Timber: A Programming Language for Real-Time Embedded Systems

Andrew P. Black, Magnus Carlsson, Mark P. Jones,
Richard Kieburtz and Johan Nordlander

Department of Computer Science and Engineering
OGI School of Science & Engineering
Oregon Health & Science University
20000 NW Walker Road
Beaverton, OR 97006-8921 USA

Technical Report Number CSE 02-002
April 2002

Timber: A Programming Language for Real-Time Embedded Systems

Andrew P. Black, Magnus Carlsson, Mark P. Jones, Richard Kieburtz and Johan Nordlander
timber@cse.ogi.edu

Department of Computer Science & Engineering
OGI School of Science & Engineering at OHSU
Beaverton, Oregon, USA

In this paper we provide a detailed but informal survey of Timber and its characteristic features. A formal semantic treatment of the language will appear in other papers; here the exposition will instead be based on short code examples. However, we also introduce the semantic model that underlies one of Timber's main contributions: the way that time is integrated into the language.

May of the features of Timber have been adopted from the reactive object-oriented concurrent functional language O'Haskell [15], which was in turn defined as an extension to the purely functional language Haskell [12]. However, the Haskellian ancestry of Timber should not cause it to be ignored by the wider (non-functional) programming language community. Indeed, Timber attempts to combine the best features of three different programming paradigms.

- Timber is an **imperative object-oriented language**, offering state encapsulation, objects with identity, extensible interface hierarchies with subtyping, and the usual complement of imperative commands such as loops and assignment. Inheritance in the style of, *e.g.*, Smalltalk, is not presently supported, but this is an area that we continue to study. The lack of inheritance is largely counterbalanced by rich facilities for parameterization over functions, methods, and templates for objects. Additional Timber features not commonly found in object-oriented languages include parametric polymorphism, type inference, a straightforward concurrency semantics, and a powerful expression sub-language that permits unrestricted equational reasoning.
- Timber can also be characterized as a strongly typed **concurrent language**, based on a monitor-like construct with implicit mutual exclusion, and a message-passing metaphor offering both synchronous and asynchronous communication. However, unlike most concurrency models, a Timber process is represented as an object, that is, as the unit of state encapsulation. Moreover, execution of a process should not be regarded as continuous, but should instead be thought of as consisting of a

sequence of **reactions** to external events (made visible to the object as messages). These reactions always run to completion (*i.e.*, they are non-blocking) and in mutual exclusion. The execution order of reactions is determined by baselines and deadlines associated with these events. When we speak of reactive objects in the sequel, this is what we mean.

- Timber is finally a **purely functional language** that supports stateful objects through the use of monads. Timber allows recursive definitions, higher-order functions, algebraic datatypes, pattern-matching, and Hindley/Milner-style polymorphism. Timber also supports type constructor classes and overloading as in Haskell, but these features are not central and the Timber extensions to Haskell do not depend on them. To this base Timber conservatively adds two major features: *subtyping*, and a monadic implementation of stateful *reactive objects*. The subtyping extension is defined for records as well as datatypes, and is supported by a powerful partial type inference algorithm that preserves the types of all programs typeable in Haskell. The monadic object extension is intended as a replacement for Haskell's standard IO model, and provides concurrent objects and assignable state variables while still maintaining referential transparency.

The exposition here largely follows the informal survey of O'Haskell in Nordlander's thesis [15]. Section 1 presents a brief overview of the base language Haskell and its syntax, before we introduce the major type system additions of Timber: *records* and *subtyping* (Sections 2 and 3). In Section 4 our approach to *type inference* in Timber is presented. The rôle of *time* is introduced in Section 6. *Reactive objects*, *concurrency*, and *encapsulated state* are discussed in Section 5. Section 7 presents some additional syntactic features of Timber, before the paper ends with an example of Timber programming (Section 8). The grammar of Timber appears in the Appendix.

1. Haskell

Haskell [1, 12] is a lazy, purely functional language, and the base upon which Timber is built. Readers familiar with Haskell may wish to skip this section; it introduces no new material, and is present to make this paper accessible to those who have not previously met the language, or who need a reminder of its features and syntax.

Functions

Functions are the central concept in Haskell. Applying a function to its arguments is written as a simple juxtaposition; that is, if f is a function taking three integer arguments, then

```
f 7 13 0
```

is an expression denoting the result of evaluating f applied to the arguments 7, 13, and 0. If an argument itself is a non-atomic expression, parentheses must be used as delimiters, as in

```
f 7 (g 55) 0
```

Operators like + (addition) and == (test for equality) are also functions, but are written between their first two arguments. An ordinary function application always binds more tightly than an operator, thus

```
a b + c d
```

should actually be read as

```
(a b) + (c d)
```

Laziness

The epithet *lazy* means that the arguments to a function are evaluated only when absolutely necessary. So, even if

```
g 55
```

is a non-terminating or erroneous computation (including, for example, an attempt to divide by zero), the computation

```
f 7 (g 55) 0
```

might succeed in Haskell, if f happens to be a function that ignores its second argument whenever the first argument is 7. This kind of flexibility can be very useful for encoding and manipulating infinite data structures, and for building functions that play the rôle of control structures.

One of the consequences of laziness is that it can sometimes become quite hard to predict when computation will actually take place, and calculating worst case execution times is correspondingly difficult. Whether the costs of laziness outweigh the benefits in a language intended for real-time programming is an open question, and one that we will continue to examine experimentally. It is important to note that none of the extensions to Haskell that we put forward in Timber relies on laziness. Thus it is perfectly reasonable to judge the merits of our extensions as if they were intended for an eager programming language, and it would be perfectly possible to give Timber an eager semantics without major surgery.

Function definitions

Functions can be defined by equations on the top-level of a program. They can also be defined locally within an expression. The following fragment defines the function f at the top-level; the function sq is defined locally within the body of f.

```
f x y z = let sq i = i * i
          in sq x * sq y * sq z
```

Note that the symbol = denotes *definitional* equality in Haskell (*i.e.*, = is neither an assignment nor an equality test). Local definition of a function within other definitions is also possible, as in

```
f x y z = sq x * sq y * sq z where
    sq v = v * v
```

Anonymous functions can be introduced with the so-called *lambda-expression*, written using the symbols \ …-> in lieu of λ…. So

```
\ x y z -> x*y*z
```

is an expression whose value is the function that multiplies together its three arguments. An identical function is defined and named product by the definition

```
product x y z = x*y*z
```

The scope of a name can be limited by a **let** expression, so

```
let product x y z = x*y*z in product
```

has as its value the same anonymous function as the original lambda expression.

Type inference

When introducing a new variable, the programmer does not in general have to declare its type. Instead, the Hindley-Milner-style type inference algorithm employed in Haskell is able to discover the most general type for each expression. This often results in the inference of a *polymorphic type, i.e.,* a type expression that includes one or more variables standing for arbitrary types.

The simplest example of a polymorphic type is that inferred for the identity function,

```
id x = x
```

The most general type that can be ascribed to the function id is a -> a: this type is polymorphic, since a is treated as if it were universally quantified, that is, “for all types a”. However, the programmer can also use an explicit type annotation to indicate a more specific type, as in

```
iid :: Int -> Int
iid x = x
```

Partial application

A function like f above that takes three integer arguments and delivers an integer result has the type

```
Int -> Int -> Int -> Int
```

Arrow associates to the right, so this means Int -> (Int -> (Int -> Int))

Hence such a function need not always be supplied with exactly three arguments. Instead, functions can be *partially applied*; a function applied to fewer than its full complement of arguments is treated as denoting an anonymous function, which in turn is applicable to the missing arguments. This means that (f 7) is a valid expression of type Int -> Int -> Int, and that (f 7 13) denotes a function of type Int -> Int. Note that this treatment is consistent with parsing an expression like f 7 13 0 as (((f 7) 13) 0).

