The Whiley Language Specification

David J. Pearce
School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
djp@ecs.vuw.ac.nz

January 2, 2014

Contents

1 Introduction **3**
- 1.1 Overview . . . 3
- 1.2 Goals . . . 3
- 1.3 History . . . 3

2 Lexical Structure **4**
- 2.1 Indentation . . . 4
- 2.2 Blocks . . . 4
- 2.3 Whitespace . . . 4
- 2.4 Identifiers . . . 4

3 Compilation Units **5**
- 3.1 Type Declarations . . . 5
- 3.2 Constant Declarations . . . 5
- 3.3 Function & Method Declarations . . . 5
- 3.4 Visibility Modifiers . . . 5
- 3.5 Packages . . . 5
- 3.6 Imports . . . 5

4 Types **6**
- 4.1 Overview . . . 6
- 4.2 Primitives . . . 6
  - 4.2.1 Any Type . . . 7
  - 4.2.2 Void Type . . . 7
  - 4.2.3 Null Type . . . 7
  - 4.2.4 Bool Type . . . 7
  - 4.2.5 Byte Type . . . 8
  - 4.2.6 Char Type . . . 8
  - 4.2.7 Int Type . . . 8
  - 4.2.8 Real Type . . . 9
- 4.3 Tuple Types . . . 9
- 4.4 Record Types . . . 9
- 4.5 Reference Types . . . 10
- 4.6 Nominal Types . . . 10
- 4.7 Collection Types . . . 10
  - 4.7.1 Set Type . . . 10
  - 4.7.2 Map Type . . . 11
  - 4.7.3 List Type . . . 11
- 4.8 Function Types . . . 11
- 4.9 Method Types . . . 11
- 4.10 Union Types . . . 12
- 4.11 Intersection Types . . . 12

4.12 Negation Types 12
4.13 Abstract Types 13
4.13.1 Recursive Types 13
4.13.2 Effective Tuples 13
4.13.3 Effective Records 13
4.13.4 Effective Collections 13
4.14 Subtyping Algorithms 13

5 Expressions **14**
5.1 Binary Expressions 14

6 Statements **16**
6.1 Assert Statement 16
6.2 Assignment Statement 16
6.3 Assume Statement 17
6.4 Return Statement 17
6.5 Throw Statement 18
6.6 Variable Declarations 18
6.7 If/Else Statements 18
6.8 While Statements 18
6.9 Do/While Statements 18
6.10 For Statements 18
6.11 Switch Statements 18
6.12 Try/Catch Statements 18

Glossary **19**

Chapter 1

Introduction

1.1 Overview

1.2 Goals

1.3 History

Chapter 2

Lexical Structure

2.1 Indentation

2.2 Blocks

2.3 Whitespace

2.4 Identifiers

Chapter 3

Compilation Units

3.1 Type Declarations

3.2 Constant Declarations

3.3 Function & Method Declarations

3.4 Visibility Modifiers

3.5 Packages

3.6 Imports

Chapter 4

Types

4.1 Overview

Discuss syntactic versus semantic types. Also, need to consider constrained types as well as type patterns.

```
Type  ::=
       |   TermType
       |   UnionType
       |   IntersectionType
```

```
TermType  ::=
           |   PrimitiveType
           |   TupleType
           |   RecordType
           |   ReferenceType
           |   NominalType
           |   CollectionType
           |   NegationType
           |   FunctionType
           |   MethodType
```

4.2 Primitives

```
PrimitiveType  ::=
                |   AnyType
                |   VoidType
                |   NullType
                |   BoolType
                |   ByteType
                |   CharType
                |   IntType
                |   RealType
```

4.2.1 Any Type

AnyType	::=	`any`

Description. The type `any` represents the type whose variables may hold any possible value.

Examples.

Semantics.

Notes. The `any` type is top in the type lattice. That is, it is the supertype of all other types.

4.2.2 Void Type

VoidType	::=	`void`

Description. The `void` type represents the type whose variables cannot exist! That is, they cannot hold any possible value. Void is used to represent the return type of a function which does not return anything. However, it is also used to represent the element type of an empty list of set.

