Getting Started with Whiley

David J. Pearce

April 8, 2015

Abstract

The aim of this document is to provide a short introduction to the Whiley programming language, in order to get you up and running quickly. However, it is not intended to be a definitive reference. We'll walk through a number of simple examples illustrating the most interesting features of Whiley, and show you how to get it up and running. We will be assuming some rudimentary knowledge of programming.

Contents

1 Introduction

The Whiley programming language has been in active development since 2009. The language was designed specifically to help the programmer eliminate bugs from his/her software. The key feature is that Whiley allows programmers to write *specifications* for their functions, which are then checked by the compiler. For example, here is the specification for the `max()` function which returns the maximum of two integers:

```
function max(int x, int y) -> (int z)
// must return either x or y
ensures x == z || y == z
// return must be as large as x and y
ensures x <= z && y <= z:
    // implementation
    if x > y:
        return x
    else:
        return y
```

Here, we see our first piece of Whiley code. This declares a function called `max` which accepts two integers `x` and `y`, and returns an integer `z`. The body of the function simply compares the two parameters and returns the largest. The two **`ensures`** clauses form the function's *post-condition*, which is a guarantee made to any caller of this function. In this case, the `max` function guarantees to return one of the two parameters, and that the return will be as large as both of them. In plain English, this means it will return the maximum of the two parameter values.

When verification is enabled the Whiley compiler will check that every function meets its specification. For our `max()` function, this means it will check that body of the function guarantees to return a value which meets the function's post-condition. To do this, it will explore the two execution paths of the function and check each one separately. If it finds a path which does not meet the post-condition, the compiler will report an error. In this case, the `max()` function above is implemented correctly and so it will find no errors. The advantage of providing specifications is that they can help uncover bugs and other, more serious, problems earlier in the development cycle. This leads to software which is both more reliable and more easily maintained (since the specifications provide important documentation).

1.1 Objectives

Although the primary purpose of Whiley is to allow us to write specifications on functions, we will not talk about that again until later in the document. Furthermore, we will not consider this aspect in detail and, for more, the reader is referred to our tutorial on verification[?].

The primary goal of this article is to introduce the core language of Whiley without worrying about verification (since this presents many challenges and adds complexity). Indeed, it is only once we've understood the basics of Whiley that we will will be ready to investigate verification. Furthermore, Whiley's core language turns out to be rather interesting even without considering verification!

1.2 Installation

There are currently three ways to get setup with the Whiley programming language:

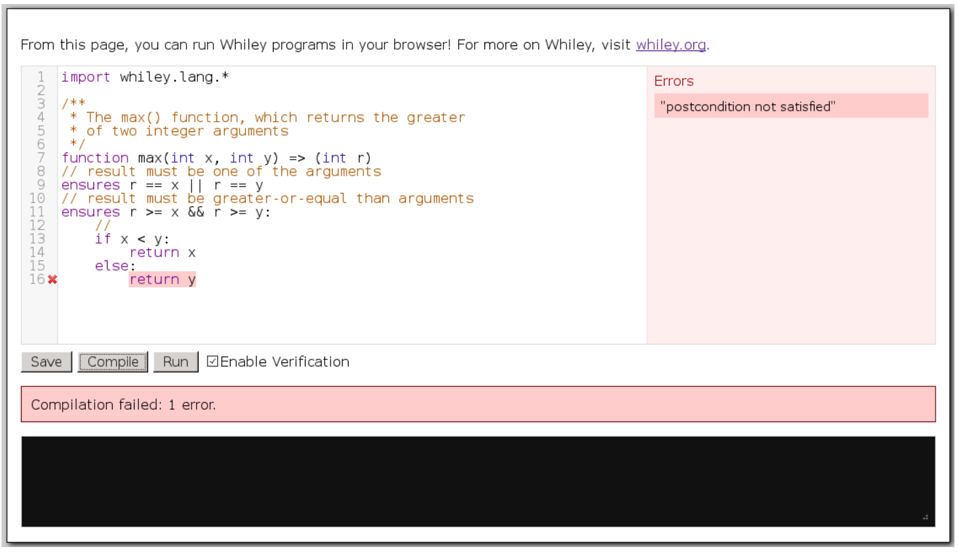
- **Web Browser.** By far the simplest way to get started with Whiley is by running it in your web browser (see Figure 1). Go to `http://whiley.org/play/` and you can get started straight away!

- **Eclipse Plugin.** If you're familiar with the Eclipse IDE or want to develop more serious programs in Whiley, then installing the Eclipse plugin is easy to do. From within Eclipse, choose

Help→Install New Software from the menu. Enter `http://whiley.org/eclipse` as the site, select the "Whiley Eclipse Plugin" and follow the on-screen instructions (see Figure 2).

- **Development Kit.** For those familiar with the command-line, installing the Whiley Development Kit (WDK) is another option. Furthermore, you'll be able to explore the source code for the Whiley system, and see how it all works! To do this, visit `http://whiley.org/downloads/`.

More information of getting started with Whiley can be found at `http://whiley.org/getting-started/`. Finally, the Whiley system is completely free and released under an open source license (BSD), and you can get the latest code from `http://github.com/Whiley`.



From this page, you can run Whiley programs in your browser! For more on Whiley, visit whiley.org.

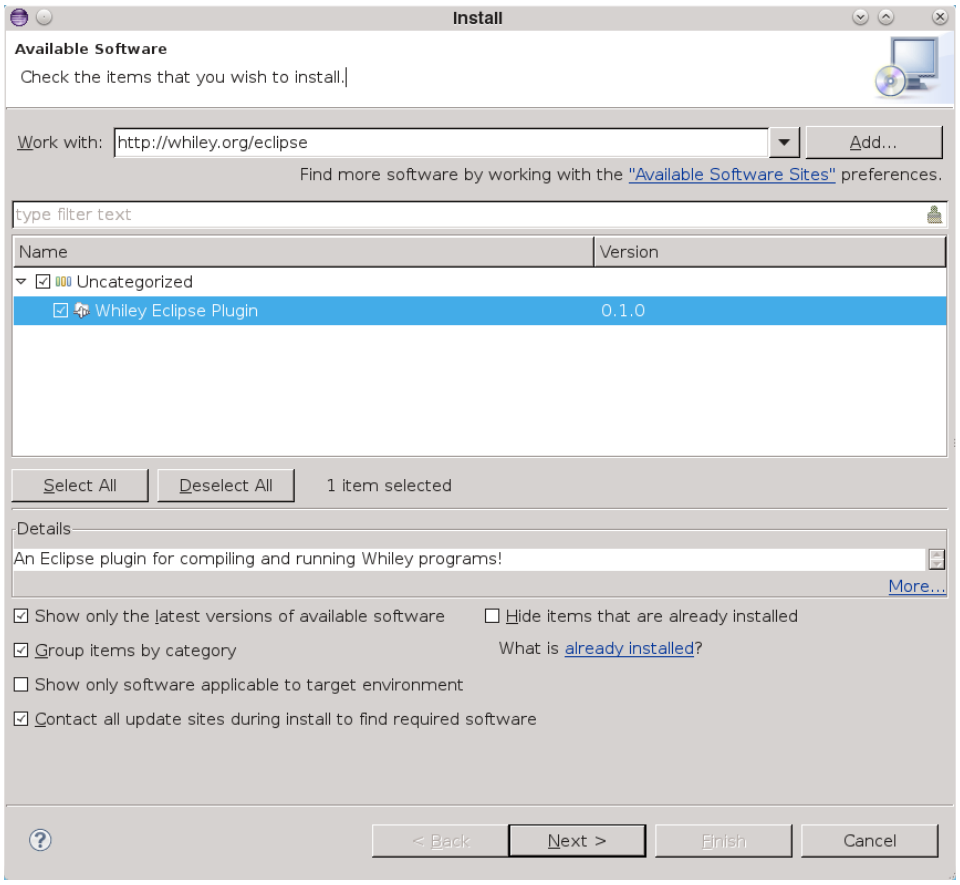
```
import whiley.lang.*

/**
 * The max() function, which returns the greater
 * of two integer arguments
 */
function max(int x, int y) => (int r)
// result must be one of the arguments
ensures r == x || r == y
// result must be greater-or-equal than arguments
ensures r >= x && r >= y:
    //
    if x < y:
        return x
    else:
        return y
```

Errors

"postcondition not satisfied"

Compilation failed: 1 error.

Figure 1: Compiling a Whiley program using a web browser (Mozilla Firefox). At the moment, the user's program is not correct and the system is reporting this as an error in red.



Install

Available Software

Check the items that you wish to install.

Work with: http://whiley.org/eclipse

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☑ Uncategorized	
☑ Whiley Eclipse Plugin	0.1.0

1 item selected

Details

An Eclipse plugin for compiling and running Whiley programs!

More...

☑ Show only the latest versions of available software

☐ Hide items that are already installed

☑ Group items by category

What is already installed?

