Report on the Moby Programming Language
(Version 0.9.18)

The Moby Project
moby.cs.uchicago.edu

Last revised: June 7, 2003

Contents

I MOBY Overview **1**

1 A tutorial introduction **2**

1.1 Getting started . . . 2

1.2 Values and expressions . . . 2

2 Functional programming in MOBY **5**

2.1 Basic values and expressions . . . 5

2.1.1 Tuples . . . 5

2.2 Modules . . . 5

2.3 Functions . . . 5

2.4 Exceptions . . . 5

2.5 Data constructors and pattern matching . . . 5

2.5.1 Pattern matching . . . 6

2.5.2 Constant definitions . . . 6

2.5.3 Expression forms . . . 6

2.6 Concrete types . . . 6

2.6.1 Datatypes . . . 6

2.6.2 Enumerations . . . 6

2.6.3 Tagtypes . . . 6

3 Object-oriented programming in MOBY **7**

3.1 Objects . . . 7

3.2 Classes . . . 8

3.3 Object construction and invariants . . . 10

3.4 `typeof` . . . 10

3.5 Class interfaces 11
3.6 Class types 12
3.7 Using tagtypes to implement checked down-casts 13
3.8 Programming with classes and modules 14

II MOBY Reference **17**

4 Introduction **18**
4.1 Program structure 18
4.2 Notation 18

5 Lexical structure **19**
5.1 White space 19
5.2 Comments 19
5.3 Identifiers 19
5.3.1 Reserved words 20
5.3.2 Underscore identifiers 20
5.4 Operators 20
5.5 Separators 20
5.6 Literals 20
5.6.1 Boolean literals 20
5.6.2 Integer literals 20
5.6.3 Floating-point literals 21
5.6.4 Character literals 21
5.6.5 String literals 21

6 Evaluation **22**
6.1 Tuples 22
6.2 Imperative features 22
6.2.1 Assignment and L-values 22
6.2.2 Mutable fields 23
6.2.3 Mutable arrays 23
6.2.4 Pointers 23
6.2.5 References 23

6.3 Exceptions . . . 23
6.3.1 Raising and handling exceptions . . . 24
6.3.2 Exception values . . . 24
6.3.3 The standard exceptions . . . 24
6.4 Threads . . . 24
6.5 Synchronization . . . 24

7 Signatures and modules **25**
7.1 Signatures . . . 25
7.2 Specifications . . . 25
7.2.1 Module specifications . . . 25
7.2.2 Type specifications . . . 26
7.2.3 Class specifications . . . 26
7.2.4 Constant specifications . . . 26
7.2.5 Value specifications . . . 27
7.3 Modules . . . 27
7.3.1 A module's implicit signature . . . 27
7.4 Parameterized modules . . . 28
7.5 Declarations . . . 29
7.5.1 Constant declarations . . . 29
7.5.2 Function declarations . . . 29
7.5.3 Value declarations . . . 30
7.5.4 Recursive declarations and scope . . . 30
7.6 Signature Matching . . . 31

8 Type declarations **32**
8.1 Type name declarations . . . 32
8.2 Object type declarations . . . 32
8.3 Datatype declarations . . . 33
8.4 Enumeration type declarations . . . 33
8.5 Tagtype declarations . . . 34

9 Classes **35**
9.1 Class declarations . . . 35

9.2 Member declarations 35
9.2.1 Field declarations 36
9.2.2 Method declarations 36
9.2.3 Maker declarations 36
9.3 Initially clauses 37
9.4 Inheritance 37
9.4.1 Method overriding 37
9.4.2 Object initialization 37

10 Types, type equality, and subtyping **38**
10.1 Type schemes 38
10.2 Types 38
10.2.1 Type tuples 38
10.2.2 Function types 39
10.2.3 Atomic types 39
10.2.4 Type constructors 39
10.2.5 Record types 39
10.2.6 Extended types 40
10.2.7 Pervasive types 40
10.3 Type equality 41
10.4 Subtyping 41

11 Expressions **42**
11.1 Blocks 42
11.2 If-then-else expressions 42
11.3 TryExpressions 43
11.3.1 Try-except expressions 43
11.3.2 Try-finally expressions 43
11.4 Raise expressions 44
11.5 Spawn expressions 45
11.6 Sync expressions 45
11.7 Binary expression 45
11.7.1 Assignment expressions 45
11.7.2 Conditional expressions 46

11.7.3 Equality expressions . . . 46
11.7.4 Relational expressions . . . 47
11.7.5 List expressions . . . 47
11.7.6 Bitwise expressions . . . 48
11.7.7 Arithmetic expressions . . . 48
11.7.8 Function composition expressions . . . 49
11.8 Prefix expressions . . . 49
11.9 Application expressions . . . 49
11.10 Postfix expressions . . . 50
11.11 Atomic expressions . . . 50
11.11.1 Function expressions . . . 50
11.11.2 Case expressions . . . 50
11.11.3 Expression tuple . . . 50
11.11.4 Constraint expression . . . 50
11.11.5 Match test expression . . . 50
11.11.6 Record expression . . . 51
11.11.7 Choice event . . . 51
11.11.8 New expression . . . 51
11.11.9 Self and super expressions . . . 51
11.11.10 Identifier expressions . . . 51

12 Match cases and patterns **52**
12.1 Match cases . . . 52
12.2 Patterns . . . 53
12.2.1 Constrained patterns . . . 53
12.2.2 Is patterns . . . 53
12.2.3 Or patterns . . . 53
12.2.4 Cons patterns . . . 54
12.2.5 Application patterns . . . 54
12.2.6 Atomic patterns . . . 54
12.2.7 Record patterns . . . 55
12.3 Pattern matching semantics . . . 55
12.4 Pattern expressions . . . 56

III Appendices **59**

A Collected MOBY syntax **60**

A.1 Identifiers 60

A.1.1 Identifier classes 60

A.1.2 Reserved symbols and keywords 60

A.2 Collected syntax 60

B The MOBY Basis **72**

B.1 Pervasives 72

B.1.1 Pervasive types 72

B.1.2 Overloaded operators 72

B.1.3 Pervasive functions 72

B.2 Standard modules 72

Part I

MOBY Overview

Chapter 1

A tutorial introduction

Let us begin with a quick introduction to the syntax and major features of the MOBY programming language.

1.1 Getting started

Following tradition, our first program is the infamous "*hello world*" example. In MOBY, one implementation is as follows:

```
module Main {
  val () = ConsoleIO.print "hello world\n"
}
```

The code in this figure constitutes a *compilation unit*. All MOBY programs are constructed from a collection of compilation units. This program consists of the module `Main`, which contains an empty binding with a right-hand-side consisting of the application of the `print` function from the `ConsoleIO` module to the string literal `"hello world\n"`. The `ConsoleIO` module is part of the MOBY basis and provides support for input and output to the console device associated with the program.

[[describe how to compile and run this program?]]

1.2 Values and expressions

MOBY is primarily a *value oriented* language — most computation is expressed in terms of computing new values from existing ones. Expressions in MOBY evaluate to a tuple of zero or more results.

Our next example is a program that computes and prints the first ten numbers in the *Fibonacci sequence* (a favorite of functional programmers):

```
module Fib {
  fun fib : Int -> Int {
    0 => 1,
    1 => 1,
    n => fib(n-1) + fib(n-2)
  }

  const N : Int = 10

  fun loop : Int -> () {
    N => (),
    i => {
      ConsoleIO.print(Int.toString(fib(i)) + "\n");
      loop (i+1)
    }
  }

  val () = loop 0

}
```

This example illustrates a number of MOBY features. First, the definition of the `fib` function uses the "*definition by cases*" syntax, where the body of the function is given as a *match*. Following the definition of the `fib` function is a *constant declaration*, which binds the value `10` to the symbolic name `N`. Symbolic constants (including datatype constructors) in MOBY always start with an upper-case character, while variables start with a lower-case character. This convention allows the compiler to distinguish between constants and variables in patterns. NB. the `Int` module is *pervasive*.

```
module Fact {
// compute the factorial function
  fun fact (n : Int) -> Int {
    if (n == 0) then 1 else n*fact(n-1)
  }

// get a positive integer value from the console
  fun getInt () -> Int {
    ConsoleIO.print ("please enter a positive integer: ");
    val input = Console.inputLn ();
    case Int.fromString input of {
      (SOME x) when (x >= 0) => x,
      _ => {
        Console.print ("That isn't a positive integer!\n");
        getInt ()
      }
    }
  }

// the factorial server
  fun server () -> () {
    val n = getInt ();
    val factN = fact (n);
    ConsoleIO.print("The factorial of " + Int.toString n +
      " is " + Int.toString factN + "\n");
    server ()  // do it again!
  }

}
```

Chapter 2

Functional programming in MOBY

MOBY is a value-oriented language in the tradition of the ML-family of languages, and it supports common features of these languages, such as polymorphism, datatypes, pattern matching, higher-order functions, and exceptions. In this chapter, we survey these features in MOBY.

2.1 Basic values and expressions

2.1.1 Tuples

MOBY supports multiple function and constructor arguments and multiple return values using *tuples*. Unlike many functional languages, tuples are not first-class. It is not possible to bind a variable to a tuple value or to instantiate a type variable with a tuple type.

[[The MOBY basis will provide type constructors for pairs and triples, and possibly a more general mechanism.]]

2.2 Modules

2.3 Functions

2.4 Exceptions

2.5 Data constructors and pattern matching

As in most functional languages, *data constructors* are used both to construct values and to deconstruct them.

2.5.1 Pattern matching

2.5.2 Constant definitions

2.5.3 Expression forms

Because of the importance of pattern matching in deconstructing data structures, MOBY provides several expression forms as shorthands.

One can test the structure of a value in a couple of ways. The "`?`" modifier can be applied to a data constructor or constant to create a predicate. For example, "`? None`" defines a predicate function that returns **`True`** when applied to `None`. It is equivalent to the function

```
fn : [t] Option(t) -> Bool { None => True, _ => False }
```

It is also possible to test a value against a more complicated pattern. For example,

```
(x is Some(_::_))
```

evaluates to **`True`** when `x` matches the pattern "`Some(_::_)`".

Lastly, MOBY provides syntactic sugar for deconstructing values. The "`#`" modifier can be applied to a data constructor to create a projection function. For example, "`# Some`" defines a projection function that returns the argument of the `Some` constructor. It is equivalent to the function

```
fn : [t] Option(t) -> t { Some(x) => x, _ => raise Match }
```

2.6 Concrete types

Most constants and data constructors are defined as part of *concrete type definitions*. MOBY provides three kinds of concrete type definition: *datatypes*, *enumerations*, and *tagtypes*.

2.6.1 Datatypes

2.6.2 Enumerations

2.6.3 Tagtypes

Chapter 3

Object-oriented programming in MOBY

MOBY supports a rich class-based style of object-oriented programming.

3.1 Objects

An *object* in MOBY is a collection of fields (both mutable and immutable) and methods. The fields and methods of an object are called its *members*. There are three operations on objects: select a field, update a field, and invoke a method. An *object type* specifies which members of an object may be used by clients to operate on it. Because of subtyping and hiding of members, an object can have different types in different contexts.

Object types must be declared, much like datatypes, but subtyping is *structural*. For example, a type for one-dimensional point objects may be written as follows:

```
objtype Point {
   method getX : () -> Float
   method move : Float -> Point
 }
```

Such a `Point` object has no visible fields and two visible methods: `getX`, which takes no arguments and returns a float, and `move`, which takes a float and returns a point. The intended semantics is that `getX` returns the point's current position, while `move` shifts the location of the point by some amount and returns the point. Because `move` returns the point, we can chain together method invocations. For example, assuming that `pt` is a point, the expression:

```
pt.move(1.0).getX()
```

moves `pt` by `1.0` and then returns its new position.

Continuing the standard example, we might define color point objects as follows:

```
objtype CPoint {
   extends Point
   method getC  : () -> Color
   method shade : Color -> CPoint
}
```

This declaration defines the `CPoint` type to be an object type that includes all of the members of `Point` (with their given types), plus two new methods: `getC` and `shade`. It is shorthand for the following:

```
objtype CPoint {
   method getX : () -> Float
   method move : Float -> Point
   method getC  : () -> Color
   method shade : Color -> CPoint
}
```

Note that the return type of the `move` method is still `Point` (an not `CPoint`).[1]

3.2 Classes

In MOBY, objects are implemented by classes. For example, the following class is an implementation of the `Point` object type.

```
class PointClass {
   field x : var Float
   public method getX () -> Float { self.x }
   public method move (dx : Float) -> Point {
       self.x := self.getX() + dx;
       self
   }
   public maker point (x0 : Float) {
       field x = x0
     }
 }
```

The body of this class lists four *members*. The first is a field named `x`; we use the **`var`** keyword to denote that `x` is a mutable field. The second and third members are two methods named `getX` and `move`. The final member is a maker function named `point`.

