive nlist : type :=
  nnil : nlist
| ncons : Fun(n:nat)(l:nlist).nlist.

Define nappend : Fun(l1 l2:nlist).nlist :=
  fun(l1 l2:nlist).
    match l1 with
      nnil => l2
    | ncons n l1' => (ncons n (nappend l1' l2))
    end.
```

The basic idea behind this function can perhaps be easier seen via these equations:

```
(nappend nnil l2) = l2
(nappend (ncons n l1') l2) = (ncons n (nappend l1' l2))
```

If the first list is empty, then we just return the second (that is what the first equation says). If the first list consists of a nat n followed by a sublist l1', then we recursively append the sublist to l2. This is done with (nappend l1' l2) in the second equation. Then we put n at the front of the result. This code has the effect of working through the first list completely until it reaches nnil, which it replaces with l2. So the first list is rebuilt starting with l2 instead of with nnil.

Now let us see the polymorphic version of append, working on polymorphic lists. We can include the definition of list given in the previous section from guru-lang/lib/list.g. Because some of the theorems in that file take a while to check, it is helpful to do the include of list.g like this:

```
Include trusted "../guru-lang/lib/list.g".
```

The "trusted" option tells GURU not to check any proofs that are Defined in the included file or recursively any files that file includes. A better long-term solution is to implement separate compilation for GURU, where once a file has been checked, a summary of its declarations and definitions is created which can be used later without rechecking the file. This has not been implemented yet, however. When GURU terminates, it will print a list (which can get rather long) of all the theorems whose proofs it is trusting due to the use of "trusted". To return: the code for append can be written as follows. Note that a much more complicated definition is given in list.g, for reasons which we will explain later. We call our version here append' to avoid a name conflict.

```
Define append' : Fun(A:type)(l1 l2:<list A>).<list A> :=
  fun append'(A:type)(l1 l2:<list A>):<list A>.
    match l1 with
      nil _ => l2
    | cons _ a l1' => (cons A a (append' A l1' l2))
    end.
```

There are, clearly, several differences from the monomorphic nappend defined above. First, append' takes a type A as its first argument. The second and third arguments are then of type <list A>, meaning that they are lists whose elements are of type A. Then append' returns a list of As. Other differences include the fact that A is passed as an argument to cons, and also in the recursive call to append. This makes sense from a classification perspective, since cons and append require a type as their first arguments, and that type has to match up in the appropriate way with later arguments. One feature we have not discussed yet is used for the patterns of the cases. Here, an underscore is written for the type argument (the first argument to nil and cons). There is no special meaning to underscore: it is just another pattern variable in this case. I am using underscore here just to indicate informally that that variable is not used later. So it is essentially an ignored variable. Again, this is all informal: GURU does not check that it is ignored.

The one subtle point we must note here is that for the second pattern, say, to classify, it must be the case that the pattern variable `a` has classifier `_`. This is because `cons`'s type says that the first argument is some type, here `_`, and the second argument then has that type. So `a` really has type `_`. How then can we treat it as if it has type `A`, for example when we pass it to `cons` in the body of the clause for `cons`? Because GURU automatically equates `_` and `A` in this situation. The algorithm GURU uses is the following.

1. Compute the type, call it T_S, for the scrutinee. Here, the scrutinee is `l1`, and its type is `<list A>`.

2. Then for each clause, compute the type, call it T_P, for the clause's pattern. In the clause for `cons`, for example, the type of the pattern is `<list _>`.

3. Match T_S against T_P. That is, treat T_P as a pattern, and pattern match it against T_S. Pattern variables occurring in T_P are considered variables which can be instantiated by this pattern matching. So here, we are matching `<list A>` against `<list _>`, where `_` is considered an instantiable variable (because `_` is a variable in the pattern `cons _ a l1'`). The instantiation that makes these two types equal is the one which maps `_` to `A`.

4. Extend definitional equality in the body of the clause to equate any variables instantiated by pattern matching with whatever they were instantiated to. So in the body of the clause for `cons`, definitional equality is extended to equate `_` and `A`.

5. Now classify the body of the case with the extended definitional equality. This explains why `(cons A a)` type checks in the body of the `cons`-clause: once we have given `A` as the first argument to `cons`, the second argument is expected to have type `A`; the given second argument "`a`" actually has type `_`; but these are considered definitionally equal in this case, so the application type checks.

We will see later situations where this simple algorithm is not sufficient, particularly for dependently type programming. In those situations, explicit type casts are used in code, which begin with the keyword `cast`. You will see some of these in library files, including `list.g` (for reasons explained in Section 6.5.3). But for polymorphic programming as we usually encounter it (certainly in OCAML or HASKELL without some of its recent extensions to the type system), this is enough to make the connection between the type of the scrutinee and the type of the pattern, in a fully automatic way.

6.4 Polymorphic Proving

Let us now prove that `append'` is associative. The theorem we wish to prove is written this way in GURU:

```
Forall(A:type)(l1 l2 l3 : <list A>).
  { (append' l1 (append' l2 l3)) = (append' (append' l1 l2) l3) }
```

There are two points to explain here. First, we are quantifying over a type `A`, similarly to the way our code for `append'` above takes in a type `A`. The second point, which is somewhat subtle, is in the phrasing of the equation we are proving. Notice that here, unlike in our code above, no type argument is given to `append'`. We just call `append'` with the two lists. Here we are seeing in action one of the main features of GURU over related proving environments like COQ. Equations are always stated and proved between terms without any type annotations. In fact, GURU's definitional equality erases type annotations from terms. So `(nil bool)` and `(nil nat)` are considered definitionally equal, when we compare classifiers. In fact, note that when we drop their type annotations, we get just `nil`, with no parentheses.

The rationale for this approach is that we are interested in reasoning about the computational behavior of programs, which in GURU does not depend on type annotations. So we may reason about the programs with their annotations dropped. Practically speaking, this greatly simplifies reasoning about code with annotations, since they are erased for purposes of equational reasoning. Tools like COQ do not erase such annotations, which then clutter proofs. The problem becomes much worse with dependently typed programming, where programs contain much more complex annotations than just a few type arguments here for polymorphic programming.

```
Define append'_assoc :
  Forall(A:type)(l1 l2 l3 : <list A>).
    { (append' l1 (append' l2 l3)) = (append' (append' l1 l2) l3) } :=
foralli(A:type).
induction(l1:<list A>)
return Forall(l2 l3 : <list A>).
        { (append' l1 (append' l2 l3)) = (append' (append' l1 l2) l3) }
with
  nil _ =>
  foralli(l2 l3 : <list A>).
    trans cong (append' * (append' l2 l3)) l1_eq
    trans join (append' nil (append' l2 l3)) (append' (append' nil l2) l3)
          cong (append' (append' * l2) l3) symm l1_eq
| cons _ a l1' =>
  foralli(l2 l3 : <list A>).
    trans cong (append' * (append' l2 l3)) l1_eq
    trans join (append' (cons a l1') (append' l2 l3))
               (cons a (append' l1' (append' l2 l3)))
    trans cong (cons a *) [l1_IH l1' l2 l3]
    trans join (cons a (append' (append' l1' l2) l3))
               (append' (append' (cons a l1') l2) l3)
          cong (append' (append' * l2) l3) symm l1_eq
end.
```

This proof does not use any features we have not already discussed. The algorithm described in the previous section for connecting the scrutinee's type and the pattern's type is used here also for the clauses of the `induction`-proof (the same algorithm is used for `case`-proofs, too). We do induction on the first list `l1`, following Hint 1, since `l1` is used twice in an analyzed argument position, here the first argument position of `append` (while `l2` is used once and `l3` not at all in such a position). Otherwise, we just perform straightforward equational reasoning, where as explained just above, we do not need to include the type annotations in any of the terms we use in equational reasoning. In particular, we give completely unannotated terms to `join` and `cong`.

6.5 The Fold-Right Function

An important polymorphic function often defined on lists is the fold-right function, which we will call `foldr'`; the name is to avoid a conflict with a more complex version `foldr` defined in `guru-lang/lib/list.g`, which we will discuss below. The basic idea is that `foldr'` should take a function "f" of type `Fun(a:A)(b:B).B` and a starting value `b` of type `B`. It will iterate `f` over the elements of the list, starting with value `b` and accumulating a resulting value of type `B` for the whole list. For example, given a list like "a1,a2,a3" of elements of type `A`, `foldr'` with `f` and `b` will return:

```
(f a1 (f a2 (f a3 b)))
```

We will see in just a moment why this is useful, but let us first write the type for `foldr'`:

```
Fun(A B:type)
   (f:Fun(a:A)(b:B).B)
   (b:B)
   (l:<list A>).B
```

I have split this across multiple lines for readability. We can see that we must first take in the types `A` and `B`. Then we take in the function `f` and the starting value `b`, and finally the list of `A`s. The code for `foldr'` is easy to write:

```
fun foldr'(A B:type)(f:Fun(a:A)(b:B).B)(b:B)(l:<list A>):B.
  match l with
    nil _ => b
  | cons _ a l' => (f a (foldr' A B f b l'))
  end.
```

When the list `l` is empty, we return our starting value `b`. When it consists of an element "`a`" and a sublist `l'`, we apply `f` to `a` and also to the result of folding `f` on `l'`.

6.5.1 Using `foldr'` to compute length

As a simple example, we can use `foldr'` to give a very concise (and non-recursive) definition of a function to compute the length of a list:

```
Define length' :=
   fun(A:type)(l:<list A>).
     (foldr' A nat fun(a:A)(n:nat).(S n) Z l).
```

First we should confirm that this type checks. For example, the third argument's type is `Fun(a:A)(n:nat).nat`, which is correct given that the first two arguments are `A` and `nat`. To understand how this definition of `length` works, let us look at what `foldr'` will return for a list "a,b,c". Writing `F` for the function `fun(a:A)(n:nat).(S n)`, the returned value is computed by:

```
(F a (F b (F c Z)))
```

The starting value we have given `foldr'` is the fourth argument, `Z`. So we see "`Z`" at the right of the expression just above. Then we have the iteration of `F` through the data of the list. We can easily see that { `(F x y) = (S y)` } for all `x` and `y`. Transforming the above expression one step at a time (just for emphasis), we see that the expression is indeed equal to the length of the list (namely, three):

```
(F a (F b (F c Z))) =
(S (F b (F c Z))) =
(S (S (F c Z))) =
(S (S (S Z)))
```

6.5.2 Using `foldr'` to map a function

Another common operation in polymorphic functional programming with lists is to transform an input list of `A`s into a list of `B`s by applying a function `f` of type `Fun(x:A).B` to the elements of the input list. This operation is called *map*. We can implement a version of this called `map'` using `foldr'`, as follows:

```
Define map' :=
  fun(A B:type)(f:Fun(x:A).B)(l:<list A>).
    (foldr' A <list B> fun(x:A)(l2:<list B>).(cons B (f x) l2) (nil B) l).
```

The second argument to `foldr'` is always the type of the data that is being computed by the function we are folding across the list's data. We wish to compute a list of `B`s from a list of `A`s, so the final result we want from our folding operation is of type `<list B>`. That explains the second argument to `foldr'`. For the third, we can confirm it has the correct type:

```
Fun(x:A)(l2:<list B>).<list B>
```

This fits the pattern for these functions: their types should be of the form `Fun(x:X)(b:Y).Y`, for some types `X` and `Y`. The function we are folding takes in the element of the list first, as `x` of type `A`, and it takes the result of folding the function over a sublist (`L`, say), and returns the result of folding the function over `(cons A a L)`. So we just use `cons` to put together `(f x)` and the result `l2` of mapping the sublist. Our starting value for this folding operation is `(nil B)`, the empty list at type `B`.

6.5.3 Some complications due to compilation

If you look in `guru-lang/lib/list.g`, you will see first our definition of polymorphic lists (from Section 6.2), and definitions of functions `foldr`, `map`, and `length` that behave just like the functions with similar names that we have defined above. But the definitions are more complicated, and indeed a bit ugly. For example, `foldr` is declared to have this type:

```
Fun(A B C: type)(owned cookie:C)
   (fcn: Fun(owned cookie:C)(owned x:A)(y:B).B)
   (b:B)(owned l : <list A>).B
```

This is similar to, but not the same as, the type for `foldr'` that we have above:

```
Fun(A B: type)
   (f: Fun(x:A)(y:B).B)
   (b:B)(l : <list A>).B
```

The difference (other than the different name for the function which is being folded – a purely syntactic difference with no semantic import) is that the type of `foldr` takes three types, `A`, `B`, and `C`; where `foldr'` takes just `A` and `B`. Then `foldr` expects a *cookie* of type `C`, and the function `fcn` that it takes as its fifth input expects to be called with such a cookie. Furthermore, the keyword `owned`, which we have not seen up to now, is used in several places. What is going on?

What is going on is that we are, for the first time, encountering code that is actually intended to be compiled to C code and executed efficiently. Up until now, all the code we have considered is not intended for compilation to C, but rather just in specifying such code. After all, what computation do we really want to do with unary natural numbers? Efficient computations should be done with machine integers, which we will discuss in a later section. Unary `nats` are much better than machine integers for specification, however, since they are much easier to reason with. The boolean datatype is used heavily in practice, but it is so simple it turns out to constitute a special case: it is just an enumeration, and those can be handled specially during compilation to C code. But with the list datatype, we actually encounter a non-trivial datatype that we wish to use in compiled code. So here, we run into several issues in GURU as it currently stands, that must be dealt with. These issues are not fundamental to the language; we could change GURU so that we did not have to deal with them. But it turns out there are some reasons for not doing so, even though it results in somewhat less pleasant code. We will explore all three of these issues in more detail later. For now, this is just a preview so we can handle the code in `lib/list.g`.

1. When GURU programs are compiled to C, memory is managed using reference counting, not garbage collection as is typical in other memory-safe languages (a memory-safe language is, at least roughly, one where we it is not possible to read uninitialized memory or read or write freed memory). GURU uses ownership annotations like `owned` here to help reduce the cost of reference counting.

2. When compiling `fun`-terms to C, the compiler requires that they do not contain any free variables. For example, a `fun`-term like `fun(x:nat).(plus x y)` is disallowed by the compiler. Such `fun`-terms are supported by functional languages like OCAML and HASKELL, and indeed, many would consider them crucial to the functional programming methodology. They are disallowed in the current version of GURU for reasons again related to memory management. A language that supports (compilation of) functions with free variables is said to support *closures* (this is the name of the compiler technique used to support functions with free variables).

3. In code which will be compiled, again for reasons related to memory management, it is sometimes necessary to use explicit type casts, even though the algorithm presented at the end of Section 6.3 succeeds in connecting the type of the scrutinee with the type of a pattern in a `match`-term. The syntax for these casts is

   ```
   cast t by P
   ```

   where `t` is a term of type `T`, and `P` is a proof of { `T` = `T'` }, for some type `T'`. The `cast` then has type `T'`.

So, our `foldr` code includes some `owned` annotations related to memory management. The type `C` for cookies is to accomodate functions that might want to rely on auxiliary data. If GURU supported closures, we would not need these cookies, since auxiliary data could be included in the function itself via free variables. Since we do not support closures, the auxiliary data must be passed in via a cookie, which is some other data structure that exists just to hold that auxiliary data.

The functions `foldr` and `map` in `lib/list.g` are written to use cookies, just in case the functions being folded or mapped need to use cookies. The `length` function just uses a trivial cookie `unit` of type `Unit`, defined in `lib/unit.g`. For purposes of proving lemmas about these functions, it should be possible to:

- Ignore ownership annotations completely in your proofs. You should never have to type a keyword like `owned` anywhere in your proofs. This is because these annotations are dropped out by definitional equality inside proofs (though not inside terms).
- Reason about those functions as if they had the simpler definitions above. They do the same thing, except that they pass cookies around in some cases where the simpler definitions do not.

When writing new code using these functions, we can also avoid ownership annotations by using the `spec` option with the `Define`-command. For example, the following definition uses `foldr` (the one from `lib/list.g` with cookies) to define a function `concat` that takes a list whose elements are lists of `A`s and appends all those elements to get just a list of `A`s. Without the `spec` option just after the `Define` keyword, this will not type check in GURU due to the lack of ownership annotations:

```
Define spec concat : Fun(A:type)(l:<list <list A>>).<list A> :=
  fun(A:type)(l:<list <list A>>).
    (foldr <list A> <list A> Unit unit
       fun(u:Unit)(l1 l2:<list A>).(append A l1 l2)
       (nil A) l).
```

6.6 Exercises

The usual restriction on using proof methods discussed in the book so far applies.

1. Prove the following theorem about the `fill` function in `guru-lang/lib/list.g`:

   ```
   Forall(A:type)(a:A)(n m:nat).
          { (fill a (plus n m)) = (append (fill a n) (fill a m)) }
   ```

   In the code for `fill`, you will see `inc a`. This `inc` is for incrementing a reference count. In proofs, `inc a` is definitionally equal to `a`.

   HINT: I proved this using induction on one of the variables (which one would it most likely be?), and I did not need any other lemmas.

2. Prove that `fill` is total.

3. Prove the following theorem, which states that mapping "`f`" and then mapping "`g`" is the same as mapping their composition. Note how `fun`-terms are written with annotations dropped, on the right hand side of the equation (for the composition of "`f`" and "`g`"). The type for cookies is taken to be just `Unit`, for simplicity.

   ```
   Forall(A B C:type)
          (f:Fun(u:Unit)(x:A).B)
          (g:Fun(u:Unit)(x:B).C)
          (l:<list A>).
     { (map unit g (map unit f l)) = (map unit fun(u x).(g u (f u x)) l) }
   ```

HINT: my proof is by induction on one of the variables (which?), and uses no other lemmas.

4. Using `foldr` defined in `guru-lang/lib/list.g`, define a function `some` with the same type as the function `all` defined at the bottom of `lib/list.g`. This function `some` should return `tt` if the given function `f` returns `tt` for some element of the list, and `ff` otherwise. In contrast, the `all` function returns `tt` if `f` returns `tt` for all elements of the list, and `ff` otherwise.

5. Write down two universally quantified formulas in GURU syntax, where each one mentions two functions from { `foldr, length, append, foldr, all, some, fill` }, and where you believe the formula to be provable in GURU. Your formulas should not be trivially provable, in the sense that they should not be provable using only `foralli` and `join`: `induction` or at least `case`-proofs should be required to prove the formulas.

Optional: prove one of the formulas you have written.

Chapter 7

Dependently Typed Programming

In the preceding chapters, we have seen how to write monomorphic and later polymorphic programs, and to prove properties about their operational (i.e., run-time) behavior. Those proofs are external to programs, in the sense that they reside outside the code for the functions in question, and from that external position prove operational properties of the code. With external verification, proofs and programs are distinct and separate artifacts.

In this chapter, we study internal verification, where a program and a proof are written as a single combined artifact. Syntactically this artifact is a program (i.e., a term), but it contains proofs inside it. These proofs are there to demonstrate to the type checker that the code has a semantically rich type, called a *dependent type*. The semantic richness and the dependency of these types arises because dependent types are allowed to mention ("depend on") terms. We have seen in the previous chapter how types can mention other types: the type `<list A>` of lists of `A`s mentions the type `A`. More concretely, `<list nat>` mentions the type `nat`. With dependent types, we can have a type like `<vec A n>`, where `n` is a `nat`, and hence a term. This `<vec A n>` is the type for lists ("vectors") of `A`s of length `n`. A dependent datatype like this is also sometimes called an *indexed* type (where `vec` is considered to be indexed by `n`). The length `n` is present in the type, which allows the expression of more complex relationships between the types of program data than we have seen up to now. For example, we may write an append function on vectors that takes a `<vec A n>` and a `<vec A m>` as inputs, and produces a `<vec A (plus n m)>` as output. At various places in the code for this vector append function, it is necessary to use a proof to convince the GURU type checker that two types are equivalent. For example, we might need to convince the type checker that `<vec A (plus n m)>` and `<vec A (plus m n)>` are equivalent. We can do this using congruence and commutativity of `plus`.

Programming with dependent types has been studied intensively in the past few years, with a number of different research languages for dependently typed programming proposed [7, 8, 9, 12, 4, 2, 15, 6]. The motivation for this is that programming with rich dependent types seems to be closer to more traditional programming than a combination of traditional programming and external theorem proving, and still can give stronger correctness properties than are possible with traditional type checking. The GHC implementation of HASKELL has recently added support for a limited kind of dependent types called Guarded Algebraic Datatypes (GADTs) [3]. Dependent types arose much earlier than these works, in type theoretic formulations of logic. A well-known example is Martin-Löf type theory [5].

Because practical dependently typed programming is rather new, there is not yet consensus on effective methodology for using it. One question is, when to use dependent types and when to use external verification (assuming a language supports both, as GURU does). One simple practical answer is that it can quickly become infeasible to use external verification for large functions. Certainly in GURU, external verification relies heavily on partial evaluation, which will become unbearable with functions more than a few tens of lines long: otherwise we will end up `join`ing one gigantic program term with another, which will bloat proofs unacceptably. In such cases, dependent types are very useful, because we need never write a proof about the large function. Instead, we give it a rich type which captures some critical properties of it, the way the type mentioned above for vector append captures the property that the length of the output list is equal to the sum of the lengths of the input lists.

7.1 Preview

In this chapter we will see how to define dependent types like `<vec A n>`, and how to write dependently typed programs operating on such types. In the next chapter we will discuss external verification of dependently typed programs.

- A dependent type like `vec` is declared using an `Inductive`-command like this:

```
Inductive vec : Fun(A:type)(n:nat).type :=
  vecn : Fun(A:type).<vec A Z>
| vecc : Fun(A:type)(n:nat)(a:A)(l:<vec A n>).
              <vec A (S n)>.
```

- Proofs of type equivalence enter code in `cast`-terms, mentioned at the end of the last chapter.
- Previously, we have seen how `match`-terms, `case`-proofs, and `induction`-proofs introduce an assumption variable like `n_eq` for a match on "`n`". This variable proves that the scrutinee is equal to the pattern in each clause of the expression. These expressions also have a second assumption variable, which is `n_Eq` for "`n`". This proves that the scrutinee's type is equal to the pattern's type. These assumptions can be needed to reason based on the fact that indices to the scrutinee's type are equal to corresponding indices of the pattern's type. Injectivity, embodied in `inj`-proofs, is used to go from the proof that the types are equal to the proof that the indices are equal.

7.2 Indexed Datatypes

The declaration for the indexed datatype `vec` for vectors (i.e., lists) of `A`s of length `n` in GURU is:

```
Inductive vec : Fun(A:type)(n:nat).type :=
  vecn : Fun(A:type).<vec A Z>
| vecc : Fun(A:type)(n:nat)(a:A)(l:<vec A n>).
             <vec A (S n)>.
```

You can find this declaration in `guru-lang/lib/vec.g` (ignoring for now the `spec` keyword that appears in one place there). In the first line, we declare `vec` to be a type constructor, similarly to the declaration we have in Section 6.2 for `list`. Here we see that `vec`'s classifier (its *kind*) is `Fun(A:type)(n:nat).type`. So when we apply `vec` – which is done at the type level using angle brackets, as we saw for `list` – we will have to supply two arguments: a type and a `nat`. So for example, `<vec nat Z>` is a correct type-level application of `vec`, and will have classifier `type`. So `<vec nat Z>` is a type.

Now let us look at the declarations of the constructors, `vecn` and `vecc`. This `vecn` corresponds to `nil` for lists: it creates the vector of `A`s of length zero. That is indeed what its type tells us. Given `A` that is a type, `vecn` will return a value of type `<vec A Z>`. To extend a vector by adding a new element to the front, we use `vecc`, corresponding to `cons` for lists. This takes in a type `A`, a `nat` "`n`", an element `a` of type `A` (this is the element to add to the front of the vector), and a subvector `l`. The subvector should have type `<vec A n>`. Since "`n`" is the length of the subvector, the length for the new vector being built by `vecc` is `(S n)`. This is because we have added one element to the front of "`l`", thus yielding a list whose length is one greater than `l`'s.

The interpretation of "`n`" in `<vec A n>` as the length of the vector is not machine-checked. We have given the index `n` this interpretation informally. We have not proven any theorem relating this `n` to the length of the vector as computed by some `length` function.

Here is an example value built using `vecc` and `vecn`:

```
(vecc nat (S Z) three (vecc nat Z four (vecn nat)))
```

This value has type `<vec nat (S (S Z))>`. This is a vector of length two (the "`(S (S Z))`" in this type). The data stored in this vector are `three` and `four` (so the list looks like "3,4"). We have a type annotation `nat` in three places. We also have annotations `Z` and `(S Z)` for the lengths of sublists. These are given as the second argument to `vecc` in each case.

7.3 Programming with Indexed Types

Let us now see how to write the append function on vectors. As mentioned above, we wish to write this function so that it has the following type:

```
Fun(A:type)(n m:nat)(l1 : <vec A n>)(l2 : <vec A m>).<vec A (plus n m)>
```

We will take in `A` that is the type of elements in the vectors, "`n`" and "`m`" that are the lengths of the vectors, and then the vectors themselves. We will return a vector of length `(plus n m)`. This function will behave very similarly to append on `lists`, except for the presence of the indices to `vec` for the lengths of the lists. It is these which require some extra work for dependently typed programs.

Just like `list` append, `vec` append will use the first list as its parameter of recursion. Let us think for a moment about the base case, when this vector `l1` is the empty vector. In this case, we wish just to return `l2`. This is exactly what `list` append does. Our only difficulty is with the types. Our "`l2`" has type `<vec A m>`, but we are supposed to return (from `vec` append) a value of type `<vec A (plus n m)>`. Of course, we are in the case where `l1` is empty, so its length "`n`" is equal to zero. That means that the type `<vec A (plus n m)>` is actually equal to `<vec A m>`, since in this case:

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

We just need to prove that type equality to GURU. This can be done by standard GURU equational reasoning with `cong`, `join`, and the rest, as long as we can get that proof that { `n = Z` }. How do we do that?

7.3.1 The assumption variable for types

We have seen how all the constructs in GURU that have clauses – `match`-terms, `case`-proofs, and `induction`-proofs – make available an assumption variable in each clause that says that the scrutinee is equal to the pattern. For example, in Section 3.6, we saw this simple `case`-proof:

```
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.
```

In each clause, the assumption variable `b_eq` is automatically declared. In the `ff`-clause its classifier is { `b = ff` }, while in the `tt`-clause its classifier is { `b = tt` }. So `b_eq` serves as a proof of the assumption that the scrutinee matches the pattern in each clause.

For dependently typed programs, we need to reason about indices to types. For this purpose, clausal constructs in GURU provide a second assumption variable. Where the first assumption variable proves, in each clause, that the scrutinee is equal to the pattern of the clause, the second assumption variable proves that the scrutinee's type is equal to the pattern's type. Where the first assumption variable is automatically named `x_eq` for scrutinee "`x`", the second is automatically named `x_Eq`.

For monomorphic code, this assumption is never interesting, since the types involved do not have any structure. For example, in the `not_not` proof above, the second assumption variable, whic is automatically named `b_Eq`, proves { `bool = bool` }. This is because the type of the scrutinee "`b`" is `bool`, and the type of the patterns (`ff` and `tt`) is `bool`. Let us return to the code for `vec` append to see a more interesting example.

7.3.2 Starting the base case of vector append

Here is the code for vector append (found in `guru-lang/lib/vec.g`), not including the clause for `vecc`:

```
fun vec_append(A:type)(n m:nat)(l1 : <vec A n>)(l2 : <vec A m>):
              <vec A (plus n m)>.
    match l1 with
      vecn _ => cast l2 by
                cong <vec A *>
                 symm trans cong (plus * m)
                             inj <vec ** *> l1_Eq
                           join (plus Z m) m
   | vecc _ n' x l1' => ...
   end.
```

The function begins as we would expect based on the type it is supposed to have. It does a `match` on `l1`. As for polymorphic `match`-terms, GURU makes the connection between `_` and `A`, and declares them definitionally equal in the bodies of the clauses (see Section 6.3). We then see a `cast`-term. The syntax of such terms is `cast t by P`, where "`t`" is a term and `P` is a proof. If "`t`" has type `T`, and `P` proves {`T = T'`}, then the whole `cast`-term has type `T'`. In other words, a `cast`-term is used to change the type of `t` from `T` to `T'`, where `P` proves these types are equal. In our case here, the goal is to change the type of `l2` from `<vec A m>` to `<vec A (plus n m)>`. Let us walk through the proof to see how this is done.

First, if we use `show` on the `trans`-proof, we will see it proves these equational steps:

```
1. (plus n m) =

2. (plus Z m) =

3. m
```

The only new part of the `trans`-proof is how we obtain the fact that {`n = Z`}. This is done with an `inj`-proof, `inj <vec ** *> l1_Eq`, which we explain next.

7.3.3 Injectivity reasoning

Suppose we have a proof `P` that {`(S x) = (S y)`}. This should mean that {`x = y`}, but up until now, we have not seen how to conclude this in GURU. The way to do so is to use *injectivity* reasoning. Term and type constructors are injective. A function f is injective iff $f(x) = f(y)$ implies $x = y$. In other words, the only way f can map x and y to the same output is if x and y are the same input. If $x \neq y$, then $f(x) \neq f(y)$.

In GURU, injectivity reasoning is done with `inj`-proofs. The general syntax is slightly complicated, so let us start with our example using successor. If `P` proves {`(S x) = (S y)`}, the `inj (S *) P` proves {`x = y`}. We use a context here to indicate which position is of interest to us (marked with $*$, as for congruence).

The general syntax of an `inj`-proof is `inj C P`, where `C` is an *injectivity context* and `P` is a proof. An injectivity context is a constructor term or a type with a hole $*$ in it, indicating a position of interest. In addition, a second hole $**$ is used to indicate positions we wish to ignore. The use of the injectivity context is as follows. We check that the proof `P` proves a formula of the form {`C[e,e1,...,en] = C[e',e1',...,en']`}, where `C[e,e1,...,en]` means the expression obtained by substituting `e` for the first hole and `ei` for the i'th occurrence of the second hole in `C`. In other words, the left and right hand sides of the equation proved by `P` share a common top-level structure described by `C`, differing in one place we care about (indicated with $*$) and several places we do not (indicated by $**$). Then the `inj`-proof proves {`e = e'`}.

7.3.4 Finishing the base case of vector append

Now we may return to the base case of vector append, to understand the proof used to cast `l2` from type `<vec A m>` to `<vec A (plus n m)>`:

```
cong <vec A *>
  symm trans cong (plus * m)
               inj <vec ** *> l1_Eq
             join (plus Z m) m
```

The `inj`-proof here uses the second assumption variable `l1_Eq` (see Section 7.3.1). This variable has the following classifier:

```
{ <vec A n> = <vec _ Z> }
```

The left hand side is the type of the scrutinee "`l1`", while the right hand side is the type of the pattern `(vecn _)`. From this we wish to conclude that {`n = Z`}. We must tell `inj` to disregard the difference between `A` and `_` in the first argument position of the two expressions. For that we use `**`, so we have `<vec ** *>` as our context. The only other point to note in our proof is that we use `cong` with a type context `<vec A *>`. This works exactly as `cong` with a term context. Here, the `cong`-proof goes from a proof that {`m = (plus n m)`} (note the use of `symm` to get this from the `trans`-proof) to a proof that:

```
<vec A m> = <vec A (plus n m)>
```

This is what we need to cast `l2` to be able to return it in the base case.

7.3.5 Finishing vector append

The full code for vector append is:

```
fun vec_append(A:type)(n m:nat)(l1 : <vec A n>)(l2 : <vec A m>):
              <vec A (plus n m)>.
  match l1 with
    vecn _ => cast l2 by
              cong <vec A *>
                symm trans cong (plus * m)
                             inj <vec ** *> l1_Eq
                           join (plus Z m) m
  | vecc _ n' x l1' =>
     cast
        (vecc A (plus n' m) x (vec_append A n' m l1' l2))
     by cong <vec A *>
           trans symm join (plus (S n') m)
                           (S (plus n' m))
                 cong (plus * m)
                   symm inj <vec ** *> l1_Eq
    end.
```

Let us look now at the `vecc`-clause. In this case, `l1` is a `vecc`-term, where the subvector is `l1'`, of type `<vec _ n'>`. The data at the front of the vector is `x` of type `_`. The type of this pattern is then

```
<vec _ (S n')>
```

As remarked for the base case, GURU makes the connection between `_` and `A`. But we need to help GURU make the connection between the length `n` of `l1` and `(S n')`. This is again done by injectivity reasoning: at the very end of the body of this clause, we have again `inj <vec ** *> l1_Eq`, proving in this case that

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

Let us see what the `cast` is doing. First, we should figure out what type the term has which is being cast. The term is

```
(vecc A (plus n' m) x (vec_append A n' m l1' l2))
```

From the type of `vec_append` specified at the very start of the recursive `fun`-term, we can compute that the term `(vec_append A n' m l1' l2)` has type `<vec A (plus n' m)>`. Notice how, in this term, the indices `n'` and `m` have to relate to the types of `l1'` and `l2`. Once we have specified to `vec_append` that the lengths of the vectors are `n'` and `m`, then we really need to give vectors whose types say they have those lengths. Turning now to the entire `vecc`-term, we can compute that it has this type:

```
<vec A (S (plus n' m))>
```

This is because we are adding an element `x` to the front of the vector built by the recursive call to `vec_append`. We have already seen that that vector has type `<vec A (plus n' m)>`. Adding one to that length leads to the type `<vec A (S (plus n' m))>`.

So the `cast`-term is changing the type of this `vecc`-term from

```
<vec A (S (plus n' m))>
```

to

```
<vec A (plus n m)>
```

The critical part of this, of course, is changing from `(S (plus n' m))` to `(plus n m)`, which we well know how to do at this point, using equational reasoning.

7.4 Binary Search Trees

As another example, we will define an indexed datatype for binary search trees in such a way that any value of that datatype truly satisfies the binary search tree property: if we go left from a node with data x in the tree, we will only ever see data less than or equal to x; and if we go right, we will see data greater than or equal to x. To enforce these invariants, the solution here is to index the type of binary search trees by lower and upper bounds on the data stored in the tree. To do this in a generic way, we do not build in the comparison, `le`, but allow it to be specified by users of our datatype. So our type for binary search trees is:

```
<bst A le l u>
```

Here, `A` is the type for data stored in the tree, `le` is the comparator, and `l` and `u` are the lower and upper bounds, respectively. Here is the declaration for this datatype:

```
Inductive bst : Fun(A:type)(le:Fun(a b:A).bool)(l u : A).type :=
  leaf : Fun(A:type)(le:Fun(a b:A).bool)(a:A).
             <bst A le a a>
| node : Fun(A:type)(le:Fun(a b:A).bool)
             (a l1 u1 l2 u2:A)
             (t1 : <bst A le l1 u1>)
             (t2 : <bst A le l2 u2>)
             (q1:{ (le u1 a) = tt})
             (q2:{ (le a l2) = tt}).
             <bst A le l1 u2>.
```

We are supposing here that we always have two children, and that leaves store data. In the next chapter, we will revisit these assumptions. Let us walk through this declaration. First, the kind for `bst` is declared to be:

```
Fun(A:type)(le:Fun(a b:A).bool)(l u : A).type
```

This means that for <bst A le l u> to be a type, we must have le be a function that takes two As as inputs, and produces a bool telling whether or not the first is less than the second. The lower and upper bounds (l and u) must have type A, naturally. The type for leaf says that a leaf storing data "a" has lower and upper bounds a: the return type for leaf is <bst A le [a] [a]>.

The type for node is more complicated. The basic idea is that we want to make a new tree with left and right subtrees t1 and t2, respectively, where the new tree stores data "a" (at the top of the tree). Here is where we enforce our invariants that going left should give you data less than or equal to "a", while going right should give you greater than or equal data. Let l1 and u1 be the lower and upper bounds on the data in t1. Then u1 should be less than or equal to "a". Similarly, if l2 and u2 are lower and upper bounds on the data in t2, we should have "a" less than or equal to l2. When node is applied, proofs of these facts must be given, for the parameters q1 and q2. We then get a new tree with lower bound l1 and upper bound u2.

Programming with datatypes like bst that have such strong invariants is not easy. For example, we might wish to write a function insert, to insert a piece of data in the bst. If we were using external verification, we would have one modestly tricky piece of code to write, and one more complicated theorem proving that the tree produced by inserting a piece of data into a binary search tree is again a binary search tree. With our indexed bst datatype, the proof and the program have to be combined, resulting in a single, more compact artifact that the separate proof and program, but one that is more complicated, perhaps, than either of them separately would be.

7.5 Summary

We have seen how to design indexed datatypes for internal verification. In some cases, our indices simply provide information about the data, as the length index for vectors does, without constraining the data at all. Such information can be used for internal verification of functions operating on the data, such as verifying the relationship between input and output lengths to the vector append function. In other cases, like the binary search tree example, indices are actually used to help state constraints on the data. We have seen how to use injectivity and casts to work with indexed data in GURU programs. In the next chapter, we will see how to prove properties of dependently typed functions.

7.6 Exercises

1. Declare an indexed datatype slist for sorted lists of nats. Your declaration should make <slist n> a type for any nat n. Your datatype should be designed in such a way that if L is of type <slist n>, then either L is empty and n is zero ("Z"), or else L is the head of the list.

2. Define a function insert which takes a nat x and a sorted list, and returns a new sorted list where x has been inserted. In more detail, your function's type should begin like this:

   ```
   Fun(x n:nat)(l:<slist n>).
   ```

   It should return a sorted list. The first question you should try to answer is, what index to slist will be used in the return type? Mine uses a certain arithmetic function applied to x and n. Writing insert then requires several lemmas about this arithmetic function, which you can find proved already in one of the files in the standard library (guru-lang/lib/*g).

3. Explain in English what the type of vec_cat, defined in lib/vec.g, is saying about what that function does. Similarly, explain what the type of vec_get, also defined in lib/vec.g, says about what that function does.

Chapter 8

Specificationality and Dependently Typed Proving

In this chapter, we discuss an important feature of GURU for dependently typed programming, which is *specificationality*. Arguments to term constructors and to recursive functions can be designated as specificational. This means that they are only to be used when type checking code, and they will disappear, just like type annotations, when reasoning about code. Indices to indexed types are very often specificational. For example, code working with vectors typically does not need the length of the vector to compute a desired output value. The length is really just specificational, allowing us to state properties about operations on vectors by relating indices of input and output vectors (as we did in the type of vector append). The one rule about specificational data, naturally enough, is that non-specificational data cannot be computed from specificational data. GURU enforces this be disallowing pattern matching (in code) on specificational data, and by requiring that specificational data can be passed to term constructors or to recursive functions only in positions which have been designated (by the term constructor or the function) as specificational. It is fine, however, to use non-specificational data in a specificational argument position. Specificationality makes external verification of dependently typed code easier, since specificational data are dropped during equational reasoning. We will see examples of that in this chapter, as we consider proving properties of dependently typed functions.

8.1 Preview

- We will see how to mark arguments as specificational using the `spec` keyword, and how this feature is used with vectors and binary search trees.

- Specificationality is also useful for a rather rarely used feature of GURU, which is `existse_term`. This is like `existse`, but is a term construct, rather than a proof construct. The name we introduce for the value which has been proven to exist must be marked as specificational, since we must prohibit non-specificational values from being computed from it.

- We will see examples of proving properties of dependently typed code. The main differences to note are that specificational data are considered annotations, similar to type annotations, and are dropped during equational reasoning; and that the form of `induction`-proofs is generalized to allow quantification over indices used by the type of the parameter of induction. So to induct over a vector, we will start with "`induction(n:nat)(l:<vector A n>)`". Without this extra generality, we would typically not be able to apply the induction hypothesis, for typing reasons.

- We will also introduce `hypjoin`, which tries to join terms (similarly to `join`), except that it also performs substitutions using a given set of proven equations.

8.2 Specificationality for Datatypes

The vector datatype we saw in the last chapter is indexed by the length of the vector. The types of functions can then state non-trivial semantic properties via relationships between indices to input and output vectors. For example, the type of vector append state a non-trivial property of the input and output vectors by stating that the length index of the output vector is the sum of the length indices of the input vectors. In this example and many others, we can regard the index as just being part of the specification of the function. The function itself does not pattern match on indices, or perform other operations with them that would affect the output value computed. Hence, these indices are typically computationally irrelevant, just like type annotations. So we should be able to drop them during compilation and during theorem proving, if we wish. Of course, to do this we need a way to distinguish between arguments (to recursive functions and term constructors) that are only specificational, like the length of a vector, and arguments that are computational, in the sense that the final output value may depend on them. GURU provides a mechanism for distinguishing computational arguments and specificational arguments, using the `spec` keyword.

8.2.1 Specificationality for vectors

Let us look at the declaration of `vec` as it actually appears in `lib/vec.g`:

```
Inductive vec : Fun(A:type)(n:nat).type :=
  vecn : Fun(A:type).<vec A Z>
| vecc : Fun(A:type)(spec n:nat)(a:A)(l:<vec A n>).
              <vec A (S n)>.
```

The argument `n` to `vecc` is labeled specificational with the `spec` keyword. This keyword is itself an annotation that will be dropped during theorem proving by definitional equality. It just indicates that arguments given for `n` are specificational, and computational values should not depend on them.

For an example of how this works, consider a vector storing just the natural number `two`. With its type annotations, this is written just as it would be without the `spec` annotation:

```
(vecc nat Z two (vecn nat))
```

If we had not marked the length argument to `vecc` as specificational, then dropping annotations would result in this term, where the length `Z` still remains in the term:

```
(vecc Z two (vecn nat))
```

But with the `spec` keyword in our declaration of `vec`, the length is considered an annotation, and dropping annotations actually yields the pleasantly simplified:

```
(vecc two vecn)
```

The vector append function may now also mark the lengths of its input vectors as specificational. From `lib/vec.g`, we have the following, where `P1` and `P2` are standing in for the same two proofs we had in the previous chapter for `vec_append` (see Section 7.3.5):

```
Define vec_append :=
fun vec_append(A:type)(spec n m:nat)(l1 : <vec A n>)(l2 : <vec A m>):
              <vec A (plus n m)>.
    match l1 with
      vecn _ => cast l2 by P1
    | vecc _ n' x l1' =>
       cast
          (vecc A (plus n' m) x (vec_append A n' m l1' l2))
       by P2
    end.
```

Note the use of the spec keyword where n and m are declared. When GURU type checks this fun-term, it will make sure that n and m are only used in specificational argument positions of applications. We can confirm by eye that indeed they are: n is not used anywhere. For m, it is used as the third argument, which is specificational, to the recursive call to vec_append; and also in the application of plus. But that application is itself used in a specificational position, and hence satisfies the criteria on use of specificational data: they may only appear in specificational argument positions. The length n' of l1' in the vecc-clause is also specificational (since the declaration of vecc says it is), but it is also used only in specificational positions.

When the code for vec_append is compiled, all data in specificational argument positions are dropped. So the term (plus n' m) will not be evaluated at run-time, since it is used (only) in a specificational argument position to vecc. It is important that (plus n' m) will be dropped in this case, since otherwise vec_append would run in time quadratic in the length of the first input list: we would execute an addition that takes time linear in this length for each recursive call. With (plus n' m) dropped, however, the function runs in time linear in the length of the first input list.

Similarly, when it is time to reason externally about vec_append, we will do so using its unannotated version, which it definitionally equals:

```
fun vec_append(l1 l2).
    match l1 with
      vecn => l2
    | vecc x l1' => (vecc x (vec_append l1' l2))
    end.
```

Notice that terms of the form cast t by P are replaced by just t when we drop annotations. Specificational input variables have been dropped from the fun-term, and similarly from patterns in match-terms. Again, this code is clearly much more succinct and easier to manage, for external reasoning, than the annotated version, and we might wish we could write only this. Annotations are the price of internal reasoning.

8.2.2 Specificationality for binary search trees

At the end of the previous chapter, we considered a polymorphic datatype of binary search trees. Here, we will work with the following monomorphic datatype of binary search trees of nats:

```
Inductive bst : Fun(l u : nat).type :=
  leaf : Fun(a:nat).<bst a a>
| node : Fun(a l1 u1 l2 u2:nat)
             (t1 : <bst l1 u1>)
             (t2 : <bst l2 u2>)
             (q1:{ (le u1 a) = tt})
             (q2:{ (le a l2) = tt}).
             <bst l1 u2>.
```

Similarly to what we had before, the type <bst l u> is for binary search trees whose data are bounded between l and u. The node constructor takes a piece of data "a" to store at the root of the new tree, and then subtrees t1 and t2. Those have their own lower and upper bounds, namely l1 and u1, l2 and u2.

One problem with this design arises because data are stored at the leaves, and nodes must have exactly two subtrees. So we cannot have, for instance, a tree with just two pieces of data in it. We can have a leaf, which has one piece of data, or a node built from two leaves, which has three; but nothing in between. Using specificationality, we can solve this problem. Let us make our lower and upper bounds specificational data. This is surely sensible, since the bounds themselves are not intended to have any computational role: they are just used to help enforce (at compile time) the binary search tree property. What will our datatype look like then? Clearly the arguments l1,u1,l2, and u2 to node should be specificational. The interesting question is, what about the argument to leaf? If we make it specificational, then computationally speaking, leaves do not store any data. The argument to leaf is specificational, indicating which lower and upper bounds the leaf should be viewed as having. (Indeed, we could imagine a different

definition where `leaf` takes possibly distinct specificational lower and upper bounds as arguments, instead of just the single argument `a`.) The definition with `spec` annotations is:

```
Inductive bst : Fun(l u : nat).type :=
  leaf : Fun(spec a:nat).<bst a a>
| node : Fun(a:nat)(spec l1 u1 l2 u2:nat)
            (t1 : <bst l1 u1>)
            (t2 : <bst l2 u2>)
            (q1:{ (le u1 a) = tt})
            (q2:{ (le a l2) = tt}).
            <bst l1 u2>.
```

To build a tree storing just `two`, say, we could write:

```
(node two two two two two (leaf two) (leaf two)
      [x_le_x two] [x_le_x two])
```

This is definitionally equal (dropping annotations) to just:

```
(node two leaf leaf)
```

The type annotation and the lower and upper bounds have all dropped away, since the corresponding argument positions of `leaf` and `node` are specificational. Proofs (using the `x_le_x` lemma from `lib/nat.g`, which says that any `x` is less than or equal to itself) are considered annotations, too, so they have also disappeared. Of course, we could wish to write just the simple unannotated term instead of the much more verbose annotated one, but the annotations are the cost of our static guarantee that the node satisfies the binary search tree property. Finally, to build a tree storing just `one` and `two`, which was not possible to do with the previous design for `bst`, without specificationality; we can write:

```
(node one one one two two (leaf one)
  (node two two two two two (leaf two) (leaf two)
      [x_le_x two] [x_le_x two])
  [x_le_x one]
  join (le one two) tt)
```

8.3 Existential Elimination in Terms

Specificationality is useful for accomodating a rarely used feature of GURU, the `existse_term` construct. This allows us to do an existential elimination inside a term. Our interest here is not so much in `existse_term`, as in the utility of specificationality in bridging between proofs and terms in dependently typed code.

Recall the `existe` proof construct (Section 5.4, where the syntax is `existse P1 P2`, with:

- `P1 : Exists(x:A).F1`
- `P2 : Forall(x : A)(u:F1).F2`

The whole `existse`-expression is then a proof of `F2`. In contrast, the `existse_term P t` is a term (not a proof), of type `B`, when:

- `P : Exists(x:A).F`
- `t : Fun(spec x : A)(u:F).B`

The two constructs are clearly based on the same idea of a universal hypothetical interpretation of existentials. In the case of `existse_term`, the subterm `t` takes in the value proven to exist, together with a proof that that value has the property described by `F`. Here, when we introduce the name `x` for that value, the `existse_term`-construct requires that we mark `x` as specificational. This ensures that our computational results cannot depend on the witness to the existential. So `existse_term P t` can be treated as definitionally equal to just `t`, similarly to the way `cast t by P` is definitionally equal to just `t`: the witness introduced by `P` is computationally irrelevant, and hence `P` can be treated as computationally irrelevant, too. This is consistent with the view of proofs as computationally irrelevant where they are used in other places in terms.

8.4 Induction Over Indexed Datatypes

Proving properties about values of indexed datatypes and about dependently typed code is similar to the proving we have done up to now, with two new points to note. First, the form of `induction`-proofs is generalized just slightly, so that we may quantify over not just the parameter of induction, but also over indices mentioned in its type. This gives us extra generality in our induction hypothesis, without which we would not be able to apply it, in many cases. Second, proofs and specificational data occurring in terms are computationally irrelevant, and hence drop out during external reasoning, just like type annotations do when reasoning about polymorphic programs.

For an example, let us prove a property of our `bst` datatype, which partially shows that we have designed it correctly. The property is that if we have a `bst` with lower bound `l` and upper bound `u`, then `l` is really less than or equal to `u`. Stated in GURU, the property is:

```
Forall(l u:nat)(b:<bst l u>). { (le l u) = tt}
```

Unfortunately, this is a sad case where Hint 1 misleads us. The only universal variable used as a parameter of recursion in a function mentioned in our theorem is `l`, as the first argument to `le`. But in fact, we would get nowhere doing induction on `l` in this case. This is an easy lemma, if we prove it by induction on `b`. I cannot think of a good rule of thumb to help guide us here, except to observe that this is really a property of our binary search trees, not of their lower and upper bounds. But how we know this, or what it means for this to be "really" a property of `b` rather than `l` or `u` is something I do not know.

Leaving aside these difficulties, let us try the proof by induction on `b`. Suppose we begin the following way, which will turn out not to work:

```
foralli(l u:nat).
induction(b:<bst l u>) return { (le l u) = tt}
```

All will be well in the base case. But in the step case, for when `b` is a `node`-value, we will run into difficulty applying our induction hypothesis. There, we will have `b` of the form `(node a l1 u1 l2 u2 t1 t2 q1 q2)`, where

- `t1 : <bst l1 u1>`
- `t2 : <bst l2 u2>`

Since `t1` and `t2` are subtrees of `b`, the requirement that the induction hypothesis be used only for structurally smaller trees is fulfilled. But we have a problem with typing. The induction hypothesis for the proof as we have begun it is:

```
Forall(b:<bst l u>). { (le l u) = tt }
```

We may only instantiate this with a subtree of type `<bst l u>`. But our subtrees `t1` and `t2` have different types (e.g., `<bst l1 u1>` for `t1`). So typing constraints will not be satisfied if we try to instantiate the induction hypothesis with a proof-level application like `[b_IH t1]`, and we will not be able to complete our proof.

The induction hypothesis we need to avoid this problem is one which includes the lower and upper bounds:

```
Forall(l u:nat)(b:<bst l u>). { (le l u) = tt }
```

Since the lower and upper bounds are now included in the quantification, a proof like `[b_IH l1 u1 t1]` will now pass the proof checker: we have instantiated the bounds in a way that allows us to instantiate the tree with `t1`. GURU's `induction` proof construct allows us to write this `induction`-proof in such a way that we get this more general induction hypothesis. The straightforward syntax is demonstrated below, as well as the easy bodies of the clauses (note that `le_trans` is for transitivity of `le`, from `lib/nat.g`):

```
  induction(l u:nat)(b:<bst l u>) return { (le l u) = tt}
    with
    leaf _ => trans cong (le l *) inj <bst ** *> b_Eq
                    [le_refl l]
  | node a _ u1 l2 _ t1 t2 q1 q2 =>
    [le_trans l a u
       [le_trans l u1 a [b_IH l u1 t1] q1]
       [le_trans a l2 u q2 [b_IH l2 u t2]]]
  end.
```

This proof is called `bst_bounds` in `lib/bst.g`.

8.5 Dependently Typed Proving

As mentioned, proofs and specificational data drop out of terms during external reasoning about them. Consider the following code for testing whether or not a piece of data is in a `bst`:

```
Define bst_in : Fun(x:nat)(spec l u:nat)(t:<bst l u>). bool :=
  fun bst_in(x:nat)(spec l u:nat)(t:<bst l u>): bool.
    match t with
      leaf _ => ff
    | node a l1 u1 l2 u2 t1 t2 q1 q2 =>
      match (eqnat x a) with
        ff =>
          match (le x a) with
            ff => (bst_in x l2 u2 t2)
          | tt => (bst_in x l1 u1 t1)
          end
      | tt => tt
      end
    end.
```

Let us prove the following theorem about this piece of code:

```
Forall(x l u:nat)(t:<bst l u>)(u:{(bst_in x t) = tt}). { (le l x) = tt }
```

That is, if `x` is in `bst t` with lower bound `l` and upper bound `u`, then `l` must be less than or equal to `x` (of course, `x` must be also less than or equal to `u`, but we do not prove that here for simplicity). Let us write our proof using refinement. You can find this proof in `lib/bst-spec.g`, as `bst_in_le1` (`bst-spec.g` is just temporary, to avoid confusion with `bst.g` as it is being used by homework 3 currently). Our starting point is just to write down the initial `foralli` and `induction` parts:

```
 foralli(x:nat).
 induction(l u:nat)(t:<bst l u>) return
   Forall(u:{(bst_in x t) = tt}). { (le l x) = tt } with
   leaf _ => truei
 | node a l1 u1 l2 u2 t1 t2 q1 q2 => truei
end.
```

8.5.1 The base case

Let us fill in the base case of this proof. There, we have that `t` is a leaf, but our assumption `u` tells us that `x` is in `t`. This will give us a contradiction, since our code for `bst_in` says that no data is stored in a leaf. We prove this using unannotated terms, as we do for all theorem proving. The proof looks like this (with the equational steps to be shown just below):

```
 foralli(x:nat).
 induction(l u:nat)(t:<bst l u>) return
   Forall(u:{(bst_in x t) = tt}). { (le l x) = tt } with
   leaf _ =>
     foralli(u:{(bst_in x t) = tt}).
     contra
       trans symm u
       trans cong (bst_in x *) t_eq
       trans join (bst_in x leaf) ff
             clash ff tt
     { (le l x) = tt }
 | node a l1 u1 l2 u2 t1 t2 q1 q2 => truei
end.
```

The equational steps are:

```
1. tt =

2. (bst_in x t) =

3. (bst_in x leaf) =

4. ff !=

5. tt
```

Here we can clearly see that all the specificational data – in this case, the lower and upper bounds – have been dropped from the `bst_in`-terms.

8.5.2 `Case`-proofs in the step case

For the step case, where the tree `t` is a `node`-value, we will do case splits following the pattern matches in `bst_in`, and then fill in the proofs in each case. Putting in just the case splits to begin with, we can write the following, where `P1` is the proof of the base case considered above:

```
 induction(l u:nat)(t:<bst l u>) return
   Forall(u:{(bst_in x t) = tt}). { (le l x) = tt } with
   leaf _ => P1
 | node a l1 u1 l2 u2 t1 t2 q1 q2 =>
     foralli(u:{(bst_in x t) = tt}).
     case (eqnat x a) by v1 _ with
       ff =>
       case (le x a) by v2 _ with
         ff => truei
       | tt => truei
       end
     | tt => truei
```

```
    end
  end.
```

Notice that in our two `case`-proofs, we use `by`-clauses to introduce names – `v1` for the first `case`-proof, `v2` for the second – for the assumptions that the scrutinee equals the pattern in each clause (see Section 5.7 for more on `by`-clauses). So for example, at the point where the first `truei` is, we have:

- `v1 : {(eqnat x a) = ff}`
- `v2 : {(le x a) = ff}`

It is burdensome to have to pick and remember the names `v1` and `v2`, which is why GURU assigns the standard names ending with "`_eq`" (and "`_Eq`" for the equality between the type of the scrutinee and the type of the pattern) when it is reasonable to do so (when the scutinee is a symbol).

8.5.3 The first subcase

The reasoning for the situation of the first `truei` is as follows (using conventional mathematical notation for less-than-or-equal-to). We need to show `l` $\leq$ `x`. We first observe that it suffices to show `l1` $\leq$ `x`. This is because GURU considers `l1` definitionally equal to `l` in the `node`-clause, after applying the algorithm of Section 6.3. In more detail, the type of the scrutinee (here, the parameter of induction) is `<bst l u>`. The type of the pattern is `<bst l1 u1>`. The algorithm of Section 6.3 tries to pattern match the latter against the former, where pattern variables may be instantiated by the pattern matching. Here the pattern variables are `l1` and `u1`. We can indeed make the `bst`-types identical by mapping `l1` to `l` and `u1` to `u`.

Continuing now with the inequality reasoning: we wish to show `l1` $\leq$ `x`. Since $x \leq a$ is false, we can conclude, using a lemma somewhat verbosely called `le_ff_implies_le` in `lib/nat.g`, that $a \leq x$ is true. We have `u1` $\leq a$ by the assumption `q1`, since our `node` constructor (which introduces `q1` as a pattern variable here) requires a proof that the upper bound of the left subtree is less than or equal to the data `a` stored by the node. Also, by the `bst_bounds` lemma which we proved in Section 8.4, we have that `l1` $\leq$ `u1`. So we have this chain of inequalities, which we can glue together with `le_trans`:

$$\texttt{l1} \leq \texttt{u1} \leq \texttt{a} \leq \texttt{x}$$

The proof in GURU syntax is:

```
[le_trans l1 a x
   [le_trans l1 u1 a [bst_bounds l1 u1 t1] q1]
   [le_ff_implies_le x a v2]]
```

We have nested (proof-level) applications of `le_trans`. The outer application goes from `l1` to `a` to `x`. The inner one goes from `l1` to `u1` to `a`. This corresponds to grouping our inequalities above like this:

$$(\texttt{l1} \leq \texttt{u1} \leq \texttt{a}) \leq \texttt{x}$$

That is, we first (corresponding to the inner `le_trans`) go from `l1` to `a`, and then to `x`.

8.5.4 The third subcase

Our proof currently looks like this:

```
induction(l u:nat)(t:<bst l u>) return
  Forall(u:{(bst_in x t) = tt}). { (le l x) = tt } with
  leaf _ =>
    foralli(u:{(bst_in x t) = tt}).
```

```
    contra
      trans symm u
      trans cong (bst_in x *) t_eq
      trans join (bst_in x leaf) ff
            clash ff tt
    { (le l x) = tt }
| node a l1 u1 l2 u2 t1 t2 q1 q2 =>
    foralli(u:{(bst_in x t) = tt}).
    case (eqnat x a) by v1 _ with
      ff =>
      case (le x a) by v2 _ with
        ff => [le_trans l1 a x
                  [le_trans l1 u1 a [bst_bounds l1 u1 t1] q1]
                  [le_ff_implies_le x a v2]]
      | tt => truei
      end
    | tt => truei
    end
 end
```

Let us fill in the third subcase, where we have the second `truei` in the proof just listed. We will come back to the second subcase after that. In the third case, we are in a situation where `(eqnat x a)` has returned `tt`. From this, we should be able to conclude that $\{$`x = a`$\}$, and indeed, there is a lemma in `lib/nat.g` to that effect:

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

We have `l1` $\leq$ `u1` $\leq$ `a` using `bst_bounds` and `q1`, so we just need to use `le_trans` and then congruence to change "`a`" to `x`. The proof in GURU is:

```
trans cong (le l *) [eqnatEq x a v1]
      [le_trans l1 u1 a
         [bst_bounds l1 u1 t1]
             q1]
```

8.5.5 The second subcase

Our proof now looks like this:

```
 induction(l u:nat)(t:<bst l u>) return
   Forall(u:{(bst_in x t) = tt}). { (le l x) = tt } with
   leaf _ =>
     foralli(u:{(bst_in x t) = tt}).
     contra
       trans symm u
       trans cong (bst_in x *) t_eq
       trans join (bst_in x leaf) ff
             clash ff tt
     { (le l x) = tt }
 | node a l1 u1 l2 u2 t1 t2 q1 q2 =>
     foralli(u:{(bst_in x t) = tt}).
     case (eqnat x a) by v1 _ with
       ff =>
       case (le x a) by v2 _ with
```

```
          ff => [le_trans l1 a x
                    [le_trans l1 u1 a [bst_bounds l1 u1 t1] q1]
                    [le_ff_implies_le x a v2]]
*       | tt => truei
        end
      | tt =>
        trans cong (le l *) [eqnatEq x a v1]
              [le_trans l1 u1 a
                 [bst_bounds l1 u1 t1]
                 q1]
     end
  end
```

We have one `truei` left to fill in, on the line marked (not GURU syntax) with a $*$. I have saved this subcase for last, because to prove it, we will make use of a new proof method in GURU called `hypjoin`. Let us try to do the proof with the methods we know, and see where we run into trouble. In this subcase, we have the following assumptions:

- `u:` {`(bst_in x t) = tt`}
- `v1 :` {`(eqnat x a) = ff`}
- `v2 :` {`(le x a) = tt`}

This corresponds to the case in our `bst_in` code where we look for `x` in the left subtree `t1`, since `x` is less than or equal to the data `a` stored at the root of the tree `t`. So `(bst_in x t)` is equal to `(bst_in x t1)` in this case. Our assumption `u` tells us that `(bst_in x t)` equals `tt`, so we can use `trans` (and `symm`) to conclude {`(bst_in x t1) = tt`}. At that point, we may apply our induction hypothesis to conclude that `l1` $\leq$ `x`.

This informal reasoning is easily formalized in GURU, except for the step where we prove