Pattern-matching

Haskell functions are often defined by a sequence of equations that *pattern-match* on their arguments, as in the following example:

```
fac 0 = 1
fac n = n * fac (n-1)
```

which is equivalent to the more conventional definition

```
fac n = if n==0 then 1 else n * fac (n-1)
```

Pattern-matching using Boolean *guard* expressions is also available, although this form is a bit contrived in this simple example.

```
fac n   | n==0        = 1
        | otherwise   = n * fac (n-1)
```

Moreover, explicit **case** expressions are also available in Haskell, as shown in this fourth variant of the factorial function:

```
fac n = case n of
          0  ->  1
          m  ->  m * fac (m-1)
```

Algebraic datatypes

User-defined types (called *algebraic datatypes* in Haskell) can be defined using data definitions, which define a kind of labeled union type with name equality and recursive scope. Here is an example of a data definition for binary trees: it declares three identifiers, BTree, Leaf and Node.

```
data BTree a   =   Leaf a
               |   Node (BTree a) (BTree a)
```

The type argument a is used to make the BTree polymorphic in the contents of its leaves; thus a binary tree of integers has the type BTree Int. The identifiers Leaf and Node are called the *constructors* of the datatype. Constructors, which have global scope in Haskell, can be used both as functions and in patterns, as the following example illustrates:

```
swap (Leaf a) = Leaf a
swap (Node l r) = Node (swap r) (swap l)
```

This function (of type BTree a -> BTree a) takes any binary tree and returns a mirror image of the tree obtained by recursively swapping its left and right branches.

Predefined types

In addition to the integers, Haskell's primitive types include characters (Char) as well as floating-point numbers (Float and Double). The type of Boolean values (Bool) is predefined, but is an ordinary algebraic datatype. Lists and tuples are also essentially predefined datatypes, but they are supported by some special syntax. The empty list is

written [], and a non-empty list with head x and tail xs is written x:xs. A list known in its entirety can be expressed as [x1,x2,x3], or equivalently x1:x2:x3:[]. Moreover, a pair of elements a and b is written (a,b), and a triple also containing c is written (a,b,c), *etc*.

As an illustration of these issues, here is a function which "zips" two lists into a list of pairs:

```
zip (a:as) (b:bs)    = (a,b) : zip as bs
zip _      _         = [] .
```

Note that the order of these equations is significant.

The names of the types of lists and tuples are analogous to the terms: [a] is the type of lists containing elements of type a, and (a,b) denotes the type of pairs formed by elements of types a and b. Thus the type of the function zip above is [a] -> [b] -> [(a,b)]. There is also degenerate tuple type (), called *unit*, which contains only the single element (), also called unit.

Strings are just lists of characters in Haskell, although conventional string syntax can also be used for constant strings, with "abc" being equivalent to ['a','b','c']. The type name String is just a *type abbreviation*, defined as:

```
type String = [Char]
```

String concatenation is an instance of general list concatenation in Haskell, for which there exists a standard operator ++, defined as

```
[] ++ bs = bs
(a:as) ++ bs = a : (as ++ bs)
```

Haskell also provides a primitive type Array, with an indexing operator ! and an "update" operator //. However, this type suffers from the fact that updates must be implemented in a purely functional way, which often means creating a fresh copy of an array each time it is modified. We will see later in this paper how monads and stateful objects enable us to support the Array type in a more intuitive, as well as a more efficient, manner.

Higher-order functions

Functions are first-class values in Haskell, so it is quite common for a function to take another function as a parameter; such a function is known as a higher-order function. map is a typical example of a higher-order function; map takes two arguments, a function and a list, and returns a new list created by applying the function to each element of the old list. map is defined as follows:

```
map f [] = []
map f (x:xs) = f x : map f x
```

The fact that map is higher-order is exposed in its type,

```
map :: (a -> b) -> [a] -> [b]
```

where the parentheses are essential. As an example of how map can be used, we construct an upper-casing function for strings by defining

```
upCase = map toUpper
```

where toUpper :: Char -> Char is a predefined function that capitalizes characters. The type of upCase must accordingly be

```
upCase :: [Char] -> [Char]
```

or, equivalently,

```
upCase :: String -> String  .
```

Layout

Haskell makes extensive use of indentation — two dimensional layout of text on the page — to convey information that would otherwise have to be supplied using delimiters. We have been using this convention in the foregoing examples, and will continue to do so. The intended meaning should be obvious. It is occasionally convenient to override the layout rules with a more explicit syntax, so it may be good to keep in mind that the two-dimensional code fragment

```
let   f x y = e1
      g i j = e2
in g
```

is actually a syntactic shorthand for

```
let { f x y = e1 ; g i j = e2 } in g   .
```

Informally stated, the braces and semicolons are inserted as follows. The layout rule takes effect whenever the open brace is omitted after certain keywords, such as **where**, **let**, **do**, **record** and **of**. When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted. For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted); if it is indented the same amount, then a new item begins (a semicolon is inserted); and if it is indented less, then the layout list ends (a close brace is inserted). A close brace is also inserted whenever the syntactic category containing the layout list ends; that is, if an illegal lexeme is encountered at a point where a close brace would be legal, a close brace is inserted. The layout rule matches only those open braces that it has inserted; an explicit open brace must be matched by an explicit close brace.

2. Records

The first Timber extension beyond Haskell is a system for programming with first-class records. Although Haskell already provides some support for records, we have chosen to replace this feature with the present system, partly because the Haskell proposal is a somewhat ad-hoc adaptation of the datatype syntax, and partly because Haskell records do not fit very well with the subtyping extension that is described in Section 3.

The distinguishing feature of Timber records is that the treatment of records and datatypes is perfectly symmetric; that is, there is a close correspondence between record selectors and datatype constructors, between record construction and datatype selection (*i.e.*, pattern-matching over constructors), and between the corresponding forms of type extension, which yield subtypes for records and supertypes for datatypes.

Consequently, we treat *both* record selectors *and* datatype constructors as *global* constants. This is the common choice where datatypes are concerned, but not so for records (see, *e.g.*, references [14] and [8]). Nevertheless, we think that a symmetric treatment has some interesting merits in itself, and that the ability to form hierarchies of record types alleviates most of the problems of having a common scope for all selector names. We also note that overloaded names in Haskell are given very much the same treatment, without creating many problems in practice.

A record type is defined in Timber by a global declaration analogous to the datatype declaration described previously. The following example defines a record type for two-dimensional points, with two selector identifiers of type Float.

```
record Point where x,y :: Float
```

The **record** keyword is also used in the term syntax for record construction. We will generally rely on Haskell's layout rule (see "Layout" on page 7) to avoid cluttering our record expressions with braces, as in the following example.

```
origin = record
           x = 0.0
           y = 0.0
```

Since the selector identifiers x and y are global, there is no need to indicate to which record type a record term belongs; the compiler can deduce that origin is a Point. If one wishes to make this clear to the human reader, one can explicitly write

```
origin:: Point
origin = record
           x = 0.0
           y = 0.0
```

but this is entirely redundant. It is a static error to construct a record term where the set of selector equations is not exhaustive for some record type.