Classes have two different kinds of clients: *object clients*, who use the objects instantiated from the class, and *class clients*, who extend the class through subclassing. Typically, the interface available to class clients is richer than that provided to object clients. Software engineering principles suggest that access to members should be restricted by default. To support this usage, MOBY provides the keyword **`public`**. Public members are available to both object and class clients, while

[1]The return type of `move` does not change because MOBY has recursive object types, not the more elaborate *mytype* mechanism [BSv95, FM95, AC96]. Because our objects are stateful, we believe the extra power of the mytype mechanism is not worth the additional complexity.

unannotated members are available only to class clients. For example, because the `x` field in class `PointClass` is not marked **`public`**, object clients cannot access it directly. The full type of the objects implemented by this class (*i.e.*, the type available to deriving classes) is equal to the type:

```
objtype XPoint {
  extends Point
  field x : var Float
}
```

which exposes the mutable `x` field.

The last part of this class definition is the *maker* `point`. Makers are special functions used to create objects.[2] Unlike many class-based languages, we do not use the class name for makers, but instead allow the programmer to specify maker names. In general, a class may contain any number of makers. The body of a maker is responsible for initializing all of the fields of the object. In this case, there is only the `x` field. The MOBY typing rules require that each maker initialize all the fields defined in its class. An object client of a class creates an object by using the **`new`** operator to invoke one of the class's makers with an appropriate argument. For example:

```
new point(0.0)
```

returns a new `Point` object located at the origin. Note that the maker `point` is available without qualification; it is visible because our minimal class mechanism does not define any form of namespace structure — that rôle is left to modules.

In addition to implementing objects, classes provide a vehicle for code reuse via inheritance. For example, we can implement color points by extending `PointClass`:

```
class CPointClass {
  inherits PointClass
  field c : var Color
  public override method move (dx : Float) -> Point {
    super.move(2*dx)
  }
  public method getC () -> Color { self.c }
  public method shade (dc : Color) -> CPoint {
    self.c := self.getC().blend(dc);
    self
  }
  public maker cpoint (x0 : Float, c0 : Color) {
    super point(x0);
    field c = c0
  }
}
```

In this case, we call `PointClass` the *superclass*, and `CPointClass` the *subclass* (or *derived* class). There are three things to note about the `CPointClass` code. First, the **`inherits`** clause

[2] We use the term *maker* (borrowed from Theta [Pro95]), instead of the more standard *constructor*, to avoid confusion with data and type constructors.

specifies that the fields and methods of `PointClass` are inherited by `CPointClass`. Second, we *override* the `move` method to make color points speedy. The use of the reserved identifier **`super`** in the body of the `move` method statically binds the call of `move` to the `PointClass` implementation. Third, the `cpoint` maker invokes the `point` maker of its parent class before initializing the `c` field.

3.3 Object construction and invariants

One important feature of class-based languages is support for the establishment and maintenance of object and class-level invariants.[3] These guarantees are especially important to base-class implementors who want to restrict how their classes may be extended. Providing such support motivates several features of the MOBY class mechanism design.

Object-level invariants are established when the object is initialized by its maker. We require that all fields defined in a class be initialized by each maker in the class and that subclasses always invoke a superclass maker. These requirements guarantee that fields are always initialized before an object is used. The author of a class can ensure that its invariants are maintained by hiding the mutable state from subclasses (*i.e.*, making the state private) and by declaring methods to be **`final`**.

Unlike many class-based languages, we do not allow access to the object's methods (via **`self`**) inside a maker. This restriction avoids the complication of a superclass maker invoking a subclass method before the subclass's fields have been initialized.[4] The main disadvantage of not being able to reference **`self`** inside makers is that class-level invariants cannot be maintained when new objects are created, since they require access to the new object. MOBY allows classes to include an **`initially`** clause, which specifies an expression to be evaluated each time an object is created. This expression may refer to **`self`**. The **`initially`** clauses are executed in the same order as makers — superclass and then subclass. Thus, the act of creating a new object involves first computing the initial values of its fields by invoking maker functions, then creating the object and passing it to the **`initially`** clauses, and finally returning the new object to the context that invoked **`new`**.

3.4 `typeof`

As a syntactic convenience, MOBY provides the **`typeof`** operator. When applied to a class name, **`typeof`** provides a name for the (public) object type associated with the class in the given scope. For example, in the scope of the `CPointClass` above, the type **`typeof`**(CPointClass) is equivalent to the type `CPoint`.

[3] Object-level invariants are properties of the state of a given object, whereas class-level invariants are properties of a collection of objects instantiated from a class.

[4] C++ addresses this problem by changing the semantics of method dispatch inside constructors, while JAVA relies on the property that all fields are initialized to some type-specific value prior to executing the constructor.

3.5 Class interfaces

An *interface* is the type (or signature) of a class; it is used when specifying a class in a module's signature.[5] For example, the following module encapsulates the implementation of the `PointClass`:

```
module Pt : {
  class PointClass : {
    implements Point
    public maker point of Float
  }
} { ... implementation of PointClass ... }
```

In this example, the `Pt` module is constrained by a signature that specifies an interface for `PointClass`. This interface specifies that `PointClass` provides an implementation of the `Point` type (and hence must provide `getX` and `move` methods with the appropriate types) and a maker named `point` that takes a float argument. Matching the `PointClass` with this interface has the effect of hiding the `x` field (*i.e.*, inside the `Pt` module, `x` is visible to deriving classes, but not outside). Once a field or method is hidden in this way, subclasses are free to define new members with the same name. Note, however, that the original member is still accessible from the superclass's methods (*e.g.*, `getX`), even if its name is reused.

The **`implements`** and **`maker`** clauses of a class interface together capture the class's object view. To avoid redundancy, MOBY has programmers express class views as incremental modifications to object views. For example, we might encapsulate the `CPointClass` as follows:

```
module CPt : {
  class CPointClass : {
    implements CPoint
    public maker cpoint of (Float, Color)
    field c : var Color
    public final method getC : () -> Color
  }
} { ... implementation of CPointClass ...}
```

The **`field`** and **`method`** clauses of the `CPointClass` interface specify a refinement (or delta) of the object view. Specifically, we have added a mutable field named `c` and noted that the `getC` method is *final* (*i.e.*, cannot be overridden by subclasses). In general, class interfaces may specify additional members and makers, final annotations on methods, and refinements to the types of methods, immutable fields, and makers.

As we have described, classes have two views: object and class, and modules define two scopes: internal and external. Combining these views gives us four distinct visibility modes, which roughly correspond to those in C++ and JAVA as follows:

	Object view	Class view
External	public	protected
Internal	package	private

[5] MOBY's notion of an interface should not be confused with that of JAVA. Interfaces in JAVA play a role that is more closely related to MOBY's object types.

Nested modules permit even finer control over visibility.

3.6 Class types

Occasionally, it is useful to tie object types to specific classes. Although it is possible to use abstract representation types and representation methods to make this connection classes, a more natural approach is to have the class names themselves play the rôle of types when the connection is needed.

To support this usage, MOBY includes *class types*. Syntactically, if `C` is a class name, then `#C` is the corresponding class type. This type can be used in any context that is in the scope of `C`. The methods and fields available from an object with a given class type depend on context: within a method body, the class type of the host class permits access to all members in the class view, while outside the class, it permits access only to public members.

The following code fragment illustrates the use of class types: the `BagM` module uses class types to grant the `union` function access to the `items` field of the `Bag` class, essentially making `union` a *friend* of the class. The signature for the `BagM` module then hides this field, so code outside the `BagM` module cannot access the `items` field directly. This use of class types ensures that the `union` function can be passed only objects instantiated from the `Bag` class or one of its descendants, guaranteeing that the `item` field will be available.

```
module BagM : {
  class Bag : {
    public method add : Int -> ()
    public maker mkBag of ()
  }
  val union : (#Bag, #Bag) -> ()
} {
  class Bag {
    public field items : var List(Int)
    public method add (x : Int) -> () {
      self.items := x :: self.items
    }
    public maker mkBag () { field items = Nil }
  }
  fun union (s1 : #Bag, s2 : #Bag) -> () {
    List.app s1.add s2.items
  }
}
```

When typing the methods of a class `C`, we give **`self`** the type `#C`. Likewise, if `B` is `C`'s superclass, then **`super`** has the type `#B`. Objects instantiated from class `C` are given type `#C`.

To connect class types and object types, we have that for any class `C`, `#C` <: **`typeof`**(`C`). Subtyping between class types follows the class hierarchy, with one side-condition. Namely, if class `C` inherits from class `B` and **`typeof`**(`C`) <: **`typeof`**(`B`), then we have that `#C` <: `#B`.

Finally, to integrate class types and class interfaces, we extend class interfaces to allow an optional **`inherits`** clause. If in a given context a class `C` has an interface that includes an "**inherits** B" clause, then we view **#**C as a subtype of **#**B. In this case, the type checker ensures **typeof**(C) <: **typeof**(B) in determining that the class interface for C is well-formed. Omitting the **inherits** clause from C's interface causes the relationship between B and C to be hidden.

For example, in the following code fragment, the signature for the CBagM module asserts that the CBag class **inherits** from the BagM.Bag class. Since **typeof**(CBag) <: **typeof**(BagM.Bag), this class interface is well-formed and we have the relationship that **#**CBag <: **#**BagM.Bag, which permits the union function to be applied to objects instantiated from the CBag class.

```
module CBagM : {
  class CBag : {
    inherits BagM.Bag
    public method size : () -> Int
    public maker mkCBag of ()
  }
} {
  class CBag {
    inherits BagM.Bag
    public field nItems : var Int
    public override method add (x : Int) -> () {
      self.nItems := self.nItems+1;
      super.add(x)
    }
    public method size () -> Int { self.nItems }
    public maker mkCBag () { super mkBag(); field nItems = 0 }
  }
}
```

At first glance, the type **typeof**(C) may seem similar to **#**C, since both types provide access to the same set of members in a given context. The key difference is that objects derived from classes completely unrelated to C may have the type **typeof**(C), whereas an object of type **#**C must have been generated from class C or one of its descendants. In other words, **typeof**(C) is an *interface* type and **#**C is an *implementation* type.

3.7 Using tagtypes to implement checked down-casts

Using MOBY's tagtype mechanism, programmers can provide a type-safe downcast mechanism. For example, suppose we wished to allow users of the Point object to downcast instantiated objects to have type CPoint, if the object had the more refined type. We first declare a tagtype that will serve to indicate from which class a given object was instantiated:

```
tagtype PointKind of Point
```

We then add a method to the Point object to reveal its associated tag:

```
objtype Point {
  ...
  method kindOf : () -> PointKind
}
```

The implementation of the `kindOf` method wraps **`self`** with the `PointKind` constructor:

```
class PointClass {
    ...
  public method kindOf () -> PointKind { PointKind self }
}
```

When we define the `ColorPoint` class, we add a new constructor to the `PointKind` tagtype:

```
tagtype CPointKind of CPoint extends PointKind
```

and override the `kindOf` method to wrap **`self`** with the new constructor:

```
class CPointClass {
  ...
  public override method kindOf () -> PointKind { CPointKind self }
}
```

With this infrastructure in place, we can write a function that provides checked downcasting:

```
fun asCPt (pt : Point) -> Option(CPoint) {
  case pt.kindOf() of
  { CPointKind cpt => Some cpt
  , _ => None
  }
}
```

3.8 Programming with classes and modules

The last example of this section is an idealized graphics application that illustrates the interactions between classes and modules. We use objects to represent the graphical elements of a picture, which we call *glyphs*. We start by defining a module signature that specifies the `Glyph` type and a base class for implementing glyphs.

```
signature BASE_GLYPH {
  class BaseGlyph : {
    public final method draw : Point -> ()
    abstract method drawGlyph : Drawable -> ()
    maker mk of ()
  }
}
```

where the type `Drawable` is the type of an object that represents a drawing surface. This signature specifies that the `drawGlyph` method is *abstract*; thus we call `BaseGlyph` an *abstract class*.

