Getting Started with Whiley

David J. Pearce

December 29, 2013

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 **`requires`** 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`.

1.3 Running Whiley

Illustrate Hello World

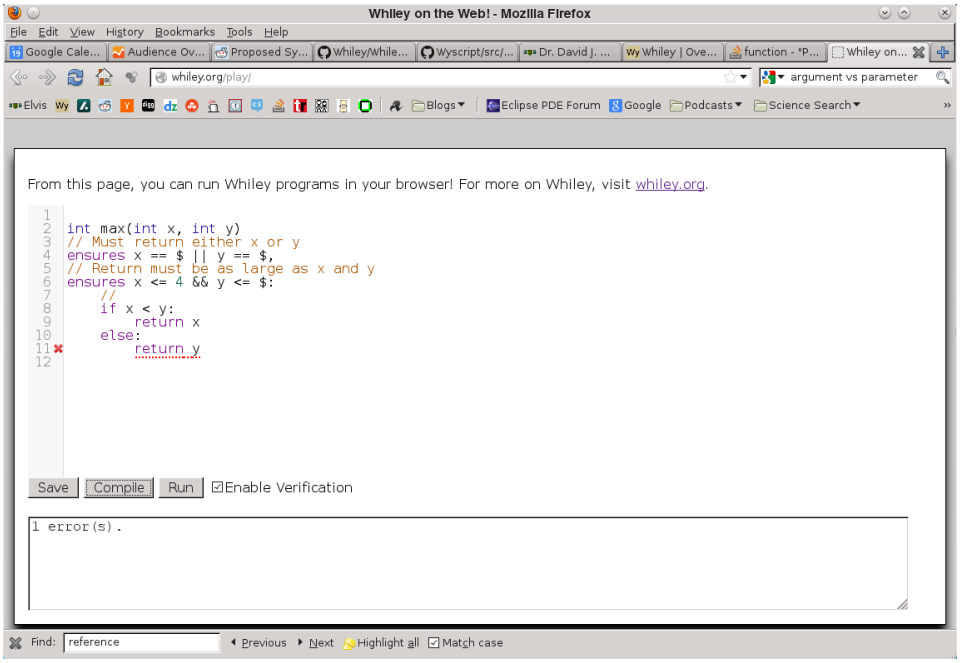

Figure 1: Compiling a Whiley program using a web browser (Mozilla Firefox). At the moment, the user's program is not correct and the system is reporting this as an error in red.