Record selection is performed in the conventional way by means of the *dot*-syntax. (Timber also has an operator **.** used to denote function composition; record selectors

should immediately follow the dot, while the operator **.** must be followed by some amount of white space). Record selection binds more tightly than function and operator application, as the following example indicates.

```
dist p = sqrt (sq p.x + sq p.y) where
    sq i = i * i
```

A record selector can moreover be turned into an ordinary prefix function if needed, by enclosing it between (. and), as in

```
xs = map (.x) some_list_of_points
```

Just as algebraic datatypes may take type arguments, so may record types. The following example shows a record type that captures the signatures of the standard equality operators.†

```
record Eq a where
    eq :: a -> a -> Bool
    ne :: a -> a -> Bool
```

This defines a record type with two selectors named eq and ne. The types of these selectors are both a -> a -> Bool, where the type a is a parameter to Eq.

A record term of type Eq Point is defined below.

```
pdict =  record
             eq = eq
             ne a b = not (eq a b)
         where eq a b = a.x==b.x && a.y==b.y
```

This example also illustrates three minor points about records: (1) record expressions are not recursive, (2) record selectors possess their own namespace (the equation eq = eq above is *not* recursive), and (3) selectors may be implemented as functions if so desired.

†. Strictly speaking, this record type is not legal since its name coincides with that of a predefined Haskell *type class*. Type classes form the basis of the *overloading* system of Haskell, whose ins and outs are beyond the scope of this survey. The name Eq has a deliberate purpose, though — it connects the example to a known Haskell concept, and it indicates the possibility of reducing the number of constructs in Timber by eliminating type class declarations in favour of record types.

3. Subtyping

The subtyping system of Timber is based on *name inequality*. This means that a possible subtype relationship between (say) two record types is determined solely by the names of the involved types, and not by consideration to whether the record types in question might have matching substructure. Name inequality is a generalization of the *name equality* principle used in Haskell for determining whether two types are equal.

The subtype relation between user-supplied types is induced as a consequence of declaring a new type as an extension of a previously defined type. This makes record subtyping in Timber look quite similar to interface extension in Java, as the following type declaration exemplifies:

```
record CPoint < Point where
    color :: Color
```

This syntax both introduces the record type CPoint and declares it to be an extension of the existing type Point. The extension is the addition of the selector color. As a consequence of this definition, CPoint is a subtype of Point, written CPoint < Point. We call CPoint < Point a subtyping rule. The meaning of this definition is that type CPoint possess the selectors x and y in addition to its own selector color.

The structure of CPoint must be observed when constructing CPoint terms, as is done in the following function.

```
addColor :: Point -> CPoint

addColor p = record  x = p.x
                     y = p.y
                     color = Black

cpt = addColor origin
```

Here addColor is defined to be a function that converts Points to CPoints by coloring them black. Notice that leaving out the equation color = Black would make the definition invalid, since the function result would then be a value of type Point instead of CPoint, contradicting the type definition.

Subtyping can also be defined for algebraic datatypes. Consider the following type modelling the colors black and white.

```
data BW = Black | White
```

This type can now be used as the basis for an extended color type:

```
data Color > BW =
        Red | Orange | Yellow | Green | Blue | Violet
```

Since its set of possible values is larger, the new type Color defined here must necessarily be a *supertype* of BW (hence we use the symbol > instead of < when extending a

datatype). The subtype rule introduced by this declaration is accordingly BW < Color, and type Color possess all of the constructors of its base type BW, in addition to those explicitly mentioned for Color. This is analogous to situation for record types formed by extension, where the extended type has all of the destructors (selectors) of its base type.

Timber allows pattern-matching to be incomplete, so there is no datatype counterpart to the static exhaustiveness requirement that exists for record types. However, the set of constructors associated with each datatype still influences the meaning of Timber programs. This is because the type inference algorithm approximates the domain of a pattern-matching construct by the smallest type that contains all of the enumerated constructors. The functions f and g, defined as

```
f Black  = 0
f _      = 1

g Black  = 0
g Red    = 1
g _      = 2
```

illustrate this point. The domain of f is inferred to be BW, while the domain of g is inferred to be Color.

Polymorphic subtype rules

Subtype definitions may be polymorphic. Consider the following example where a record type capturing the standard set of comparison operators is formed by extending the type Eq defined above.

```
record Ord a < Eq a where
    lt, le, ge, gt :: a -> a -> Bool
```

The subtype rule induced by the definition of Ord states that for all types a, a value of type Ord a also supports the operations of Eq a. Ord a must also support the operations lt, le, ge and gt

Polymorphic subtyping works just as well for datatypes. Consider the following example, which provides an alternative definition of the standard Haskell type Either.

```
data Left a = L a

data Right a = R a

data Either a b > Left a, Right b
```

The first declaration, actually defines both a new datatype Left a and a new constructor for values of that type, called L. The declaration of Right is parallel.

The last declaration is an example of type extension with multiple basetypes. Like interface extension in Java, this declaration introduces *two* polymorphic subtype rules; one that says that for all a and b, a value in Left a also belongs to Either a b, and one that says that for all a and b, a value in type Right b also belongs to Either a b. The declara-

tion of Either also shows that a datatype declaration need not declare any new constructors.

Depth subtyping

Subtyping is a reflexive and transitive relation. This, any type is a subtype of itself, and S < T and T < U implies S < U for all types S, T, and U. The fact that type constructors may be parameterized makes subtyping a bit more complicated, though. For example, under what circumstances should we be able to conclude that Eq S is a subtype of Eq T?

Timber incorporates a flexible rule that allows *depth subtyping* within a type constructor application, by taking the *variance* of a type constructor's parameters into account. By variance we mean the rôle that a type variable has in the set of type expressions in its scope—does it occur in a function argument position, in a result position, in both these positions, or perhaps not at all?

In the definition of the record type Eq above

```
record Eq a where
    eq :: a -> a -> Bool
    ne :: a -> a -> Bool
```

all occurrences of the parameter a are in an argument position. For these cases Timber prescribes *contravariant* subtyping, which means that Eq S is a subtype of Eq T only if T is a subtype of S. Thus we have that Eq Point is a subtype of Eq CPoint. This means that an equality test developed for Points can also be applied to CPoints, *e.g.*, it can be used to partition colored points into equivalence classes.

The parameter of the datatype Left, on the other hand, occurs only as a top-level type expression (that is, in a result position). In this case subtyping is *covariant*, which means for example that Left CPoint is a subtype of Left Point. As an example of *invariant* subtyping, consider the record type

```
record Box a where
    in  :: a -> Box a
    out :: a
```

Here the type parameter a plays the rôle of a function argument as well as a result, so both the co- and contravariant rules apply at the same time. The net result is that Box S is a subtype of Box T only if S and T are identical types. There is also the unlikely case where a parameter is not used at all in the definition of a record or datatype:

```
data Contrived a = Unit
```

Clearly a value of type Contrived S also has the type Contrived T for any choice of S and T, thus depth subtyping for this *nonvariant* type constructor can be allowed without any further preconditions. The motivation behind these rules is of course the classical rule for subtyping of function types, which states that S -> T is a subtype of S' -> T' only if S' is a subtype of S, and T is a subtype of T' [7]. Timber naturally supports this rule, as well as covariant subtyping for the built-in aggregate types: lists, tuples, and

arrays. Depth subtyping may be transitively combined with declared subtypes to deduce subtype relationships that are intuitively correct, but perhaps not immediately obvious. Some illustrative examples follow.

