In: "Combinators and Functional Programming Languages"
G.Cousineau, P-L.Curien and B.Robinet Editors,
Lecture Notes in Computer Science n.242, Springer-Verlag,
1986

Amber

Luca Cardelli[1]

AT&T Bell Laboratories, Murray Hill, NJ 07974

Introduction

The Amber language embeds many recent ideas in programming language design, and tries to introduce all the features in their minimal, essential, form. One of its main goals is to safely blend static typing with the dynamic requirements of a system programming language. For this purpose, multiple inheritance and persistent objects are integrated in a strongly typed language. Other features include graphics, higher-order functions, modules and concurrency.

Amber is a spin-off of the ML programming language [Milner 84]. The ML language is now being standardized, and as such is not very suitable for experimentation. Amber is intended as a tool for trying out new ideas in language implementation, language design, and language environments, while being deeply influenced by the ML experience.

As a programming language, Amber was defined to experiment with a new style of polymorphism [Cardelli 84b] which, unlike the ML-style parametric polymorphism [Milner 78], is based on a notion of type inclusion, and can be used to interpret many programming concepts found in object-oriented languages [Goldberg Robson 83]. In this view, the main features of functional and object-oriented languages can be naturally integrated, and the combination of higher-order functions and multiple inheritance can be strongly typed. Some typechecking anomalies are still present in Amber, and current research is aimed at solving them and integrating inclusion polymorphism with parametric polymorphism.

Type inclusion also plays an important role in modularization. Amber programs can be partitioned into modules and separately compiled. Modules have import-export lists for types and values. When a type is imported, its actual definition is not accessible: this is a form of data abstraction realized through the module mechanism, and implies that modules can be compiled in any order. It is possible to specify that two imported types, although unknown, are one a subtype of the other, so that inheritance can be made to work across module boundaries.

At the programming system level, the implementation is heavily based on the ability to export and import arbitrary values to/from persistent storage. This feature is provided at the lowest level, and guarantees the preservation of any circularity or sharing present in the

[1] Current addresss: DEC SRC, 130 Lytton Ave, Palo Alto, CA 94301.

data structures. As the language is statically scoped and the implementation of functions is based on closures, exporting a function will automatically export everything the function needs to run when it is imported. Complex data structures, like trees or fonts, can be exported and imported without having to write ad-hoc routines to unparse and reparse them. Separate compilation is also based on persistence, and involves importing and exporting module data structures. Moreover, persistent data are strongly typed, so that a type error is generated (at run time) if one happens to import the wrong kind of object. A limitation is that one can only import/export whole objects; hopefully a scheme will be found to store partial objects, along the lines of [Atkinson Bailey Chisholm Cockshott Morrison 83].

At the implementation level, the Amber system is organized in three layers. At the bottom there is a kernel which provides input-output, graphics, heap management and (in the future) process scheduling. Heap management consists of data allocation, collection and persistent storage. This level is largely independent of any particular language, and only deals with four basic data formats: immediates (such as booleans, integers and pointers), strings of bytes, arrays of immediates and bitmaps.

The second layer is an abstract machine, similar to the one in [Cardelli 84a], which provides an instruction set and data types (such as records, functional closures, etc.) based on the underlying data formats. Programs are at the moment encoded as byte streams to be interpreted, but nothing would prevent them from being compiled to machine code (except for portability considerations). This level is mildly language-dependent, and can be used to implement different languages in the same generic class with few changes.

The third layer is the compiled Amber compiler, and its source which is written in Amber. The compiler has already gone through dozens of generations, each time recompiling a new version of itself. The system has been ported to three different machines and currently runs on an Apple Macintosh.

Features

This section summarizes the main language features, before going into more detail in the following sections.

Amber is interactive. Every time a phrase is entered at the top level, the phrase is analyzed, compiled and executed. A phrase can be as simple as evaluating 1+2, or as complicated as compiling and linking a program module.

Amber is statically scoped. Every variable is bound to the nearest enclosing defining occurrence of that variable. Functions referring to global variables can be treated as self-contained objects which can be passed to other functions, returned from other functions, stored into data structures and exported to persistent storage. Static scoping is also respected at the top level.

Amber is a safe language. Static and dynamic typechecking ensures the consistent use of data and operations. Typechecking is mostly static, with some provisions for dynamic checks, as described below.

Amber has safe dynamic types. Any value in the language can be bundled into a dynamic value, which carries its full type with it. Dynamic values can later be coerced to some specific type, stripping them of the dynamic type information. The coercion process

only succeeds if the specified type matches the dynamic type.

Amber has persistence. Any dynamic value (and hence any value, data or program) can be exported to persistent storage, and later imported and coerced, probably during a different program session. This process preserves sharing and circularities within the exported objects.

Amber has modules. As a special case of persistent objects, program modules can be separately compiled and later linked into the system. The linking process performs typechecking across module boundaries and preserves the sharing of data and programs in common ancestor modules. A module is a set of declarations with an import list (declarations imported from other modules) and an export list (declarations which can be imported by other modules).

Amber has multiple inheritance. The type system is based on an implicit inheritance relation between types, so that some fundamental aspects of object-oriented languages can be modeled within a strongly typed system.

Amber has signals. Primitive operations produce signals on error conditions (like divide by zero). These signals can be trapped and recovery actions can be taken. User-defined signals can also be generated, and they are treated in the same way as primitive signals.

Amber has concurrency. The concurrency model is synchronous, handshake communication on typed channels between pairs of processes. Channels are denotable values and can transmit any value (including other channels). Any number of processes can input or output on the same channel. Processes are not denotable values, but can be dynamically created and linked to other processes. Process scheduling is non-preemptive and context switches happen after every communication.

Amber has graphics. A set of primitives gives access to a bitmap display and pointing device, allowing the construction of editors and window systems within the language. Graphics is viewed as an essential part of the language.

Amber is written in itself. A one-pass compiler produces a portable intermediate abstract machine code, which can then be interpreted or assembled, depending on the implementation. The abstract machine, called Chaos, supports graphics, persistence, dynamic data structure allocation and garbage collection. It can be defined without any reference to the language itself and provides facilities for building self-compiling compilers.