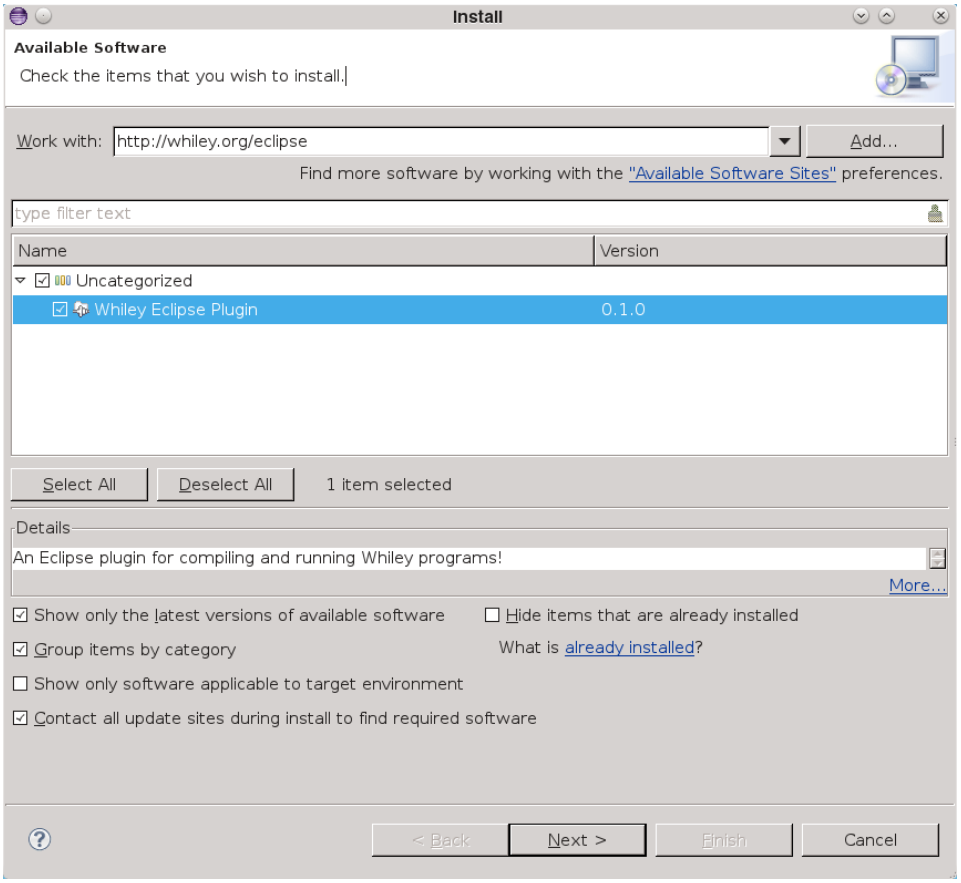

Figure 2: Installing the Whiley Eclipse Plugin from within Eclipse.

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, characters, 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:

```
[real] vectorMultiply([real] vector, real scalar):
    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=>string}`) 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*$\times$ *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=>string} map, string value) => {string}:
    {string} 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$\times$value pair (i.e. `k,v`) in a from integers to strings (i.e. `{int=>string}`) using the **`for`** statement.

2.3 Records and Tuples

2.4 Strings and Characters

2.5 Value Semantics

In Whiley, all compound structures (e.g. lists, sets, and records) have *value semantics*. This means they are passed and returned by-value (as in Pascal, MATLAB or most functional languages). But, unlike functional languages (and like Pascal), values of compound types can be updated in place.

Value semantics implies that updates to a variable only affects that variable, and that information can only flow out of a function through its return value. Whiley has no general, mutable heap comparable to those found in object-oriented languages. Consider:

```
int f([int] xs):
    ys = xs
    xs[0] = 1
    ...
```

The semantics of Whiley dictate that, having assigned `xs` to `ys` as above, the subsequent update to `xs` does not affect `ys`. Arguments are also passed by value, hence `xs` is updated inside `f()` and this does not affect `f`'s caller. That is, `xs` is not a *reference* to a list of **`int`**; rather, it *is* a list of **`int`**s and assignments to it do not affect state visible outside of `f()`.

Whilst this approach may seem inefficient, a variety of techniques exist (e.g. reference counting) to ensure efficiency (see e.g.[**???**]). Indeed, the underlying implementation does pass compound structures by reference and copies them only when absolutely necessary.

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*[? ?] along with *variable retyping*. The following example illustrates how this works (where the body of `indexOf()` is left out for brevity):

```
function indexOf(string str, char c) => null|int:
   ...

function split(string str, char c) => [string]:
   var idx = indexOf(str,c)
   // idx has type null|int
   if idx is int:
        // idx now has type int
        int below = str[0..idx]
        int above = str[idx..]
        return [below,above]
   else:
        // idx now has type null
        return [str] // no occurrence
```

Here, `indexOf()` returns the first index of a character in the string, or **null** if there is none. The type **null**|**int** is a *union type*, meaning it is either an **int** or **null**.

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.[?]). 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[2]. In the research literature, there have been many proposals to solve this problem using static type systems (e.g.[? ? ? ? ? ? ? ? ?]). 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.

? **Untagged Unions** . Often confusion surrounding untagged versus tagged unions. The latter are more common, and sometimes known as *sum types*.

3.2 Recursive Types

To represent tree-like structures, Whiley provides *recursive types* which are similar to the abstract 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)
int length(LinkedList l):
  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 §2.5 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 Types

Statically typed languages, such as Java, employ nominal typing for recursive data types. This results in rigid hierarchies which are often difficult to extend[?]. In contrast, Whiley employs *structural subtyping* of records[?] to give greater flexibility.

Suppose we wish to extend our expressions from Figure 3 with assignment statements. A common issue arises as the left-hand side of an assignment is a restricted form of expression, often called an *lval*. In a language like Java, we can capture this nicely using interfaces:

```
interface Expr { ... }
interface LVal { ... }
class ListAccess implements Expr,LVal { ...}
class Var implements Expr,LVal { ... }
class Int implements Expr { ... }
```

However, suppose the code for expressions was part of an existing library, and we are trying to add statements after the fact. In a language like Java, this presents a problem as we cannot retroactively insert the necessary `LVal` interface to `Var` and `ListAccess`.

In Whiley, adding the notion of an `LVal` is easy to do retroactively because of structural subtyping:

```
define LVal as Var | ListAccess
define Assign as {LVal lhs, Expr rhs}

Expr parseExpression():
    ...

null|Assign parseAssign():
    le = parseExpression()
    match(":=")
    re = parseExpression()
    if le is LVal:
        return {lhs: le, rhs: re}
    else:
        return null // syntax error
```

Here, `LVal` is implicitly a subtype of `Expr` — i.e. there is no need for an explicit declaration of this, as would be required in Java. That is, they can be defined entirely separately from each other (e.g. in different files, packages or entirely separate programs) — and yet, `LVal` remains a subtype of `Expr`.

3.4 Structural Subtyping.

Whiley permits subtyping between recursive structural types.

```
define Link as {int data, LinkedList next}
define LinkedList as null | Link
define OrderedList as null | {
  int data, int order, OrderedList next
}

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

Here, we have defined a standard linked list and a specialised "ordered" list where `order < next.order` for each node (see e.g. [?]). Whiley type checks this function by showing that `OrderedList` is a structural subtype of `LinkedList` — despite this relationship not being identified explicitly in the program. Type checking in the presence of recursive structural types is a well-known and challenging problem [? ? ?] which is further compounded in Whiley by the presence of flow-sensitive reasoning.

4 Functional vs Imperative

4.1 Function Purity

Much research has been done on functional purity for object-oriented languages (e.g.[? ? ? ?]), because it makes many things more tractable, including: *automatic parellelisation*[? ?], *software verification*[? ? ? ?], *query systems*[? ?], *compiler optimisations*[? ? ?] and more.

4.2 Objects and References

4.3 Simulating Interfaces

4.4 Concurrency

5 Example: Calculator

Figure 3 provides a simple implementation of expressions, along with code for evaluating them. The types `Expr` and `Value` are algebraic data types, with the latter defining the set of allowed values. Type `Op` is an enumeration, whilst `BinOp` and `ListAccess` are records which form part of `Expr`. Parameter `env` is a map from variables to `Values`. Finally, **`null`** is used as an error condition to indicate a "stuck" state (i.e. the evaluation cannot proceed).

The code in Figure 3 makes extensive use of runtime type tests to distinguish different expression forms (e.g. "`e` **`is int`**"). These work in a similar fashion to Java's `instanceof` operator, with one important difference: they operate in a flow-sensitive fashion and automatically *retype* variables after the test. As an example, consider the type test "`e` **`is int`**" on Line 11. On the true branch, variable `e` is automatically retyped to have type **`int`**. Likewise, on the false branch, `e` is now known *not* to have type **`int`** (and any attempt to retest this yields a compile-time error).