Relation:	Interpretation:
Left CPoint < Either Point Int	*If either some kind of point or some integer is expected, a colored point will certainly do.*
Ord Point < Eq CPoint	*If an equality test for colored points is expected, a complete set of comparison operations for arbitrary points definitely meets the goal.*

Restrictions on subtyping

The general rule when defining types by subtyping is that the newly defined subtype or supertype may be any type expression that is not a variable. There are, however, some restrictions, for example, it is illegal to define **record** S a < Bool, because the supertype is not a record type. We will not dwell on the restrictions here; more information can be found in reference [15].

4. Automatic type inference

In a polymorphic language, expressions have several types. A *principal type* is a type sufficiently general for all of the other types to be deducible from it.

In Haskell, the polymorphic function

```
twice f x = f (f x)
```

has the principal type

```
(a -> a) -> a -> a
```

from which every other valid type for twice, *e.g.*, (Point -> Point) -> Point -> Point, can be obtained as a substitution instance.

However, it is well known that polymorphic subtyping systems need types qualified by *subtype constraints* in order to preserve a notion of principal types. To see this, assume that we allow subtyping, and that CPoint < Point. Now twice can also have the type

```
(Point -> CPoint) -> Point -> CPoint
```

which is not an instance of the principal Haskell type. In fact, there can be no simple type for twice that has both (Point -> Point) -> Point -> Point and (Point -> CPoint) -> Point -> CPoint as substitution instances, since the greatest common anti-instance of these types, (a -> b) -> a -> b, is not a valid type for twice.

Thus, to obtain a notion of principality in this case, we must restrict the possible instances of a and b to those types that allow a subtyping step from b to a; that is, we must associate the subtype constraint b < a with the typing of twice. In Timber, subtype

constraints are attached to types using the " =>" symbol†, so the principal type for twice can be written

```
(b < a ) => (a -> b) -> a -> b .
```

This type has two major drawbacks compared to the principal Haskell type: (1) it is syntactically longer than most of its useful instances because of the subtype constraint, and (2) it is no longer unique modulo renaming, since it can be shown that, for example,

```
(b < a, c < a, b < d) => (a -> b) -> c -> d
```

is also a principal type for twice. In this simple example the added complexity that results from these drawbacks is of course manageable, but even just slightly more involved examples soon get out of hand. The problem is that, in effect, *every application node* in the abstract syntax tree can give rise to a new type variable and a new subtype constraint. Known complete inference algorithms tend to illustrate this point very well, and even though algorithms for simplifying the type constraints have been proposed that alleviate the problem to some extent, the general subtype constraint simplification problem is at least NP-hard. It is also an inevitable fact that no conservative simplification strategy can ever give us back the attractive type for twice that we have in Haskell.

For these reasons, Timber relinquishes the goal of complete type inference, and employs a *partial* type inference algorithm that gives up generality to gain consistently readable output. The basic idea is to let functions like twice retain their original Haskell type, and, in the spirit of monomorphic object-oriented languages, infer subtyping steps only when both the inferred and the expected type of an expression are known. This choice can be justified on the grounds that (a -> a) -> a -> a is still likely to be a sufficiently general type for twice in most situations, and that the benefit of consistently readable output from the inference algorithm will arguably outweigh the inconvenience of having to supply a type annotation when this is not the case. We certainly do not want to prohibit exploration of the more elaborate areas of polymorphic subtyping that need constraints, but considering the cost involved, we think that it is reasonable to expect the programmer to supply the type information in these cases.

As an example of where the lack of inferred subtype constraints might seem more unfortunate than in the typing of twice, consider the function

```
min x y = if less x y then x else y
```

which, assuming less is a relation on type Point, will be assigned the type

```
Point -> Point -> Point
```

by Timber's inference algorithm. A more useful choice would probably have been

```
(a < Point) => a -> a -> a
```

---

†. The syntax is inspired by the way that type classes are expressed in Haskell.

but, as we have indicated, such a constrained type can only be attained in Timber by means of an explicit type annotation. On the other hand, note that the *principal* type for min,

```
(a < Point, b < Point, a < c, b < c) => a -> b -> c
```

is still more complicated, and unnecessarily so in most realistic contexts.

An informal characterization of our inference algorithm is that it improves on ordinary polymorphic type inference by allowing subtyping under function application when the types are known, as in

```
addColor cpt
```

In addition, the algorithm computes least upper bounds for instantiation variables when required, so that, *e.g.*, the list

```
[cpt, pt]
```

will receive the type

```
[Point]
```

Greatest lower bounds for function arguments will also be found, resulting in the inferred type

```
CPoint -> (Int,Bool)
```

for the term

```
\ p -> (p.x, p.color == Black)   .
```

Notice, though, that the algorithm assigns constraint-free types to *all* subterms of an expression, hence a compound expression might receive a less general type, even though its principal type has no constraints. One example of this is

```
let twice f x = f (f x) in twice addColor pt
```

which is assigned the type Point, not the principal type CPoint.

Unfortunately, a declarative specification of the set of programs that are amenable to this kind of partial type inference is still an open problem. Completeness relative to a system that lacks constraints is also not a realistic property to strive for, due to the absence of principal types in such a system. However, experience strongly suggests that the algorithm is able to find solutions to most constraint-free typing problems that occur in practice—in fact, an example of where it mistakenly fails has yet to be found in our experience with O'Haskell and Timber programming. Moreover, the algorithm is provably complete with respect to the Haskell type system, and hence possesses another very important property: programs typeable in Haskell retain their inferred types when con-

sidered as Timber programs. Additionally, the algorithm can also be shown to accept all programs typeable in the core type system of Java (see section 5.3 of [15]).

5. Reactive objects

The dynamic behavior of a Timber program is the composition of the behavior of many state-encapsulating, time sensitive *reactive object*s executing concurrently. In this section we will survey this dynamic part of the language as it is seen by the programmer. The number of new concepts introduced here is quite large, so we proceed step by step and ask the reader to be patient while the structure of the language unfolds.

Some of the language features will initially appear to be incompatible with a purely functional language. However, it is in fact the case that all constructs introduced in this section are syntactically transformable into a language consisting of only the Haskell kernel and a set of primitive monadic constants. This "naked" view of Timber will not be pursued here; the interested reader is referred to chapter 6 of Nordlander's Thesis [15].

Objects and methods

Objects are created by executing a **template** construct, which defines the *initial state* of an object together with its *communication interface*. Unlike Smalltalk or Java, there is no notion of class; in this way Timber is similar to classless languages like Emerald [16] and Self [17].

The communication interface can be a value of any type, but it will usually contain one or more *methods*. (It can be useful to put other things in the communications interface too, as we will see in some of the examples.) Methods allow the object to react to *message sends*. We use the term *message send* in its usual sense in object-oriented programming: a message send is directed from one object (the sender) to another object (the receiver); in response, the receiver first selects and then executes a method. Message send is sometimes called method invocation or even method call, but the term message send is preferred because it emphasises that the coupling between the message and the method occurs in the receiving object.

A method takes one of two forms: an asynchronous **action** or a synchronous **request**. An **action** lets the sender continue immediately, and thus introduces concurrency. Consequently, actions have no result. A synchronous **request** causes the sender to wait for the method to complete, but allows a result value to be passed back to the sender.