Data Types

There is a basic distinction between single values and multiple values. A single value is any ordinary value which can be manipulated, like a number, a record or a function. Multiple values are formed by tupling: `(3,true)` is a tuple (pair) of `3` and `true`, and it has type `(Int,Bool)`. A tuple of a single value, like `(3)`, is considered as a single value, and is equivalent to `3`. A tuple of tuples like `(3,(true,"foo"),5)` is equivalent to its flattening: `(3,true,"foo",5)`. A null tuple `()` is *no* value, and it is absorbed when nested in other tuples: `(3,(),4)` is the same as `(3,4)`.

Similarly, there is a distinction between single types (the types of single values) and multiple types (the types of tuples). Multiple values and types can only be used in restricted contexts, as described in the sequel.

Ground Types

- There is a trivial type called Unit, and the only value of this type is the constant unity.
- The boolean type is Bool, with constants true and false.
- The integer type Int has constants 0, ~1, 1, ~2, 2, ...

  Moreover, a quoted character (e.g. 'a) denotes its ascii integer value.
- Strings of ascii characters have type String with constants "", "abc123", "this is a string". The escape character is \, which can be used to introduce " and \ in strings.
- Bitmaps have type Bitmap, and there are no constants of this type.
- Regions (related to bitmaps) have type Region, and there is a constant nullregion.

Tuples

A tuple (3,true) has a tuple type (Int,Bool). The null tuple () has type (). Expressions can be multiple-valued, and functions can take multiple arguments (i.e. tuples of arguments) and return multiple values (i.e. tuples of results). However, program variables can only denote single (i.e. non-tuple) values, and data structures can only be composed of single values. The reason for these restrictions is that tuple creations and manipulations do not involve dynamic storage allocation: tuples are always built and manipulated on the stack.

Records

Records are unordered, labeled sets of values: {a = 3, b = true} has type {a : Int, b : Bool}, where a and b are labels. The above record has also type {a : Int}, in the sense that it has an a field which is an Int, and we may not care about other fields. Individual fields of a record can be declared to be updatable by using the symbol =>, instead of =, in records, and :>, instead of :, in record types. Hence {a => 3, b = true} has type {a :> Int, b : Bool}, and the a field can be updated to a different integer.

Variants

Variants are labeled values, and variant types are unordered, labeled sets of types: [a = 3] has type [a : Int]. An object of type [a : Int, b : Bool] is either an a variant with Int contents, or a b variant with Bool contents; in this sense, the variant [a = 3] has also type [a : Int, b : Bool]. Individual cases of variant types can be declared to be updatable by using the symbol =>, instead of =, in variant objects, and :>, instead of :, in variant types. Hence [a => 3] has type [a :> Int, b : Bool] and and it can be updated to a different integer (changing the tag is not allowed).

Arrays

Array types can be formed out of arbitrary single types, for example integer arrays have type Array(Int). Arrays of the same type can have different lengths, and the length is fixed at creation time. Array indices are zero-based and array bounds are dynamically checked. Arrays are updatable.

Functions

Function types can be made out of single or multiple types; Int -> Int is the type of functions from integers to integers, (ex.: fun(x:Int) x+1). The type (Int,Int) -> (Bool,Bool) is the type of functions of two integer arguments and two boolean results, (ex.: fun(x:Int, y:Int) (x=0, y=0)). Function types are always single types.

Dynamic

Any single value can be converted to a dynamic value, which is obtained by tagging the value with its full type. Dynamic values can be dynamically typechecked and coerced back to non-dynamic values. Dynamically typed values have type Dynamic.

Channels

Communication channels can be made out of any other single type, including channels. An integer channel has type Channel(Int).

Operations

Equality

The operator = can be applied to objects of any type, provided that its arguments have compatible types. It tests the equality of objects of type Unit (always true), Bool and Int, and the identity of objects of any other type, i.e. whether they are the same object in store.

Assignment

Assignable variables (described later) can be updated by the operation var a = e, which takes an assignable variable (a) and an expression (e) and returns the null tuple ().

Ground Types

- There are no operations on the Unit type.
- Boolean operations are the conditional (e.g. if true then 3 else 4) and the while (e.g. while true repeat ()) expressions, plus the infix operators \/ (or), /\(and) and the function not. The type of a conditional is the type join of the types of its branches. The type of a while expression is () (the null tuple type), and its body must also have type ().
- Integer operations are (infix) +, -, *, /, %(modulo), <, >, <= and >=.
- Strings can contain characters whose representation is in the range 0..255. The first element of a string has index 0. String operations are:

```
string: (Int, Int) -> String
```

string(size,char) returns a string of given size, all initialized to the same character.

```
length: String -> Int
```

length(string) returns the length of a string.

getascii: (String, Int) -> Int

getascii(string, n) returns the ascii representation of the n-th element of a string.

putascii: (String, Int, Int) -> ()

putascii(string, n, char) updates the n-th element of a string by a new ascii value.

sub: (String, Int, Int) -> String

sub(src, index, size) extracts the substring of src starting from index and of length size.

setsub: (String, Int, String) -> ()

setsub(dst, index, src) affects dst by replacing src for the substring of dst starting at index and of size length(src).

stringblit: (String, Int, Int, String, Int) -> ()

stringblit(src, srcIndex, length, dst, dstIndex) copies a substring of the source, starting from the source index, into the destination, starting from the destination index. If the two strings are the same, the copy order is chosen so that the result is correct even if src and dst overlap.

search: (String, String, Int, Bool) -> Int

search(chars, string, from, direction) searches for the first occurrence (forward if the direction is true, backward if false) of any of the characters in chars in the string string, starting from some position in the string. Returns the index of the occurrence, if found, or signals search if not found.

equal: (String, String) -> Bool

equal(string1,string2) compares the contents of two strings.