☐ Show only software applicable to target environment

☑ Contact all update sites during install to find required software

Figure 2: Installing the Whiley Eclipse Plugin from within Eclipse.

2 Quick Walkthrough

This section provides a quick walk through of the main concepts and ideas in the Whiley language. Through a series of short examples, we'll introduce the basic building blocks of the language.

2.1 Booleans and Numbers

As found in many languages, Whiley supports a range of primitive datatypes for representing boolean, integers, real numbers, bytes, etc. Of these, the most commonly used are:

- **Booleans** are denoted by the type `bool`. This is the simplest of the primitive datatypes, and has only two possible values: `true` or `false`.
- **Integers** are denoted by the type `int`. Integers in Whiley are *unbounded*. This means that, in theory at least, a variable of type int can take on *any possible integer value*; this differs from many other languages (e.g. Java), which limit the number of possible values (e.g. following 32-bit two's complement).
- **Real Numbers** are denoted by the type `real`. Reals in Whiley are *unbounded rationals*. This means that, in theory at least, a variable of type real can take on *any possible rational value*. Again, this offers significantly better precision than, for example, `float` or `double` types based on IEEE754 as found in other languages (e.g. Java).

A very simple example which illustrates the `int` and `bool` types is the following:

```
function isLessThan(int x, int y) -> bool:
    //
    if x < y:
        return true
    else:
        return false
```

This declares a simple function which returns `true` if the first parameter, `x`, is less than the second, `y`, and `false` otherwise.

?

Indentation Syntax. From the above example you should notice that Whiley, unlike many languages, does not use curly braces (i.e. `{ ... }`) to demarcate blocks of code. Instead, Whiley uses *indentation syntax* which was popularised by the Python programming language. The start of a new code block is signalled by a preceding **:** on the previous line. The new block must be indented by at least one space (the actual amount doesn't matter) and all subsequent statements with the same indentation are included.

?

Ints versus Reals. Unlike many other languages, Whiley provides a strong relationship between values of type `int` and those of type `real`. Specifically, every `int` value can be represented precisely as a `real` value. Although it may seem surprising, this is not true for many other languages (e.g. Java), where there are `int` (resp. `long`) values which cannot be represented using `float` (resp. `double`)[1].

2.2 Sets, Lists and Maps

Like many modern programming languages, Whiley provides built-in types for representing collections. The following illustrates a short function which multiplies a vector by a scalar:

```
function vectorMultiply([real] vector, real scalar) -> [real]:
    //
    for i in 0 .. |vector|:
        vector[i] = vector[i] * scalar
    return vector
```

This illustrates a few of the common collection operations. Firstly, the size of a collection is obtained using the *length* operator (i.e. `|vector|` returns the length of `vector`). Secondly, the **`for`** loop is useful for iterating over the elements of a collection. In this case, `0 .. |vector|` returns a *list* of consecutive integers from `0` up to (but not including) `|vector|`. Finally, the *list access* operator, `vector[i]`, returns the element at index `i`. The three different kinds of collections supported in Whiley are:

- **Sets** (e.g. `{int}`) provide the simplest form of collection, and are constructing using a *set constructor* (e.g. `{1,2,3}`). They support the *set union* (e.g. `xs + ys`), *set intersection* (e.g. `xs & ys`) and *set difference* (e.g. `xs - ys`) operators. One can test for inclusion using the *element of* operator (e.g. `x in xs`), or the *subset* (e.g. `xs ⊂ ys`) and *subset or equal* (e.g. `xs ⊆ ys`) operators.

- **Maps** (e.g. `{int=>[int]}`) provide a halfway point between sets and lists. They are similar to dictionaries in Python or the `Map` interface in Java. They can be viewed as a set of *key*× *value* pairs, where every key maps to exactly one value. Maps are constructed using a *map constructor* (e.g. `{1=>"hello",2=>"world"}`), and elements are accessed using the *map access* operator (e.g. `map[i]`).

- **Lists** (e.g. `[int]`) are similar to arrays (e.g. in Java), but they can also be resized. As can be seen above, they support the *list access* operator (e.g. `vector[i]`). They also support the *list append* operator (e.g. `[1,2,3] ++ [4,5,6]`) and *sublist* operator (e.g. `vector[0..2]`). Finally, lists are constructed using a *list constructor* (e.g. `[1,2,3]`).

All collection kinds can be iterated using the built-in **`for`** loop construct. For example, here is a function to iterate a map looking for a particular value:

```
// Return the set of all keys which map to a given value
function keysOf({int=>[int]} map, [int] value) -> {int}:
    //
    {int} result = {} // initialise result with empty set
    for k,v in map:        // loop over every key,value pair in map
        if v == value:
            result = result + {k} // add matching keys to result set
    // return result
    return result
```

This function iterates over each key×value pair (i.e. `k,v`) in a from integers to integer lists (i.e. `{int=>[int]}`) using the **`for`** statement.

2.3 Records and Tuples

Aside from collection types, Whiley also allows provides *records* and *tuples* for grouping items together. Records are similar to `structs` (as found in C) and objects (as found in JavaScript). A record is constructed from one or more *fields*, each of which has a unique name and type. For example, the following defines a simple record type for representing 2D points and a function for translating their position:

```
// A point has two integer fields named x and y
type Point is {int x, int y}

// Translate a given point by an x and y delta
function translate(Point p, int dx, int dy) -> Point:
    return {
      x: p.x + dx, // new x value is old value plus dx
      y: p.y + dy  // new y value is old value plus dy
    }
```

Here, a *user-defined type* named `Point` has been defined. This is a record type containing two **int** fields, `x` and `y`. In the `translate()` function, a *record literal* is used to construct a new `Point` to be returned.

Records are an important mechanism for giving meaning to data in Whiley. For example, consider the following declaration of a rectangle:

```
// A point has two integer fields named x and y
type Point is {int x, int y}

// A rectangle has a position, and a width and height
type Rectangle is {
  Point position, // position of top-left corner
  int width,      // width of the rectangle
  int height      // height of the rectangle
}
```

Here, we see that a rectangle has a position, a width and a height. The names of the fields are important for conveying their meaning in the real-world.

Sometimes, using a record type is a little more than necessary because the data being described is really quite simple. In such cases we can use a *tuple* in Whiley, which provide a lightweight mechanism for grouping data. For example, we might consider that using a record to defined our `Point` is a little verbose. Instead, we could rework the example to use a tuple as follows:

```
// Translate a given point by an x and y delta
function translate(Point p, int dx, int dy) -> Point:
    int x, int y = p
    return (x + dx), (y + dy)
```

Since there is a widely used notation for describing 2D points — namely (x, y) — it makes sense in this case to use a tuple over a record. Tuples are also useful for describing functions which return multiple values. For example, the following illustrates a simple function which swaps the order of its parameters:

```
function swap(int x, int y) -> (int, int):
    return y, x
```

Here, we see that the return type is a tuple and this provides a convenient and lightweight mechanism for returning multiple values. Following on from this, Tuple types in Whiley also supports *destructuring syntax* (e.g. as found in Python). The following example illustrates this:

```
    int x
    int y
    ...
    x, y = swap(x,y) // destructuring assignment
    ...
```

Here, we see that variables x and y are assigned their respective component of the tuple returned from swap. Again, this syntax simplies the use of functions with multiple return values.

2.4 Strings and Characters

Unlike many programming languages, Whiley does not provide explicit data types representing strings and characters. The reason for this is that such data types can already be encoded within the language and, by doing so, we can easily support the full range of different encodings (e.g. ASCII versus UTF-8, etc). In Whiley, a string is simply a list of integers. The following example illustrates the well-known replace() function:

```
import string from whiley.lang.ASCII
import char from whiley.lang.ASCII

function replace(string str, char old, char n) => string:
  //
  int i = 0
  while i < |str|:
     if str[i] == old:
        str[i] = n
     i = i + 1
  return str
```

This uses the standard ASCII representation of strings and characters, which are imported from the standard library.

3 Flexible Types

The previous section introduced us to the basic types found in Whiley, such as integers (**`int`**), rationals (**`real`**) and booleans (**`bool`**). However, unlike many languages, Whiley provides a flexible and powerful approach to typing which go well beyond the basic forms. In this section, we will examine this in more detail.

3.1 Flow Typing

To improve the programmer experience and reduce unnecessary tedium, Whiley employs a *flow typing* system. What this means is that the type of a variable can vary at different points within a function. To make this work, Whiley employs *union types*[2;3] along with *variable retyping*. The following example illustrates how this works (where the body of `indexOf()` is left out for brevity):

```
function indexOf([int] items, int item) -> null|int:
   int i = 0
   while i < |items|:
      if items[i] == item:
         return i
      i = i + 1
   return null

function split([int] items, int item) -> [[int]]:
   int|null idx = indexOf(items,item)
   // idx has type null|int
   if idx is int:
       // idx now has type int
       [int] below = items[0..idx]
       [int] above = items[idx..]
       return [below,above]
   else:
       // idx now has type null
       return [items] // no occurrence
```

Here, `indexOf()` returns the first index of an item in a list of items, or **`null`** if there is none. The type **`null|int`** is a *union type*, meaning it is either an **`int`** or **`null`**. The `split()` function splits a list into two pieces based on the first occurrence of a given `item`, or leaves the list as is otherwise. It calls `indexOf()` to determine the first occurrence of `item` in `items`. Observe that variable `idx` has been declared as type **`var`**, meaning the compiler will automatically infer the best possible type for it.