The body of a method, finally, is a sequence of *commands*, which can basically do three things: update the local state, create new objects, and send messages to other objects. The following template defines a simple counter object:

```
counter =   template
                val := 0
            in record
                   inc =
                       action val := val + 1
                   read =
                       request return val
```

Executing this **template** command creates a new counter object with its own state variable val, and returns an interface through which this new object can be accessed. The interface is a record containing two methods: an asynchronous **action** inc, and a synchronous **request** read. Sending the message inc to this interface will cause the **action** val := val + 1 to be executed, with the effect that the counter object behind the interface will update its state, concurrently with the continued execution of the sender. In contrast, sending the message read will essentially perform a rendezvous with the counter object, and return its current value to the sender.

Procedures and commands

Actions, requests and templates are all expressions that denote commands; such expressions have a type in a *monad* called Cmd. A monad is a structure that allows us to deal with effect-full computations in a mathematically consistent way by formalizing the distinction between *pure* computations (those that simply compute values), and *impure* computations, which may also have effects, such as changing a state or accessing an external device.

Cmd is actually a type constructor: Cmd a denotes the type of commands that may perform *effects* before returning a value of type a. If Counter is a record type defined as

```
record Counter where
    inc :: Cmd ()
    read :: Cmd Int
```

then the type of the example shown above is given by

```
counter :: Cmd Counter
```

This says that counter is a command that, when executed, will return a value of type Counter.

The result returned from the execution of a monadic command may be bound to a variable by means of the *generator* notation. For example

```
do  c <- counter
    ...
```

means that the command counter is executed (*i.e.*, a counter object is created), and c is bound to the value returned—the interface of the new counter object.

Note that a generator expression cannot be the final command in a **do**-construct. There would be no point in using it in such a position, because the bound variable could never be referenced. Executing a command and discarding the result is written without the

left-arrow. For example, the result of invoking an asynchronous method is always the uninteresting value (), so the usual way of incrementing counter c is

```
do  c <- counter
    c.inc
```

This **do**-construct is by itself an expression: it represents a command sequence, that is, an anonymous procedure, like progn in Lisp or like a block in Smalltalk. *When this command sequence is executed,* it executes its component commands in sequence. Thus, the command counter is executed first. This in turn executes the **template** expression shown on page 17, which has the effect of creating a counter object and returning its interface record, which is bound to c. Next, the command c.inc is executed, which sends the message inc to the counter object. The result is ().

The value returned by a procedure is the value returned by its last command, so the type of the above expression is Cmd ().

Since the read method of c is a synchronous **request** that returns an Int, we can write

```
do  c <- counter
    c.inc
    c.read
```

and obtain a procedure with the type Cmd Int.

Just as for other commands, the result of executing the read method can be captured by means of the generator notation:

```
do  c <- counter
    c.inc
    v <- c.read
    return v
```

This procedure is actually equivalent to the previous one. The identifier return denotes the built-in command that, when executed, produces a result value (in this case simply v) without performing any effects. Unlike most imperative languages, however, return is not a branching construct in Timber—a return in the middle of a command sequence means only that a command without effect is executed and its result is discarded. This is pointless, but not incorrect. For example

```
do  ...
    return (v+1)
    return v
```

is simply identical to

```
do  ...
    return v
```

A procedure constructed with **do** can be named just like any other expression:

```
testCounter = do  c <- counter
                  c.inc
                  c.read
```

testCounter is thus the name of a simple procedure that, *when executed*, creates a new counter object and returns its value after it has been incremented once. The counter itself is then simply forgotten, which means that space used in its implementation can be reclaimed by the garbage collector.

A very useful procedure with a predefined name is

```
done = do return ()
```

which plays the rôle of a *null* command in Timber.

Notice the difference between the equals sign = and the generator arrow <-.

- The symbol = denotes definitional equality: it defines the name on the left to be equivalent to the expression on the right, which may be a command (as in this example) or a simple value, such as a factorial function defined on page 5; = can appear at the top level, or in **let** and **where** clauses. Since a definition can never have any effect, the order of execution of a set of definitions made with = is irrelevant, and mutual recursion is allowed.
- In contrast, the generator arrow <- can appear only inside a **do**-construct. The right hand side must be a command, that is, it must have the type Cmd a for some value type a. The effect is to *execute* the command, and to bind the resulting value to the identifier on the left hand side. A sequence of generator bindings is executed in the order written, and the values bound by the earlier bindings can be used in the later expressions, but not vice-versa.

Notice that the definition of counter in “Objects and methods” on page 16 does not by itself create an object. What it does is define counter to be a command. If and when that command is executed, an object will be created; if counter is executed three times, three new counter objects are created. Thus, the command counter is analogous to the expression CounterClass new in Smalltalk or new Counter() in Java: it is the means to make a new object.

Assignable local state

The method-forming constructs **action** and **request** are syntactically similar to procedures, but with different operational behaviors. Whereas calling a procedure means that its commands are executed by the caller, sending a message triggers execution of the commands that make up the corresponding method within the receiving object. Thus, methods have no meaning other than within an object, and the **action** and **request** keywords are accordingly not available outside the **template** syntax.

In actions and requests, as well as in procedures that occur within the scope of an object, two additional forms of commands are available: commands that obtain the values of state variables (for example the command return val in the counter object), and commands that assign new values to these variables (e.g. val := val + 1).

The use of state variables is restricted in order to preserve the purely functional semantics of expression evaluation.

- First, references to state variables may occur only within commands. For example

  ```
  template
      x := 1
  in x
  ```

  is statically illegal, since the state variable x is not visible outside the commands of a method or a procedure (*i.e.*, it can only be used inside a **do**, **action** or **request**).

- Secondly, there are no aliases in Timber, which means that state variables are not first-class values. Thus the procedure declaration

  ```
  someProc r = do r := 1
  ```

  is illegal even if someProc is applied only to integer state variables, because r is syntactically a parameter, not a state variable. Parameterization over some unknown state can instead be achieved in Timber by turning the parameter candidates into full-fledged objects.

- Thirdly, the scope of a state variable does not extend into nested objects. This makes the following example ill-formed:

  ```
  template
      x := 1
  in
      template
          y := 2
      in
          do x := 0
  ```

- Fourthly, there is a restriction that prevents other local bindings from shadowing a state variable. An expression like the following is thus disallowed:

  ```
  template
      x := 1
  in \ x -> ...
  ```

  While not necessary for preserving the purity of the language, this last restriction has the merit of making the question of assignability a simple lexical matter, as well as emphasizing the special status that state variables enjoy in Timber.

A word about overloading

Sequencing by means of the **do**-construct, and command injection (via return), are not limited to the Cmd monad. Indeed, just as in Haskell, these fundamental operations are *overloaded* and available for any type constructor that is an instance of the *type class* Monad [9, 11]. Type classes and the overloading system will not be covered in this paper, partly because this feature constitutes a virtually orthogonal complement to the subtyping system of Timber, and partly because we do not capitalize on overloading in any essential way. In particular, monadic programming in general will not be a topic of this paper.

Nevertheless, we are about to introduce one more monad that is related to Cmd by means of subtyping. We will therefore take the liberty of reusing return and the **do**-syntax for this new type constructor, even though strictly speaking this means that the overloading system must come into play behind the scenes. The same trick is also employed for the equality operator == in a few places. However, the uses of overloading that occur in this paper are all statically resolvable, so our naive presentation of the matter is intuitively quite correct. We feel that glossing over Haskell's most conspicuous type system feature in this way avoids more confusion than it creates.

The O monad

While all commands are members of the monad Cmd, commands that refer to or assign to the local state of an object belong to a richer monad O s, where s is the type of the local state. Accordingly, O s a is the type of state-sensitive commands that return results of type a. An assignment command always returns (), whereas a state-referencing command can return any type. Any procedure that contains a state-referencing command is itself a state-referencing command, and will therefore have a type O s.