- Bitmaps are rectangular regions of pixels with a coordinate system. A pixel can have two values: white pixels are false, and black pixels are true. Operations are:

bitmap: (Int, Int, Int, Int) -> Bitmap

bitmap(x, y, hor, ver) returns a new white bitmap of horizontal size hor, vertical size ver, and top left corner at x,y.

bitmaptile: Bitmap -> (Int, Int, Int, Int)

bitmaptile(bitmap) returns the x, y, hor, ver information of a bitmap.

pixel: (Bitmap, Int, Int) -> Bool

pixel(bitmap, x, y) returns the value of a pixel (in the bitmap coordinates) in a bitmap. If the coordinates are out of bounds, the result is false.

bitblit: (Bitmap, Int, Int, Int, Int, Bitmap, Int, Int, Int, Region) -> ()

bitblit(src, srcX, srcY, hor, ver, dst, dstX, dstY, op, clip) transfers a rectangular region of size hor,ver from the source to the destination bitmap. The region starts at srcX,srcY in the source and is transferred to dstX,dstY in the destination, clipped to the clip region (only points inside the clip region of dst are affected by the operation); the region may fall partially or totally outside the source or the destination: pixels outside the source have value false. The transfer is done by replacing the destination region pixel-by-pixel by a boolean operation of the source and destination, encoded by the op argument. op, in the range 0..3, means respectively: *copy, or, xor, clear*.

texture: (Bitmap, Bitmap, Int, Int, Int, Int, Int, Region) -> ()

texture(pattern, bitmap, x, y, hor, ver, op, clip) fills the rectangle x, y, hor, ver of bitmap (further clipped by the clip region) by replicating the pattern bitmap. The pattern is aligned to the origin (0,0) in bitmap coordinates, so that independently textured regions merge correctly. The op argument works as in bitblit. The size of the pattern is fixed (e.g. 8x8) and implementation-dependent.

line: (Bitmap,Bitmap, Int, Int, Int, Int, Int, Region) -> ()

line(texture, bitmap, fromX, fromY, toX, toY, penHor, penVer, op, clip) draws a line from the point fromX,fromY to the point toX,toY in bitmap coordinates, with pen size penHor,penVer and pattern texture (an 8x8 bitmap), using the operation op, and clipping to the clip region. op is the same as in the bitblit operation, with the source always taken to be true.

screen: () -> Bitmap

screen() returns the screen bitmap. Changes to this bitmap are immediately reflected on the screen.

cursor: () -> (Int,Int)

cursor() returns the current position of the cursor on the screen bitmap, in screen

coordinates.

```
setcursor: (Int,Int) -> ()
```

setcursor(X,Y) moves the cursor to a given position on the screen.

```
button: () -> Bool
```

button() returns the current state of the button (only one button is assumed to exist) on the pointing device; true if pressed, false if released.

```
cursoricon: () -> Bitmap
```

cursoricon() returns a bitmap which contains the current cursor icon. Changes to the contents of this bitmap will affect the cursor icon on the screen.

```
cursortip: (Int,Int) -> (Int,Int)
```

cursortip(x,y) changes the *tip* (in cursor icon coordinates) of the cursor, i.e. the point which determines which pixel on the screen is being pointed at. Returns the old tip.

```
makeregion: Bitmap -> Region
```

makeregion(bitmap) turns a bitmap into a region to be used for clipping in bitblit, texture and line. Blank bitmaps are turned into nullregion. A region is an arbitrary set of points in a coordinate system. Unlike bitmaps, regions do not have boundaries.

```
offsetregion: (Region,Int,Int) -> Region
```

offsetregion(region,dx,dy) translates a region with respect to its coordinate system and returns the new translated region.

```
insetregion: (Region,Int,Int) -> Region
```

insetregion(region,dh,dv) shrinks (or expands, for negative dx and dy) a region, by moving all its points inward (or outward) according to the given displacement.

```
unionregion: (Region, Region) -> Region
sectregion: (Region, Region) -> Region
diffregion: (Region, Region) -> Region
xorregion: (Region, Region) -> Region
```

return the union, intersection, difference and xor of two regions.

pointinregion: (Int,Int,Region) -> Bool

pointinregion(x,y,region) tests whether a point belongs to a region.

tilesectregion: (Int,Int,Int,Int,Region) -> Bool

tilesectregion(x,y,hor,ver,region) tests whether a tile intersects a region.

equalregion: (Region,Region) -> Bool

equalregion(region1,region2) tests whether two regions are the same set of points in the same coordinate system, except that two empty regions are always equal.

Tuples

There are no operations on tuples (multiple values). Tuples can be built by the syntax $(e_1, \ldots, e_n)$ and can be taken apart in function arguments and declarations.

Records

Record fields can be extracted by the notation r.a, which selects the a component of a record r. If a record field is updatable, the assignment set r.a = e (which returns the null tuple) can be used to update it.

Variants

Variants can be inspected by the case statement: case v $[a_1 = x_1]$ e_1 ... $[a_n = x_n]$ e_n otherwise e. The contents of the variant v are bound to x_i if the variant tag is a_i, and e_i is evaluated; otherwise e is evaluated. The scope of each x_i is the respective e_i. If one does not care about binding some of the x_i variables, they can be omitted together with the respective = signs. If a variant case is updatable, the assignment set v[a] = e (which returns the null tuple) can be used to update it.

Arrays

Arrays can be created by array(12,"a"), which builds an array of 12 elements, all "a". The size of an array is given by arraysize(a). Arrays are indexed by index(a,i) and updated by update(a,i,v) (which returns the null tuple); the first element of an array has index 0.

Functions

The only operation on functions is function application: f(a). The function is evaluated first, then the argument, and then the application is performed. The argument can be a tuple, e.g. f(a,b,c).

Dynamics

Dynamic values are created by the expression dynamic e, where e can have any single

type. They are coerced to some known type t by coerce e to t; if the dynamic value e does not have type t, a run-time error is produced. Dynamic values can be copied to persistent storage by extern(name,e), where name is a string (a file name) and e has type Dynamic. Persistent values are reimported by intern(name), which returns a dynamic.