Because it is an abstract class, its makers cannot be public and objects cannot be created from the class. Also, the class interface for `BaseGlyph` marks the `draw` method as **`final`**, which means that derived classes cannot override or hide it. Although no implementation is given here, the intuition is that method `draw` sets up any conditions necessary for drawing, perhaps translates the relevant coordinate system to the origin, *etc.*, and then invokes the protected `drawGlyph` method to do the actual drawing. This design illustrates *method factoring*: drawing code common to all glyphs is *factored* into the `draw` method supplied by the base class.

Since we might want to support different kinds of drawables (*e.g.*, computer screens and postscript), we collect the subclasses of `BaseGlyph` into a module that is parameterized by the `BASE_GLYPH` signature:

```
module Glyphs (G : BASE_GLYPH) {
  class LineGlyph {
    inherits G.BaseGlyph
    field p1 : Point
    field p2 : Point
    override method drawGlyph (d : Drawable) -> () { d.drawLine(p1, p2) }
    public maker line (p1 : Point, p2 : Point) {
      super mk();
      field p1 = p1;
      field p2 = p2
    }
  }
   ... implementations of other glyph classes ...
}
```

In this code fragment, `LineGlyph` is a subclass of `G.BaseGlyph` and provides the implementation of the `drawGlyph` method for drawing lines. This example illustrates both module and class-based code reuse. We use a parameterized module to abstract over the base-class of `LineGlyph`, which allows multiple class hierarchies to be defined by applying `Glyphs` to different base classes, while using inheritance to factor out code common to all glyphs.

Part II

MOBY Reference

Chapter 4

Introduction

4.1 Program structure

1. At the top-most level, a MOBY program is structured as a collection of source groups. A source group may be specific to a given application or it may be a library that can be reused by many applications.
2. A source group consists of a set of MOBY compilation units. A MOBY compilation unit is either a *signature*, a *module*, or a *parameterized module*. Associated with each compilation unit is a *prelude* that describes various aspects about the compilation context used to compile the unit.
3. A signature consists of a sequence of specifications.
4. A module or parameterized module has a body that consists of a sequence of declarations.
5. Declarations.
6. Expressions.
7. At the lowest level is MOBY's *lexical structure*, which is covered in Chapter 5.

4.2 Notation

Chapter 5

Lexical structure

MOBY programs are written using a subset of the ASCII character set.[1] The sequence of ASCII characters that make up a MOBY compilation unit are reduced to a sequence of *input elements*, which are white space, comments, and tokens.

5.1 White space

The following characters are considered *white space* characters: the horizontal tab (ASCII 9), the newline (ASCII 10), the vertical tab (ASCII 11), the pagefeed character (ASCII 12), the carriage return (ASCII 13), and the space character (ASCII 32).

The sequence of characters that make up a MOBY compilation unit are logically divided into lines by *line terminators*, which are a subset of the white-space characters. A line terminator is either a line feed, an carriage return, or a carriage return followed by a line feed. To accomodate certain operating systems, we allow the ASCII SUB (or control-Z) character as the last charater of an input stream.

5.2 Comments

MOBY supports two forms of comment: single line comments, which begin with the two characters '`//`', and end with the next line terminator (or the end of the input stream); and traditional comments, which are bracketed by '`/*`' and '`*/`'. Traditional comments may be nested. The character sequences '`/*`' and '`*/`' are ignored in single-line comments, while '`//`' is ignored inside nested comments.

5.3 Identifiers

[[describe capitalization convention]]

[1] It is expected that we might generalize this to Unicode at some point.

5.3.1 Reserved words

abstract	**case**	**class**	**const**	**datatype**
deconst	**else**	**enumtype**	**except**	**extends**
field	**final**	**finally**	**fn**	**fun**
if	**implements**	**include**	**inherits**	**is**
isnot	**ivar**	**local**	**maker**	**method**
module	**mvar**	**new**	**objtype**	**of**
override	**public**	**raise**	**self**	**signature**
spawn	**super**	**sync**	**tagtype**	**then**
try	**type**	**typeof**	**val**	**var**
when	**with**			

5.3.2 Underscore identifiers

Identifiers with leading underscores ('_') are reserved for experimentation with new language features.

5.4 Operators

5.5 Separators

5.6 Literals

A *literal* is a syntactic representation of a primitive value. MOBY has syntax for boolean, integer, floating-point, character, and string literals.

5.6.1 Boolean literals

There are two boolean literals: **`True`** and **`False`**. A boolean literal is always of type `Bool`.

5.6.2 Integer literals

IntegerLiteral
  ::= *DecimalLiteral*
  | *HexLiteral*

DecimalLiteral
  ::= *DecimalDigit*$^+$

DecimalDigit
::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
HexLiteral
::= **0 x** *HexDigit*$^+$
| **0 X** *HexDigit*$^+$

HexDigit
::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **a** | **A** | **b** | **B** | **c** | **C** | **d** | **D** | **e** | **E** | **f** | **F**

5.6.3 Floating-point literals

FloatLiteral
::= *DecimalDigit*$^+$ **.** *DecimalDigit*$^+$ *Exponent*^{opt}
| *DecimalDigit*$^+$ *Exponent*

Exponent
::= **e** Sign^{opt} *DecimalDigits*$^+$
| **E** Sign^{opt} *DecimalDigits*$^+$

Sign
::= **+** | **-**

5.6.4 Character literals

5.6.5 String literals

[[The space character is the only whitespace character allowed inside string literals]]

[[Use SML syntax for multi-line string literals]]

Chapter 6

Evaluation

6.1 Tuples

6.2 Imperative features

Unlike traditional imperative languages (*e.g.*, C), MOBY does not have mutable variables, but it does support mutable data structures. MOBY allows the fields of object and record types to be declared as mutable (using the **`var`** keyword) and also provides several types of mutable arrays.

6.2.1 Assignment and L-values

MOBY uses the special operator `:=` for all forms of assignment to mutable data structures. To make the assignment syntax uniform, we distinguish those expressions that can appear on the left-hand-side of the `:=` operator in the typing rules. We note this distinction using the judgement form

$$\Gamma \vdash e \triangleright \mathbf{lval}\,\tau$$

when e denotes an *L-value* of type τ. With this notation, the typing rule for assignment is

$$\frac{\Gamma \vdash e_1 \triangleright \mathbf{lval}\,\tau_1 \quad \Gamma \vdash e_2 \triangleright \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_1}{\Gamma \vdash e_1 \texttt{:=} e_2 \triangleright ()}$$

L-values are not first-class; for example, it is not possible to pass an l-value as a function argument. When an l-value is used in a non-assignment context, we just forget the annotation, as expressed in the following typing rule:

$$\frac{\Gamma \vdash e \triangleright \mathbf{lval}\,\tau}{\Gamma \vdash e \triangleright \tau}$$

6.2.2 Mutable fields

6.2.3 Mutable arrays

$$\frac{\Gamma \vdash e_1 \triangleright \texttt{Array}(\tau) \quad \Gamma \vdash e_2 \triangleright \texttt{Int}}{\Gamma \vdash e_1 \,\texttt{[}e_2\texttt{]} \triangleright \textbf{lval}\,\tau}$$

6.2.4 Pointers

While l-values are not first-class, it is possible to reify an l-value into a *pointer value*. The MOBY pervasive environment includes the following definitions for pointers:

```
type Ptr(t)
const NULL : [t] Ptr(t)
val isNull : [t] Ptr(t) -> Bool
```

Pointer values are created by taking the address of an l-value using the prefix **&** operator. This expression form has the following typing rule:

$$\frac{\Gamma \vdash e \triangleright \textbf{lval}\,\tau}{\Gamma \vdash \texttt{\&}e \triangleright \texttt{Ptr}(\tau)}$$

A pointer can be used as an l-vlaue by applying the prefix ***** operator, which has the following typing rule:

$$\frac{\Gamma \vdash e \triangleright \texttt{Ptr}(\tau)}{\Gamma \vdash \texttt{*}e \triangleright \textbf{lval}\,\tau}$$

6.2.5 References

Since reference cells are a common case of mutable data structures, the MOBY pervasive environment provides the `Ref` type constructor and supporting operations.

```
type Ref(t)
val ref : [t] t -> Ref(t)
val deref : [t] Ref(t) -> t
```

Note that the `deref` function returns a value, instead of an l-value, which means that it cannot be used for assignment. To support assignment of reference cells, MOBY overloads the prefix ***** operator with the following typing rule:

$$\frac{\Gamma \vdash e \triangleright \texttt{Ref}(\tau)}{\Gamma \vdash \texttt{*}e \triangleright \textbf{lval}\,\tau}$$

6.3 Exceptions

Like most modern languages, MOBY provides a mechanism for signalling and handling exceptional and error conditions. There are actually two mechanisms for supporting exceptions: a control-flow mechanism for aborting the current computation and escaping to some enclosing handler, and a mechanism for representing information about the situation that caused the abort.

6.3.1 Raising and handling exceptions

6.3.2 Exception values

6.3.3 The standard exceptions

```
tagtype Exn
tagtype System of (String) extends Exn
tagtype MatchFail extends Exn
tagtype BindFail extends Exn
tagtype Range extends Exn
tagtype Subscript of (Int) extends Exn
tagtype DivByZero extends Exn
tagtype Fail of String extends Exn
```

6.4 Threads

6.5 Synchronization

Chapter 7

Signatures and modules

7.1 Signatures

SignatureDecl
  ::= **`signature`** *SigId* **`{`** *Specification** **`}`**
  | **`signature`** *SigId* **`=`** *SigId* (**`with`** **`{`** *TypeReveal*$^+$ **`}`**)^{opt}

[[Need to allow qualified names on lhs of type reveal.]]

7.2 Specifications

Specification
  ::= **`include`** *SigId*
  | *ModuleSpec*
  | *TypeSpec*
  | *ClassSpec*
  | *ConstSpec*
  | *ValueSpec*

7.2.1 Module specifications

ModuleSpec
  ::= **`module`** *ModuleId* **`:`** *Signature*

Signature
  ::= **`{`** *Specification** **`}`**
  | *SigId* (**`with`** **`{`** *TypeReveal*$^+$ **`}`**)^{opt}

7.2.2 Type specifications

TypeSpec
::= **type** *TypeId TypeParams*^{opt}
| *TypeReveal*
| *DataTypeDecl*
| *EnumTypeDecl*
| *TagTypeDecl*
| *ObjectTypeDecl*

TypeReveal
::= **type** *TypeId TypeParams*^{opt} **=** *Type*
| **type** *TypeId TypeParams*^{opt} **<:** *Type*

TypeParams
::= **(** *TypeVar* (**,** *TypeVar*)* **)**

7.2.3 Class specifications

ClassSpec
::= **class** *ClassId TypeParams*^{opt} **:** *ClassInterface*

ClassInterface
::= **{** *InheritsSpec*^{opt} *ImplementsSpec*^{opt} *MemberSpec** **}**

ImplementsSpec
::= **implements** *NamedType* (**,** *NamedType*)*

MemberSpec
::= **public**^{opt} **field** *Label* **:** *ExtendedType*
| **public**^{opt} *MethodSpec*
| **public**^{opt} **maker** *MakerId* **of** *Types*

MethodSpec
::= **abstract method** *Label* **:** *TypeScheme*
| **method** *Label* **:** *TypeScheme*
| **final method** *Label* **:** *TypeScheme*

7.2.4 Constant specifications

ConstSpec
::= **const** *DataCon* **:** *TypeParams*^{opt} *Type* (**of** *Types*)^{opt}
| **deconst** *DataCon* **:** *TypeParams*^{opt} *Type* (**of** *Types*)^{opt}

7.2.5 Value specifications

ValueSpec
 ::= **`val`** *ValueId* **`:`** *TypeScheme*
 | **`val`** *DataCon* **`:`** *TypeScheme*

7.3 Modules

ModuleDecl
 ::= **`module`** *ModuleId* (**`:`** *Signature*)^{opt} *ModuleBody*
 | **`module`** *ModuleId* (**`:`** *Signature*)^{opt} **`=`** *ModuleDef*

ModuleBody
 ::= **`{`** *Declaration** **`}`**

ModuleDef
 ::= *Path*^{opt} *ModuleId*
 | *ModuleId* **`(`** (*ModuleExp* (**`,`** *ModuleExp*)*)^{opt} **`)`**

ModuleExp
 ::= *ModuleBody*
 | *ModuleDef*

7.3.1 A module's implicit signature

Each module definition defines an implicit signature, which governs how the outside world sees the contents of the module. The implicit signature of a module can be determined by a simple syntactic transformation of its declarations into specicifations. If the resulting signature is not well formed, then the module is not well formed. The translation of a declaration to zero or more specifications is as follows:

- If the declaration is annotated with **`local`** then there is no corresponding specification.
- If the declaration has the form ...
- If the declaration is a type declaration, then the specification is identical.
- If the declaration is a constrained class declaration of the form

  ```
  class C : ClassInterface { ... }
  ```

  then the specification is

  ```
  class C : ClassInterface
  ```

- If the declaration is an unconstrained class declaration of the form ...
- If the declaration is a value binding of the form

  $$\texttt{val}\ (x_1\ :\ t_1,\ \ldots,\ x_n\ :\ t_n)\ \texttt{=}\ \textit{Expression}$$

  Then there are *n* corresponding specifications

  $$\begin{array}{l}\texttt{val}\ x_1\ :\ t_1\\ \ldots\\ \texttt{val}\ x_n\ :\ t_n\end{array}$$

  [[What about type parameters??]]