In the above example, Whiley's flow typing system seamlessly ensures that **`null`** is never dereferenced. This is because the type **`null|int`** cannot be treated as an **`int`**. Instead, one must first check it is an **`int`** using a type test, such as "`idx` **`is int`**". Whiley automatically *retypes* `idx` to **`int`** when this is known to be true, thereby avoiding any awkward and unnecessary syntax (e.g. a cast as required in many languages).

?

Null References. In many languages (e.g. C/C++, Java, etc) the use of **`null`** is a significant source of error (see e.g.[4]). For example, in Java dereferencing the **`null`** value gives rise to a `NullPointerException`, which is regarded as the most common form of error in Java[?]. The issue is that, in such languages, one can treat *nullable* references as though they are *non-null* references[5]. In the research literature, there have been many proposals to solve this problem using static type systems (e.g.[6;7;8;9;10;11;12;13]). Unfortunately, at the time of writing, very few languages have incorporated such ideas.

? **Intersections and Negations.** Whiley also supports so-called *intersection* and *negation* types. Whilst these can be expressed directly in source code, they are generally less useful than unions.

3.2 Recursive Types

To represent tree-like structures, Whiley provides *recursive types* which are similar to the algebraic data types found in functional languages (e.g. Haskell, ML, etc). For example:

```
// A linked list is either the empty list or a link
type LinkedList is EmptyList | Link

// The empty list contains no links
type EmptyList is null

// A single link in a linked list
type Link is {int data, LinkedList next}

// Return the length of a linked list (i.e. the number of links it contains)
function length(LinkedList l) -> int:
  if l is null:
    return 0 // l now has type null
  else:
    return 1 + length(l.next) // l now has type int data, LinkedList next
```

Here, `LinkedList` is a recursive type representing a linked list (i.e. a sequence of zero or more links). The empty list is defined as **`null`**, whilst each link contains a `data` field. The type `LinkedList` is defined in terms of itself (i.e. it is recursive) and describes linked lists of arbitrary size.

? **Value Semantics.** As discussed in §**??** all compounds structures in Whiley are passed by value, *including recursive types*. This differs from common languages (e.g. Java), where linked structures are typically composed from *references* to link objects. This means, for example, that linked structures in such languages can share substructures, leading to subtle and hard-to-find bugs. In Whiley linked structures, such as `LinkedList`, can never share substructure.

The above example also serves as another illustration of flow typing in Whiley. More specifically, on the false branch of the type test "`l` **`is null`**", variable `l` is automatically retyped to `{`**`int`**` data, LinkedList next}` — thus ensuring the subsequent dereference of `l.next` is safe. No casts are required as would be needed for a conventional imperative language (e.g. Java). Finally, like all compound structures, the semantics of Whiley dictates that recursive data types are passed by value (or, at least, appear to be from the programmer's perspective).

3.3 Structural vs Nominal Types

Statically typed languages, such as Java, employ *nominal typing* for recursive data types. This means that two otherwise identical types with different names are considered distinct and, for example, a variable of one type cannot flow into the other. In contrast, Whiley employs *structural typing* of records[?] to give greater flexibility. This means that the name of a type is, generally speaking, unimportant. Instead, identical types (i.e. those with identical *structure*) with different names are still considered identical in Whiley. For example:

```
// Define the notion of a "rectangle"
type Rectangle is { int x, int y, int width, int height }
// Define the notion of a "bounding box"
type BoundingBox is { int x, int y, int width, int height }

// Define a function for computing the area of a rectangle
function area(Rectangle rect) -> int:
    return rect.width * rect.height
```

In this example, the types `Rectangle` and `BoundingBox` are *identical* and can be used interchangeably. For example, if we have a variable of type `BoundingBox`, we can safely pass it to the `area()` function above to compute its area.

3.4 Coercions

A *coercion* converts a value of one type into a corresponding value of another type. For example, in Whiley, we can coerce the **`int`** value "`0`" into the **`real`** value "`0.0`". Many programming languages permit both *implicit* and *explicit* coercions, with the latter more commonly referred to as *casting*. Implicit coercions occur without explicit direction from the programmer, and are often considered dangerous because of this.

? **Lossless Coercions?** The Java programming language attempts to enforce a requirement that implicit coercions are *lossless*. Thus, any coercion which may result in a loss of information must be made explicit through the use of a cast. Unfortunately, Java does permit implicit lossy coercions by, for example, allowing **`int`** values to be implicitly coerced into `float` values — because not every **`int`** value can be represented by a `float` in Java.

To address the issues surrounding implicit coercions, Whiley only permits implicit coercions which are lossless. That is, which do not result in a loss of information. In contrast, *lossy* coercions require an explicit cast be used to "force" the coercion. The following example illustrates:

```
type Link is {int data, LinkedList next}
type LinkedList is null | Link
type OrderedList is null | {
  int data, int order, OrderedList next
}
```

Here, we have defined a standard linked list and a specialised "ordered" list. The intuition is that `order < next.order` for each node in an `OrderedList` (although the details of how this is done are unimportant here). These two types are not considered identical because they have different structure (i.e. `OrderedList` has an additional field, `order`). However, there is still a subtyping relationship between them (i.e. `OrderedList` subtypes `LinkedList`). Thus, an instance of `OrderedList` can be used where a `LinkedList` was expected. For example:

```
function sum(LinkedList l) -> int:
    if l is null:
        return 0
    else:
        return l.data + sum(l.next)
```

This defines a simple recursive function for computing the sum of the elements of a `LinkedList`. Instances of `OrderedList` can be passed into this function by *coercing* them to instances of `LinkedList`:

```
function sum(OrderedList l) -> int:
    return sum((LinkedList) l)
```

Here, we have used an explicit coercion (i.e. a cast) from `OrderedList` to `LinkedList`. This must be done explicitly because the coercion is lossy because the field `order` is discarded during the coercion.

> ? **Lossless Coercions.** Whiley (unlike Java) supports lossless coercion from **`int`** values to **`real`** values. This is because arithmetic in Whiley is unbounded and, hence, every value of **`int`** type has a corresponding value of **`real`** type (though not vice-versa).

3.5 Subtyping

An important concept in many modern programming languages is that of *subtyping*. This defines a relationship between otherwise different types (i.e. which do not have identical structure). Subtyping allows data from a variable of one type to flow into a variable of another. However, unlike a coercion, subtyping does not physically change the value itself.

The most common form of subtyping in Whiley is through the use of union types. Indeed, we have encountered this already. The following illustrates a very simple example:

```
// A circle is defined by its position and radius
type Circle is { real x, real y, real radius }

// A rectangle is defined by its position and dimensions
type Rectangle is { real x, real y, real width, real height }

// A shape is either a circle or a rectangle
type Shape is Circle | Rectangle

// Determine the area of a shape
function area(Shape s) -> real:
   if s is Rectangle:
       // case for rectangle
       return s.width * s.height
   else:
       // case for circle
       return Math.PI * (s.radius * s.radius)
```

Here, we see that a `Shape` is either a `Rectangle` or `Circle`. We say that `Rectangle` and `Circle` are subtypes of `Shape`. The Whiley compiler knows that there are only two possibilities and, hence, automatically retypes variable `s` to `Circle` on the false branch.

> ? **A** useful analogy to help understanding the concept of a subtype is that of a *subset*. In this way, we think of types as representing the set of values that their variables may hold. Then, one type is a subtype of another if its set of values is a subset of the other's. Conversely, one type is a *supertype* of another if its set of values is a *superset* of the other's. Furthermore, a union type "`T1 | T2`" corresponds to a set union of those values represented by "`T1`" and "`T2`".

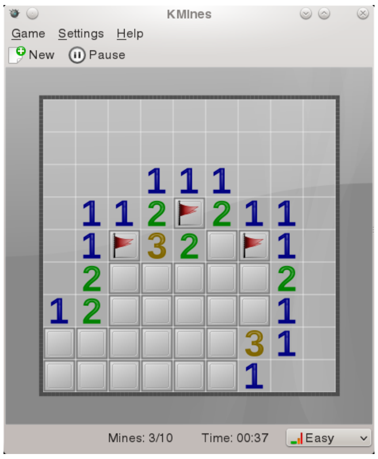
Another situation where subtyping occurs in Whiley is with the types **`any`** and **`void`**. Specifically, every type is a subtype of **`any`**, whilst every type is a *supertype* of **`void`**. To understand this, consider the following:

```
function toInteger(any x) -> int:
    if x is int:
        return x
    else if x is real:
        return Math.round(x)
    else if x is [any]:
        return |x|
    else:
        return 0 // default value
```

In this function, parameter `x` can hold *any possible value on entry*. It could be an integer, a real, a list, a set, a record, etc. In this case, we have picked a few examples which we can easily convert into **`int`** values. Note that the type `[any]` describes all possible lists, including e.g. instances of `[int]`, `[[int]]`, etc.