The type of the local state of an object with more than one state variable is a tuple type; there is no information about the names of the state variables encoded in the type of the state. For example, consider the definitions

```
a = template
      x := 1
      f := True
in ...
```

and

```
b = template
      count := 0
      enable := False
in ...
```

The commands a and b, when executed, both generate objects with local states of type (Int,Bool).

Procedures defined within an object are *parametric* in the state on which they operate. The state of the object within which the procedure is eventually executed is, in effect, provided to it as an implicit parameter. There exists no connection at runtime between a value of some O type (a procedure or a method) and the object in which its definition is syntactically nested.

What this means is that, as long as the state types match, a procedure declared within one object can be used as a local procedure within another object. This does not constitute a loophole in Timber's object encapsulation, because the state accessed by that procedure will be the state of the caller. It remains true that the only way in which an object may affect the state of another object is by sending a message to *that object*. However, the ability to export procedures provides a way of sharing code between templates. It is very much like inheritance in class-based languages, which permits one class to re-use the code originally defined in another. In such a language, encapsulation is preserved

because the state on which the re-used code operates that of the currently executing object.

Object Identity

A sometimes controversial issue in the design of object-oriented languages is whether clients should be able to compare two object references for identity. That is, given two identifiers a and b that name objects, can a client ask whether a and b are in fact the same object?

Allowing such an identity test breaches encapsulation, because whether two possibly distinct interfaces are actually implemented by the same object is an implementation decision that may be changed and which should accordingly be hidden. However, failing to provide an efficient identity test can impose an unreasonable burden on the programmer. For a more complete discussion of these issues, see reference [6].

Timber makes the compromise of letting the programmer decide whether or not identity comparison shall be possible. Objects themselves cannot be compared, so encapsulation is preserved. However, Timber provides a special variable self, which is implicitly in scope inside every **template** expression, and which may not be shadowed. All occurrences of self have type Ref s, where s is the type of the current local state. The value of self uniquely identifies a particular object at runtime.

It should be noted that the variable self in Timber has nothing to do with the *interface* of an object (in contrast to, for example, this in C++ and Java). This is a natural consequence of the fact that a Timber object may have multiple interfaces — some objects may even generate new interfaces on demand (recall that an interface is simply a value that contains at least one method).

To facilitate straightforward comparison of arbitrary object reference values, Timber provides the primitive type ObjRef with the built-in subtype rule

```
Ref a < ObjRef   .
```

By means of this rule, all object references can be compared for equality (using the overloaded primitive ==) when considered as values of the supertype ObjRef. Timber moreover provides a predefined record type ObjIdentity, which forms a convenient base from which interface types supporting a notion of object identity can be built.

```
record ObjIdentity where
    self :: ObjRef
```

For example, suppose that we wish to define ICounter as a subtype of the counter type whose objects can be compared for identity.

```
record ICounter < ObjIdentity, Counter

iCounter :: Template ICounter
```

```
iCounter =  template
                c <- counter
            in record
                self = self
                inc = c.inc
                read = c.read
```

Now we can compare the identity of two iCounters:

```
do  c1 <- iCounter
    c2 <- iCounter
    return (c1 == c2)        -- always false
```

Expressions vs. commands

Although commands are first-class values in Timber, there is a sharp distinction between the *execution* of a command, and the *evaluation* of a command considered as a functional value. The following examples illustrate this point.

```
f :: Counter -> (Cmd (), Cmd Int)
f cnt = (cnt.inc, cnt.read)
```

The identifier f defined here is a function, not a procedure: it cannot be *executed*; it can only be applied to arguments of type Counter. The fact that the returned pair has command-valued components does not change the status of f. In particular, the occurrence of sub-expressions cnt.inc and cnt.read in the right-hand side of f does *not* imply that the methods of some counter object are invoked when evaluating applications of f. Extracting the first component of a pair returned by f is also a pure evaluation with no side-effects. However, the result in this case is a command value, which has the specific property of being *executable*.

By placing a command in the body of a procedure, the command becomes subject to execution, *whenever the procedure itself is executed*. Such a procedure is shown below.

```
do  c <- counter
    fst (f c)
```

The second line applies f to a counter object, resulting in a pair. The standard function fst extracts the first element of the pair, which is a command. This command is executed when the procedure (the **do** construct) in which it is defined is executed.

The separation between *evaluation* and *execution* of command values can be made more explicit by introducing a name for the evaluated command. This is achieved by the **let**-command, which is a purely declarative construct: as usual, the equality sign denotes definitional equality.

```
do  c <- counter
    let newCmd = fst (f c)      -- Now newCmd denotes a command
        newCmd                  -- this causes the command to be executed
```

Hence the two preceding examples are actually equivalent, and in each case a counter will be created once and incremented once. The following fragment is yet another equivalent example,

```
do   c <- counter
     let aCmd = fst (f c)      -- Now aCmd and bCmd
         bCmd = fst (f c)      -- both name the same command
     in aCmd
```

whereas the next procedure has a different operational behaviour (here the inc method of c will actually be invoked twice).

```
do   c <- counter
     let newCmd = fst (f c)
     in  newCmd
         newCmd
```

A computation that behaves like f above, but which also has the effect of incrementing the counter it receives as an argument, must be expressed as a procedure.

```
g :: Counter -> Cmd (Cmd (), Cmd Int)
g cnt = do   c.inc
             return (c.inc, c.read)
```

Note that the type system clearly separates the effectfull computation from the pure one: the result type of f is a value, whereas the result type of g is a command.

Likewise, the type system demands that computations that depend on the current state of some object be implemented as procedures. For example,

```
h :: Counter -> Int
h cnt = cnt.read * 10
```

is not type correct, since cnt.read is not an integer — it is a *command* that, *when executed*, returns an integer. If we really want to compute the result of multiplying the counter value by 10 we can write

```
h :: Counter -> Cmd Int
h cnt = do   v <- cnt.read
             return (v * 10)
```

The fact that h calls the read method of the counter is reflected in the return type of h, which is Cmd Int.

Subtyping in the O monad

We have already indicated that the Cmd and O s monads are related by subtyping. This is formally expressed as a built-in subtype rule.

```
Cmd a < O s a
```

This rule can be read as a higher-order relation: “all commands in the monad Cmd are also commands in the monad O s, for any s”.

One way of characterizing the Cmd monad is as a refinement of the O monad that represents those commands that are independent of the current local state. Timber takes this idea even further by providing three more primitive command types, which are related to the Cmd monad via the following built-in subtyping rules.

```
Template a < Cmd a
Action < Cmd ()
Request a < Cmd a
```

The intention here is to provide more precise typings for the **template**, **action**, and **request** constructs. For example, Template a is the type of a **template** that, when executed, constructs an object with an interface of type a. Thus, the type inferred for counter defined on page 17 is actually Template Counter (rather than Cmd Counter), and the types of its two methods are Action and Request Int. The record type Counter can of course be updated to take advantage of this increased precision.

```
record Counter where
    inc :: Action
    read :: Request Int
```

Unlike the refinement step of going from O s to Cmd, which actually makes more programs typeable because of the rank-2 polymorphism, the distinction between Cmd and its subtypes has mostly a documentary value. However, by turning a documentation practice into type declarations, the type system can be relied on to guarantee certain operational properties. For example, a command of type Template a cannot change the state of any existing objects when executed: object instantiation only *adds* objects to the system state. Moreover, commands of type Action or Template a are guaranteed to be deadlock-free, since a synchronous method can never possess any of these types. Note that none of these properties hold for a general command of type Cmd a.