Channels

Channels are created by the channel(t) form, where t is any single type, which returns a new channel of type Channel(t). Communication is achieved by the select construct, which attempts communication simultaneously on several channels:

```
select    ch1 ? x => e1
or        ch2 ! e => e2
or ...
```

where ch_1, ch_2 are channel-valued expressions, and e_1 , e_2 are expressions of any type (their types have to match), x is a variable of a type corresponding to ch_1 messages, and e is an expression of type corresponding to ch_2 messages. Question marks (?) are for input communications, where the message is bound to a variable, and exclamation marks (!) are for output communications, where the result of an expression is passed as a message. For channels of type Unit, the variable following ?, and the expression following ! can be omitted.

Only one of the channels of a select is chosen for communication at any given time. When this happens, the corresponding expression (following =>) is evaluated and its value is the value of the whole construct. select may wait indefinitely, until at least one of its channels is ready for communication.

Communication happens by handshake between a process waiting for input and another process waiting for output on the same channel. Any number of processes can be waiting for input or output on the same channel, and a communicating pair is selected according to some scheduling strategy (which may be unfair). Channels are legal values and can be passed on other channels of the appropriate type.

All the output messages of a select are evaluated before any communication attempt. Communications can happen while evaluating the message for an output communication.

Signals

Primitive operations produce signals, instead of values, on error conditions. The name of the signal is usually the same as the name of the operation which generates it. If a signal is not trapped and reaches the top level, a message is printed.

```
3/0                    signals  /

getascii("",0)         signals  getascii
```

Signals can be trapped by the on sig e_1 in e_2 construct, which evaluates e_2 and

normally returns its value, but if the signal sig is produced, then evaluates e_1. If a signal different from sig is produced in e_2, or if a signal is produced in e_1, the signal propagates outside the on construct.

```
on getascii 'a in getascii("",0)      yields      'a

on foo 'a in getascii("",0)           signals     getascii
```

User-defined signals are generated by the signal sig : type construct, where sig (which is not evaluated) is an identifier which is the name of the signal, and type is the type of the signal. Signals are usually generated to avoid returning a value of some type, and the type of a signal is the type of the value they replace.

```
if false then 3 else signal foo : Int      signals    foo

on foo 4 in signal foo : Int               yields     4
```

Signals propagate dynamically, back along the chain of function calls leading to them. Hence a signal trap like on foo x in f(a) will trap any foo signal generated in the execution of f(a), although the definition of f and the signaling of foo may be in a different syntactic scope.

Local Declarations

Local declarations are introduced by the let-do construct. This is a sequence of let-clauses and do-clauses in any order; the last clause must be a do-clause. A let-clause introduces and initializes new variables, which can be used in all the succeeding clauses; a do-clause evaluates an expression for side effects, or to produce a result. Clauses are evaluated in order, and the value and type of the last clause are the value and type of the whole construct.

```
let a = 3
let f = fun () a+1
do f()
```

Declarations and evaluations can involve multiple values:

```
let (a,b) = (3,4)
let f = fun () (a+1,b+1)
do f()
```

Updatable variables can be declared and initialized by the form let var a = e, and they can then be assigned by the form var a = e. assignment takes a variable (previously declared by var), and an expression, and returns the null tuple. In declarations and

assignments the var keyword must be followed by a single variable.

```
let var a = 3
do var a = a + 1;
```

Global Declarations

Global (or top-level) value declarations are introduced by the keyword value, which behaves very much like a let; assignable global variables are declared by value var.

```
value fact =
    fun(n: Int)
        let var result = 1
        let var count = n
        do while count > 0 repeat
            (do var result = result * count
             do var count = count - 1)
        do result;
```

Global type declarations are introduced by the keyword type. Types can only be declared globally.

```
type Point = {x : Int, y : Int};
type (A, B) = (Int, Bool);
type IntTriple = (Int, Int, Int);
```

A type declaration can introduce a new type name (e.g. Point). Or it can introduce a tuple of names (e.g. (A, B)) respectively bound to types in a tuple of types. Or it can bind a type name to a tuple of types (note that the equivalent situation in value and let declarations is illegal).

A type name declared in this way is a simple abbreviation. It is equivalent to its defining type expression and to any other type name equivalent to that expression.

Recursion

Recursive and mutually recursive types are defined by the forms:

```
rec(a) t
rec(a1, ... , an) (t1, ... , tn)
```

This produces a tuple of types, which can be used in type definitions:

```
type IntList =
    rec(List) [nil : Unit, cons : {first : Int, rest : List}]
```

```
type (A,B) =
    rec(A,B) (... , ...)
```

Recursive and mutually recursive objects and functions are obtained by the expressions:

```
rec(a: t) e
rec(a1: t1, ... , an: tn ) (e1, ... , en)
```

where a_i are variables, t_i are types, and e_i are constructors. A constructor is either a record expression, a variant expression, an array expression (array(...)) or a function expression. The result is a tuple of all the mutually recursive objects defined.

```
value fact =
    rec(f: Int -> Int)
        fun(n: Int)
            if n=0 then 1 else n * f(n-1);

value (f,g) =
    rec(f: Int -> Int, g: Int -> Int ) (fun(x:Int) ... , fun(y:Int) ...);

value ones =
    rec(ones: IntList) [cons = {first = 1, rest = ones}];
```

Here is another technique for defining mutually recursive functions. The two functions are independently built as recursive functions, and the second is passed back to the first to achieve mutual recursion (the first is known to the second by the order of definition):

```
value f =
    rec(f: (Int->Int, Int) -> Int)
        fun(g: Int->Int, a:Int) ... g(...) ... f(g,...) ... ;
value g =
    rec(g: Int->Int)
        fun(b: Int) ... g(...) ... f(g,...) ... ;
```

This is useful when f and g are defined in two distinct modules, as modules cannot be mutually recursive.

Inheritance

Typechecking is based on an inclusion relation between types, which is also referred to as type inheritance. Type inclusion is only determined by the structure of type terms, even when the types are recursive. Type definitions and type names are only used as abbreviations.

Every type is included in itself. The most useful non-trivial case of inclusion has to do with record types. A record type T with fields a : Int, b : Bool is considered as the set of all records with an integer value labeled a, a boolean value labeled b , and possibly more fields. A type U with an additional field c : String is included, as a set, in T (as all the records in U are also in T, but there are two-field records which are in T but not in U). Hence a record type with more fields is included in a record type with fewer fields, provided that the labels match and the respective field types are included. An updatable field is included in a corresponding non-updatable field if the corresponding types are included. A non-updatable field is never included in a corresponding updatable field. An updatable field is included in a corresponding updatable field if the corresponding types are equal.