Figure 3 also employs runtime type tests to identify and propagate errors. For example, having evaluated the left- and right-hand sides of a `BinOp`, we check on Line 21 that both are **`int`** values (i.e. not list values or **`null`**). After the check, Whiley's flow-sensitive type system automatically retypes both `lhs` and `rhs` to **`int`**. For `ListAccess` expressions, we check on Line 39 that `src` is a list value, and that `index` is an **`int`**. The latter is achieved with "`index` **`is int`**". As `src` is retyped within the condition itself, the subsequent use of `|src|` on Line 40 is type safe.

Implementing our expression language in a statically-typed language, such as Java, would require code that was more cumbersome, and more verbose than that of Figure 3. One reason for this is that, in languages like Java, variables must be *explicitly* retyped after `instanceof` tests. That is, we must insert casts to update the types of tested variables and, since variables can have only one type in Java, introduce temporary variables to hold these new types. For example, after a test "`e instanceof BinOp`" we must introduce a new variable, say `r`, with type `BinOp` and assign `e` to `r` using an appropriate cast. A Java implementation would also (most likely) break up the test on Line 39, since it would otherwise need two identical casts (one inside the condition for `|src|`, and one on the true branch for `src[index]`).

In an object-oriented language, such as Java, a direct conversion of Figure 3 might not be optimal. Instead, the *visitor pattern*[?] can be used to distinguish different expression forms. Using the visitor pattern reduces the amount of explicit retyping required. This is because the different expression forms are explicitly given as parameters to the visitor methods. However, using the visitor pattern is a heavyweight solution which is not suitable in all situations. In particular, it would not eliminate all forms of explicit retyping from Figure 3. In this case, explicit variable retyping will still be required to properly handle the different values returned from `eval()`. For example, to check `BinOp`s are evaluated on **`int`** operands (Line 21), and that `src` gives a list and `index` an **`int`** (Line 39).

Code Reuse. Whiley's flow type system can expose greater opportunities for code reuse:

```
1  {string} usedVariables(Expr e):
2      if e is Var:
3          return {e}
4      else if e is BinOp || e is ListAccess:
5          l = useVariables(e.lhs)
6          r = useVariables(e.rhs)
7          return l + r   // set union
8      else if e is [Expr]:
9          ...
10     else:
11         return {}
```

On Line 5, variable `e` has type `BinOp|ListAcccess`. The use of `e.lhs` at this point is type safe, since we can perform operations common to all types of a union and, in particular, unions of records expose common fields (similar to a *common initial sequence* for `union`s of `struct`s in C[?] §6.3.2.3).

```
define Var as string
define Op as { ADD, SUB, MUL, DIV }
define BinOp as { Op op, Expr lhs, Expr rhs }
define ListAccess as { Expr lhs, Expr rhs }

define Value as int | [Value] | null

define Expr as int | Var | BinOp | [Expr] | ListAccess

Value eval(Expr e, {Var→Value} env):
   if e is int:
       return e
   else if e is Var && e in env:
       // look up variable's value
       return env[e]
   else if e is BinOp:
       // evaluate left and right expressions
       lhs = eval(e.lhs,env)
       rhs = eval(e.rhs,env)
       // sanity check
       if !(lhs is int && rhs is int):
           return null // stuck
       // evaluate result
       switch e.op:
           case ADD:
               return lhs + rhs
           case SUB:
               return lhs - rhs
           case MUL:
               return lhs * rhs
           case DIV:
               if rhs != 0:
                   return lhs / rhs
   else if e is ListAccess:
       // evaluate src and index expressions
       src = eval(e.lhs,env)
       index = eval(e.rhs,env)
       // santity check
       if src is [Value] && index is int
          && index >= 0 && index < |src|:
           return src[index]
   else if e is [Expr]:
       lv = []
       // evaluate items in list constructor
       for i in e:
           v = eval(i,env)
           if v == null:
               return v
           else:
               lv = lv + [v]
       return lv
   // some kind of error occurred, so propagate upwards
   return null
```

Figure 3: Whiley code for a simple expression tree and evaluation function. This makes extensive use of type tests, both for distinguishing expressions and error handling. Flow-sensitive typing greatly simplifies the code, which would otherwise require numerous unnecessary casts.