Examples.

Semantics.

Notes. The void type is a subtype of everything; that is, it is bottom in the type lattice.

4.2.3 Null Type

NullType	::=	`null`

Description. The null type is a special type which should be used to show the absence of something. It is distinct from void, since variables can hold the special `null`¿ value (where as there is no special ”`void`” value).

Examples.

Semantics.

Notes. With all of the problems surrounding `null` and `NullPointerException`s in languages like Java and C, it may seem that this type should be avoided. However, it remains a very useful abstraction to have around and, in Whiley, it is treated in a completely safe manner (unlike e.g. Java).

4.2.4 Bool Type

```
BoolType ::= bool
```

Description. Represents the set of boolean values (i.e. `true` and `false`).

Examples.

Semantics.

Notes.

4.2.5 Byte Type

```
ByteType ::= byte
```

Description. Represents a sequence of 8 bits.

Examples.

Semantics.

Notes. Unlike for many languages, there is no representation associated with a byte. For example, to extract an integer value from a byte, it must be explicitly decoded according to some representation (e.g. two's compliment) using an auxillary function (e.g. `Byte.toInt()`).

4.2.6 Char Type

```
CharType ::= char
```

Description. Represents a unicode character.

Examples.

Semantics.

Notes.

4.2.7 Int Type

```
IntType ::= int
```

Description. Represents the set of (unbound) integer values.

Examples.

Semantics.

Notes. Since integer types in Whiley are unbounded, there is no equivalent to Java's `MIN_VALUE` and `MAX_VALUE` for **`int`** types.

4.2.8 Real Type

```
RealType ::= [real]
```

Description. Represents the set of (unbound) rational numbers.

Examples.

Semantics.

Notes.

4.3 Tuple Types

```
TupleType ::= [(] Type ( [,] Type )+ [)]
```

Description. A tuple type describes a compound type made up of two or more subcomponents. It is similar to a record, except that fields are effectively anonymous.

Examples.

Semantics.

Notes.

4.4 Record Types

```
RecordType ::= [{] Type Ident ( [,] Type Ident )* [ [,] [...] ] [}]
```

Description. A record is made up of a number of fields, each of which has a unique name. Each field has a corresponding type. One can think of a record as a special kind of "fixed" map (i.e. where we know exactly which entries we have).

Examples.

Semantics.

Notes. Syntax for functions? Open versus closed records?

4.5 Reference Types

```
ReferenceType ::= & Type
```

Description. Represents a reference to an object in Whiley.

Examples.

Semantics.

Notes.

4.6 Nominal Types

```
NominalType ::= Ident
```

Description. The existential type represents the an unknown type, defined at a given position.

Examples.

Semantics.

Notes.

4.7 Collection Types

4.7.1 Set Type

```
SetType ::= { Type }
```

Description. A set type describes set values whose elements are subtypes of the element type. For example, `{1,2,3}` is an instance of set type `{int}`; however, `{1.345}` is not.

Examples.

Semantics.

Notes.

4.7.2 Map Type

```
MapType ::= { Type => Type }
```

Description. A map represents a one-many mapping from variables of one type to variables of another type. For example, the map type `{int=>real}` represents a map from integers to real values. A valid instance of this type might be `{1=>1.2,2=>3.0}`.

Examples.

Semantics.

Notes.

4.7.3 List Type

```
ListType ::= [ Type ]
```

Description. A list type describes list values whose elements are subtypes of the element type. For example, `[1,2,3]` is an instance of list type `[int]`; however, `[1.345]` is not.

Examples.

Semantics.

Notes.

4.8 Function Types

```
FunctionType ::= function ( [ Type ( , Type )* ] ) => Type
```

Description.

Examples.

Semantics.

Notes.

4.9 Method Types

```
MethodType ::= method ( [ Type ( , Type )* ] ) => Type
```

Description.

Examples.

Semantics.

Notes.