For variant types, a variant type with fewer fields is included in one with more fields, provided that the labels match and the corresponding field types are included. An updatable field is included in a corresponding non-updatable field if the corresponding types are included. A non-updatable field is never included in a corresponding updatable field. A updatable field is included in a corresponding updatable field if the corresponding types are equal.

An array type is included in another if the respective base types are equal.

A channel type is included in another if the respective base types are included.

A tuple type is included in another if the respective components are included.

A function type T -> T' is included in U -> U' if U is included in T and T' is included in U' (note the reversal of inclusion in the domains). In a function application, the type of the argument must be included in the type of the domain of the function.

Finally, a recursive type rec(t) T is included in a recursive type rec(u) U, if assuming t included in u implies T included in U. Inclusions where only one of the types is recursive are covered by the fact that any type T is equivalent to rec(t) T, if t does not occur in T.

Here are the basic types used by the graphics system: points, frames (rectangles with horizontal and vertical dimensions) and tiles (frames in a reference system which determines the position of their top left corners).

```
type Point = {x : Int, y : Int};
type Frame = {hor : Int, ver : Int};
type Tile = {x : Int, y : Int, hor : Int, ver : Int};

value origin = {x = 0, y = 0};
value unitFrame = {hor = 1, ver = 1};
value unitTile = {x = 0, y = 0, hor = 1, ver = 1};
```

According to the inclusion rules, every tile is a point (the type Tile is included in the type Point) and every tile is a frame (Tile is included in Frame). Another way of saying this is that Tile inherits the properties (fields) of Point and Frame, or that Tile is a subtype of both Point and Frame. When a type inherits from several other types it is said to enjoy multiple inheritance.

The type inclusion rules also assert that the argument of a function must have a subtype

of the type of the domain of the function. Hence, all the functions accepting points can also accept tiles; similarly, all functions accepting frames accept tiles:

```
value xCoord = fun (p: Point) p.x;
value horSize = fun (f: Frame) f.hor;

xCoord(unitTile)        yields  0
horSize(unitTile)       yields  1
```

Inheritance also works at higher functional levels. In the following example, a function expecting a function of type Tile -> Int as an argument can be applied to an argument of type Point -> Int, as Point -> Int is included in Tile -> Int:

```
value applyUnitTile = fun (f: Tile -> Int) f(unitTile);

applyUnitTile(xCoord)       yields  1
```

Functions can be embedded in record fields, and recursion can be used to refer to other components of the same record:

```
type ActivePoint =
    rec(ActivePoint)
        {x :> Int, y :> Int, double : () -> ActivePoint};

value makeActivePoint =
    fun (theX: Int, theY: Int)
        rec(self: ActivePoint)
            {   x => theX,
                y => theY,
                double =
                    fun ()
                        do set self.x = 2 * self.x
                        do set self.y = 2 * self.y
                        do self
            }

xCoord(makeActivePoint(1,1).double().double())    yields   4
```

Note how recursive types and recursive objects are used to model the concept of *self* found in object-oriented languages.

Modules

Modules are self-contained program units which can import/export names from/to other

modules. Modules can be compiled separately; the import/export dependencies must form a directed acyclic graph (mutual dependencies are not allowed). Modules can be compiled in any order, independently of their dependencies. They can only be defined at the top level.

When a type is imported from another module, the definition of that type is not accessible in the current module. In order to create and use objects of that type, appropriate operations have to be imported from the other module. The following example shows some of the possible relations between imported types and values.

```
module "A"
    import "B"
        type T, U
        value var b : T, f : T -> U;
    type TtoT = T -> T;
    import "C"
        value c1 : Int, c2 : TtoT;
    export
        type  A
        value g : A -> T, var a : Int;

type A = {t : T, u : U};

value g = fun (x: A) x.t;

value var a = 0;

end;
```

Note how type definitions like `type TtoT = T -> T` can be intermixed with `import` lists. It is also possible to intermix `type "file.typ"`, in which case the type definitions in the file `file.typ` are expanded in place.

An imported type can be declared to be included in some other imported type, even if the definitions of both types are not accessible in the current module. This way inheritance can function across module boundaries. The following (incomplete) example shows how to import points, frames and tiles from different modules, and how to declare their inclusion relations. In the body of the module, tiles can be used as points and as frames.

```
module "Graphics"
    import "Point"  type Point  value ... ;
    import "Frame"  type Frame  value ... ;
    import "Tile"
        type Tile in Point, Frame
        value ... ;
    export ... ;
```

```
...
end;
```

Type inclusion is used in several ways when matching module interfaces: (a) the user of a module can import fewer names than are actually defined in the module; (b) every name can be exported with a type that includes (is less specific than) the real type of that name; (c) every name can be imported with a type that includes the exported type of that name (e.g. a record with three fields can be imported as a record with two fields); and (d) the above mentioned inclusion specifications of imported types must be verified.

When a module definition is evaluated, a structure is stored into persistent storage representing that compile module. To interactively import some of the contents of a compiled module, one uses the form (only at the top level):

```
import "A"
    type A
    value var a : Int;
```

A top-level import invokes the linker to link all the modules necessary to produce the desired values. The linker performs typechecking across module boundaries and evaluates the module bodies (a module definition does not involve any evaluation). For every single import statement, each module is evaluated exactly once, the first time it is imported: succeeding imports of the same module will use the values which have already been evaluated. This caching of imported modules ensures sharing of code and data structures which are imported through different paths. However, every import statement independently re-imports and re-evaluates modules, independently of the modules cached by previous imports.

Concurrency

Processes are created by the process e form, which starts a new process in parallel with the current one, executing the expression e. The current process is not stopped: the result of starting a new process is the null tuple.

The current process is stopped either when executing a stop command, which kills it, or when requesting a communication by select. In the latter case, other waiting processes get a chance to run; the current process is reactivated only if there is a possibility of communication.