```
{ (bst_in x t) = (bst_in x t1) }
```

Let us try to prove that in GURU using our assumptions `u`,`v1`, and `v2` listed above, and see what goes wrong. Certainly our first step is to change `t` to `(node a t1 t2)`, with this proof:

```
cong (bst_in x *) t_eq
```

Now we have `(bst_in x (node a t1 t2))`, and we would like to prove this equals `(bst_in x t1)`. That is only true, of course, because of the way the pattern matches go when evaluating `(bst_in x (node a t1 t2))`, as described by our assumptions `v1` and `v2`. The proof is intolerably verbose:

```
trans join (bst_in x (node a t1 t2))
           match (eqnat x a) with
              ff => match (le x a) with
                       ff => (bst_in x t2)
                     | tt => (bst_in x t1)
                     end
           | tt => tt
           end
trans cong match * with
              ff => match (le x a) with
                       ff => (bst_in x t2)
                     | tt => (bst_in x t1)
                     end
           | tt => tt
           end
        v1
```

```
trans join match ff with
              ff => match (le x a) with
                      ff => (bst_in x t2)
                    | tt => (bst_in x t1)
                    end
           | tt => tt
           end
      match (le x a) with
        ff => (bst_in x t2)
      | tt => (bst_in x t1)
      end
trans cong match * with
             ff => (bst_in x t2)
           | tt => (bst_in x t1)
           end
        v2
      join match tt with
             ff => (bst_in x t2)
           | tt => (bst_in x t1)
           end
      (bst_in x t1)
```

The problem here is we have had to repeat parts of the full code for `bst_in` several times, as we alternate partial evaluation (via `join`) and congruence reasoning to take into account how `(eqnat x a)` and `(le x a)` are assumed in this case to have evaluated. This is clearly unacceptable, as it results in a large proof which is dependent on the exact details of the way `bst_in` is written. Any change to `bst_in`, even one which does not change its observable pattern of recursive calls, will break this proof. Fortunately, there is a better way.

8.6 Hypjoin

As you have no doubt observed, proof in GURU is largely a very manual process. True, `join` allows us to take fairly large equational steps all at once, but other than that, we have so far seen no automation to help us prove theorems, until now. GURU implements one fairly powerful automated theorem proving method, called `hypjoin`. The theory and implementation of `hypjoin` are due to Adam Petcher [10]. The syntax is:

```
hypjoin t1 t2 by P1 ... Pn end
```

In other words, we give `hypjoin` two terms, `t1` and `t2`, and as many proofs `P1` through `Pn` as we like, between the `by` and `end` keywords. Then `hypjoin` will try to prove that `t1` equals `t2`, using the formulas proved by `P1` through `Pn`. Those formulas are required to be equations. The remarkable property of `hypjoin` is that, under some natural assumptions about termination of recursive functions used in `t1`, `t2` and the equations; and assuming the equations are consistent (a contradiction cannot be derived); then `hypjoin` will succeed if and only if there is a proof that `t1` equals `t2` which uses just equational reasoning (including `cong`), `join`, and the given equations. In other words, `hypjoin` is *sound* – if it claims {`t1 = t2`} is provable, then it really is – and also complete – if they are provably equal in the way described, then `hypjoin` will indeed report that they are.

Here is a very simple example demonstrating the use of `hypjoin`. Consider the following theorem about `le`, called `leZ` in `lib/nat.g`:

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

We can easily prove this by case-splitting on `a`, and then using equational reasoning:

```
foralli(a:nat).
  case a with
```

```
    Z => trans cong (le Z *) a_eq
                join (le Z Z) tt
  | S a' => trans cong (le Z *) a_eq
                  join (le Z (S a')) tt
  end
```

Since the subproofs in the two clauses are both the kind that `hypjoin` is supposed to be able to find automatically, we can use `hypjoin` to simplify this proof:

```
foralli(a:nat).
  case a with
    Z => hypjoin (le Z a) tt by a_eq end
  | S a' => hypjoin (le Z a) tt by a_eq end
  end.
```

In each case, we are instructing `hypjoin` to try to join `(le Z a)` and `tt`, using the equation proved by `a_eq`. Essentially, `hypjoin` evaluates the two terms just the way `join` would, but where `join` would get stuck pattern-matching on a variable (or other non-value), `hypjoin` tries to keep going by replacing the scrutinee of the pattern-match using one of the equations. In this example, in the first case, `a_eq` proves {`a = Z`}. To see how this works in detail, we need to recall that the definition of `le` in `lib/nat.g` is:

```
Define le : Fun(a b:nat).bool :=
   fun (a b: nat). (or (lt a b) (eqnat a b)).
```

So `hypjoin` will evaluate `(le Z a)` first like this (writing `-->*` for evaluation):

```
1. (le Z a)}                    -->*
2. (or (lt Z a) (eqnat Z a))    -->*
3. (or match a with
          Z => ff
       | S a' => tt
       end
       match a with
          Z => tt
       | S a' => ff
       end)
```

The `match`-terms come from the definitions of `lt` and `eqnat`. At this point, regular evaluation is stuck, since we are matching in both cases on a variable, namely "a". But where `join` would stop here, `hypjoin` continues by substituting `Z` for `a`, since the proof `a_eq` given to `hypjoin` proves {`a = Z`}. So we continue with:

```
4. (or match Z with
          Z => ff
       | S a' => tt
       end
       match Z with
          Z => tt
       | S a' => ff
       end)                     -->*
5. (or ff tt)                   -->*
6. tt
```

The evaluation performed by `hypjoin` in the other case is similar, except we end up substituting `(S a')` for `a`.

8.6.1 Default clauses

As a final finishing touch, since the subproofs (of our original proof for `leZ`) in the case where `a` is `Z` and where `a` is `(S a')` are syntactically identical, we can use a `default` clause for the `case`-proof to write the subproof just once. The syntax is that you can begin a `case`-proof (or `match`-term, or `induction`-proof) with a clause whose pattern is just "`default`". The body of the clause will be repeated for all constructors which do not have a subsequent pattern in the `case`-proof. If the `default` clause is the only one given (as it will be in our case here), you must write "`default c`" for the pattern, where `c` is the type constructor for the type of the scrutinee. In this case, `c` is just `nat`, and our proof looks like this:

```
foralli(a:nat).
case a with
  default nat => hypjoin (le Z a) tt by a_eq end
end.
```

8.6.2 Finishing the `bst` proof

To return to the final missing subcase of our `bst` proof, we can replace the page-long proof listed in Section 8.5.5 with just one call to `hypjoin`, resulting in this short proof (invoking the induction hypothesis) for the missing case of our theorem:

```
[t_IH l1 u1 t1
   symm
   trans symm u
         hypjoin (bst_in x t) (bst_in x t1)
         by v1 v2 t_eq end]
```

8.7 Summary

We have seen how the `spec` keyword may be used in GURU to designate certain argument positions of term constructors or recursive functions as specificational. Specificational data and proofs are treated as annotations (like type annotations from polymorphic programming), and are dropped during compilation and during theorem proving. This greatly reduces clutter when reasoning externally about dependently typed code. We have seen how a more general form of `induction`-proof is used when doing induction over an indexed datatype. We have also seen a rather complex theorem about the dependently typed function `bst_in` for checking if a piece of data is in a `bst`. During the course of our proof, we saw the `hypjoin` proof construct, which extends the reach of partial evaluation as provided by `join`, to make use of equational hypotheses proven by a set of proofs given to `hypjoin`.

Chapter 9

Resource Management with CARRAWAY

In the previous chapters, we have seen how to write pure functional programs and prove properties about them. Starting with this chapter, we will consider how to write more realistic programs in GURU than just pure functional ones. We will see how to do basic input/output, and how to implement mutable data structures including arrays. The single core issue that it turns out we must address to do this while retaining the ability to prove properties about GURU code is the issue of resource management. Input/output channels, mutable data structure, and other computational resources can be accomodated in our functional setting by viewing them as resources to be managed. In this chapter, we study resource management in GURU. Recently, I have factored out the resource management subsystem of GURU into a stand-alone tool called CARRAWAY. CARRAWAY has its own input language, which is like a simplified version of GURU's, without proofs, indexed types, and nested functions. In addition, CARRAWAY has features which make it possible to describe, as part of the input to CARRAWAY, a variety of resource management policies. GURU programs will be compiled to CARRAWAY programs, which can then be compiled to C code by the CARRAWAY compiler.

9.1 What is a Resource?

Let us see why the idea of a resource is crucial to implementing features like mutable state in GURU. Certain resource management policies enable us to take a pure functional view of operations on resources. The operations as we desire to implement them are not functional, because they destructively modify the program's state in some way (as when we update the value in an array) or do not depend functionally on that state as we normally think of it (as when we get the time of day, for example). But if we manage them carefully, their behavior will be indistinguishable from a different set of operations which are purely functional. For efficient execution – for example, when we compile code – we will use the non-functional implementation. For formal reasoning, however, we can use the functional model.

A simple example, which we will consider in more detail in a later chapter, is that of mutable arrays. Suppose there is at most one reference to a given array at a given point during execution. In our functional model, updating a value in the array is done by creating a new array that is just like the old one, except where it holds the new value. In our non-functional implementation, we destructively modify the array. If there were a second reference to the array, the value of that reference would have to change when the array is updated via the first reference. Such a change would be a non-local effect: variable `x` magically takes on a different value based on something we have done via variable `y`. But if there is only one reference to the array, our functional and non-functional models are operationally indistinguishable, since then there is no need for non-local communication of the fact that the array has changed.

In this array example, it is crucial to ensure that there is at most one reference to the array. This property turns out to be crucial to management of other resources, too. We will consider a resource to be something that only one entity can make use of at a time. This is true of real-world resources. For example, consider a bicycle. We can view it as a resource, if we consider that only one person can ride it at a time. But, you might ask, what if someone sits on the seat and someone else on the handlebars? Or what if it is a tandem bicycle, with two seats? In that case the bicycle is not a monolithic resource, but rather consists of several resources: the handlebars and the seat would each be a resource, and similarly for each of the two seats of the tandem.

By viewing a resource as something that only one entity can use at a time, we get a simple fundamental idea that can form the basis for managing program resources. We need one more idea, however, as shown perhaps a bit more naturally via another example. Consider a box set of the Harry Potter books. We could view each book as a resource, if we consider that only one person can read it at a time. (In this case that is clearly a bit crude, since two people could certainly read through it together simultaneously, but never mind.) If you want to read "The Sorcerer's Stone" and I want to read "The Goblet of Fire", there is no problem, since each is a separate resource. But what if our friend wants to borrow the whole box set? That is fine, as long as all the books are free at the moment. So in a sense, the box set is a single resource, consisting of the individual books as resources. To use the box set, no one can be using any of the individual books. In the general situation, we have one main resource which consists of several subsidiary ones. To use the main resource, none of the subsidiary ones can be in use. If someone is using one of the subsidiary resources, we will say the subsidiary resource is *pinning* the main resource, and that the main resource is *pinned* by that subsidiary one. A pinned resource cannot be used until all the resources pinning it have been returned.

This serves as our basic foundation for resource management in the CARRAWAY language: resources which can only be used by one entity at a time, and which can be pinned by other resources.

9.2 CARRAWAY Overview

This section gives an overview of the basic concepts in CARRAWAY. Subsequent sections will go through the syntax and semantics in detail, via examples. CARRAWAY input files consist of commands, similarly to GURU input files. CARRAWAY commands support three main activities:

- Declaring resource types and primitive operations for manipulating resources of those types (commands `ResourceType`, `Primitive`, `Init`).
- Declaring datatypes, which may be either inductive datatypes, for which pattern-matching and the creation of values using the type's term constructors is allowed; or else unspecified datatypes, which can then only be created by primitives (command `Datatype`).
- Defining functions and global variables (commands `Function` and `Global`).

CARRAWAY reads input files ending in ".w" and translates them to C files with the same name except ending instead in ".c". CARRAWAY checks that resources are used properly by functions and globals, according to the declared resource-managing primitives. It emits code to allocate memory cells appropriately when term constructors are applied. Cells store a numeric tag identifying which constructor of the datatype they belong to. Pattern-matching terms are translated to C `switch` statements that switch on that tag. Primitives are basic resource-manipulating operations. A primitive is specified by giving a CARRAWAY type that shows how it affects the resources it is given as arguments, along with a piece of C code that actually implements the primitive.

CARRAWAY's algorithm for checking that resources are used properly is based on tracking resources to check that there is at most one reference to a resource at any given time, and that pinned resources are not used until the resources pinning them have been returned. To return a resource, we just *consume* it, which we can think of as returning it to the underlying runtime system. To consume a resource, we either:

- pattern-match on it (if it is an element of an inductive datatype),
- pass it as an argument to a function whose type declares that it consumes the resource, or else
- return it from the function

User-defined functions and primitives have some flexibility to declare that they do or do not consume certain arguments. Constructors do not, however: they are considered to consume all their arguments. Arguments to functions and primitives may be designated as treating the resource in one of the following three ways.

- consuming the resource, including possibly returning it directly or inside a data structure built with term constructors. This is the default.

- consuming the resource, but not returning it, neither directly nor inside a data structure. The annotation for this is a caret (^).

- not consuming the resource. The annotation for this is an exclamation point (!).

There is a built-in resource type called `untracked` for things that are not resources, and do not need to be tracked. Additionally, types are runtime data in CARRAWAY, and values whose classifier is `type` are also considered untracked data, not resources. Functions are not considered resources, and are not tracked if they are passed to constructors or other functions.

Pattern-matching terms, whose syntax is similar to `match`-terms in GURU, consume the scrutinee, unless it has type `untracked` or else comes from an input that was marked as not consumed. Except in those cases, the pattern-match consumes the scrutinee either at the end of each case, which is the default; or else at the start of each case, for which a $ annotation is used (between the `match` keyword and the scrutinee). CARRAWAY uses user-specified initialization functions to initialize subdata extracted in the clauses of a pattern-match. This allows different resource management policies to initialize extracted subdata in different ways.

Running CARRAWAY: The CARRAWAY program is `guru-lang/bin/carraway`, which works very similarly to `guru-lang/bin/guru`.

9.3 Reference Counting for Inductive Data

The simplest resource management policy implemented in CARRAWAY is one for managing the memory allocated for elements of inductive types. Unlike in other functional programming languages, GURU (also CARRAWAY) does not rely on garbage collection to manage memory safely. The reason for avoiding garbage collection is that it can severely impact performance in memory-pressured situations (see [**?**, **?**]). In GURU, all our data are inductive data. Since bigger data are built from strictly smaller data, there can be no cycles in our data. Define the reference graph of the program at a particular point in execution as follows. We will allocate a cell (piece of memory) for each application of a constructor. The set of cells which are reachable from a program variable is the set of nodes of the reference graph. There is an edge from one cell to another if the piece of data corresponding to the first cell has the piece of data corresponding to the second as one of its subdata. So immediately after executing $(S\ Z)$, our reference graph will have two cells: one for the application of S, and the other for the use of Z (which we should think of here as a degenerate application of Z to 0 arguments). There is a single edge in the graph, pointing from the cell for the application of S to the cell for the (degenerate) application of Z. With this definition of reference graph, all executions of all programs have acyclic reference graphs at all points in time.

Whenever one is guaranteed to have only acyclic reference graphs, reference counting can be used to manage memory. In each cell we keep an integer value, called the *refcount*, which tells how many references there are to this cell from other cells or from program variables. Whenever a new reference is added, we increment the refcount. When a reference is dropped – for example, when the scope of a local variable referring to the cell ends – then the refcount is decremented. If the refcount falls to 0, that means the cell is garbage, since no one is referring to it. The cell can be recycled in that case, either by returning it to the runtime system (e.g., by calling `free()`), or used for another constructor application. If the refcount overflows, indicating more than the maximum number of references we can store with the bits set aside in the cell for that purpose, then we will treat the cell as *immortal*, and never recycle it. This may leak memory, since all those references may eventually be dropped and we will still not be able to recycle the cell. But it will not corrupt memory by recycling a cell that is really still in use.

A compiler can easily insert code to increment and decrement reference counts for us. But executing these increments and decrements at runtime adds overhead to the execution, which can be avoided in many cases, as we will discuss. So GURU and CARRAWAY provide `inc` and `dec` functions, which we programmers use explicitly to indicate when the reference count should change. To make sure that we perform `inc` and `dec` operations correctly, we treat reference counted data as a resource, and track it with CARRAWAY's reference tracking system, as described next.

9.4 Reference Counting in CARRAWAY

In this section, we will see how to describe reference counted data as a CARRAWAY resource type, and `inc` and `dec` as primitives operating on that type. The code to do this may currently be found in `guru-lang/tests/carraway/unowned.w`. This file begins with a declaration of the `unowned` resource type, for reference counted data:

```
ResourceType unowned with consume_unowned : Fun(A:type)(^ r:unowned).void <<END
  void gconsume_unowned(int A, void *r) {
    dec(r);
    if (op(r) < 256)
      release(A,r);
  }
END
```

Here we see an example of the `ResourceType` command. We have the name of the resource type, which is `unowned` in this case. Then the `with` keyword, and then the declaration of a primitive function for consuming resources of this resource type. That function must be named "`gconsume_T`", where `T` is the name of the resource type. The type of that function comes next, which must be exactly the one given here. That type says that `gconsume_unowned` is a function that takes in a type as its first argument, and then an `unowned r` as its second. The caret annotation means that `r` will not be returned by this function. The function returns `void`, which means that it does not return a value at all. Finally, there is the punctuation "<<" followed immediately by a word (here "`END`") which will mark the end of the raw C code portion which follows. This C code defines how the `unowned` resource is consumed. In our case, we use functions specially provided by CARRAWAY to decrement `r`'s refcount. The refcount is stored in the high 24 bits of the first word (given by `op(r)`) of `r`'s memory cell. So to check if the refcount falls to 0, we check whether that first word falls below 256. If so, we call another function provided by CARRAWAY, to return the memory associated with this cell to the runtime system. The raw C code, here and for all primitives, is expected to have the same name as the CARRAWAY one, except with a "g" prefixing it.

Next, in `unowned.w`, we have declarations of primitive functions for incrementing and decrementing reference counts. From a resource tracking perspective, incrementing a reference count creates a new resource, without consuming the original one; and decrementing consumes a resource. This is indicated by the types of the primitives. The `inc` function states, using the `!` annotation, that it does not consume its input, but it does produce a new output. The `dec` function states, via the caret annotation (^), that it consumes and does not return its input.

```
Primitive inc : Fun(!y:unowned).unowned <<END
  void *ginc(void *y) {
    inc(y);
    return y;
  }
END

Primitive dec : Fun(A:type)(^y:unowned).void <<END
  #define gdec(A,y) gconsume_unowned(A,y)
END
```

Finally, in `unowned.w`, there is an `Init`-command to declare an initialization function for `unowned` resources.

```
Init ginit_unowned_unowned : Fun(A:type)(! x:unowned)(y:unowned).unowned <<END
  void *ginit_unowned_unowned(int A,void *x,void *y) {
    ginc(y);
    return y;
  }
END
```

This function will be automatically used by CARRAWAY to initialize `unowned` subdata `y` when pattern matching on an `unowned` scrutinee `x`. Since the scrutinee will be consumed in that case, initialization needs to increment the subdatum's refcount, since otherwise consumption of the scrutinee could cause the subdatum also to be consumed.

9.5 Programming with Reference-Counted Data

The file `nat.w` in `guru-lang/tests/carraway/` gives an example of a datatype declaration and CARRAWAY programs for reference-counted unary natural numbers. The datatype declaration is somewhat similar to what we have in GURU:

```
Datatype nat := Z : unowned | S : Fun(x:unowned & nat).unowned.
```

The difference is in the input and output types we give for the constructors. Since this is returning reference counted data, the input and output types are `unowned` in all cases. The notation "`& nat`" indicates that the resource type is `unowned` and the datatype is `nat`. All constructors must list datattypes for their input arguments, each of which (as mentioned above) is considered consumed and possibly returned by the constructor. The datatypes are used when we recycle a cell.

Now, let us finally see some examples of programming with reference-counted data. The definitions of `plus` and `mult` are typical. The code for `plus` requires no uses of `inc` and `dec` at all:

```
Function plus(x:unowned)(y:unowned).unowned :=
  match x with
    Z => y
  | S x' => (S (plus x' y))
  end.
```

Let us think about why this code uses resources correctly, even though it contains no `inc`s or `dec`s. The inputs `x` and `y` are marked as ones that are consumed by `plus`, and possibly returned (either directly or in a data structure). The pattern match on `x` consumes it at the end of each case (as mentioned in Section 9.2). So we are sure that `x` is going to be consumed by the time the function returns. In the `Z`-clause, we return `y`, which as mentioned in Section 9.2 is considered a way of consuming it. So `x` and `y` are consumed in that case. In the `S`-clause, we produce a new reference `x'`. This is initialized by incrementing its reference count, since this is what the code given in the `Init`-command from `unowned.w` does (see the previous section). This reference, and `y`, are consumed by the recursive call to `plus`. That call produces a new reference, which is consumed by the call to `S`, which in turn produces a new reference, which is consumed by returning it from the function.

Now let us look at the code for `mult`, where we do need to use `inc` and `dec`:

```
Function mult(x:unowned)(y:unowned).unowned :=
  match x with
    Z => do (dec nat y) Z end
  | S x' => (plus (inc y) (mult x' y))
  end.
```

In the `Z`-clause, we need to consume `y`, because unlike in the `Z`-clause for `plus`, we here return just `Z`, and drop `y`. We use our `dec` function from above, in a `do`-term. The syntax for those is just `do` $t_1 \cdots t_n$ `end` (where n is at least 2). This just executes t_1 through t_n in order, returning the value returned by t_n. In the `S`-clause, we must call `inc` on `y`, since it is used twice.

Let us see what kind of error messages we would get from CARRAWAY if we left off either of these. First, suppose we leave off the `dec` of `y` in the `Z`-clause:

```
Function mult(x:unowned)(y:unowned).unowned :=
  match x with
    Z => Z
  | S x' => (plus (inc y) (mult x' y))
  end.
```

The error message from CARRAWAY is:

```
"/home/stump/guru-lang/tests/carraway/nat.w", line 20, column 3: simulation error.

Two match-cases consume different sets of earlier references.

1. the first case: gZ

2. the second case: gS

3. a reference created at: "/home/stump/guru-lang/tests/carraway/nat.w", line 17, column 25

4. the first case does not consume it.

5. the second case consumes it at: "/home/stump/guru-lang/tests/carraway/nat.w", line 20, column 27
```

This message is telling us that the two clauses of the match have different behavior with regard to references that exist before the match begins. CARRAWAY requires that behavior to be the same in all clauses. The line and column number mentioned in (3) is (in my modified `nat.w` containing this code) is for input variable `y`. The first case does not consume `y`, as (4) states; and the second does, as (5) states. Suppose instead we leave off the `inc` of `y` in the `S`-clause:

```
Function mult(x:unowned)(y:unowned).unowned :=
  match x with
    Z => do (dec nat y) Z end
  | S x' => (plus y (mult x' y))
  end.
```

Then the error message from CARRAWAY is the following, which just notes that we are consuming a resource twice:

```
"/home/stump/guru-lang/tests/carraway/nat.w", line 20, column 13: simulation error.

A reference that was already consumed is being consumed again.

1. the reference created at: "/home/stump/guru-lang/tests/carraway/nat.w", line 17, column 25

2. first consumed at: "/home/stump/guru-lang/tests/carraway/nat.w", line 20, column 21
```

9.6 Pinning References and `owned`

In order to avoid incrementing and decrementing reference counts, `owned.w` in `guru-lang/tests/carraway` defines a resource type `owned`, for references which are owned by another entity, which is thus pinned. Since they are owned by another entity, we do not need to decrement their reference counts: the owning entity cannot be consumed until the owned one is, since the owned one pins the owning one. Thus, we cannot get into the situation where the reference count of the owning entity falls to zero while we are using the owned entity. That situation, of course, would jeopardize memory safety, since recycling the owning cell might cause the owned cell to be reclaimed as well, while we still have a reference to it. By insisting that the owned cell is consumed before the owning cell is, we ensure this cannot happen.

Here are the CARRAWAY commands defining the `owned` resource type and giving the central primitive, `inspect`, which operates on `owned` data:

```
ResourceType owned with consume_owned : Fun(A:type)(^x:owned).void <<END
  #define gconsume_owned(A,x)
END

Primitive inspect : Fun(!x:unowned).<owned x> <<END
```

```
  void *ginspect(void *x) {
    return x;
  }
END
```

Consuming an owned resource does nothing, as the C code given for `consume_owned` shows (that code is a C macro definition, defining "`gconsume_owned(A,x)`" to be nothing, so such expressions just disappear). Inspecting an `unowned` resource does not consume it: the `!` annotation given with the input `x` in the `Fun`-type for `inspect` shows that. But the result of `inspect` is a pinning `owned` reference, as indicated by the return type `<owned x>`. The notation for a pinning type is `<T x1 ... xn>`, where `T` is a resource type and `x1` through `xn` are symbols for pinned entities. Additional primitives for `owned` data allow us to pass back to an `unowned` reference by incrementing the refcount of the `owned` data. This is done consuming the owned reference with the primitive `owned_to_unowned`, and not consuming the owned reference with the primitive `inc_owned`.