Of the type constructors mentioned here, Cmd, Template, and Request are all covariant in their single argument. This also holds for the type O s a in case of its second argument. However, the O constructor, like all types that support both dereferencing and assignment, must be invariant in its state component. Similarly, Ref is also invariant.

The main template

So far, we have seen how to define functions and procedures, and have emphasized that procedures are executed only when some other procedure calls them. How, then, is the execution of a Timber program started?

A Timber program should have a special template called main. This template is parameterized by an *environment* that gives the program the ability to interact with the rest of the system. The type of the main template must be

```
main :: Environment -> Template Program
```

The definitions of the types Environment and Program depend on the capabilities of the particular system for which the Timber program is written. The section “Reactivity” on page 27 provides more details of these types.

When a Timber program is started, the system will apply main to an environment parameter. The resulting template is then used by the system to create an object which constitutes the system's interface to the running program. This interface is required to contain an action start, which the system executes to initialize the program.

Here is the traditional “Hello World” program in Timber:

```
main env = template
               -- no state
           in record
               start = action
                           env.putStr "Hello World!\n"
```

Concurrency

In general, execution of a Timber program is concurrent: many commands may potentially be active simultaneously. However, each Timber object behaves like a monitor: its methods execute in mutual exclusion, so at most one of its methods can be active at any given time. Since all state is encapsulated in some object, this ensures orderly update of the state.

In the following example, two contending clients send messages to a counter object. Mutual exclusion between method executions in the counter guarantees that there is no danger of simultaneous updates to the counter’s state.

```
proc cnt = template
               -- this object has no state of its own
           in record
                  doIt = action cnt.inc

f env =    do c <- counter
               p <- proc c
               p.doIt
               c.inc
               v <- c.read
               env.putStr (show v)
```

Here proc is function that returns a template (a particular kind of Cmd). The command p <- proc c (inside the **do**) parameterizes proc by the counter object c and *executes* proc c: the result is a new object with a single method called doIt. The message send p.doIt starts execution of this method, which then executes autonomously and asynchronously, because the method is an action.

Methods are not guaranteed to be executed in the order that the corresponding messages are sent. Their execution is instead scheduled subject to timing constraints, which will be discussed in Section 6. However, in the absence of explicit timing constraints, if one message send precedes (in the sense of Lamport’s “happened before” relation) another send to the same object, it is safe to assume that the corresponding methods will be exe-

cuted in the same order. This is also illustrated by the previous example, as we now explain.

In the procedure f the message inc is sent to p before the message read. There are no explicit timing annotations on the message sends c.inc and c.read. Thus, inc will be executed before read, and it is safe to assume that the value v returned from the read request is at least 1. In contrast, the send of doIt to p initiates a concurrent activity, because doIt is an action. Nothing can be said about whether the doIt action of object p will be scheduled to send its inc message before, in between, or after the two messages sent in procedure f.

Reactivity

Objects alternate between indefinitely long phases of inactivity and periods of method execution that must be finite, unless the programmer has explicitly written an infinite loop. Given a sufficiently fast processor, in many applications the method executions may be considered to be instantaneous. When used with very short duration methods, Timber then approximates Berry's *perfectly synchronous* model of computation [3].

The existence of value-returning synchronous methods does not change the fact that method executions are finite, since, assuming that the system is not in deadlock, there are no *other* commands that may block indefinitely, and hence sending a request cannot block indefinitely either. Thus, it is important that the computing environment also adheres to this reactive view, by *not* providing any operations that might block a process indefinitely.

This means, for example, that a Timber program cannot read input from a console using a blocking primitive. Instead, interactive Timber programs install *callback methods* in the computing environment, with the intention that these methods will be invoked whenever the event that they are set to handle occurs. As a consequence, Timber programs do not generally terminate when the start action of the main template returns; instead, they are considered to be alive as long as there is at least one active object or one installed callback method in the system. (Alternatively, the environment may provide a quit method that terminates the whole program).

The overall form of a Timber program is thus *not* a (potentially) infinite main loop. Instead, a Timber program defines a set of objects and binds events in the environment to message sends to those objects. When the events occur, the messages will be sent, and the corresponding methods will be scheduled for execution.

The actual shape of the interface to the computing environment must of course be allowed to vary with the type of application being constructed. The current Timber implementation supports several environment types, including TixEnv, BotEnv, and StdEnv, which model the computing environments offered by a Tk server with graph building extensions [2], a mobile Robot, and the *stdio* fragment of a Unix operating system, respectively.

As an illustration of the use of environments, let us see how a text-based Timber program might work in a minimal Unix-like computing environment:

```
record StdEnvironment where
        putStr :: String -> Action
        quit   :: Action

record StdProgram where
        start  :: Action
        char   :: Char -> Action
        signal :: Int -> Action

main :: StdEnvironment -> Template StdProgram
```

The program can send the message putStr to the environment to output strings. The system will deliver characters and signals to the program by executing the actions char and signal when a new character is typed or a signal is generated.

For an example of a more elaborate environment interface, the reader is referred to the vehicle controller discussed in Section 8.

6. Time

So far, our discussion of Timber has (intentionally) avoided the topic of time. This is conventional in the definition of programming languages; ignoring time has the great advantage of allowing conforming implementations of a language to exist on varied hardware and software platforms. Unfortunately, ignoring time makes a language unsuitable for programming real-time systems, with the result that embedded systems—almost alone in the universe of modern software—are frequently programmed in assembly language or in a way that must escape from the programming language and appeal to the primitives of a real-time operating system for all critical operations.

With Timber we attempt to find some middle ground by allowing the programmer to place *bounds* on the execution time of actions, while allowing the implementation the freedom to schedule the actions within those bounds. We use the notion of *deadline*—the latest time before which an action must complete, and *baseline*—the earliest time after which the action may commence. We call the closed interval bounded by a baseline and a deadline a *timeline*; while an action is executing the current time will normally be within the timeline for that action. It is possible to read the current time directly, but since this will vary from one execution to the next, the timeline is in practice more useful.

Specifying Time

Timber has two built-in datatypes for time: TimeInstant and TimeDuration. TimeInstant refers to a calendar date and time; TimeDuration to the interval between two TimeInstants†. Neither the precision nor the accuracy of the clock against which

†. The names *TimeDuration* and *TimeInstant* are taken from ISO 8601[10] and the XML Schema Specifications for DataTypes [5]

times are measured are dictated by the Timber language. This means that implementations are free to provide as coarse or as fine a notion of time as their applications require or their platforms permit. However, Timber does require that time is monotonic with respect to the Lamport Logical Clock [13]. That is, if an action a "happened before" an action b, then the current time observable in a must not be later than the current time observable in b. Note that, because of the finite granularity of the clock, the two times may be equal.