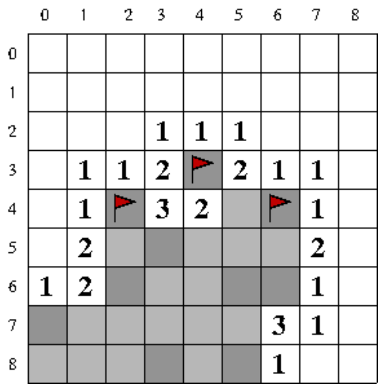
4 Example: Minesweeper

In this section, we will develop a simple implementation of the well-known *Minesweeper* game. Typically the game is played through a graphical user interface, illustrated as follows:



Here, we can see the main aspects of the game. The *game board* is a two-dimensional grid of *squares*. Each square holds *nothing* or a *bomb* and is in one of the three states: *hidden*, *exposed* or *flagged*. An exposed square shows either the total number of bombs in the nine adjacent squares, referred to as its *rank*. If an exposed square contains a bomb, then the game is over and the player has lost. Flagged squares are protected and cannot be exposed unless they are *unflagged*. The intuition here is that the player marks those squares believed to contain a bomb.

Let's analyse the above board. In the following diagram of the above minesweeper game, gray squares represent hidden squares in the game. For our benefit we've split them into two categories: those which contain a bomb (dark gray); and, those which don't (light gray):



In our discussion, we'll use (x, y) to indicate a position on the board where x gives the horizontal column, and y the vertical row. So, for example, the squares $(2, 4)$, $(4, 3)$ and $(6, 4)$ are all marked with a flag. Indeed, we can see that the player has correctly flagged the three bombs in these squares, and that there are seven remaining to be identified and flagged. Of course, unlike us, the player cannot see exactly where the bombs are. However, he/she can easily determine that the square $(2, 6)$ must contain a bomb. This is because the exposed square at $(1, 4)$ has a rank of 1, and a bomb is already flagged at $(2, 4)$. Therefore, there can be no bomb in square $(2, 5)$ as otherwise the rank of square $(1, 4)$ would be incorrect. Finally, the rank of the square at $(1, 5)$ is 2 with only three unexposed squares, of which one is known already to contain a bomb and the other is known *not* to contain a bomb. Therefore, the $(2, 6)$ must contain a bomb.

The player plays the game by repeatedly selecting a square to expose. When all squares are exposed, except for those containing bombs, the game is over and the player wins. However, if a square holding a bomb is exposed, then the game is over and the player loses. A *blank* square is one with no adjacent bombs. When a blank square is exposed, every adjacent blank square is recursively exposed.

4.1 Squares

We're now going to begin implementing the game of Minesweeper in Whiley. To start with, we'll implement the game board in Whiley and provide functions for manipulating it; then, we'll implement the game-play itself.

The first aspect of the game board we'll implement is the concept of a *square*. There are essentially two broad categories of square in the game: *exposed squares* and *hidden squares*. Therefore, our implementation will reflect this. Exposed squares either have a *rank* or are *blank* (i.e. have a rank of zero). Furthermore, they may or may not hold a bomb. We can implement this in Whiley like so:

```
type ExposedSquare is {
  int rank,          // Number of bombs in adjacent squares
  bool holdsBomb     // true if the square holds a bomb
}
```

Here, we can see that an integer field called `rank` is used to store the rank of the square. Likewise, a boolean field called `holdsBomb` is used to indicate whether or not the square holds a bomb. To simplify creating values of type `ExposedSquare`, it is common to additionally provide one or more *constructors*. These are functions of the same name which create values of the given type. Here is our `ExposedSquare` constructor:

```
// ExposedSquare constructor
function ExposedSquare(int rank, bool bomb) -> ExposedSquare:
    return { rank: rank, holdsBomb: bomb }
```

Hidden squares may or may not hold a bomb, and may or may not have been flagged. We can implement this in Whiley as follows:

```
type HiddenSquare is {
  bool holdsBomb,   // true if the square holds a bomb
  bool flagged      // true if the square is flagged
}
```

As before, a boolean field called `holdsBomb` is used to signal whether or not the square holds a bomb. Likewise, a boolean field called `flagged` signals whether or not the square is flagged. Again, we define a constructor as follows:

```
// HiddenSquare constructor
function HiddenSquare(bool bomb, bool flag) -> HiddenSquare:
    return { holdsBomb: bomb, flagged: flag }
```

We can now define the concept of a square in our Whiley implementation by combining the notions of exposed and hidden squares together as follows:

```
type Square is ExposedSquare | HiddenSquare
```

Here, the type `Square` is a union of the types `ExposedSquare` and `HiddenSquare`. In otherwords, it is either an `ExposedSquare` or a `HiddenSquare`. Notice that we don't provide a constructor for `Square`. This is because a `Square` is an abstract concept formed from the composition of two existing types with their own constructors.

4.2 Board

Using our above `Square` data type, we can now define the game board in our Whiley implementation as follows:

```
type Board is {
   [Square] squares,   // List of squares making up the board
   int width,          // Width of the game board (in squares)
   int height          // Height of the game board (in squares)
}
```

The main component of `Board` is the `squares` list. Although this is a one-dimensional list, we'll see shortly that it is treated in a two dimensional way. The remaining fields record the width and height of the board, which is needed in order to safely manipulate the board. To accompany this data type, we define a simple constructor as follows:

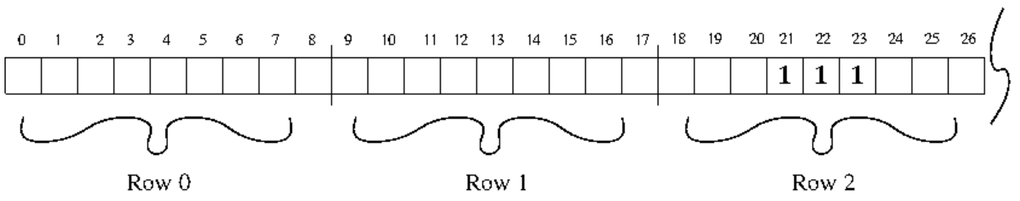
```
// Create a board of given dimensions which contains no bombs, and
// where all squares are hidden.
function Board(int width, int height) -> Board:
    [Square] squares = []
    //
    for i in 0 .. width * height:
      squares = squares ++ [HiddenSquare(false,false)]
    //
    return {
        squares: squares,
        width: width,
        height: height
    }
```

This constructor creates a `Board` of given width and height containing only hidden squares and no bombs. Later, we will return to consider how to randomly place bombs on the board.

We'll now provide some simple helper functions for updating the board. First, we provide a function to read the `Square` at a given position on a `Board`:

```
// Return the square on a given board at a given position
function getSquare(Board b, int col, int row) -> Square:
    int rowOffset = b.width * row // calculate start of row
    return b.squares[rowOffset + col]
```

This function performs a simple calculation to determine the start of the row in the `Board.squares` list. To understand this calculation, we need to view the board in a 1-Dimensional manner, as follows:



Here, we can see how each row is laid out in the 1-Dimensional `Board.squares` list. To calculate the start of a given row, we multiply the row number by the width of the board. Then, to calculate a given column within that row, we simply add the column number. For example, the position $(3, 2)$ represents column 3, row 2; therefore, the position in the example board above would be: $(2 * 9) + 3 = 21$.

The corresponding function to `getSquare()` provides a way to change the square at a given position on the board:

```
// Set the square on a given board at a given position
function setSquare(Board b, int col, int row, Square sq) -> Board:
    int rowOffset = b.width * row // calculate start of row
    b.squares[rowOffset + col] = sq
    return b
```

Here, the same calculation is performed as before to determine the actual position within the `Board.squares` list. This time, the `Board.squares` array is updated with the new `Square`. Note that we must return the updated board in order for this change to be visible (see **??** for more on this). Notice also that we are not attempting to control how the `Board.squares` list may be updated. That is, any `Square` can be passed into this function, even it doesn't make sense in the wider context of the game. This is because these simple provide a general-purpose mechanism for manipulating a `Board`.

4.3 Game Play

Having defined the data types for the Minesweeper game, we can use these to implement the actions of the game. In particular, the user can *flag squares* and *expose squares*. We also need to know when its *game over* and the player has either *won* or *lost*. The easiest of these is that for flagging squares:

```
// Flag (or unflag) a given square on the board. If this operation is not permitted, then do nothing
// and return the board; otherwise, update the board accordingly.
function flagSquare(Board b, int col, int row) -> Board:
   Square sq = getSquare(b,col,row)
   // check whether permitted to flag
   if sq is HiddenSquare:
      // yes, is permitted so reverse flag status and update board
      sq.flagged = !sq.flagged
      b = setSquare(b,col,row,sq)
   //
   return b
```

This function checks whether the square on the board is hidden or not. If so, the flagged status of that square is flipped (i.e. if it was not flagged then it is now, etc). As before, we must return the updated board in order for any change to be visible.