Here is a process which duplicates itself indefinitely, but only after receiving an ok from its parent:

```
value rabbit =
    rec(rabbit: Channel(Unit ) -> ())
        fun (ok: Channel(Unit))
            select
                ok ? =>
```

```
            let (one, two) = (channel(Unit), channel(Unit))
            do process rabbit(one)
            do process rabbit(two)
            do select one ! => ()
            do select two ! => ()
            do stop;

value zillions =
    fun ()
        let ok = channel(Unit)
        do process rabbit(ok)
        do select ok ! => ();
```

A different flavor of process is obtained by the form realtime process e. Real-time processes behave very much like ordinary processes, but they are scheduled very frequently (typically 60 times a second) by preempting ordinary processes. This frequent scheduling is represented as a communication on a special channel, called tick, which is reserved for use by real-time processes.

Real-time processes should complete their tasks (i.e. perform a new select) in very short time. Other restrictions may apply to them, for example they should not allocate dynamic storage during their normal activity.

Here is a real-time process handling mouse events, for the benefit of slower (ordinary) processes. It provides, on the transition channel, the latest mouse button transition detected since the last time the channel was used.

```
type Trans = [noTrans : Unit, upTrans : Unit, downTrans : Unit];

value noTrans = [noTrans = unity];
value upTrans = [upTrans = unity];
value downTrans = [downTrans = unity];

value transitionOf =
    fun (old: Bool, new: Bool)
        if old then if new then noTrans else upTrans
        else if new then downTrans else noTrans;

realtime process
    let var lastTrans = noTrans
    let var lastButton = false
    do while true repeat
        select
            transition ! lastTrans  =>
                var lastTrans = noTrans
```

```
or  tick ? =>
        let thisButton = button()
        do var lastTrans = transitionOf(lastButton,thisButton)
        do var lastButton = thisButton;
```

Keyboard interaction is handled by a channel key: Channel(Int,Bool), where the integer is a key number, or an encoding of the keyboard status, and the boolean is true for pressed and false for released.

Inspecting dynamic values

A set of primitives are provided to inspect and manipulate unknown dynamic objects, e.g. ones obtained by intern, and which we do not know how to coerce. One of these primitives returns the type of a dynamic object, as an object of type Type. The type Type is not primitive, and is defined below.

```
type (Type, FunctionType, TaggedTypeList, RecType) =
   rec (Type, FunctionType, TaggedTypeList, RecType)
       ([unit  : Unit,
         bool  : Unit,
         int   : Unit,
         string : Unit,
         bitmap : Unit,
         region : Unit,
         function : FunctionType,
         record  : TaggedTypeList,
         variant : TaggedTypeList,
         array : Type,
         channel : Type,
         dynamic : Unit,
         rec : RecType
        ],

        {dom : Type, cod : Type},

        [nil : Unit,
         cons :
             {tag : String, type : Type,
              updatable : Bool, rest : TaggedTypeList}
        ],

        {printName : String, type : Type}
       );
```

```
type TaggedValue =
    {tag : String, value : Dynamic};

type TaggedValueList =
    rec(TaggedValueList)
        [nil : Unit,
         cons :
             {tag : String, value : Dynamic, rest : TaggedValueList}
        ];
```

The following set of primitives allows one, among other things, to print the type and the value of an unknown dynamic object.

```
typeOf: Dynamic -> Type
```

Returns an encoding of the type of a dynamic object. Objects of type Type are ordinary values, and are only useful as a guide for applying appropriate coercions to dynamic objects. This does not imply any ability to manipulate Amber types as values.

```
exposeFunction: Dynamic -> (Dynamic -> Dynamic)
```

If the argument of exposeFunction is a dynamic object containing a function f: A -> B, then the result is the function fun(x: Dynamic) dynamic f (coerce x to A).

```
exposeRecord: Dynamic -> TaggedValueList
```

If the argument of exposeRecord is a dynamic object containing a record {a = e}, or {a => e}, then the result is a list [cons = {tag = "a", value = dynamic e, rest = [nil = unity]}] (similarly for records with more fields). Information about which fields are assignable can be extracted by typeOf.

```
exposeVariant: Dynamic -> TaggedValue
```

If the argument of exposeVariant is a dynamic object containing a variant [a = e], then the result is {tag = "a", value = dynamic e}

```
exposeArray: Dynamic -> Array(Dynamic)
```

If the argument of exposeArray is a dynamic object containing an array with elements e_1 ... e_n, then the result is a new array with elements dynamic e_1 ... dynamic e_n.

```
exposeRec: Dynamic -> Dynamic
```

If the argument of exposeRec is a dynamic object with a recursive type of the form {printName = a, type = t} and value v, then the result is a dynamic object with type t and value v.

Examples

The following Amber program defines the Fibonacci function:

```
value fib =
    rec(fib: Int -> Int)
        fun (n: Int)
            if n < 2 then 1
            else fib(n-1) + fib(n-2);
```

This is a call-by-value lambda-calculus interpreter.

```
module "Eval"
export
    type Ide
    type Term
    type Env
    type Value
    value empty: Env
    value extend: (Ide,Value,Env) -> Env
    value lookup: (Ide,Env) -> Value
    value eval: (Term,Env) -> Value;

type Ide = String;

type Term =
    rec(Term)
        [ide : Ide,
         fun : {bind : Ide, body : Term},
         apply : {fun : Term, arg : Term}
        ];

type (Value,Env) =
    rec(Value,Env)
        ({bind : String, body : Term, env : Env},
          Ide -> Value);

value empty =
    fun (lookedFor: Ide) signal empty:Env;
```

```
value extend =
    fun (ide: Ide, value: Value, env: Env)
        fun (lookedFor: Ide)
            if equal(lookedFor,ide) then value else env(lookedFor);

value lookup =
    fun (ide: Ide, env: Env) env(ide);

 value (eval,apply) =
    rec(eval: (Term,Env) -> Value, apply: (Value,Value) -> Value)
        (fun (term: Term, env: Env)
            case term of
                [ide = i]  lookup(i,env)
                [fun = f] {bind = f.bind, body = f.body, env = env}
                [apply = a] apply(eval(a.fun,env), eval(a.arg,env))
            otherwise signal eval:Value,
         fun (fun: Value, arg: Value) to Value is
            eval(fun.body, extend(fun.bind, arg, fun.env))
        );

end;
```

References

[Albano Cardelli Orsini 85] A.Albano, L.Cardelli, R.Orsini: **Galileo: a strongly typed, interactive conceptual language**, *Transactions on Database Systems*, June 1985, 10(2), pp. 230-260.