Note that **`local`** is a fairly crude tool to define an abstract interface; our intention is that one will use signature ascription to define abstraction boundries at the package (or source group) level, and that the modules in a given group will have access to each other's internal types.

7.4 Parameterized modules

ParamModuleDecl
  ::= **`module`** *ModuleId* **`(`** *ModuleParams*^{opt} **`)`** (**`:`** *Signature*)^{opt} *ModuleBody*
  | **`module`** *ModuleId* **`(`** *ModuleParams*^{opt} **`)`** (**`:`** *Signature*)^{opt} **`=`** *ModuleDef*

ModuleParams
  ::= *ModuleId* **`:`** *Signature* (**`,`** *ModuleId* **`:`** *Signature*)*

[[Each instance of a parameterized module has its own state.]]

Here is an example of a generic list sorting module that is parameterized over the element type and ordering function.

```
module ListSortFn (E : {type T val cmp : (T, T) -> Order})
{
  fun sort (l : List(E.T)) -> List(E.T) {
    fun ins : (E.T, List(E.T)) -> List(E.T) {
      (x, Nil) => Cons(x, Nil),
      (x, Cons(y, ys)) => if (E.cmp(x, y) is Greater)
        then Cons(y, ins(x, ys))
        else Cons(x, Cons(y, ys))
    };
    fun lp : (List(E.T), List(E.T)) -> List(E.T) {
      (Nil, l) => l,
      (Cons(x, xs), l) => lp (xs, ins(x, l))
    };
    lp (l, Nil)
  }
}
```

7.5 Declarations

Declaration
::= **`local`**^{opt} *ModuleDecl*
| **`local`**^{opt} *TypeDecl*
| **`local`**^{opt} *ClassDecl*
| **`local`**^{opt} *ConstDecl*
| **`local`**^{opt} *FunDecl*
| **`local`**^{opt} *ValueDecl*

Declarations make up the body of a module. A declaration may be annotated with the **`local`** keyword, which means that it is not visible outside its module. All other declarations are visible outside the module, although they may be hidden later using signature ascription (see Section **??**). MOBY provides declaration forms for submodules, types, classes, constants, functions, and values. Module declarations have already been discussed in Section 7.3, and we postpone the discussion of type and class declarations to chapters 8 and 9 respectively.

7.5.1 Constant declarations

ConstDecl
::= **`const`** *DataCon* **`:`** *TypeScheme* **`=`** *Pattern*
| **`const`** *DataCon* **`:`** *TypeParams*^{opt} *Type* **`of`** *Params* **`=`** *Pattern*

Constant declarations introduce symbolic names for constant values. Unlike in the case of value bindings, these names are in the class of data-constructor identifiers and can be used in pattern matching. Furthermore, MOBY allows constants to be parameterized over a tuple of free variables.

The right-hand-side of a constant declaration must be a *pattern expression*. These restricted patterns are described in Section 12.4.

7.5.2 Function declarations

FunDecl
::= **`fun`** *ValueId FunDef*

FunDef
::= *BoundTypeVars*^{opt} *Params*$^{+}$ **`->`** *Type Block*
| **`:`** *TypeScheme MatchCase*

There are two forms of function definition: the first supports curried definitions without pattern matching, whereas the second supports an equational style of definition without currying. The function definition syntax is also used in the syntax of function expressions (see Section 11.11.1) and method declarations (see Section 9.2.2). The following example illustrates both forms of function declaration:

```
fun map [s, t] (f : s -> t) (l : List(s)) -> List(t) {
  fun mapf : List(s) -> List(t) {
    Nil => Nil,
    Cons(x, xs) => Cons(f x, mapf xs)
  };
  mapf l
}
```

The `map` function is a curried definition and its body is a block consisting of the definition of `mapf` and its application to the list `l` (blocks are described in Section 11.1). The definition of `mapf` uses the equational style.

[[explain `s` and `t`.]]

7.5.3 Value declarations

ValueDecl
  ::= **`val`** *BoundTypeVars*^{opt} *Params* **=** *Expression*
  | **`val`** *BoundTypeVars*^{opt} *Param* **=** *Expression*

7.5.4 Recursive declarations and scope

Type, class and function declarations in MOBY may be *recursive*. To allow programmers flexibility in the way they organize their code, MOBY allows declarations to include forward references to other identifiers bound at the same level[1]. To avoid nonsensical definitions and recursion through nested modules, however, there are some restrictions. Let $d_1 \cdots d_n$ be a sequence of declarations, let V_i be the set of names bound by declaration d_i, and let F_i be the free names of the right hand side of d_i. Then, we say that $v \in V_i$ is *fully defined* at declaration $j >= i$, if for every $v' \in F_i$, it is the case that there exists $k < j$ with $v' \in V_k$. Given this definition, we place five restrictions on the declarations that make up a module body:

1. At the point of the last declaration in the module body, all names defined in the module must be fully defined.
2. Names may not be redefined.

   [[we might weaken this restriction to apply only to type and module names]]

3. The names defined by type abbreviations (see Section 8.1) and value declarations (see Section 7.5.3) must be fully defined at their point of declaration.

[1] By *level*, we mean the module nesting level.

4. Only fully defined names are visible inside a nested module.

5. Type bounds must be fully defined.

The third restriction is sufficient to avoid nonsensical definitions and the fourth avoids the problem of recursion through nested modules. For example, the following declarations are well-formed:

```
fun f (x : Int) : Int { g (x-1) }
fun g (y : Int) : Int { y+1 }
module M {
  val a = f 1
  fun h () -> Int { a }
}
val w = M.h()
```

Notice that `w` has a dependency on `f` and `g` via the nested module `M`. Since module `M` imports `f`, we cannot move the definition of `g` to after `M`. If we did so, then `f` would not be fully defined at the point of `M`'s declaration and thus would not be visibile inside `M`.

7.6 Signature Matching

Chapter 8

Type declarations

TypeDecl
::= *TypeNameDecl*
| *DataTypeDecl*
| *EnumTypeDecl*
| *TagTypeDecl*
| *ObjectTypeDecl*

8.1 Type name declarations

TypeNameDecl
::= **`type`** *TypeId TypeParams*^{opt} **`=`** *Type*

8.2 Object type declarations

ObjectTypeDecl
::= **`objtype`** *TypeId TypeParams*^{opt} *ObjectMembers*
| **`objtype`** *TypeId TypeParams*^{opt} **`=`** *NamedType*

ObjectMembers
::= **`{`** *ObjectMember** **`}`**

ObjectMember
::= **`extends`** *NamedType*
| **`field`** *Label* **`:`** *ExtendedType*
| **`method`** *Label* **`:`** *TypeScheme*

8.3 Datatype declarations

DataTypeDecl
 ::= **datatype** *TypeId TypeParams*^{opt} *DataTypeDef*
 | **datatype** *TypeId TypeParams*^{opt} **=** *NamedType*

DataTypeDef
 ::= **{** *DataConDef* (**,** *DataConDef*)* **}**

DataConDef
 ::= *DataCon* (**of** *Types*)^{opt}

8.4 Enumeration type declarations

EnumTypeDecl
 ::= **enumtype** *TypeId EnumTypeDef*
 | **enumtype** *TypeId* **=** *Path*^{opt} *TypeId*

EnumTypeDef
 ::= **{** *DataCon* (**,** *DataCon*)* **}**

Enumeration type declarations introduce an ordered collection of nullary data constructors (or data constants). Because the constants are ordered, two enumeration types are considered equal only when they have the same constants in the same order.

In addition to its constants, an enumeration type declaration introduces a collection of operations. These operations allow efficient mapping between integers and the enumeration type, as well as testing the order of elements. To avoid namespace pollution, we treat the enumeration type as a *pseudo-module* and use dot notation for most of the operations. For an enumeration type declaration

enumtype `T { ` C_1, ..., C_n `}`

the following operations are defined:

Operation	Type	Description
`T.toInt`	`T -> Int`	Convert to integer; C_1 maps to 0 and C_n maps to $n-1$.
`T.fromInt`	`Int -> T`	Convert an integer to `T`; 0 maps to C_1 and $n-1$ maps to C_n. Out of range values cause the exception `Range` to be raised.
`T.first`	`T`	The first element of the enumeration (*i.e.*, C_1).
`T.last`	`T`	The last element of the enumeration (*i.e.*, C_n).
`T.succ`	`T -> T`	Returns the successor of its argument; raises the exception `Range` on C_n.
`T.pred`	`T -> T`	Returns the predecessor of its argument; raises the exception `Range` on C_0.
`T.compare`	`T -> Order`	Compares the order of two elements.

In addition to these operations, all enumeration types have the standard infix relational and equality operations defined on them.

8.5 Tagtype declarations

TagTypeDecl
::= **`tagtype`** *TypeId TypeParams*^{opt} (**`of`** *Types*)^{opt} (**`extends`** *NamedType*)^{opt}
| **`tagtype`** *TypeId TypeParams*^{opt} **=** *NamedType*

Tagtypes provide the programmer with a mechanism for defining tree-structured type hierarchies, with checked downward coercions. A tagtype declaration of the form

`tagtype` `T` **`of`** `(`t_1`, ..., `t_n`)`

defines a *root tagtype* `T`. The name `T` also serves as a data constructor. A tagtype declaration of the form

`tagtype` `S` **`of`** `(`s_1`, ..., `s_m`)` **`extends`** `T`

defines a tagtype `S` that is a subtype of the type `T`. In order that the declaration of `S` be valid, we require that $m \geq n$ and that for $1 \leq i \leq n$ we have $\Gamma \vdash s_i <: t_i$.

There is a standard predefined tagtype, called `Exn`, that is used to represent exceptions.

Chapter 9

Classes

Classes play two important rôles in MOBY: they are the mechanism used to implement objects and they support code reuse and specialization via implementation inheritance.

9.1 Class declarations

ClassDecl
  ::= **`class`** *ClassId* (**`:`** *ClassInterface*)^{opt} *ClassDef*
  | **`class`** *ClassId* (**`:`** *ClassInterface*)^{opt} **`=`** *NamedClass*

ClassDef
  ::= **`{`** (**`inherits`** *NamedClass*)^{opt} *MemberDecl** *InitiallyClause*^{opt} **`}`**

NamedClass
  ::= *Path*^{opt} *ClassId* *TypeArgs*^{opt}

9.2 Member declarations

There three kinds of class members: fields, methods, and makers.

MemberDecl
  ::= **`public`**^{opt} *FieldDecl*
  | **`public`**^{opt} *MethodDecl*
  | **`public`**^{opt} *MakerDecl*

Member declarations may be annotated as **`public`**, which means that they are visible outside the class (and its subclasses). For fields and methods, the **`public`** annotation means that they are visible in the type of the objects generated from the class, while for makers the public annotation means that the maker may be used to generate new objects (*cf.*, Section 11.11.8).

9.2.1 Field declarations

Fields are the *instance variables* of MOBY objects. A field declaration defines the name and type of the field:

FieldDecl
  ::= **`field`** *Label* **`:`** *ExtendedType*

Unlike many object-oriented languages, fields are immutable by default in MOBY. To define a mutable field, one uses the **`var`** annotation.