The next function we'll implement is that for exposing a square. This requires that the square to be exposed is not already exposed. Furthermore, in the case of a blank square, then the expose method is recursively applied to blank squares. An important sub-computation to this process is that of determining the rank of a given square. That is the number of bombs contained in adjacent squares. Here is our implementation of this sub-function:

```
function determineRank(Board b, int col, int row) -> int:
    int rank = 0
    // Calculate the rank
    for r in Math.max(0,row-1) .. Math.min(b.height,row+2):
        for c in Math.max(0,col-1) .. Math.min(b.width,col+2):
            Square sq = getSquare(b,c,r)
            if sq.holdsBomb:
                rank = rank + 1
    //
    return rank
```

This function iterates through the nine squares which directly surround that specified by `col` and `row`, whilst excluding those which are off the board. The functions `Math.min()` and `Math.max()` are imported from the standard library and determine the maximum (resp. minimum) of their arguments. Also, note that we can access the field `holdsBomb` without determining whether `sq` is hidden or not. This is because `holdsBomb` is contained in both `ExposedSquare` and `HiddenSquare` and, hence, is guaranteed to be present for any `Square`.

Using the `determineRank()` function above, we can now specify the following function for exposing a given square on the board:

```
// Attempt to recursively expose blank hidden square, starting from a given position.
function exposeSquare(Board b, int col, int row) -> Board:
    // Check whether is blank hidden square
    Square sq = getSquare(b,col,row)
    int rank = determineRank(b,col,row)
    if sq is HiddenSquare:
        // yes, so expose square
        sq = ExposedSquare(rank,sq.holdsBomb)
        b = setSquare(b,col,row,sq)
        if rank == 0:
            // now expose neighbours
            return exposeNeighbours(b,col,row)
    //
    return b
```

This function does one of two things depending on the square being exposed. First, the square is exposed by by creating an `ExposedSquare` and updating the board. Then, if that square is blank (i.e. has a rank of zero), then it and its neighbours are recursively exposed by calling `exposeSquare()` again:

```
// Recursively expose all valid neighbouring squares on the board
function exposeNeighbours(Board b, int col, int row) -> Board:
    for r in Math.max(0,row-1) .. Math.min(b.height,row+2):
       for c in Math.max(0,col-1) .. Math.min(b.width,col+2):
           b = exposeSquare(b,c,r)
    //
    return b
```

The final function we need for our implementation of minesweeper is that for determining when the game is actually over and, furthermore, whether the player has won or not. This examines the board to see whether there are any exposed squares (in which case, the game is over and the player lost). Furthermore, it checks whether or not every hidden square contains a bomb (in which case, the game is over and the player won). This function is implemented as follows:

```
function isGameOver(Board b) -> (bool,bool):
    bool isOver = true
    bool hasWon = true
    for i in 0 .. |b.squares|:
        Square sq = b.squares[i]
        if sq is HiddenSquare && !sq.holdsBomb:
            // Hidden square which doesn't hold a bomb so game may not be over
            isOver = false
        else if sq is ExposedSquare && sq.holdsBomb:
            // Exposed square which holds a bomb so game definitely over
            isOver = true
            hasWon = false
            break
    //
    return isOver, hasWon
```

This function iterates through every square on the board, and checks for two cases: firstly, whether a hidden square exists which does not contain a bomb; secondly, whether there is an exposed bomb. Note that, if an exposed bomb is found the loop exits immediately; however, if a hidden square is found which doesn't hold a bomb we must continue to examine the remaining squares to see whether an exposed bomb exists or not.

4.4 Simple Text Interface

At this point, we can now put the game together and do some preliminary testing to check everything is working as expected. To do this, we can run the code through some simple scenarios from the console. Later, we can extend it with a Graphical User Interface for our game using the Java Swing library.

The first and most important aspect of our simple text interface is a method for drawing the board onto the screen. The following method does this using the `System.Console`:

```
method printBoard(Board board, System.Console console):
    for row in 0 .. board.height:
        // Print Side Wall
        console.out.print("|")
        for col in 0 .. board.width:
            Square sq = getSquare(board,col,row)
            if sq is HiddenSquare:
                if sq.flagged:
                    console.out.print("P")
                else:
                    console.out.print("O")
            else if sq.holdsBomb:
                console.out.print("*")
            else if sq.rank == 0:
                console.out.print("␣")
            else:
                console.out.print(sq.rank)
        // Print Side Wall
        console.out.println("|")
```

This method prints the board using a single character to represent each square, where '`X`' represents hidden squares, numbers represent exposed squares with a given rank and space represents an exposed blank square. For example, here is a simple board drawn in this way:

```
|11        |
|X1 111    |
|1212X1 111|
| 1X211 1XX|
| 111   2XX|
```

Here we can see, for example, that there must be a bomb at square $(0, 1)$. Using this textual representation of a board, we can begin to see the game working. To do this, we'll provide a mechanical notion of a player's move (rather than, say, allowing the player to actually select his/her move). This is done with the following type:

```
// expose signals the player exposes a square (true) or flags it (false)
type Move is { bool expose, int col, int row }

constant MOVES is [
    {expose: true, col: 0, row: 0},        // First move, expose square 0,0
    {expose: false, col: 0, row: 1},       // Second move, flag square 0,1
    {expose: true, col: 2, row: 0}         // Third move, expose square 2,0
]
```

This defines a sequence of moves where the player exposes the square at $(0,0)$, then flags the square at $(0,1)$ and, finally, exposes the square at $(2,0)$.

To execute our sequence of moves on a minesweeper `Board` we need to construct an outer method which first initialises a new `Board`, then reads each `Move` in sequence from `MOVES` and apply's the appropriate function (i.e. `exposeSquare()` or `flagSquare()`) to the `Board`. The following illustrates the first part of a method for doing exactly this:

```
public method main(System.Console console):
    Board board = Board(10,5)
    // Place bombs on the board
    for b in [ (0,1), (2,3), (3,3), (4,4), (4,2), (6,4) ]:
      int x, int y = b
      board = setSquare(board,x,y,HiddenSquare(true,false))
    ...
```

This method creates a new `Board` and places six bombs on to it. The position of each bomb is given using a tuple whose first element is the column, and second element the row. The remainder of the function then iterates through each `Move`, applies it to the `Board` and prints out the updated board at each step:

```
public method main(System.Console console):
    ...
    //
    for m in MOVES:
      // Apply the move
      if m.expose:
        console.out.println("Player␣exposes␣" ++ m.col ++ "," ++ m.row)
        board = exposeSquare(board,m.col,m.row)
      else:
        console.out.println("Player␣flags␣" ++ m.col ++ "," ++ m.row)
        board = flagSquare(board,m.col,m.row)
      // Print the board
      printBoard(board,console)
    // Done
```

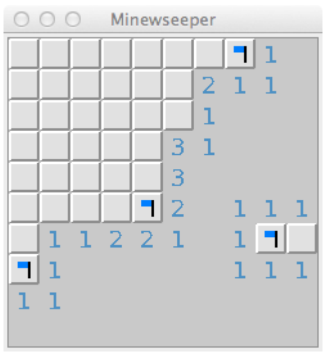
We can now run this little program to check that our minesweeper program appears to be working correctly. The output from doing so should look like this:

```
Player exposes square at 0,0
|1XXXXXXXXX|
|XXXXXXXXXX|
|XXXXXXXXXX|
|XXXXXXXXXX|
|XXXXXXXXXX|
Player flags square at 0,1
|1XXXXXXXXX|
|PXXXXXXXXX|
|XXXXXXXXXX|
|XXXXXXXXXX|
|XXXXXXXXXX|
Player exposes square at 2,0
|11        |
|P1 111    |
|X223X1    |
|XXXXX311  |
|XXXXXXX1  |
```

From this, it certainly seems that the program is working correctly. Note, however, that this is only one test and not enough to be completely certain. Furthermore, we would want to extend our loop above to check whether the game is over and the player has won or lost.

4.5 Graphical User Interface

At this point, we're now going to outline how one can go about turning this code into a real game of minesweeper using a Graphical User Interface (GUI). Since, at the time of writing, the Whiley language has no support for graphical interfaces we instead recommend implementing the GUI in Java using Swing, and then connect that to our Whiley minesweeper code. Here is how our final implementation of the minesweeper game using a Java GUI looks:



In order to connect the Java GUI with our Whiley program, we need to use the *Foreign Function Interface* (see Appendix A for more on this). The essential idea is that, by marking our functions with the modifier **`export`**, they are visible to Java and can be called directly from Java Code. In addition to providing a Graphical User Interface the Java code also provides a mechanism to generate random numbers, allowing us to randomly place bombs on the board.