[Apple 85] Apple Computer Inc. **Inside Macintosh**, Addison-Wesley, 1985.

[Atkinson Bailey Chisholm Cockshott Morrison 83] M.P.Atkinson, P.J.Bailey, K.J.Chisholm, W.P.Cockshott, R.Morrison: **An approach to persistent programming**, *Computer Journal* 26(4), November 1983.

[Burstall Lampson 84] R.M.Burstall, B.Lampson: **A kernel language for abstract data types and modules**, in *Sematics of Data Types*, Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[Cardelli 84a] L.Cardelli: **Compiling a functional language**, *Conference Record ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[Cardelli 84b] L.Cardelli: **A semantics of multiple inheritance**, in *Semantics of Data Types*, G.Kahn, D.B.MacQueen and G.Plotkin Ed. Lecture Notes in Computer Science n.173, Springer-Verlag 1984.

[Dahl Nygaard 66] O.Dahl, K.Nygaard: **Simula, an Algol-based simulation language**, *Comm. ACM*, Vol 9, pp. 671-678, 1966.

[Goldberg Robson 83] A.Goldberg, D.Robson: **Smalltalk-80. The language and its implementation**, Addison-Wesley, 1983.

[Gordon] M.Gordon. **Adding Eval to ML**, private communication (about dynamic types).

[INMOS 84] INMOS Ltd.: **occam programming manual**, Prentice Hall 1984.

[MacQueen 84] D.B.MacQueen: **Modules for Standard ML**, *Proc. Symposium on Lisp and Functional Programming*, Austin, Texas, August 6-8 1984, pp 198-207. ACM, New York.

[Milner 78] R.Milner: **A theory of type polymorphism in programming**, *Journal of Computer and System Science 17*, pp. 348-375, 1978.

[Milner 80] R.Milner: **A calculus of communicating systems**, Lecture Notes in Computer Science, n.92, Springer-Verlag, 1980.

[Milner 84] R.Milner: **A proposal for Standard ML**, *Proc. Symposium on Lisp and Functional Programming*, Austin, Texas, August 6-8 1984, pp. 184-197. ACM, New York.

[Mycroft 83] A.Mycroft: **Dynamic types in ML**, to appear.

[Wirth 83] N.Wirth: **Programming in Modula-2**, Texts and Monographs in Computer Science, Springer-Verlag 1983.

Shadow 4.0 Release

This section is supposed to tell you what is really in the current shadow - or, more often, what is not there yet.

Actual use of the system

The Macintosh "Switcher" program is automatically loaded when booting from the Amber system disk (if not, run the Switcher first). From the "File" menu of the Switcher select "Load Set...", and then open "Amber Set". This will load the Amber system and the editor.

Every top-level phrase must be terminated by semicolon:

```
type A = Int;
value a = 3;
a+1;
```

The variable it always denotes the last value which has been generated at the top-level. Declarations and erroneous computations do not count. it is just an ordinary program variable, and is statically scoped.

A file containing an Amber program (e.g. a set of definitions and expressions; one or more modules; etc.) can be loaded by:

```
infile("filename");
```

Identifiers are either alphanumeric (starting with a letter), or symbolic (composed by !@#$%^&*_-+=|\/?<>). All symbolic identifiers are automatically infix, and must be declared as functions of two arguments:

```
value ** = fun(x:Int, y:Int) (x*x)+(y*y);
3 ** 4;
```

All infix identifiers have the same precedence and associate to the right, even the primitive ones
like +, * and =. Infix identifiers are also accepted in function position:

```
+(3,4);
```

Comments are enclosed in backquoted parentheses, and can be nested:

```
`(This is a `(This is a comment within a comment)` comment)`
```

In case of syntactic or typechecking errors, the evaluation of the faulty phrase is inhibited, so no harm will result. Looping programs will crash the system.

Mac interface

There are two windows, called "Text" and "Graphics". All the character I/O takes place in the text window, and all the operations involving the screen() bitmap have effect in the graphic window.

The text window is split into two panels: the upper one is the *past* and the lower one is the *future*. They are separated by the *present* line.After entering the system you can just type phrases, and everything will behave like a normal terminal.

If you type in the future panel (which is the active one, initially) the text accumulates until you hit carriage return. Before hitting carriage return, you can backspace or mouse-edit the text. When you hit carriage return all the text disappears from the future panel, crosses the present line and shows up again in the past panel. Whenever some text crosses the present line, it is also sent to Amber for execution; the answers will appear in the past panel. You can hit the enter key, instead of carriage return, to create multi-line text in the future panel. When you are ready to execute it, just hit carriage return (anywhere in the future panel).

You can also type and mouse-edit in the past panel, and cut and paste across panels (the enter key in the past panel behaves like carriage return). Text in the past panel can be sent to Amber directly, without having to paste it into the future panel, by selecting it and choosing "Exec" in the "Edit" menu. "Exec" always refers to the selection in the past panel, even if the future panel is the active one.

Past and future can be independently scrolled vertically, and simultaneously scrolled horizontally. The past panel fills up after a while; the lines scrolling off the top of the full panel are lost forever. The future panel also fills up, but no lines are ever lost. The panel stops scrolling and the bottom lines fall off the bottom: they become inaccessible (except by cleaning up the panel higher up), but they are there. You can move the present line by dragging it up and down.

Cut and past between the text window and the clipboard does not work at the moment, so you cannot transfer text directly to/from the editor.

The graphic window pops up automatically whenever a graphic operation is executed, and stays there until the text window is selected again from the "Windows" menu. The graphic window cannot be moved or reshaped.

At the moment, the contents of the graphic window are not saved: whenever a window is put on top of the graphic window, the graphics under it is lost.