In languages like Java, exploiting code reuse in this way requires careful planning, as common types must be explicitly related in the class hierarchy. In contrast, Whiley's flow-sensitive type system lets us exploit opportunities for code reuse in an ad-hoc fashion, as and when they occur.

5.1 Objective Objects

6 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.

6.1 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*.

6.2 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:

```
define nat as int where $ >= 0
define pos as int where $ > 0

nat f(pos x) ensures $ != x:
    return x-1
```

Here, the **define** statement 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 $\mathtt{T}_1$ and $\mathtt{T}_2$ have the same *underlying* type, then $\mathtt{T}_1$ is a subtype of $\mathtt{T}_2$ iff the constraint on $\mathtt{T}_1$ implies that of $\mathtt{T}_2$. Consider the following:

```
define anat as int where $ >= 0
define bnat as int where 2*$ >= $

bnat f(anat x):
    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 $ from both sides). The Whiley compiler is able to reason that these types are equivalent and statically verifies that this function is correct.

6.3 Quantification

6.4 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).

6.5 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

nat sum([nat] items):
    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:

```
nat sum([nat] items):
    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 §6.4 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`.

6.6 Explicit Assumptions

6.7 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.

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

7 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.

7.1 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 r)
// 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!

7.2 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.

7.3 Specifying Property 3 — Return Least Index

7.4 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 }:
    //
    i = 0
    while i < |items|:
       if items[i] == item:
           return i
       i = i + 1
    //
    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 4 illustrates what happens when we compile the above code with verification enabled.

```
1  int| null indexOf([int] items, int item)
2  // If return is an int r, then items[r] == item
3  ensures !($ is int) || items[$] == item,
4  // If return is null, then no element x in items where x == item
5  ensures !($ is null) || no { x in items | x == item },
6  // If return is an int i, then no index j where j $<$ i and items[j] == item
7  ensures !($ is int) || no { j in 0 .. $ | items[j] == item }:
8      //
9      i = 0
10     while i < |items|:
11         if items[i] == item:
12             return i
13         i = i + 1
14     //
15     return null
16
```

"index out of bounds (negative)"

Figure 4: 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.

7.5 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 §6.4) 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.

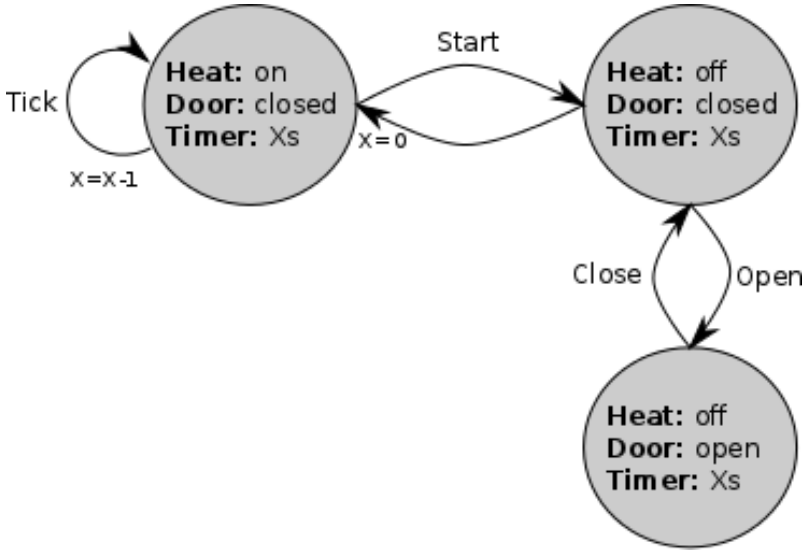

Figure 5: A state-machine diagram for the microwave oven

8 Example: Microwave Oven

A Foreign Function Interface

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