Finally, complete code for minesweeper game can be downloaded from GitHub here:

`http://github.com/Whiley/WyBench/tree/master/src/107_minesweeper`

This includes both the simple text interface and a Java-based Graphical User Interface. The game is very playable and quite fun!

5 Verification

As discussed in the introduction, an important feature of Whiley is *verification*. That is made up of two aspects: firstly, the ability to write specifications for functions and methods in Whiley; secondly, the ability of the compiler to check the body of a function or method meets its specification.

Unfortunately, specification is not always straightforward and can require considerable attention to detail. Nevertheless, with practice, it can easily fit into the routine of day-to-day development. In this section, we'll explore the basics of verification in Whiley using some small examples. In the following sections, we'll look at larger and more realistic examples.

5.1 Preconditions and Postconditions

A *precondition* is a condition over the parameters of a function that is required to be true when the function is called. The body of the function can then use this to make assumptions about the possible values of the parameters. Likewise, a *postcondition* is a condition over the return values of a function which is required to be true after the function is called. As a very simple example, consider the following function which accepts a positive integer and returns a non-negative integer (i.e. natural number):

```
function decrement(int x) -> (int y)
// Parameter x must be greater than zero
requires x > 0
// Return must be greater or equal to zero
ensures y >= 0:
    //
    return x - 1
```

Here, the **requires** and **ensures** clauses define the function's precondition and postcondition. With verification enabled, the Whiley compiler will verify that the implementation of this function meets its specification. In fact, we can see this for ourselves by manually constructing an appropriate *verification condition* (that is, a logical condition whose truth establishes that the implementation meets its specification). In this case, the appropriate verification condition is `x > 0 ==> x-1 >= 0`. Unfortunately, although constructing a verification condition by hand was possible in this case, in general it's difficult if not impossible for more complex functions.

The Whiley compiler reasons about functions by exploring the different control-flow paths through their bodies. Furthermore, as it learns more about the variables used in the function, it automatically takes this into account. For example:

```
function abs(int x) -> (int y)
// Return value cannot be negative
ensures y >= 0:
    //
    if x >= 0:
        return x
    else:
        return -x
```

The Whiley compiler verifies that the implementation of this function meets its specification. At this point, it is worth considering in more detail what this really means. Since the Whiley compiler performs verification at *compile-time*, it does not consider specific values when reasoning about a function's implementation. Instead, it considers all possible input values for the function which satisfy its precondition. In other words, when the Whiley compiler verifies a function's implementation meets its specification, this means it does so *for all possible input values*.

5.2 Data Type Invariants

The above illustrates a function specification given through explicit pre- and post-conditions. However, we may also employ *constrained types* to simplify it as follows:

```
type nat is (int n) where n >= 0
type pos is (int p) where p > 0

function f(pos x) -> (nat n)
// Return must differ from parameter
ensures n != x:
    //
    return x-1
```

Here, the **type** declaration includes a **where** clause constraining the permissible values for the type ($ represents the variable whose type this will be). Thus, `nat` defines the type of non-negative integers (i.e. the natural numbers). Likewise, `pos` gives the type of positive integers and is implicitly a subtype of `nat` (since the constraint on `pos` implies that of `nat`). We consider that good use of constrained types is critical to ensuring that function specifications remain as readable as possible.

The notion of type in Whiley is more fluid than found in typical languages. In particular, if two types $\texttt{T}_1$ and $\texttt{T}_2$ have the same *underlying* type, then $\texttt{T}_1$ is a subtype of $\texttt{T}_2$ iff the constraint on $\texttt{T}_1$ implies that of $\texttt{T}_2$. Consider the following:

```
 type anat is (int x) where x >= 0
 type bnat is (int x) where 2*x >= x

 function f(anat x) -> bnat:
     return x
```

In this case, we have two alternate (and completely equivalent) definitions for a natural number (we can see that `bnat` is equivalent to `anat` by subtracting `x` from both sides). The Whiley compiler is able to reason that these types are equivalent and statically verifies that this function is correct.

5.3 Quantification

The pre/post-conditions and invariants we have seen above were for constraining primitive types. But, what if we want to say constrain all elements in collection? In that case, we need to use a *quantifier*, as this allows us to iterate all elements in a collection. Suppose we wanted to define the type of all integer lists whose elements are non-negative. We can do this use the *universal* **all** quantifier in Whiley like so:

```
type ListOfNats is ([int] items)
// every x in items must be greater-or-equal-to zero
where all { x in items | x >= 0 }
```

The invariant given in the **where** clause simply states that every element in the list must greater-or-equal-to zero. The **all** quantifier is normally read as "for all".

In Whiley, there are a range of different quantifiers one can use. For example, the above could be rewritten using the **no** quantifier, like so:

```
type ListOfNats is ([int] items)
// no x in items is less-than zero
where no { x in items | x < 0 }
```

Similarly, we could rewrite the above using an *existential* quantifier. Unlike a universal quantifier (which applies to all elements) an existential quantifier applies to just one element. We can rewrite our `ListOfNats` data type using the `some` quantifier in Whiley, like so:

```
type ListOfNats is ([int] items)
// does not exist an x in items where is less-than zero
where !some { x in items | x < 0 }
```

The `some` requires that there is one (or more) elements which meet the criteria, and is normally read as "there exists". Thus, a literal reading of the above invariant would be: *it is not the case that there exists an element x in items where x < 0*. Essentially, however, this has the same effect as for the **no** quantifier.

5.4 Loop Invariants

A loop invariant is a property which holds before and after each iteration of the loop. There are three key points about loop invariants:

1. The loop invariant must hold on entry to the loop.
2. Assuming the loop invariant holds at the start of the loop body (along with the condition), it must hold at the end.
3. The loop invariant (along with the negated condition) can be assumed to hold immediately after the loop.

To illustrate these three aspects, we'll use some simple loop examples. For example, consider the following example:

```
function f(int x) -> (int y)
// return cannot be negative
ensures y >= 0:
    //
    int i = 0
    while i < x where i > 0:
        i = i + 1
    //
    return i
```

Loop invariants in Whiley are indicated by the **where** clause. Thus, in the above example, the loop invariant is "`i > 0`". Compiling the above program with verification enabled will fail with an error. This is because the loop invariant does not hold on entry to the loop (item 1 above).

5.5 Strategies for Loop Invariants

Loop invariants can be tricky to get right, and there are some useful tricks which can simplify things. We'll now consider some examples to illustrate this.

Example 1. Summing over a list of natural numbers is guaranteed to produce a natural number. The following Whiley program illustrates this:

```
type nat is (int x) where x >= 0

function sum([nat] items) -> nat:
    int r = 0
    int i = 0
    //
    while i < |items| where i >= 0 && r >= 0:
        r = r + items[i]
        i = i + 1
    //
    return r
```

The Whiley compiler statically verifies that `sum()` does indeed meet this specification. This is true in Whiley because integer arithmetic is *unbounded* — meaning it does not suffer from overflow as other languages do (e.g. Java). The loop invariant is necessary to help the Whiley compiler verify this function. However, we can avoid the need for a loop invariant by declaring variables `i` and `r` more precisely:

```
function sum([nat] items) -> nat:
    nat r = 0
    nat i = 0
    //
    while i < |items|:
        //...
```

This time, we have declared the variables `i` and `r` as having type `nat`. The Whiley compiler will now enforce the `nat` property for `i` and `r` at all points in the function, and the loop invariant is no longer required.

Example 2. Generally speaking, the loop condition and invariant are used independently to increase knowledge. However, sometimes they need to be used in concert. Consider the following function for initialising a list of a given size:

```
function create(int count, int value) -> ([int] r)
// Cannot create negatively sized lists!
requires count >= 0
// Returned list must have count elements
ensures |r| == count:
   //
   int i = 0
   [int] r = []
   while i < count:
      r = r ++ [value]
      i = i + 1
   //
   return r
```

This example uses the list append operator (i.e. `r ++ [value]`) and is surprisingly challenging to verify. An obvious approach is to connect the size of `r` with `i` as follows:

```
   ...
   while i < count where |r| == i:
      ...
```

Unfortunately, this loop invariant is not strong enough to allow this function to be verified. To understand this, recall from §5.4 that, after a loop is complete, the loop invariant holds along with the *negated* condition. Thus, after the above loop, we have `i >= count && |r| == i` which is insufficient to establish `|r| == count`. In fact, we can resolve this by using an *overriding loop invariant* as follows:

```
   ...
   while i < count where i <= count && |r| == i:
      ...
```

In this case, `i >= count && i <= count && |r| == i` holds after the loop and, hence, it follows that `|r| == count`.