9.2.2 Method declarations

A method declaration is either abstract or concrete:

MethodDecl
  ::= **`abstract`** **`method`** *Label* **`:`** *TypeScheme*
  | **`final`**^{opt} **`override`**^{opt} **`method`** *Label* *FunDef*

Abstract method declarations are denoted by the keyword **`abstract`** and serve as placeholders for a concrete definition to provided by a subclass. If a class has one or more abstract methods, the class is called an *abstract class*. Abstract classes may not have public makers.

Concrete method declarations are similar to function declarations; they have a function type and body consisting of a block. Concrete methods may be annotated with the **`final`** keyword, which means that they may not be overridden by subclasses, and/or with the **`override`** keyword, which means that the method declaration is overriding a declaration in its superclass. Inside the body of a method the reserved identifiers **`self`** and **`super`** are in scope (*cf.*, Section 11.11.9). The reserved identifier **`self`** is bound to the object containing the method and is used to accessthe object's fields and methods. The reserved identifier **`super`** is used to access super-class methods.

9.2.3 Maker declarations

MakerDecl
  ::= **`maker`** *MakerId Params MakerBlock*

MakerBlock
  ::= **`{`** *MakerStmt* (**`;`** *MakerStmt*)* **`}`**
  | **`{ }`**

MakerStmt
  ::= **`super`** *MakerId Expression*
  | **`field`** *Label* **`=`** *Expression*
  | *Statement*

Only public makers may be used to create objects (*cf.*, Section 11.11.8).

A maker block must have unit type (*i.e.*, return no results).

9.3 Initially clauses

InitiallyClause
  ::= **`initially`** *Expression*

An **`initially`** clause must have unit type (*i.e.*, return no results).

9.4 Inheritance

One of the primary rôles played by classes is as a mechanism for code reuse and code specialization. This rôle is supported by inheritance.

9.4.1 Method overriding

9.4.2 Object initialization

Chapter 10

Types, type equality, and subtyping

MOBY is a *typeful* language and understanding its type system is a prerequesite to understanding the language.

10.1 Type schemes

TypeScheme
  ::= *BoundTypeVars*^{opt} *Type*

BoundTypeVars
  ::= `[` *BoundTypeVar* (`,` *BoundTypeVar*)* `]`

BoundTypeVar
  ::= *TypeVar* (`<:` *Type*)^{opt}

10.2 Types

10.2.1 Type tuples

Types
  ::= *TypeTuple*
  | *Type*

TypeTuple
  ::= `(` (*Type* (`,` *Type*)$^{+}$)^{opt} `)`

Type tuples are used in the argument and result position of function types and in the argument position of data constructors.

10.2.2 Function types

Type
  ::= *FunType*
  | *AtomicType*

FunType
  ::= *AtomicTypes* **->** *FunType*
  | *AtomicTypes* **->** *AtomicTypes*

AtomicTypes
  ::= *TypeTuple*
  | *AtomicType*

Note that the **->** type constructor is right associative.

10.2.3 Atomic types

AtomicType
  ::= *TypeVar*
  | *NamedType*
  | **$** *TypeTuple*

10.2.4 Type constructors

NamedType
  ::= $Path^{opt}$ *TypeId* $TypeArgs^{opt}$
  | **typeof (** *NamedClass* **)**
  | **#** *NamedClass*

TypeArgs
  ::= **(** *Type* (**,** *Type*)* **)**

10.2.5 Record types

RecordType
  ::= **{|** $LabeledTypes^{opt}$ **|}**

LabeledTypes
  ::= *Label* **:** *ExtendedType* (**,** *Label* **:** *ExtendedType*)*

10.2.6 Extended types

ExtendedType
::= *TypeScheme*
| **var** *Type*

Extended types are used in the specification of fields in record types (see Section 10.2.5) and object types (see Section 8.2). Using an extended type, one can specify that a field is polymorphic[1] or that it is mutable. There three kinds of mutable fields: those annotated with the **var** keyword are simply updatable slots; those annotated with the **ivar** keyword are write-once fields with synchronous reading (these have so-called I-variable semantics); and those annotated with the **mvar** keyword are read/write cells with synchronous reading (these have so-called M-variable semantics).

10.2.7 Pervasive types

MOBY provides the following pervasive types and type constructors:

Bool

Int This type is the type of 32-bit 2's compliment signed integers.

Long This type is the type of 64-bit 2's compliment signed integers.

Integer This type is the type of arbitrary precision signed integers.

Float This type is the type of IEEE Std 754-1985 single precision floating values.

Double This type is the type of IEEE Std 754-1985 double precision floating values.

Extended This type is the type of IEEE Std 754-1985 extended-double precision floating values. Note that the size of this type is system dependent.

Char This type is the type of 8-bit ASCII characters.

Rune This type is the type of 32-bit UNICODE characters.

String This type is the type of immutable sequences of 8-bit characters.

Exn This tagtype is the base tagtype for exception values.

Order This enumeration type is used to denote the relation between elements of ordered types. It has the following definition:

```
enumtype Order { Less, Equal, Greater }
```

Option(t)

[1] We call this "*almost first-class polymorphism*."

`List(t)`

`Vector(t)`

`Array(t)`

`Ref(t)`

`Ptr(t)`

10.3 Type equality

10.4 Subtyping

Chapter 11

Expressions

11.1 Blocks

Block
  ::= **{** (*Statement* **;**)* *Expression* **}**
   | **{ }**

Statement
  ::= *Expression*
   | *Binding*

Binding
  ::= **let** *Patterns* **=** *Expression*
   | **fun** *ValueId FunDef* (**and** *ValueId FunDef*)*

Blocks are an expression form that combines the let-binding construct found in many functional languages with expression sequencing. A block consists of a sequence of zero or more statements followed by an expression. A statement is either a function or value binding, or a unit typed expression. The scope of a binding is from the binding to the end of the block. In the case of a function binding, the body of the function is included in the function's scope (to allow recursion). Mutually recursive functions are supported by joining the bindings with the keyword "**and**."

11.2 If-then-else expressions

Expression
  ::= **if** *Expression* **then** *Expression* **else** *Expression*

The *if expression* has the expected semantics. We require that the type of the first expression be `Bool` and that the types of the then and else clauses be the same.

11.3 TryExpressions

TryExpression
  ::= **`try`** *Expression* **`except`** *MatchCase*
  | **`try`** *Expression* **`finally`** *Expression*

11.3.1 Try-except expressions

The try-except expression is used to handle exceptions (also see the discussion of exceptions in Section 6.3).

11.3.2 Try-finally expressions

The expression

$$\texttt{\textbf{try}}\ e_1\ \texttt{\textbf{finally}}\ e_2$$

executes the expression e_1 (called its *body*) until e_1 terminates and then it executes e_2 (called its *finally clause*). The finally clause will be excuted no matter how the body terminates (*i.e.*, even if it raises an exception or exits the hosting thread). If the body (e_1) terminates normally, then after the finally clause (e_2) completes execution the result of the body is returned. If e_1 terminates by raising an exception, then after the finally clause is executed, the exception packet is propagated to the next enclosing try-except or try-finally expression in the call-stack (see also the discussion of exceptions in Section 6.3). If the body (e_1) terminates by exiting the hosting thread, then when the execution of the finally clause (e_2) completes, control is passed to the next enclosing try-finally expression (if any). If the finally clause (e_2) raises an exception, it is as if the whole try-finally expression had raised the exception.

The type of the whole expression is the type of the body and we require that the finally clause have unit type (*i.e.*, return no results).

$$\frac{\Gamma \vdash e_1 \rhd \tau \quad \Gamma \vdash e_2 \rhd ()}{\Gamma \vdash \texttt{\textbf{try}}\ e_1\ \texttt{\textbf{finally}}\ e_2 \rhd \tau}$$

The try-finally expression is used to robustly free resources. For example, will always close the stream `s`, even if the function application terminates without returning (*e.g.*, if it raises an exception or calls `exit`).

```
{ val s = TextIO.openIn "somefile";
  try someFunction (s)
  finally TextIO.closeIn s
}
```

To illustract how try-finally and try-except expressions are unwound, consider the following example:

```
try {
  try {
    try {
      try {
        try raise BAR finally ConsoleIO.print "1";
        ConsoleIO.print "a"
      } finally ConsoleIO.print "2";
      ConsoleIO.print "b"
    } except { BAR => ConsoleIO.print "3" };
    ConsoleIO.print "c"
  } finally ConsoleIO.print "4";
  ConsoleIO.print "d"
} except { _ => ConsoleIO.print "5" }
```

This will print "`123c4d`." But if we change the second finally clause to raise the exception `BAZ` as follows:

```
try {
  try {
    try {
      try {
        try raise BAR finally ConsoleIO.print "1";
        ConsoleIO.print "a"
      } finally {ConsoleIO.print "2"; raise BAZ};
      ConsoleIO.print "b"
    } except { BAR => ConsoleIO.print "3" };
    ConsoleIO.print "c"
  } finally ConsoleIO.print "4";
  ConsoleIO.print "d"
} except { _ => ConsoleIO.print "5" }
```

then the expression will print "`1245`." Note that in both examples, the finally clauses always execute.

[[If we have some form of thread finalization, then the finally clauses should be executed when a thread is finalized. Also when a thread is alerted.]]

11.4 Raise expressions

Expression
  ::= **`raise`** *Expression*

The expression

`raise` e

evaluates e to an *exception packet* and then passes the packet to the dynamically innermost enclosing try-except or try-finally expression.

Because the raise expression does not return a value to its evaluation context, it can have any type.

$$\frac{\Gamma \vdash e \rhd \mathtt{Exn} \quad \Gamma \vdash \tau \rhd \mathbf{Ok}}{\Gamma \vdash \mathtt{\textbf{raise}}\ e \rhd \tau}$$

11.5 Spawn expressions

Expression
::= **`spawn`** *Expression*

The spawn expression creates a new thread of control to execute the expression.

11.6 Sync expressions

Expression
::= **`sync`** *Expression*

The sync expression causes the hosting thread to synchronize on the synchronous event that the expression evaluates to.

11.7 Binary expression

MOBY provides a fixed set of infix operators that can be used to form binary expressions. The infix operators are given in Table 11.1 (listed in order of increasing precedence).

11.7.1 Assignment expressions

AssignmentExpr
::= *ConditionalOrExpr* **`:=`** *ConditionalOrExpr*
| *ConditionalOrExpr*

The assignment operator is used to modify mutable storage. The left-hand side of an assignment must be an *l-value*; l-values include mutable fields in objects and records, array elements, and reference cells. The MOBY typing rules distinguish between l-values and regular values as is discussed in Section 6.2. The **`:=`** operator cannot be overloaded.

Table 11.1: MOBY infix and prefix operators

	Infix operators
`:=`	assignment operator (lowest precedence)
`||`	conditional or operator
`&&`	conditional and operator
`==` `!=`	equality operators
`<=` `<` `>=` `>`	relational operators
`@` `:`	list operators (right associativity)
`<<` `>>` `>>>`	bitwise shift operators
`|`	bitwise or operator
`&`	bitwise and operator
`+` `-`	additive operators
`*` `/` `%`	multiplicative operators
`**`	exponentiation
`^`	function composition (highest precedence)

11.7.2 Conditional expressions

MOBY provides two infix operators for conditional evaluation.

ConditionalOrExpr
 ::= *ConditionalOrExpr* **||** *ConditionalAndExpr*
 | *ConditionalAndExpr*

ConditionalAndExpr
 ::= *ConditionalAndExpr* **&&** *EqualityExpr*
 | *EqualityExpr*

These operators are not strict in their right-hand-side argument and can be viewed as short-hand for **`if`**-expressions as follows:

$$e_1 \;\texttt{||}\; e_2 \equiv \texttt{if } e_1 \texttt{ then True else } e_2$$
$$e_1 \;\texttt{\&\&}\; e_2 \equiv \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else False}$$

The **||** and **&&** operators cannot be overloaded.

11.7.3 Equality expressions

EqualityExpr
 ::= *EqualityExpr EqualityOp RelationalExpr*
 | *RelationalExpr*

EqualityOp
  ::= **==** | **!=**

MOBY supports testing of equality on the following pervasive types:

`Bool`	`Char`	`String`	`Int`
`Long`	`Integer`	`Float`	`Double`
`Extended`			

Equality is also defined on all enumeration types, object types, and record types with mutable fields. In addition, `Array(`τ`)`, `Ref(`τ`)`, and `Ptr(`τ`)` are equality for any type τ. In the case of mutable types (*e.g.*, objects and arrays), two values are the same if they are the same memory object. For non-mutable equality types, two values are equal if they are the same value. The only exception to this is for the `Float` and `Double` types, which use the IEEE Std 754-1985 equality operation (*i.e.*, $x \neq x$ when x is a NaN) [IEE85]. Note that unlike SML, equality does not extend to structured values and there is no notion of polymorphic equality.