4.10 Union Types

```
UnionType ::= IntersectionType ( | IntersectionType )+
```

Description. A union type represents a type whose variables may hold values from any of its "bounds". For example, the union type **`null|int`** indicates a variable can either hold an integer value, or **`null`**.

Examples.

Semantics.

Notes. There must be at least two bounds for a union type to make sense.

4.11 Intersection Types

```
IntersectionType ::= TermType ( & TermType )+
```

Description.

Examples.

Semantics.

Notes.

4.12 Negation Types

```
NegationType ::= ! Type
```

Description. A negation type represents a type which accepts values *not* in a given type.

Examples.

Semantics.

Notes.

4.13 Abstract Types

4.13.1 Recursive Types

4.13.2 Effective Tuples

4.13.3 Effective Records

4.13.4 Effective Collections

4.14 Subtyping Algorithms

Discussion of soundness and completeness.

Expr	::=	$Cond\ [\ (\ \boxed{\texttt{\&\&}} \mid \boxed{\vert\vert}\ )\ Expr\ ]$	*// Expressions*
Cond	::=	$Append\ [\ Cop\ Expr\ ]$	*// Condition Expressions*
Append	::=	$Range\ [\ \boxed{\texttt{++}}\ Expr\ ]$	*// Append Expressions*
Range	::=	$AddSub\ [\ \boxed{\texttt{..}}\ Expr\ ]$	*// Range Expressions*
AddSub	::=	$MulDiv\ [\ (\ \boxed{+} \mid \boxed{-}\ )\ Expr\ ]$	*// Additive Expressions*
MulDiv	::=	$Index\ [\ (\ \boxed{\star} \mid \boxed{/} \mid \boxed{\%}\ )\ Expr\ ]$	*// Multiplicative Expressions*
Index	::=	???	*// Index Expressions*

Figure 5.1: Syntax for Binary Expressions

Chapter 5

Expressions

Expression blah blah.

5.1 Binary Expressions

Term	::=	*// Terms*	
	|	$Constant$	*// Constant expressions*
	|	$Identifier$	*// Identifier expressions*
	|	$Expr_1 \; (\boxed{\texttt{,}} \; Expr_i \;)^+$	*// Tuple expressions*
	|	$\boxed{\texttt{(}} \; Expr \; \boxed{\texttt{)}}$	*// Bracketed expressions*
	|	$\boxed{\texttt{\|}} \; Expr \; \boxed{\texttt{\|}}$	*// Size expressions*
	|	$Identifier \; \boxed{\texttt{(}} \; [\; Expr_1 \; (\boxed{\texttt{,}} \; Expr_i \;)^+ \;] \; \boxed{\texttt{)}}$	*// Invocation expressions*
	|	$(\boxed{\texttt{-}} \mid \boxed{\texttt{!}} \mid \boxed{\texttt{\textasciitilde}} \mid \boxed{\texttt{\&}} \mid \boxed{\texttt{*}} \;) \; Expr$	*// Unary expressions*
	|	$\boxed{\texttt{new}} \; Expr$	*// Allocation expressions*
	|	$\boxed{\{} \; [\; Expr_1 \; (\boxed{\texttt{,}} \; Expr_i \;)^* \;] \; \boxed{\}}$	*// Set expressions*
	|	$\boxed{\{} \; [\; Expr_1 \; \boxed{\texttt{=>}} \; Expr'_1 \; (\boxed{\texttt{,}} \; Expr_i \; \boxed{\texttt{=>}} \; Expr'_i \;)^* \;] \; \boxed{\}}$	*// Map expressions*
	|	$\boxed{\texttt{[}} \; [\; Expr_1 \; (\boxed{\texttt{,}} \; Expr_i \;)^* \;] \; \boxed{\texttt{]}}$	*// List expressions*
	|	$\boxed{\{} \; [\; n_1 \boxed{\texttt{:}} \; Expr_1 \; (\boxed{\texttt{,}} \; n_i \; \boxed{\texttt{:}} \; Expr_i \;)^* \;] \; \boxed{\}}$	*// Record expressions*