5.6 Function Invocation

To keep verification tractable, the Whiley compiler verifies each function in a program one at a time, independently of others.[1] Thus, when verifying a given function, it assumes that all other functions correctly meet their specification. Of course, if this is not the case, then this will eventually be discovered as the compiler progresses through the program. For example, consider this program:

```
function f(int x) -> (int y)
// Return cannot be negative
ensures y >= 0:
    //
    return x

function g() -> (int y)
// Return cannot be negative
ensures y >= 0:
    //
    return f(1)
```

This program will not verify because the implementation of `f()` does not meet its specification. For example, `f(-1)` gives `-1` but the post-condition for `f()` allows only non-negative integers to be returned. However, the Whiley compiler will verify that the implementation of `g()` meets its specification as, when doing this, it assumes that `f()` meets it specification.

5.7 Explicit Assumptions

In many cases it is possible to sufficiently encode a function's meaning within its specification to allow a program to verify. However, occasionally, what we need is to difficult for the Whiley compiler to handle. A good example is the `sum()` function, which we can try to specify as follows:

```
function sum([int] xs) -> (int r)
requires |xs| > 0
// Base case: list of size 1
ensures |xs| == 1 ==> r == xs[0]
// General case: list of size greater than 1
ensures |xs| > 1 ==> r == xs[0] + sum(xs[1..]):
   //...
```

Here, we have carefully encoded the meaning of `sum()` within its specification. Unfortunately, at the time of writing, the Whiley compiler reason accurately about this specification, and this makes exploiting known mathematical properties in subsequent specifications impossible. For example, the following illustrates a simple function for reversing a list:

```
// Rotate head item to back of list
function reverse([int] xs) -> ([int] ys)
// Permutation does not change sum
ensures |xs| > 0 ==> sum(xs) == sum(ys):
   //
   int i = 0
   [int] zs = xs
   //
   while i < |xs| where i >= 0 && |xs| == |zs|:
       int j = |xs| - (i+1)
       xs[i] = zs[j]
   return xs
```

[1] This corresponds to performing an *intra-procedural* analysis, compared with a more involved *inter-procedural* analysis.

Unfortunately, there is no hope to automatically verify the seemingly straightforward property that the sum is preserved by this function. Instead, one can use an `assume` statement in Whiley to override the verifier. Such a statement provides a way to instruct the verifier to blindly assume something holds. For this example, we have:

```
    ..
    assume |xs| == 0 || sum(xs) == sum(zs)
    return xs
```

Placing this before the **`return`** statement allows the verifier to pass `reverse()`. Of course, the use of `assume` statements is potentially unsafe and relies on correct judgement.

6 Example: IndexOf Function

To better illustrate verification in Whiley, we'll develop the specification for a slightly more challenging function. This is the `indexOf()` function, described as follows:

```
// Return the lowest index in the items list which equals the given item.
// If no such index exists, returns null.
function indexOf([int] items, int item) => int|null:
    ...
```

This is a common function found in the standard libraries of many programming languages. The body of the function examines each element of the `items` list and check whether or not it equals `item`. To start with, we won't worry too much about the body of the `indexOf()` function. Instead, we'll progressively build up the specification until we are happy with it. Then, we'll give an implementation of the function which meets this specification.

To specify this function, we want to ensure three properties:

1. If the return is an integer `i`, then `items[i] == item`.
2. If the return is **`null`**, there is no index `j` where `items[j] == item`.
3. If the return is an integer `i`, then there is no index `j` where `j < i` and `items[j] == item`.

These properties determine how a correct implementation of the `indexOf()` function should behave. We refer to them as the *specification* of the `indexOf()` function.

6.1 Specifying Property 1 — Return Valid Index

The first of the above properties is the easiest, so lets start by specifying that in Whiley. At the same time, we'll also give an initial implementation which satisfies this partial specification:

```
function indexOf([int] items, int item) -> (int|null i)
// If return value is an int i, then items[i] == item
ensures i is int ==> items[i] == item:
    //
    if |items| > 0 && items[0] == item:
        return 0
    else:
        return null
```

Here, we can see property (1) above written as an **`ensures`** clause in Whiley. In particular, the phrase "the return value is an integer" is translated into the condition "`i is int`". Likewise, the implication operator (i.e. `==>`) is used to say "If ... then ...". We've also given an initial implementation for the `indexOf()` function which simply checks whether or not `items[0] == item`. This implementation meets the specification we have so far although, obviously, this is an incomplete implementation of the `indexOf` function!

6.2 Specifying Property 2 — Return Null if No Match

Property (2) from our list above is more difficult to specify, because it requires *quantification*. There are several quantifiers available in Whiley, including: **`all`**, which allows us to say "for all elements in a list something is true"; and **`no`**, which allows us to say "there is no element in the list where something is true".

In Whiley, we can express property (2) from above in several different ways. The most direct translation would be:

```
...
// If return is null, there is no index j where items[j] == item
ensures i is null ==> no { j in 0..|items| | items[j] == item }:
    ...
```

Here, the expression `|items|` gives the length of the items list, whilst the range expression `0..|items|` returns a list of consecutive integers from `0` up to, but not including, `|items|`. Instead of using the **`no`** quantifier, we could have equally used the **`all`** quantifier, like so:

```
...
// If return is null, there is no index j where items[j] == item
ensures i is null ==> all { j in 0..|items| | items[j] != item }:
    ...
```

The above, however, is perhaps not as clear as the first translation. Finally we can, in this case, avoid talking about indices altogether like so:

```
...
// If return is null, there is no index j where items[j] == item
ensures i is null ==> no { x in items | x == item }:
    ...
```

The above simple says "there is no element `x` in `items` where `x == item`". Although this is also not the most direct translation of the original property, it is a rather convenient translation which achieves the same thing.

6.3 Specifying Property 3 — Return Least Index

Property (3) from our list above is similar to property (2), except that we not considering all elements of `items`:

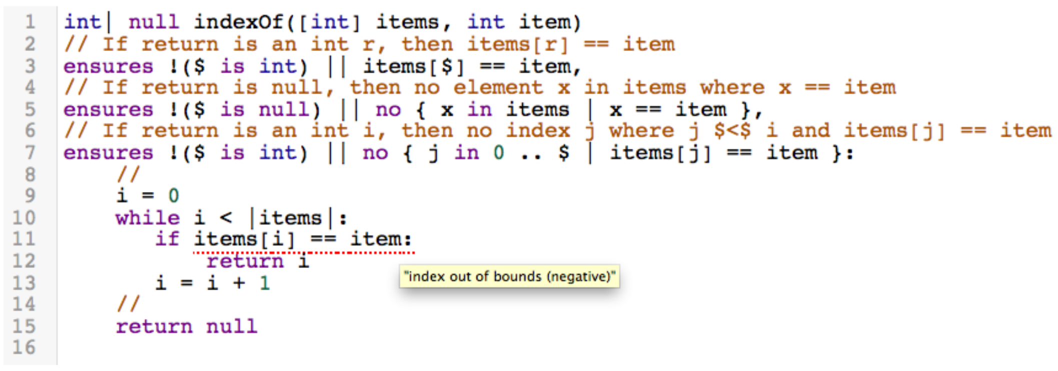
```
...
// If return is an int i, then no index j where j < i and items[j] == item
ensures i is int ==> no { j in 0 .. i | items[j] == item }:
```

As before, an **`all`** quantifier could be used. However, we cannot avoid talking about indices this time as not all elements of `items` are being considered.

6.4 Working Implementation

At this point, we can now give the complete specification for the `indexOf()` function, along with an initial implementation:

```
function indexOf([int] items, int item) -> (int|null i)
// If return is an int r, then items[r] == item
ensures i is int ==> items[i] == item
// If return is null, then no element x in items where x == item
ensures i is null ==> no { x in items | x == item }
// If return is an int i, then no index j where j < i and items[j] == item
ensures i is int ==> no { j in 0 .. i | items[j] == item }:
    //
    int i = 0
    while i < |items|:
       if items[i] == item:
           return i
       i = i + 1
```



```
int| null indexOf([int] items, int item)
// If return is an int r, then items[r] == item
ensures !($ is int) || items[$] == item,
// If return is null, then no element x in items where x == item
ensures !($ is null) || no { x in items | x == item },
// If return is an int i, then no index j where j $<$ i and items[j] == item
ensures !($ is int) || no { j in 0 .. $ | items[j] == item }:
    //
    i = 0
    while i < |items|:
        if items[i] == item:
            return i
        i = i + 1
    //
    return null
```

"index out of bounds (negative)"

Figure 3: Illustrating our first working version of the `indexOf` function being compiled with verification enabled. The compiler is reporting an error stating "*index out of bounds (negative)*". This is because the compiler believes `i` may be negative at this point. Although we know this is not true, we must write a *loop invariant* to help the compiler see this.

```
    //
    return null
```

The implementation of `indexOf()` given above meets the function's specification. Unfortunately, whilst this is true, the Whiley compiler needs help to determine this. Figure 3 illustrates what happens when we compile the above code with verification enabled.