[[We probably should allow equality of tuples of equality types.]]

[[We may want to support equality on monomorphic structured values in the future.]]

11.7.4 Relational expressions

RelationalExpr
  ::= *RelationalExpr RelationalOp ListExpr*
  | *ListExpr*

RelationalOp
  ::= **<** | **<=** | **>=** | **>**

The relational operators are overloaded on the types: `Bool`, `Int`, `Long`, `Integer`, `Float`, `Double`, `Char`, and `String`.

11.7.5 List expressions

ListExpr
  ::= *ShiftExpr ListOp ListExpr*
  | *ShiftExpr*

ListOp
  ::= **@** | **::**

MOBY provides infix operators for consing an element on a list (`::`) and appending two lists (`@`). Unlike the other infix operators, the list operators associate to the right. The `::` operator is actually the list-cell data constructor and can also be used in patterns.

11.7.6 Bitwise expressions

ShiftExpr
  ::= *ShiftExpr ShiftOp BitwiseOrExpr*
  | *BitwiseOrExpr*

ShiftOp
  ::= **<<** | **>>** | **>>>**

BitwiseOrExpr
  ::= *BitwiseOrExpr* **|** *BitwiseAndExpr*
  | *BitwiseOrExpr* **\/** *BitwiseAndExpr*
  | *BitwiseAndExpr*

BitwiseAndExpr
  ::= *BitwiseAndExpr* **&** *AdditiveExpr*
  | *BitwiseAndExpr* **/** *AdditiveExpr*
  | *AdditiveExpr*

MOBY provides shifting and bitwise logical operations on the `Int`, `Long`, and `Integer` types. As in JAVA, **<<** is left-shift, **>>** is arithmetic right shift (*i.e.*, with sign extension), and **>>>** is logical right shift (*i.e.*, no sign extension). The infix operator **|** computes the bitwise logical or of its arguments, while the operator **&** computes the bitwise logical and. MOBY defines two additional operators, **\/** and **/**, for use by libraries (*e.g.*, for set union and intersection).

11.7.7 Arithmetic expressions

AdditiveExpr
  ::= *AdditiveExpr AdditiveOp MultiplicativeExpr*
  | *MultiplicativeExpr*

AdditiveOp
  ::= **+** | **-**

MultiplicativeExpr
  ::= *MultiplicativeExpr MultiplicativeOp Expr*
  | *ExponentialExpr*

MultiplicativeOp
  ::= ***** | **/** | **%**

ExponentialExpr
  ::= *ExponentialExpr* `**` *CompositionExpr*
  | *CompositionExpr*

The operators `+`, `-`, `*`, and `/`, are overloaded on all of the numeric types (`Int`, `Long`, `Integer`, `Float`, and `Double`) with their usual meaning. In addition, the `+` operator is used to denote string concatenation. The modulus operator (`%`) is overloaded on the integer types The exponentiation operator (`**`) is overloaded on the floating-point types (`Float` and `Double`).

11.7.8 Function composition expressions

CompositionExpr
  ::= *PrefixExpr* `^` *CompositionExpr*
  | *PrefixExpr*

The right-associative operator `^` can be used to compose two functions. Because of the way that the MOBY type system treats tuples, function composition is a special operator with its own typing judgement:

$$\frac{\Gamma \vdash e_1 \triangleright (\tau''_1, \ldots, \tau''_k) \to (\tau'_1, \ldots, \tau'_n) \quad \Gamma \vdash e_2 \triangleright (\tau_1, \ldots, \tau_m) \to (\tau''_1, \ldots, \tau''_k)}{\Gamma \vdash e_1 \hat{\ } e_2 \triangleright (\tau_1, \ldots, \tau_m) \to (\tau'_1, \ldots, \tau'_n)}$$

[[Another way to think about this is that ^ is overloaded on all combinations of polymorphic functions, where the arities match up correctly.]]

11.8 Prefix expressions

PrefixExpr
  ::= *PrefixOp* *PrefixExpr*
  | *ApplicationExpr*

PrefixOp
  ::= `*` | `&` | `!` | `~` | `-`

11.9 Application expressions

ApplicationExpr
  ::= *ApplicationExpr* *PostfixExpr*
  | *PostfixExpr*

11.10 Postfix expressions

PostfixExpr
 ::= *PostfixExpr* **.** *Label*
 | *PostfixExpr* **[** *Expression* **]**
 | *AtomicExpr*

11.11 Atomic expressions

AtomicExpr
 ::= **fn** *FunDef*
 | **case** *Expression* **of** *MatchCase*
 | **(** (*Expression* (**,** *Expression*)*)^{opt} **)**
 | **(** *Expression* **:** *Type* **)**
 | **(** *Expression* **is** *Pattern* **)**
 | **(** *Expression* **isnot** *Pattern* **)**
 | *ChoiceEvent*
 | *Block*
 | **new** *Path*^{opt} *MakerId*
 | **self**
 | **super .** *Label*
 | **nack_event**
 | **rdy_event**
 | **(** *Operator* **)**
 | *Path*^{opt} *ValueId*
 | *DataConstructor*
 | **#** *DataConstructor*
 | **?** *DataConstructor*
 | *Literal*

11.11.1 Function expressions

11.11.2 Case expressions

11.11.3 Expression tuple

11.11.4 Constraint expression

11.11.5 Match test expression

It is possible to test if the value computed by an expression matches a variable-free pattern using the infix **is** and **is not** keywords. The semantics of these forms can be defined as follows:

$$(e \ \mathbf{is} \ p) \equiv \mathbf{case} \ e \ \mathbf{of} \ \{ \ p \ \mathbf{=>} \ \mathbf{True,} \ \_ \ \mathbf{=>} \ \mathbf{False} \ \}$$
$$(e \ \mathbf{isnot} \ p) \equiv \mathbf{case} \ e \ \mathbf{of} \ \{ \ p \ \mathbf{=>} \ \mathbf{False,} \ \_ \ \mathbf{=>} \ \mathbf{True} \ \}$$

11.11.6 Record expression

11.11.7 Choice event

ChoiceEvent
  ::= **{|** *WrappedEvent* (**,** *WrappedEvent*)* **|}**

WrappedEvent
  ::= *Expression*
  | *Expression* **** *Pattern* **=>** *Expression*
  | *Expression* **** *MatchCase*

11.11.8 New expression

11.11.9 Self and super expressions

The self and super expressions are only allowed inside method bodies.

11.11.10 Identifier expressions

Chapter 12

Match cases and patterns

12.1 Match cases

MatchCase
  ::= **{** *MatchRule* (**,** *MatchRule*)* **}**

MatchRule
  ::= *Patterns* (**when** *Expression*)^{opt} **=>** *Expression*

Match cases are used to define function bodies (see Sections 7.5.2 and 11.11.1), exception handlers (see Section 11.3), and event wrappers (see Section 11.11.7). A match case consists of a sequence of one, or more, match rules. Each match rule consists of a pattern or pattern tuple, an optional guard expression, and an action expression. When matched agains a value, the pattern may bind variables to sub-terms of the value. The scope of these variables is the guard expression (if present) and the action expression.

A match case is evaluated against a value by testing each rule against the value until a match is found.[1] The order of the rules matters. A match rule is tested against a value by first checking to see if the value matches the pattern and, if so, then testing to see if the guard expression evaluates to true. When a matching rule is found, the corresponding action expression is evaluated and its result is returned as the result of the match case. If no match is found, the standard `Match` exception is raised. Note that the compiler is expected to issue a warning on nonexhaustive match cases (except when used for exception handlers) and to issue an error on redundant match cases.

The typing judgements for match rules are

$$\frac{\Gamma \vdash \mathit{pat} \triangleright (\tau, VE) \quad \Gamma + VE \vdash \mathit{exp} \triangleright \tau'}{\Gamma \vdash \mathit{pat}\ \texttt{=>}\ \mathit{exp} \triangleright \tau \rightarrow \tau'}$$

$$\frac{\Gamma \vdash \mathit{pat} \triangleright (\tau, VE) \quad \Gamma + VE \vdash \mathit{exp}_1 \triangleright \texttt{Bool} \quad \Gamma + VE \vdash \mathit{exp}_2 \triangleright \tau'}{\Gamma \vdash \mathit{pat}\ \textbf{when}\ \mathit{exp}_1\ \texttt{=>}\ \mathit{exp}_2 \triangleright \tau \rightarrow \tau'}$$

[1] Obviously, more efficient implementation strategies are possible.

12.2 Patterns

Patterns
  ::= *PatternTuple*
  | *Pattern*

PatternTuple
  ::= **(** *Pattern* (**,** *Pattern*)$^+$ **)**

12.2.1 Constrained patterns

Pattern
  ::= *IsPattern* **:** *Type*
  | *IsPattern*

The type checking rule for constrained patterns uses contravariant subtyping.

$$\frac{\Gamma \vdash \mathit{pat} \triangleright (\tau', VE) \quad \Gamma \vdash \mathit{ty} \triangleright \tau \quad \Gamma \vdash \tau <: \tau'}{\Gamma \vdash \mathit{pat} \ \mathtt{:}\ \mathit{ty} \triangleright (\tau, VE)}$$

12.2.2 Is patterns

IsPattern
  ::= *ValueId* **is** *OrPattern*
  | *ValueId* **isnot** *OrPattern*
  | **_ isnot** *OrPattern*
  | *OrPattern*

The **is** pattern operator allow one to bind a variable to the term that matches the right-hand-side pattern. The **isnot** pattern operator is like the **is** pattern operator, except that the pattern successfully matches a term only when the term does not match the right-hand-side pattern. We require that the right-hand-side pattern of an **isnot** operator be variable free.

12.2.3 Or patterns

OrPattern
  ::= *OrPattern* (**|** *ConsPattern*)$^+$
  | *ConsPattern*

The infix "**|**" operator forms the *or* of two patterns. We require that the variables bound in the two sub-patterns of an or pattern be the same and that each bound variable has the same type in each sub-pattern. An or pattern matches a value when either the left pattern matches the value or the left pattern does not match and the right pattern matched the value. In other words, the pattern is tested from left to right. While this does not affect success or failure of the match, it does affect the binding of variables. For example, in the pattern

```
a::_ | _::a_::_
```

the right pattern is covered by the left, so that `a` will always be bound to the head of the list on a successful match. In such a case, we say that the right pattern is *redundant* and the compiler should issue an error.

The typing rule for or patterns is

$$\frac{\Gamma \vdash pat_1 \triangleright (\tau_1, VE_1) \quad \Gamma \vdash pat_2 \triangleright (\tau_2, VE_2) \quad \Gamma \vdash \tau_1 = \tau_2 \quad \Gamma \vdash VE_1 = VE_2}{\Gamma \vdash pat_1 \,\mathbf{|}\, pat_2 \triangleright (\tau_1, VE_1)}$$

12.2.4 Cons patterns

ConsPattern
  ::= *ApplyPattern* **::** *ConsPattern*
  | *ApplyPattern*

The infix "**::**" operator is the list cons constructor.

12.2.5 Application patterns

ApplyPattern
  ::= *Path*^{opt} *DataCon PatternTuple*
  | *Path*^{opt} *DataCon AtomicPattern*
  | *AtomicPattern*

The typing rule for constructor application allows for *contravariant* subtyping

$$\frac{\Gamma(con) = \tau' \to \tau \quad \Gamma \vdash pat \triangleright (\tau'', VE) \quad \Gamma \vdash \tau' <: \tau''}{\Gamma \vdash con\,\mathbf{(}pat\mathbf{)} \triangleright (\tau, VE)}$$

12.2.6 Atomic patterns

AtomicPattern
  ::= **(** *Pattern* **)**
  | **_**
  | *ValueId*
  | *DataConstructor*
  | *Literal*
  | **-** *NumericLiteral*

Parenthesis can be used to override the natural associativity and precedence of patterns. For example, the pattern "`(a :: b) :: c`" matches a list of lists. Record patterns are described in the next section. The pattern "`_`" is called a *wildcard* and matches any value without binding any variables. A *ValueId* pattern matches any value and binds the variable to the value. A nullary data constructor pattern matches itself. A literal pattern matches the corresponding value.