Files

The "Amber" icon (a griffin) starts up the normal Amber system. The system also needs "amber.dex" and "ops.dex" to run. Files created by the system have unicorns as icons.

The "Amber" icon has a 200K heap space, and can run under the Switcher. The "Amber Trace" icon is the same, but also allows tracing. The "Big Amber" icon has a 350K heap space, and cannot run under the Switcher. To give you a feeling of heap sizes, "Big Amber" is enough to recompile the whole Amber system (5000 lines partitioned in 23 modules) with only three or four garbage collections.

The conventional file extensions are ".amb" for source files, ".dex" for files produced by extern, and ".mod" for compiled modules.

Input-Output

Input-output is rather rudimentary, and subject to change. There is an idea of an input stream and an output stream (initially connected to the top-level), used implicitly by input and output operations.

The input stream is redirected to a file by:

```
infile: String -> ()
```

if an end-of-file is found, or if an infile("") is executed, the input is returned to the previous stream. The output stream is redirected to a file by:

```
outfile: String -> ()
```

if an outfile("") is executed, the output returns to the previous stream.

Input operations are:

```
input: () -> Int
```

read a character

```
caninput: () -> Bool
```

test if there are characters to be read

Output operations are:

```
output: Int -> ()
```

write a character

```
outstring: String -> ()
```

write a string

Side-effects

The sections on records and variants talk about control of side-effects by the => and :> notation (instead of = and :). This is not implemented: all record fields are assignable and variants are not assignable; => and :> are not legal syntax in the current shadow.

Graphics

The following bitmap operations are not implemented: setcursor, cursoricon, cursortip. None of the region operations are implemented, except for nullregion.

Concurrency

Not implemented at all.

Dynamics

The operations typeOf and expose... are not implemented.

Recursive definitions

Only functions can be recursively defined at the moment; not records, variants etc. The effect of recursive records etc. can however be achieved by inserting extra fun().

Reset

The top-level command reset; will clear the type and value environments, which otherwise keep growing during interaction. It is not normally necessary to reset. Module compilations do not affect the environments, hence you don't need to reset between compilations.

Debugging

Not much of it. There is a generic print function, a trace facility, and a few system flags.

The function print accepts any value and prints its internal format, no matter what its type is. Several print options are described below. This routine is also used by the trace facility, and the setting of the print options affects the trace output.

The trace facility only works with the "Amber Trace" system, which is distinct from with the normal "Amber" system (and slightly slower). When tracing is enabled, all the function calls and function returns are reported, together with parameters and results.

The system function takes a string and interprets it to set system flags.

"t-"	disable tracing
"t0"	trace the inner execution of the compiler
"t1"	trace user programs
"s+"	start tracing signals and traps
"s-"	stop tracing signals and traps
"pdn"	set maximum print depth to n, for n in 0..9
"pwn"	set maximum print width (for arrays and strings) to n*100 items
"pln"	set total amount of output to approx n*100 items per print
"pb+"	enable printing of non-ascii strings
"pb-"	disable printing of non-ascii strings
"pa+"	enable printing of arrays of immediates (e.g. int or bool)
"pa-"	disable printing of arrays of immediates

Bugs

There are no known bugs at present (other than dubious features). The system recompiles itself quite happily and the garbage collector seems to be very robust.

Lexicon

char ::= any printable character (excluding blank)

letter ::= **a**..**z**, **A**..**Z**

digit ::= **0**..**9**

special ::= **!@#$%^&*_-+=|\/?<>**

alphanum ::= sequence of letter or digit starting with a letter

symbol ::= sequence of special

number ::= sequence of digit possibly prefixed by **~**

stringchar ::= blank or any char but **"** and ****

stringescape ::= **** followed by any char or blank

string ::= any sequence of stringchar or stringescape enclosed in **"**

Comments are enclosed between backquoted parentheses **`(** and **)`** , and can be nested.

Keywords:

Array array arraysize case coerce do dynamic else end export false fun if import in index let module nullregion on otherwise rec repeat reset set signal then to true type unity update value var while ->

ide ::=	alphanum | symbol	but not a keyword
typeide ::=	alphanum	but not a keyword
oper ::=	symbol	but not a keyword
label ::=	alphanum	
signal ::=	alphanum | symbol	

Syntax

Terminal symbols are in bold, non-terminals and meta-syntactic notation in roman.

... ...	sequencing (strongest binding power)
... | ...	alternative (weakest binding power)
[...]	optionally
{ ... }	zero or more times
{ ... / ... }	zero or more times (left dots) separated by (right dots).
(...)	grouping

top ::=
(phrase | module | import | **reset**) **;**

phrase ::=
type typebind **=** type |
value bind **=** exp |
exp

typebind ::=
typeide |
({ typeide / **,** } **)**

type ::=
Unit | **Bool** | **Int** | **String** | **Bitmap** | **Region** | **Dynamic** |
typeide |
({ type / **,** } **)** |
{ { label **:** type / **,** } **}** |
[{ label **:** type / **,** } **]** |
type **->** type |
Array (type **)** |
rec ({ typeide / **,** } **)** type

bind ::=
[**var**] ide |
({ ide / **,** } **)**

exp ::=
unity | **true** | **false** | **'**char | number | string |
ide |
({ exp / **,** } **)** |

{ { label **=** exp / **,** } **}** |
[label **=** exp **]** |
fun ({ ide **:** type / **,** } **)** exp |
if exp **then** exp **else** exp |
while exp **repeat** exp |
dynamic exp |
coerce exp **to** type |
exp **.** label |
set ide **.** label **=** exp |
case exp { **[** label [**=** ide] **]** exp } **otherwise** exp |
exp **(** { exp / **,** } **)** |
exp oper exp |
{ **let** bind **=** exp | **do** exp } **do** exp |
rec ({ ide **:** type / **,** } **)** exp |
var ide **=** exp |
signal signal **:** type |
on signal exp **in** exp

module ::=
module string
{ import **;** | **type** (string | typebind **=** type) **;** }
export interface **;**
{ phrase **;** }
end

import ::=
import string interface

interface ::=
{ **type** typeide [**in** typeide { **,** typeide }] |
value [**var**] ide **:** type }