Figure 5.2: Syntax for Term Expressions

Constant	::=	*// Constants*	
	|	$(\boxed{\texttt{0}} \mid \boxed{\texttt{1}} \;)^+ \; \boxed{\texttt{b}}$	*// Boolean constants*
	|	$(\boxed{\texttt{0-9}} \;)^+$	*// Integer constants*
	|	$(\boxed{\texttt{0-9}} \;)^+ \; \boxed{\texttt{.}} \; (\boxed{\texttt{0-9}} \;)^+$	*// Decimal constants*
	|	$\boxed{\texttt{null}}$	*// Null constant*

Figure 5.3: Syntax for Constant Expressions

Identifier	::=	$(\boxed{\texttt{\_}} \mid \boxed{\texttt{a-z}} \mid \boxed{\texttt{A-Z}} \;) \; (\boxed{\texttt{\_}} \mid \boxed{\texttt{a-z}} \mid \boxed{\texttt{A-Z}} \mid \boxed{\texttt{0-9}} \;)^*$	*// Identifiers*

Figure 5.4: Syntax for Identifiers

Chapter 6

Statements

6.1 Assert Statement

```
AssertStmt  ::=  assert Expr
```

Description. Represents an *assert statement* of the form "**assert** e", where e is a boolean expression.

Examples. The following illustrates:

```
function abs(int x) => int:
    if x < 0:
        x = -x
    assert x >= 0
    return x
```

Notes. Assertions are either *statically checked* by the verifier, or turned into *runtime checks*.

6.2 Assignment Statement

```
AssignStmt  ::=  LVal = Expr
```

Description. Represents an *assignment statement* of the form `lhs = rhs`. Here, the `rhs` is any expression, whilst the `lhs` must be an `LVal` — that is, an expression permitted on the left-side of an assignment.

Examples. The following illustrates different possible assignment statements:

```
x = y          // variable assignment
x.f = y        // field assignment
x[i] = y       // list assignment
x[i].f = y     // compound assignment
```

The last assignment here illustrates that the left-hand side of an assignment can be arbitrarily complex, involving nested assignments into lists and records.

Semantics.

Notes.

6.3 Assume Statement

```
AssumeStmt  ::=  assume Expr
```

Description. Represents an *assume statement* of the form "`assume e`", where `e` is a boolean expression.

Examples. The following illustrates a simple function which uses an `assume` statement to meet its postcondition:

```
function abs(int x) => int:
    assume x >= 0
    return x
```

Notes. Assumptions are *assumed* by the verifier and, since this may be unsound, are always turned into *runtime checks*.

6.4 Return Statement

```
ReturnStmt  ::=  return [Expr]
```

Description. Represents a *return statement* with an optional expression is referred to as the *return value*.

Examples. The following illustrates a simple function which returns the increment of its parameter `x`:

```
function f(int x) => int:
    return x + 1
```

Here, we see a simple **`return`** statement which returns an **`int`** value.

Notes. The returned expression (if there is one) must begin on the same line as the return statement itself.

6.5 Throw Statement

```
ThrowStmt   ::=   throw Expr
```

Description.

Examples.

Notes.

6.6 Variable Declarations

```
VarDecl   ::=   Type Ident [ = Expr ]
```

Description. Represents a *variable declaration* which has an optional expression assignment referred to as an *variable initialiser*. If an initialiser is given, then this will be evaluated and assigned to the variable when the declaration is executed.

Examples. Some example variable declarations are:

```
int x
int y = 1
int z = x + y
```

Notes.

6.7 If/Else Statements

6.8 While Statements

6.9 Do/While Statements

6.10 For Statements

6.11 Switch Statements

6.12 Try/Catch Statements

Glossary

expression A combination of constants, variables and operators that, when evaluated, produce a single value. Expressions in certain circumstances may have side effects.. 14, 19

variable initialiser An optional expression used to initialise variable(s) declared as part of a variable declaration.. 18