12.2.7 Record patterns

RecordPat
::= **{|** *LabeledPat* (**,** *LabeledPat*)* **|}**

LabeledPat
::= *Label*
| *Label* **=** *Pattern*
| *Label* **is** *Pattern*
| *Label* **isnot** *Pattern*

12.3 Pattern matching semantics

$$\frac{}{v \vdash x \Longrightarrow \{x \mapsto v\}}$$

$$\frac{}{v \vdash \_ \Longrightarrow \{\}}$$

$$\frac{v = \text{ValueOf}(\mathit{lit})}{v \vdash \mathit{lit} \Longrightarrow \{\}}$$

$$\frac{v \neq \text{ValueOf}(\mathit{lit})}{v \vdash \mathit{lit} \Longrightarrow \textbf{FAIL}}$$

$$\frac{v \vdash p_1 \Longrightarrow \Sigma}{v \vdash p_1 \,|\, p_2 \Longrightarrow \Sigma}$$

$$\frac{v \vdash p_1 \Longrightarrow \textbf{FAIL} \quad v \vdash p_2 \Longrightarrow \Sigma}{v \vdash p_1 \,|\, p_2 \Longrightarrow \Sigma}$$

$$\frac{v \vdash p_1 \Longrightarrow \textbf{FAIL} \quad v \vdash p_2 \Longrightarrow \textbf{FAIL}}{v \vdash p_1 \,|\, p_2 \Longrightarrow \textbf{FAIL}}$$

$$\frac{v \vdash p \Longrightarrow \Sigma}{v \vdash x \texttt{ is } p \Longrightarrow \Sigma \pm \{x \mapsto v\}}$$

$$\frac{v \vdash p \Longrightarrow \mathbf{FAIL}}{v \vdash x \texttt{ is } p \Longrightarrow \mathbf{FAIL}}$$

$$\frac{v \vdash p \Longrightarrow \{\}}{v \vdash x \texttt{ isnot } p \Longrightarrow \mathbf{FAIL}}$$

$$\frac{v \vdash p \Longrightarrow \mathbf{FAIL}}{v \vdash x \texttt{ isnot } p \Longrightarrow \{x \mapsto v\}}$$

$$\frac{v_1 \vdash p_1 \Longrightarrow \Sigma_1 \quad \cdots \quad v_n \vdash p_n \Longrightarrow \Sigma_n}{(v_1, \ldots, v_n) \vdash (p_1, \ldots, p_n) \Longrightarrow \Sigma_1 \cup \cdots \cup \Sigma_n}$$

12.4 Pattern expressions

The definition of constants requires a syntactically restricted class of patterns called *pattern expressions*.[2] A pattern expression is a pattern that does not contain or patterns, **`is`** or **`isnot`** patterns, or wild cards.

[2] We use this terminology because the syntax is essentially the intersection of patterns and expressions [AR92].

Bibliography

[AC96] Abadi, M. and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, NY, 1996.

[AR92] Aitken, W. E. and J. H. Reppy. Abstract value constructors: Symbolic constants for standard ml. *Technical Report TR 92-1290*, Department of Computer Science, Cornell University, June 1992. A shorter version appears in the proceedings of the "ACM SIGPLAN Workshop on ML and its Applications," 1992.

[BSv95] Bruce, K., A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings of the Ninth European Conference on Object-Oriented Programming*, vol. 952 of *Lecture Notes in Computer Science*, New York, NY, 1995. Springer-Verlag, pp. 26–51.

[FM95] Fisher, K. and J. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, **1**(3), 1995, pp. 189–220. Preliminary version appeared in TACS '94 proceedings.

[IEE85] IEEE Computer Society, New York, NY. *IEEE Standard for Binary Floating-Point Arithmetic (IEEE Std 754-1985)*, 1985.

[Pro95] Programming Methodology Group, MIT LCS, Cambridge, MA. *Theta Reference Manual (Preliminary Version)*, February 1995. Available from `http://www.pmg.lcs.mit.edu/Theta.html`.

Part III

Appendices

Appendix A

Collected MOBY syntax

A.1 Identifiers

A.1.1 Identifier classes

MOBY has a number of distinct classes of identifiers. In most cases, these are distinguished by context, but in some cases a capitalization convention is used to distinguish between classes.

Class	**Case**	**Description**
ModuleId	Upper	Module IDs
SigId	Upper	Signature IDs
TypeId	Upper	Type and type constructor IDs
TypeVar	Lower	Type variables
ClassId	Upper	Class IDs
Label	Lower	Object and record labels
MakerId	Lower	Maker IDs
DataCon	Upper	Data-constructor IDs
ValueId	Lower	Value IDs

A.1.2 Reserved symbols and keywords

A.2 Collected syntax

CompilationUnit
  ::= *SignatureDecl*
  | *ModuleDecl*
  | *ParamModuleDecl*

SignatureDecl
  ::= **`signature`** *SigId* **`{`** *Specification** **`}`**
  $\mid$ **`signature`** *SigId* **`=`** *SigId* (**`with`** **`{`** *TypeReveal*$^{+}$ **`}`**)^{opt}

Specification
  ::= **`include`** *SigId*
  $\mid$ *ModuleSpec*
  $\mid$ *TypeSpec*
  $\mid$ *ClassSpec*
  $\mid$ *ConstSpec*
  $\mid$ *ValueSpec*

ModuleSpec
  ::= **`module`** *ModuleId* **`:`** *Signature*

Signature
  ::= **`{`** *Specification** **`}`**
  $\mid$ *SigId* (**`with`** **`{`** *TypeReveal*$^{+}$ **`}`**)^{opt}

TypeSpec
  ::= **`type`** *TypeId* *TypeParams*^{opt}
  $\mid$ *TypeReveal*
  $\mid$ *DataTypeDecl*
  $\mid$ *EnumTypeDecl*
  $\mid$ *TagTypeDecl*
  $\mid$ *ObjectTypeDecl*

TypeReveal
  ::= **`type`** *TypeId* *TypeParams*^{opt} **`=`** *Type*
  $\mid$ **`type`** *TypeId* *TypeParams*^{opt} **`<:`** *Type*

TypeParams
  ::= **`(`** *TypeVar* (**`,`** *TypeVar*)* **`)`**

ClassSpec
  ::= **`class`** *ClassId* *TypeParams*^{opt} **`:`** *ClassInterface*

ClassInterface
  ::= **`{`** *InheritsSpec*^{opt} *ImplementsSpec*^{opt} *MemberSpec** **`}`**

InheritsSpec
  ::= **`inherits`** *NamedClass*

ImplementsSpec
  ::= **`implements`** *NamedType* (**`,`** *NamedType*)*

MemberSpec
  ::= **`public`**^{opt} **`field`** *Label* **`:`** *ExtendedType*
  | **`public`**^{opt} *MethodSpec*
  | **`public`**^{opt} **`maker`** *MakerId* **`of`** *Types*

MethodSpec
  ::= **`abstract`** **`method`** *Label* **`:`** *TypeScheme*
  | **`method`** *Label* **`:`** *TypeScheme*
  | **`final`** **`method`** *Label* **`:`** *TypeScheme*

ConstSpec
  ::= **`const`** *DataCon* **`:`** *TypeParams*^{opt} *Type* (**`of`** *Types*)^{opt}
  | **`deconst`** *DataCon* **`:`** *TypeParams*^{opt} *Type* (**`of`** *Types*)^{opt}

ValueSpec
  ::= **`val`** *ValueId* **`:`** *TypeScheme*
  | **`val`** *DataCon* **`:`** *TypeScheme*

ModuleDecl
  ::= **`module`** *ModuleId* (**`:`** *Signature*)^{opt} *ModuleBody*
  | **`module`** *ModuleId* (**`:`** *Signature*)^{opt} **`=`** *ModuleDef*

ModuleBody
  ::= **`{`** *Declaration** **`}`**

ModuleDef
  ::= *Path*^{opt} *ModuleId*
  | *ModuleId* **`(`** (*ModuleExp* (**`,`** *ModuleExp*)*)^{opt} **`)`**

ModuleExp
  ::= *ModuleBody*
  | *ModuleDef*

ParamModuleDecl
  ::= **`module`** *ModuleId* **`(`** *ModuleParams*^{opt} **`)`** (**`:`** *Signature*)^{opt} *ModuleBody*
  | **`module`** *ModuleId* **`(`** *ModuleParams*^{opt} **`)`** (**`:`** *Signature*)^{opt} **`=`** *ModuleDef*

ModuleParams
  ::= *ModuleId* **`:`** *Signature* (**`,`** *ModuleId* **`:`** *Signature*)*

Declaration
 ::= **`local`**^{opt} *ModuleDecl*
 | **`local`**^{opt} *TypeDecl*
 | **`local`**^{opt} *ClassDecl*
 | **`local`**^{opt} *ConstDecl*
 | **`local`**^{opt} *FunDecl*
 | **`local`**^{opt} *ValueDecl*

TypeDecl
 ::= *TypeNameDecl*
 | *DataTypeDecl*
 | *EnumTypeDecl*
 | *TagTypeDecl*
 | *ObjectTypeDecl*

TypeNameDecl
 ::= **`type`** *TypeId TypeParams*^{opt} **`=`** *Type*

DataTypeDecl
 ::= **`datatype`** *TypeId TypeParams*^{opt} *DataTypeDef*
 | **`datatype`** *TypeId TypeParams*^{opt} **`=`** *NamedType*

DataTypeDef
 ::= **`{`** *DataConDef* (**`,`** *DataConDef*)* **`}`**

DataConDef
 ::= *DataCon* (**`of`** *Types*)^{opt}

EnumTypeDecl
 ::= **`enumtype`** *TypeId EnumTypeDef*
 | **`enumtype`** *TypeId* **`=`** *Path*^{opt} *TypeId*

EnumTypeDef
 ::= **`{`** *DataCon* (**`,`** *DataCon*)* **`}`**

TagTypeDecl
 ::= **`tagtype`** *TypeId TypeParams*^{opt} (**`of`** *Types*)^{opt} (**`extends`** *NamedType*)^{opt}
 | **`tagtype`** *TypeId TypeParams*^{opt} **`=`** *NamedType*

ObjectTypeDecl
 ::= **`objtype`** *TypeId TypeParams*^{opt} *ObjectMembers*
 | **`objtype`** *TypeId TypeParams*^{opt} **`=`** *NamedType*

ObjectMembers
 ::= **`{`** *ObjectMember** **`}`**

ObjectMember
  ::= **`extends`** *NamedType*
   | **`field`** *Label* **`:`** *ExtendedType*
   | **`method`** *Label* **`:`** *TypeScheme*

TypeScheme
  ::= *BoundTypeVars*^{opt} *Type*

BoundTypeVars
  ::= **`[`** *BoundTypeVar* (**`,`** *BoundTypeVar*)* **`]`**

BoundTypeVar
  ::= *TypeVar* (**`<:`** *Type*)^{opt}

Types
  ::= *TypeTuple*
   | *Type*

TypeTuple
  ::= **`(`** (*Type* (**`,`** *Type*)$^+$)^{opt} **`)`**

Type
  ::= *FunType*
   | *AtomicType*

FunType
  ::= *AtomicTypes* **`->`** *FunType*
   | *AtomicTypes* **`->`** *AtomicTypes*

AtomicTypes
  ::= *TypeTuple*
   | *AtomicType*

AtomicType
  ::= *TypeVar*
   | *NamedType*
   | **`$`** *TypeTuple*

NamedType
  ::= *Path*^{opt} *TypeId* *TypeArgs*^{opt}
   | **`typeof`** **`(`** *NamedClass* **`)`**
   | **`#`** *NamedClass*

TypeArgs
  ::= **`(`** *Type* (**`,`** *Type*)* **`)`**

ExtendedType
  ::= *TypeScheme*
  | **var** *Type*

ClassDecl
  ::= **class** *ClassId* (**:** *ClassInterface*)^{opt} *ClassDef*
  | **class** *ClassId* (**:** *ClassInterface*)^{opt} **=** *NamedClass*

ClassDef
  ::= **{** (**inherits** *NamedClass*)^{opt} *MemberDecl** *InitiallyClause*^{opt} **}**

NamedClass
  ::= *Path*^{opt} *ClassId* *TypeArgs*^{opt}

MemberDecl
  ::= **public**^{opt} *FieldDecl*
  | **public**^{opt} *MethodDecl*
  | **public**^{opt} *MakerDecl*