6.5 Verified Implementation

Although our implementation of `indexOf()` given above is correct, it currently does not verify. Although this distinction may seem unimportant, it goes to the heart of what verification is about. That is, we know the implementation of `indexOf()` is correct because we, *as humans*, have looked at it and believe it is. Whilst may be a reasonable approach for small examples, it certainly is not for larger and more complex programs. Humans are fallible and we can easily believe something is true when it is not. Therefore, we want a mechanical system which can examine a program and report "*Yes, I agree that this is correct*". Whiley provides such a system when verification is enabled.

Unfortunately, Whiley is not as smart as a human and often there will be things we know that it does not. In such cases, we need to help Whiley by adding hints into our programs. In this case, we need to add some loop invariants (recall §5.4) to help Whiley verify our implementation of `indexOf()`. The first part of the loop invariant we need is straightforward. Since `i` is modified in the loop, we need am invariant to ensure `i >= 0` when `items[i]` is accessed:

```
    ...
    i = 0
    while i < |items| where i >= 0:
        ...
        i = i + 1
```

We can see that this invariant holds on entry to the loop (i.e. since `i = 0` on entry). Furthermore, if `i >= 0` then `i+1 >= 0` follows and, hence, the loop invariant holds after each iteration.

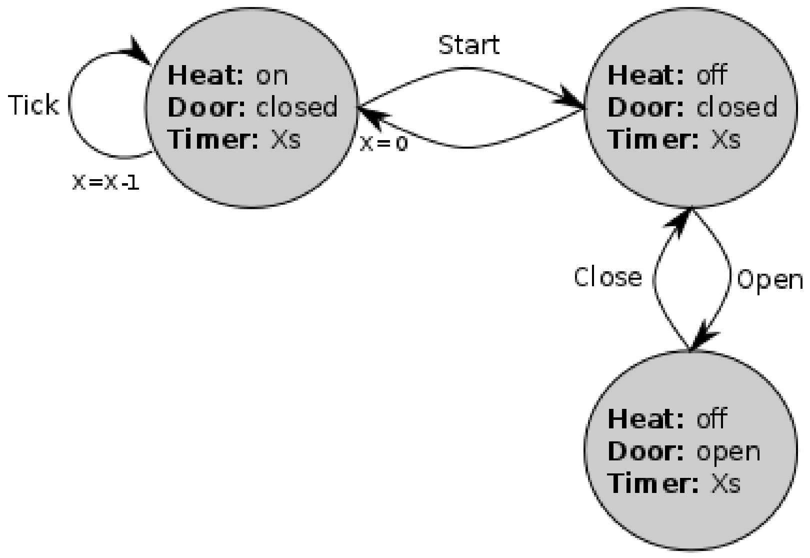


Figure 4: A state-machine diagram for the microwave oven

7 Example: Microwave Oven

For the next case study, we examine the classic and well-known *microwave oven* problem. In essence, we have a microwave oven with a door and a heating element. An important safety condition is that the heating element cannot be on when the door is open, to protect against burns. This example is interesting, because it demonstrates that Whiley can operate effectively as a modelling language, as well as a general-purpose programming language.

7.1 Overview

Figure 4 provides a state-machine diagram of the microwave oven. The microwave state has three components: a heating element, which is set either *on* or *off*; a door which (via a sensor) is recorded as either *open* or *closed*; and, finally, a timer value indicating how for many seconds heating should occur.

In addition to the microwave state, a number of external events are permitted. First, a start button is used to signal the microwave should begin heating; second, the door may be opened or closed; finally, an internal clock is used to signal time passing (in one second intervals) to the state machine.

7.2 Microwave State

The state of the microwave is represented in Whiley using a record containing the main three components, along with an appropriate invariant. The following illustrates:

```
type nat is (int x) where x >= 0

// First, define the state of the microwave.
type Microwave is {
        bool heatOn, // if true, the oven is cooking
        bool doorOpen, // if true, the door is open
        nat timer // timer setting (in seconds)
} where !doorOpen || !heatOn
```

Here, boolean values are used to represent the state of the heating element, and the door sensor, whilst a natural number represents the timer value. The invariant states that either the door is closed, or the heating element is off.

7.3 Events

Events can be modelled in Whiley using functions which map one microwave state to another. Here are the two functions representing the *open* and *close* events:

```
// A door closed event is triggered when the sensor detects that the door is closed.
function doorClosed(Microwave m) => Microwave
requires m.doorOpen:
        //
        m.doorOpen = false
        return m

// A door opened event is triggered when the sensor detects that the door is opened.
function doorOpened(Microwave m) => Microwave
requires !m.doorOpen:
        //
        m.doorOpen = true
        m.heatOn = false
        return m
```

Here, we can see that preconditions on the functions act as guards restricting when the events may fire. The Whiley compiler will statically verify that the `Microwave` invariant holds for the turn value, assuming it held for the parameter. Thus, failing to set `m.heatOn = false` in `doorOpened()` results in a compile time error which signal that the safety property is not be enforced.

Likewise, we can specify the *start* event and, in doing so, we must ensure the safety property is enforced. Specifically, when the heating element is turned on, the door must be closed:

```
// Signals that the "start cooking" button has been pressed.
function startCooking(Microwave m) => Microwave:
        //
        // Here, we check the all important safety property
        if !m.doorOpen:
                m.heatOn = true
        return m
```

Here we can see that, if the door is open when the start button is pressed, nothing happens. Again, failing to check whether the door is open in `startCooking()` results in a compile time error which signal that the safety property is not be enforced.

Appendix

A Foreign Function Interface

The *Foreign Function Interface (FFI)* provides a mechanism to enable code written in Whiley to interact with code written in another programming language. In this section, we will discuss those mechanisms provided in Whiley for this purpose. Of course, the Whiley system cannot make any guarantees about such external code and care must be taken to ensure it treats types and specifications correctly.

A.1 Overview

The foreign function interface represents a boundary between internal (i.e. Whiley) code and external code written in other (i.e. *foreign*) languages. Whiley provides two modifiers for functions and methods which allow information to flow across this boundary. These are:

- The `native` modifier, which declares a function or method which is *implemented* in a foreign language. That is, a function or method whose implementation is written in another language. Code written in Whiley can call this function, and the compiler is responsible for correctly routing this to the actual function body.
- The `export` modifier, which declares a function or method that is *visible* to foreign code. That is, a function or method which may be called from foreign code. This provides a way for foreign code to invoke Whiley function and methods.

These modifiers provide two different mechanisms for allowing information to flow between code written in Whiley and code written in a foreign language. As an example, recall the Minesweeper example from §4. Suppose we wanted to implement a Graphical User Interface for our minesweeper game in Java. There are two ways we could approach this:

- **Whiley as "Master"**. In this case, the Whiley code is considered the primary implementation and provides the entry point to the application. In contrast, the Java code is consider subservient and is always called from Whiley.
- **Whiley as "Slave"**. In this case, things are the other way around. The Java code is considered the primary implementation and provides the application's entry point. The Whiley code, on the other hand, simply provides supporting functions and methods for this.

For this particular situation, it probably makes most sense to follow the "Whiley as Slave" approach. This makes sense if we view our completed Minesweeper game as an instance of the *Model-View-Controller (MVC)* pattern. According to this, the Whiley implementation of Minesweeper illustrated in §4 corresponds to the *model*. The Graphical User Interface would then correspond to the *view*.

B Verification Conditions

Talk about how to generate and see verification conditions.

References

[1] J. Gosling, G. Steele B. Joy, and Gilad Bracha. *The Java Language Specification, 3rd Edition*. Prentice Hall, 2005.

[2] Franco Barbanera and Mariangiola Dezani-Cian Caglini. Intersection and union types. In *In Proc. TACS*, pages 651–674, 1991.

[3] A. Igarashi and H. Nagira. Union types for object-oriented programming. *Journal of Object Technology*, 6(2), 2007.

[4] Tony Hoare. Null references: The billion dollar mistake, presentation at qcon, 2009.

[5] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[6] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attributes. In *Proceedings of the confererence on Compiler Construction (CC)*, pages 334–554, 2001.

[7] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 302–312. ACM Press, 2003.

[8] T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 6(9):455–475, 2007.

[9] Maciej Cielecki, Jędrzej Fulara, Krzysztof Jakubczyk, and Lukasz Jancewicz. Propagation of JML non-null annotations in Java programs. In *Proceedings of the conference on Principles and Practices of Programming in Java (PPPJ)*, pages 135–140. ACM Press, 2006.

[10] Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*, pages 227–247. Springer, 2007.

[11] Chris Male, David J. Pearce, Alex Potanin, and Constantine Dymnikov. Java bytecode verification for @NonNull types. In *Proceedings of the confererence on Compiler Construction (CC)*, pages 229–244, 2008.

[12] Laurent Hubert. A non-null annotation inferencer for java bytecode. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, pages 36–42. ACM, 2008.

[13] Laurent Hubert, Thomas Jensen, and David Pichardie. Semantic foundations and inference of non-null annotations. In *Proceedings of the International conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 132–149. Springer-Verlag, 2008.