```
Primitive inc_owned : Fun(!y:owned).unowned <<END
  void *ginc_owned(void *y) {
    inc(y);
    return y;
  }
END

Primitive owned_to_unowned : Fun(^y:owned).unowned <<END
  void *gowned_to_unowned(void *y) {
    inc(y);
    return y;
  }
END

Primitive clone_owned : Fun(! y:owned).<owned y> <<END
  void *gclone_owned(void *y) {
    return y;
  }
END
```

We can additionally clone an owned reference with `clone_owned`. Notice that the result of this primitive is a new `owned` reference which pins the `owned` reference `y` given to the primitive. Thus, we can build up chains of ownership: an unowned `x` may be pinned by an `owned y`, which in turn (thanks to `clone_owned`) may be pinned by an `owned z`. CARRAWAY provides the construct to collapse two links in such a chain into one. In the situation just described, "`z`" will cause `z` to pin `x` directly, and no longer pin `y`.

The files `test.w` and `test2.w` in `guru-lang/tests/carraway` give several examples of programming with `owned` data. For example, here is an alternative definition of `plus` on unary natural numbers, which manages to do no decrementing of refcounts at all (in contrast, the code for `plus` given in Section 9.5 will decrement one refcount for each `S`-cell of the first argument):

```
Function plus2(^ x:owned)(^ y:owned).unowned :=
  match x with
    Z => (owned_to_unowned y)
  | S x' => (S (plus2 x' y))
  end.
```

The `owned` reference `x` is still consumed by the `match`, but consuming an `owned` reference does not cause the refcount to be decremented. When we return `y` in the `Z`-clause, we have to increment `y`'s refcount, since it is being consumed, and we have marked `x` and `y` as consumed but not returned (with the caret annotation). The advantage

of marking these as not returned is that when `plus2` is called, CARRAWAY's resource tracking algorithm knows that since these references are definitely gone by the time this function exits, at that point they will no longer pin any references they were pinning at the start of the function call. Notice that if we returned `x` or `y`, it would not be safe to drop their pins of those other references: `x` (say) would still exist in the system, and could still be used to access the pinned reference. So it must remain pinned.

Finally, in `owned.w` there are several `Init`-commands for initializing subdata at the start of match cases:

```
Init ginit_unowned_owned : Fun(A:type)(! x:unowned)(y:owned).owned <<END
  #define ginit_unowned_owned(A,x,y) y
END

Init ginit_owned_owned : Fun(A:type)(! x:owned)(y:owned).owned <<END
  #define ginit_unowned_owned(A,x,y) y
END

Init ginit_owned_unowned : Fun(A:type)(! x:owned)(y:unowned).<owned x> <<END
  #define ginit_owned_unowned(A,x,y) y
END
```

The argument `x` is always the scrutinee, and the argument `y` the subdatum. The first `Init` says that if the scrutinee is `unowned` and the subdatum is `owned`, then the subdatum is still `owned` following initialization. Similarly if the scrutinee is `owned` instead of `unowned` (the second `Init`). But if the scrutinee is `owned` and the subdatum is `unowned`, then we initialize the subdatum to an `unowned` piece of data which pins the scrutinee. So in this case, we propagate the property of being `owned` from scrutinee to subdata.

9.7 Standard Input

The file `stdin.w` in `guru-lang/tests/carraway/` gives a simple interface to a textual standard input channel based on the our resource management ideas. The file begins by declaring two opaque datatypes, `stdin_t` and `char`. These are opaque in the sense that they are not inductively defined. We do not have constructors for them. The CARRAWAY code is:

```
Datatype stdin_t with gdelete_stdin_t : Fun(^x:stdin_t).void <<END
  #define gdelete_stdin_t(x) fclose(x)
END

Datatype char with gdelete_char : Fun(^c:char).void <<END
  #define gdelete_char(c)
END
```

When an opaque datatype is defined, a primitive function to recycle the memory for elements of that datatype must also be defined. The datatype `stdin_t` is the type for the standard input channel. We recycle an element of this type by calling the C library function "`fclose`" to close the channel. C programs usually do not close standard input when they are done with it, but this example shows how we can use the delete function to return a resource to the runtime system. Deleting a character does nothing, since characters do not occupy heap-allocated memory.

Next in `stdin.w`, we have a primitive declaration for `stdin` itself. We declare this to be `unique`, which is a resource type with no primitives (except to consume the resource). This resource type is defined in `unique.w`, and can be used for resources where we really require strict unique usage, without any additional management features like increment and decrementing refcounts.

```
Primitive stdin : unique <<END
  #include <stdio.h>

  #define gstdin stdin
END
```

Next, we have primitives to get the current character from `stdin`, and to advance to the next character. Characters, which have *datatype* `char`, have resource type `untracked`, because they do not require heap-allocated memory. Hence, they are not really a resource, and we do not need to track them. We could, of course, if we needed to do so, but in this case, it is more convenient not to track them. To get the current character, we use `curc`, which does not consume the `stdin` resource. The C code for `curc` uses a global variable called `curc` to keep track if we have a current character from `stdin`, or else need to call the C library function `fgetc` to get the next character. To advance to the next character, the primitive `nextc` just clears this global variable, signaling that the `curc` primitive should indeed call `fgetc`.

```
Primitive curc : Fun(!x:unique).untracked <<END

  void *curc = 0;

  int gcurc(void *s) {
     if (curc == 0) {
int tmp = fgetc((FILE *)s);
curc = (tmp == -1 ? 0 : tmp);
     }
     return curc;
  }
END

Primitive nextc : Fun(^x:unique).unique <<END
  void *gnextc(void *x) {
    curc = 0;
    return x;
  }
END
```

Finally, additional primitives are included to check if a character marks the end of the file (`eof`), print a character (`printc`) and close standard input (`close`). The file `stdin.w` then defines functions `read_all` to read all the characters possible from `stdin` and return them in a `ulist` (discussed in a moment), and `print_list` to print all the characters in such a list using `printc`.

```
Primitive eof : Fun(c:untracked).untracked <<END
  int geof(int x) {
    return x == 0;
  }
END

Primitive printc : Fun(c:untracked).void <<END
  void gprintc(int c) {
    putchar(c);
    fflush(stdout);
  }
END

Primitive close : Fun(^x:unique).void <<END
  void gclose(void *s) {
    fclose(s);
  }
END
```

9.8 Lists and Polymorphism

The file `list.w` in `guru-lang/tests/carraway/` defines two inductive datatypes for lists. One if for lists of `unowned` elements, and the other is for lists of `untracked` elements. While these types are polymorphic in the *datatype* of the data stored in the list, they are monomorphic in the resource type of that data. CARRAWAY does not support resource type polymorphism at the moment. Such polymorphism appears challenging to support, since functions may need to take different actions depending on the resource types of data, and it is not clear how to write such functions in a uniform way. The datatype definitions for `list` (`unowned` elements) and `ulist` (`untracked` elements) are:

```
Datatype list := nil : unowned
               | cons : Fun(A:type)(x:unowned & A)(l:unowned & list).unowned.

Datatype ulist := unil : unowned
                | ucons : Fun(x:untracked)(l:unowned & ulist).unowned.
```

These are similar to the datatype definition of `nat` (Section 9.5), except that the datatype listed for `x` in the `cons` constructor for `list` is `A`, which is the first argument of `cons`. When `cons` is applied, we must supply an inductive datatype for the first argument. That argument will not be thrown away during compilation, but really used at runtime, so that where we need to recycle the memory of a `cons`-cell, we know which function should be used to consume the resource `x` (which might require its memory to get recycled, at which point the datatype it belongs to must be known). The `list.w` file defines `append` and `length` functions for `lists`.

9.9 Exercises

Note that the CARRAWAY program is `guru-lang/bin/carraway`, which runs similarly to `guru-lang/bin/guru`. If you put your code in `hw4.w`, you can process the file with CARRAWAY by running:

```
guru-lang/bin/carraway hw4.w
```

This will create a file call `hw4.c`. You can compile that file using `gcc` (the Gnu C Compiler) like this:

```
gcc -o hw4 -O4 hw4.c
```

The "`-o hw4`" part instructs `gcc` to name the binary executable it is producing "`hw4`". The "`-O4`" option tells it to use optimization level 4.

1. Define a function `length` computes the length of a `ulist` (not a `list`). Critically, your function definition should begin this way (indicating the input and output resource types):

   ```
   Function length(l1:unowned).unowned :=
   ```

2. Define an `append` function on `ulists` (again, not `lists`), beginning like this:

   ```
   Function append(l1:unowned)(l2:unowned).unowned :=
   ```

3. To test your functions, use a `Global`-command to write a piece of code which calls `read_all` (from `list.w`) to read all the characters from `stdin` into a `ulist`, then append that `ulist` to itself, and finally compute its length. A good starting point for this can be found in `guru-lang/tests/carraway/test3.w`.

   A few more steps will result in an interesting test. First, type `limit stacksize unlimited` into your shell (if you are using the default shell, which is `tcsh`; if you are using `bash`, type `ulimit -s unlimited`).

This raises the amount of stack memory your program is allowed to consume, which is necessary in this case. Run your compiled `hw4` executable like this: `time ./hw4 < shared196/labs/wrnpc11.txt`. This will cause the contents of the file `shared196/labs/wrnpc11.txt` to be sent to `stdin` of your `hw4` program. Placing `time` at the beginning will just cause the running time used to be printed when the program terminates.

4. Define a `length` function which instead begins like this:

   ```
   Function length(^l1:owned).unowned :=
   ```

5. Define an `append` function which instead begins like this:

   ```
   Function append(^l1:owned)(^l2:owned).unowned :=
   ```

6. Use a `Global`-command to write a similar test as the one of problem (3) for these new functions. You may wish to put the new functions and the new test in a separate file. Test the resulting executable as for problem (3). Compare the running time of this executable with the version from problem (3).

7. Write a function `sublist`, which takes a `nat n` and a `list l`, and returns the sublist of `l` which starts `n` levels deep in `l`. So `(sublist (S Z) (cons nat Z (cons nat (S (S Z)) nil)))` should return `(cons nat (S (S Z)) nil)`. Your function should start this way:

   ```
   Function sublist(^n:owned)(!l:owned).<owned l> :=
   ```

   Recall that matching on a resource which is marked not to be consumed will not consume it. So matching on `l` will not consume `l`. But the subdata of `l` will be initialized according to the initialization rules for `owned` scrutinees (see Section 9.6).

Bibliography

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

[2] C. Chen and H. Xi. Combining Programming with Theorem Proving. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, Tallinn, Estonia, September 2005.

[3] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple Unification-Based Type Inference for GADTs. In J. Reppy and J. Lawall, editors, *ICFP*, pages 50–61, 2006.

[4] D. Licata and R. Harper. A Formulation of Dependent ML with Explicit Equality Proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University School of Computer Science, December 2005.

[5] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

[6] C. McBride and J. McKinna. The View from the Left. *Journal of Functional Programming*, 14(1), 2004.

[7] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent Types for Imperative Programs. In James Hook and Peter Thiemann, editors, *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 229–240, 2008.

[8] U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

[9] E Pasalic, J. Siek, W. Taha, and S. Fogarty. Concoqtion: Indexed Types Now! In G. Ramalingam and E. Visser, editors, *ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation*, 2007.

[10] A. Petcher. Deciding Joinability Modulo Ground Equations in Operational Type Theory. Master's thesis, Washington University in Saint Louis, May 2008. Available from http://www.cs.uiowa.edu/~astump.

[11] 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.

[12] T. Sheard. Type-Level Computation Using Narrowing in Ωmega. In *Programming Languages meets Program Verification*, 2006.

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

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

[15] E. Westbrook, A. Stump, and I. Wehrman. A Language-based Approach to Functionally Correct Imperative Programming. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, 2005.