FieldDecl
  ::= **field** *Label* **:** *ExtendedType*

MethodDecl
  ::= **abstract** **method** *Label* **:** *TypeScheme*
  | **final**^{opt} **override**^{opt} **method** *Label* *FunDef*

MakerDecl
  ::= **maker** *MakerId* *Params* *MakerBlock*

MakerBlock
  ::= **{** *MakerStmt* (**;** *MakerStmt*)* **}**
  | **{ }**

MakerStmt
  ::= **super** *MakerId* *Expression*
  | **field** *Label* **=** *Expression*
  | *Statement*

InitiallyClause
  ::= **initially** *Expression*

ConstDecl
  ::= **const** *DataCon* **:** *TypeScheme* **=** *Pattern*
  | **const** *DataCon* **:** *TypeParams*^{opt} *Type* **of** *Params* **=** *Pattern*

FunDecl
  ::= **fun** *ValueId* *FunDef*

FunDef
$::=$ *BoundTypeVars*^{opt} *Params*$^{+}$ **`->`** *Type Block*
$|$ **`:`** *TypeScheme MatchCase*

Params
$::=$ **`(`** $($*Param* $($**`,`** *Param*$)^{*})^{opt}$ **`)`**

Param
$::=$ *ValueId* **`:`** *Type*

ValueDecl
$::=$ **`val`** *BoundTypeVars*^{opt} *Params* **`=`** *Expression*
$|$ **`val`** *BoundTypeVars*^{opt} *Param* **`=`** *Expression*

Block
$::=$ **`{`** $($*Statement* **`;`**$)^{*}$ *Expression* **`}`**
$|$ **`{ }`**

Statement
$::=$ *Expression*
$|$ *Binding*

Binding
$::=$ **`let`** *Patterns* **`=`** *Expression*
$|$ **`fun`** *ValueId FunDef* $($**`and`** *ValueId FunDef*$)^{*}$

Expression
$::=$ **`if`** *Expression* **`then`** *Expression* **`else`** *Expression*
$|$ *TryExpression*
$|$ **`raise`** *Expression*
$|$ **`spawn`** *Expression*
$|$ **`sync`** *Expression*
$|$ **`event`** *Expression*
$|$ *AssignmentExpr*

TryExpression
$::=$ **`try`** *Expression* **`except`** *MatchCase*
$|$ **`try`** *Expression* **`finally`** *Expression*

AssignmentExpr
$::=$ *ConditionalOrExpr* **`:=`** *ConditionalOrExpr*
$|$ *ConditionalOrExpr*

ConditionalOrExpr
$::=$ *ConditionalOrExpr* **`||`** *ConditionalAndExpr*
$|$ *ConditionalAndExpr*

ConditionalAndExpr
 ::= *ConditionalAndExpr* **&&** *EqualityExpr*
 | *EqualityExpr*

EqualityExpr
 ::= *EqualityExpr EqualityOp RelationalExpr*
 | *RelationalExpr*

EqualityOp
 ::= **==** | **!=**

RelationalExpr
 ::= *RelationalExpr RelationalOp ListExpr*
 | *ListExpr*

RelationalOp
 ::= **<** | **<=** | **>=** | **>**

ListExpr
 ::= *ShiftExpr ListOp ListExpr*
 | *ShiftExpr*

ListOp
 ::= **@** | **::**

ShiftExpr
 ::= *ShiftExpr ShiftOp BitwiseOrExpr*
 | *BitwiseOrExpr*

ShiftOp
 ::= **<<** | **>>** | **>>>**

BitwiseOrExpr
 ::= *BitwiseOrExpr* **|** *BitwiseAndExpr*
 | *BitwiseOrExpr* **\/** *BitwiseAndExpr*
 | *BitwiseAndExpr*

BitwiseAndExpr
 ::= *BitwiseAndExpr* **&** *AdditiveExpr*
 | *BitwiseAndExpr* **/** *AdditiveExpr*
 | *AdditiveExpr*

AdditiveExpr
 ::= *AdditiveExpr AdditiveOp MultiplicativeExpr*
 | *MultiplicativeExpr*

AdditiveOp
  ::= **+** | **-**

MultiplicativeExpr
  ::= *MultiplicativeExpr MultiplicativeOp Expr*
  | *ExponentialExpr*

MultiplicativeOp
  ::= ***** | **/** | **%**

ExponentialExpr
  ::= *ExponentialExpr* ****** *CompositionExpr*
  | *CompositionExpr*

CompositionExpr
  ::= *PrefixExpr* **^** *CompositionExpr*
  | *PrefixExpr*

PrefixExpr
  ::= *PrefixOp PrefixExpr*
  | *ApplicationExpr*

PrefixOp
  ::= ***** | **&** | **!** | **~** | **-**

ApplicationExpr
  ::= *ApplicationExpr PostfixExpr*
  | *PostfixExpr*

PostfixExpr
  ::= *PostfixExpr* **.** *Label*
  | *PostfixExpr* **[** *Expression* **]**
  | *AtomicExpr*

AtomicExpr

::= **fn** *FunDef*
| **case** *Expression* **of** *MatchCase*
| **(** (*Expression* (**,** *Expression*)*)^{opt} **)**
| **(** *Expression* **:** *Type* **)**
| **(** *Expression* **is** *Pattern* **)**
| **(** *Expression* **isnot** *Pattern* **)**
| *ChoiceEvent*
| *Block*
| **new** *Path*^{opt} *MakerId*
| **self**
| **super .** *Label*
| **nack_event**
| **rdy_event**
| **(** *Operator* **)**
| *Path*^{opt} *ValueId*
| *DataConstructor*
| **#** *DataConstructor*
| **?** *DataConstructor*
| *Literal*

DataConstructor

::= *Path*^{opt} *DataCon*
| **$**

ChoiceEvent

::= **{|** *WrappedEvent* (**,** *WrappedEvent*)* **|}**

WrappedEvent

::= *Expression*
| *Expression* **** *Pattern* **=>** *Expression*
| *Expression* **** *MatchCase*

MatchCase

::= **{** *MatchRule* (**,** *MatchRule*)* **}**

MatchRule

::= *Patterns* (**when** *Expression*)^{opt} **=>** *Expression*

Patterns

::= *PatternTuple*
| *Pattern*

PatternTuple

::= **(** *Pattern* (**,** *Pattern*)$^+$ **)**

Pattern
  ::= *IsPattern* **`:`** *Type*
  | *IsPattern*

IsPattern
  ::= *ValueId* **`is`** *OrPattern*
  | *ValueId* **`isnot`** *OrPattern*
  | **`_ isnot`** *OrPattern*
  | *OrPattern*

OrPattern
  ::= *OrPattern* (**`|`** *ConsPattern*)$^{+}$
  | *ConsPattern*

ConsPattern
  ::= *ApplyPattern* **`::`** *ConsPattern*
  | *ApplyPattern*

ApplyPattern
  ::= *Path*^{opt} *DataCon PatternTuple*
  | *Path*^{opt} *DataCon AtomicPattern*
  | *AtomicPattern*

AtomicPattern
  ::= **`(`** *Pattern* **`)`**
  | **`_`**
  | *ValueId*
  | *DataConstructor*
  | *Literal*
  | **`-`** *NumericLiteral*

Literal
  ::= **`True`**
  | **`False`**
  | *CharLit*
  | *StringLit*
  | *NumericLiteral*

NumericLiteral
  ::= *IntegerLiteral*
  | *FloatLiteral*

Path
  ::= (*ModuleId* **`.`**)$^{+}$

Operator
::= **`==`** | **`!=`** | **`<`** | **`<=`** | **`=>`** | **`>`** | **`@`** | **`::`** | **`<<`** | **`>>`** | **`>>>`** | **`|`** | **`\/`** | **`&`** | **`/\`** | **`+`** | **`-`**

$| \; \texttt{*} \; | \; \texttt{/} \; | \; \texttt{\%} \; | \; \texttt{**} \; | \; \texttt{\^{}} \; | \; \texttt{!} \; | \; \texttt{\~{}}$

Appendix B

The MOBY Basis

B.1 Pervasives

The pervasive definitions are those modules, types, and values that are available to an MOBY program without explicit importing.

B.1.1 Pervasive types

B.1.2 Overloaded operators

B.1.3 Pervasive functions

B.2 Standard modules

```
val ignore : [t] t -> ()

val ! : Bool -> Bool
val == : (Bool, Bool) -> Bool   // overloaded
val != : (Bool, Bool) -> Bool   // overloaded
val < : (Bool, Bool) -> Bool    // overloaded
val <= : (Bool, Bool) -> Bool   // overloaded
val > : (Bool, Bool) -> Bool    // overloaded
val >= : (Bool, Bool) -> Bool   // overloaded

val str : Char -> String
val ord : Char -> Int
val chr : Int -> Char
val succ : Char -> Char          // overloaded
val pred : Char -> Char          // overloaded
val == : (Char, Char) -> Bool   // overloaded
val != : (Char, Char) -> Bool   // overloaded
val < : (Char, Char) -> Bool    // overloaded
val <= : (Char, Char) -> Bool   // overloaded
val > : (Char, Char) -> Bool    // overloaded
val >= : (Char, Char) -> Bool   // overloaded

val + : (String, String) -> String  // overloaded
val concat : List(String) -> String
val length : String -> Int           // overloaded
val == : (String, String) -> Bool   // overloaded
val != : (String, String) -> Bool   // overloaded
val < : (String, String) -> Bool    // overloaded
val <= : (String, String) -> Bool   // overloaded
val > : (String, String) -> Bool    // overloaded
val >= : (String, String) -> Bool   // overloaded

val require : [t] Option(t) -> t
val optional : [t] (Option(t), t) -> t
```

Figure B.1: Pervasive functions

```
val << : (Int, Int) -> Int         // overloaded
val >> : (Int, Int) -> Int         // overloaded
val >>> : (Int, Int) -> Int        // overloaded
val | : (Int, Int) -> Int          // overloaded
val & : (Int, Int) -> Int          // overloaded
val + : (Int, Int) -> Int          // overloaded
val - : (Int, Int) -> Int          // overloaded
val * : (Int, Int) -> Int          // overloaded
val / : (Int, Int) -> Int          // overloaded
val % : (Int, Int) -> Int          // overloaded
val - : Int -> Int                 // overloaded
val ~ : Int -> Int                 // overloaded
val == : (Int, Int) -> Bool        // overloaded
val != : (Int, Int) -> Bool        // overloaded
val < : (Int, Int) -> Bool         // overloaded
val <= : (Int, Int) -> Bool        // overloaded
val > : (Int, Int) -> Bool         // overloaded
val >= : (Int, Int) -> Bool        // overloaded
val abs : Int -> Int               // overloaded
val min : (Int, Int) -> Int        // overloaded
val max : (Int, Int) -> Int        // overloaded
val isNeg : Int -> Bool            // overloaded
val sign : Int -> Int              // overloaded
```

Figure B.2: Pervasive integer operations

```
val + : (Flt, Flt) -> Flt       // overloaded
val - : (Flt, Flt) -> Flt       // overloaded
val * : (Flt, Flt) -> Flt       // overloaded
val / : (Flt, Flt) -> Flt       // overloaded
val % : (Flt, Flt) -> Flt       // overloaded
val - : Flt -> Flt               // overloaded
val ~ : Flt -> Flt               // overloaded
val == : (Flt, Flt) -> Bool     // overloaded
val != : (Flt, Flt) -> Bool     // overloaded
val < : (Flt, Flt) -> Bool      // overloaded
val <= : (Flt, Flt) -> Bool     // overloaded
val > : (Flt, Flt) -> Bool      // overloaded
val >= : (Flt, Flt) -> Bool     // overloaded
val abs : Flt -> Flt             // overloaded
val min : (Flt, Flt) -> Flt     // overloaded
val max : (Flt, Flt) -> Flt     // overloaded
val isNeg : Flt -> Bool          // overloaded
val sign : Flt -> Int             // overloaded
val isNAN : Flt -> Bool
val isInf : Flt -> Bool
val ordered : (Flt, Flt) -> Bool
val copySign : (Flt, Flt) -> Flt
val scalb : (Flt, Int) -> Flt
val logb : Flt -> Int
val nextAfter : (Flt, Flt) -> Flt
```

Figure B.3: Pervasive floating-point operations