The datatype TimeInterval is used to represent the interval between and including two TimeInstants. The operators until, from and ending can be used to construct TimeIntervals:

```
until   :: TimeInstant    -> TimeInstant    -> TimeInterval
from    :: TimeDuration   -> TimeInstant    -> TimeInterval
lasting :: TimeInstant    -> TimeDuration -> TimeInterval
ending  :: TimeDuration   -> TimeInstant    -> TimeInterval
```

In the following, tod1 is a TimeInstant, and hours is a TimeDuration. tod2 is defined to denote a timeInstant that is one hour later than tod1:

```
tod2 = tod1 + (1 * hours)
```

The following definitions all specify the same TimeInterval, namely, the interval between tod1 and tod2.

```
i1 = tod1 `until` tod2
i2 = (1 * hours) `from` tod1
i3 = tod1 `lasting` (1 * hours)
i4 = (1 * hours) `ending` tod2
```

The operators baseline, deadline and duration can be used to examine a TimeInterval:

```
baseline :: TimeInterval   -> TimeInstant
deadline :: TimeInterval   -> TimeInstant
duration :: TimeInterval   -> TimeDuration
```

In the scope of the above let expression, the following are true:

```
baseline i1 == tod1
deadline i1 == tod2
duration i1 == (1 * hours)
```

Timelines for Actions

Every method execution in a Timber program has an associated timeline. Normally, this timeline is the same as the timeline of the method that initiated the execution; indeed, this is always the case for requests. However, for actions, it is possible to specify a different timeline, as we will see shortly.

Actions invoked by the environment are also assigned timelines. For example, the timeline for the start action is determined by the operating system command that initiates it. Normally it has a baseline representing the TimeInstant at which the program is started,

and a deadline specifying when initialization must be completed. To give another example: when the environment receives an interrupt from a sensor, it sends a message to some Timber object that initiates an action. The timeline for this action might extend from the instant that the interrupt arrives until the instant when the sensor readings are no longer guaranteed to be available in the device register.

Specifying the Timeline

If a Timber action with timeline τ sends a message initiating an action *A*, then the default timeline for *A* is also τ. The baseline for *A* can be specified to be something other than τ.baseline than by means of the construct **after** b *A*, where b is a TimeDuration. This gives the action *A* a baseline of b + τ.baseline; *A*'s deadline is τ.deadline. Similarly, the deadline for *A* can be specified by means of the construct **before** d *A*, where d is a TimeDuration; this initiates the action *A* with a deadline of d + τ.baseline. In this case *A*'s baseline is τ.baseline.

The **before** and **after** constructs give the programmer an explicit way of specifying which aspects of a reaction are time-critical. If an action *A* sends a message that initiates an action *B* in some other object, the deadline for *B* will by default be the same as that of *A* itself. However, by using the **before** command, the deadline for *B* can be changed to be later than the deadline for *A*.

Whether it is appropriate to change the deadline in this way depends entirely on the application. For example, *A* may be a time-critical reaction to a real-time event, but *B* may be a housekeeping operation that can be deferred indefinitely; in this case, *B* may be given a very much more generous (even infinite) deadline. In contrast, if completion of *B* is part of the required response to the external event, then it may be necessary to give *B* the same deadline as *A*.

By using a recursive message send that specifies a new baseline, it is possible to express *periodic* scheduling. For example, the following controller schedules itself with a period of 0.1 seconds:

```
controller = action
                do_periodic_stuff
                after (0.1 * seconds) controller
```

It is important to note that the n^{th} execution of this action will have terminated before the $(n+1)^{th}$ execution starts.

Execution Model

The model of concurrent execution used by Timber is based on the idea of the Chemical Abstract Machine [4]. The state of an executing program is envisioned as a "soup" of molecules. Sometimes these molecules react together, becoming absorbed and producing new molecules as a result.

There are two kinds of molecules in the Timber "soup": *objects* and *messages*.

Objects. Objects are always *named*. The names bear no relationship to any identifier that might be used to reference an object in the Timber program. Instead, a name should be thought of as a unique identifier that distinguishes an object from all others.

Objects can either be *active* or *inactive*. An active object is denoted o:Obj⟨C, τ, s⟩. Such an object, named o, is executing the command sequence C in response to a message sent from the object named s, using the timeline τ. An inactive object is denoted o:Obj⟨⟩.

Messages. Messages are denoted Msg⟨o, C, τ, s⟩, which amounts to a message targeted at object o, containing the command sequence C, to be executed with the timeline τ, on behalf of the invoking object s. If the message corresponds to an invoked action, we will use the special identifier _ for the invoking object.

Object creation. When an object is created by executing a template command, a new object o:Obj⟨⟩ is created, using a fresh name o. The state variables of o is initialized as described by the template, and sub-objects are recursively created. All actions and requests in the template are also *associated* with the name o, so that messages can be sent to the correct target. The interface (containing the associated actions and requests), is returned.

Action message send. When a (asynchronous) action message is sent, a new message of the form Msg⟨o, C, τ, _⟩ is created, where o is the target object associated with the action, C is the command sequence in the action, and τ is the timeline specified for the action (see "Specifying the Timeline" on page 30). The identity of the sender is irrelevant in this case, and so is denoted by _.

Request message send. When a (synchronous) request message is sent, a new message of the form Msg⟨o, C, τ, s⟩ is created, where o is the target object associated with the request, C is the command sequence in the action, and τ is the timeline of the invoking method. The invoking method is blocked, awaiting a reply from o.

Dispatching of a message. If an idle object o:Obj⟨⟩ and a message Msg⟨o, C, τ, s⟩ that targets o both exist at the same time, then the message can be *dispatched*. This means that both o and the message are consumed and are replaced by the active object o:Obj⟨C, τ, s⟩. Note that this dispatch is constrained by the *scheduling rules* outlined in the next section.

Completing an action. When an active object has finished executing an action command sequence, it is transformed into the idle object.

Completing a request. When an active object has finished executing a request command sequence, it is on the form o:Obj⟨**return** e, τ, s⟩. It will be transformed into the idle object o:Obj⟨⟩, and the object s that originally sent the request message is unblocked. The return value of the message send is the value of e.

Scheduling

In the Timber execution model, scheduling reduces to the problem of choosing which message to dispatch next. The exact scheduling algorithm is not a part of the Timber language specification. Instead, we envisage the scheduler as a "plug in component": different schedulers may be chosen to meet the needs of different applications.

However, any scheduler *must* preserve the following properties:

1. No message may be dispatched before its baseline.

2. If two messages to the same object o, A = Msg⟨o, m, τ_1, _ ⟩ and B = Msg⟨o, n, τ_2, _ ⟩ are both eligible for dispatch, and A was sent *before* B, in the sense of Lamport's "happened before" relation, then B can only be dispatched before A if
   - τ_2.baseline < τ_1.baseline, or
   - τ_2.baseline = τ_1.baseline **and** τ_2.deadline < τ_1.deadline.

The second property guarantees that the order is preserved in a sequence of message sends from one object to another, provided that all the messages have the same timeline. However, if a programmer explicitly gives a later message an earlier baseline or an earlier deadline, then the later message may be dispatched before the earlier one.

Example of Reduction Semantics

Recall our definition of the counter template:

```
counter =   template
              val := 0
            in record
              inc    = action val := val + 1
              read   = request return val
```

Suppose we have an active object o:Obj⟨C, τ, _ ⟩, where is C is the following command sequence:

```
c <- counter
     c.inc
     c.inc
     v <- c.read
     env.putStr (show v)
```

Here is how the system can evolve:

```
o:Obj⟨c <- counter
     c.inc
     c.inc
     v <- c.read
     env.putStr (show v), τ, _ ⟩
```

Unfold definition of counter, create new object, with fresh name o1. Return interface with methods associated with o1

```
o:Obj⟨c <- return (record    inc    = action⟨o1⟩ val := val + 1
                             read   = request⟨o1⟩ return val)
     c.inc
     c.inc
     v <- c.read
     env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 0]
```

Bind c to returned expression

```
o:Obj⟨let c = record   inc    = action⟨o1⟩ val := val + 1
                       read   = request⟨o1⟩ return val)
      c.inc
      c.inc
      v <- c.read
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 0]
```

Evaluate c.inc

```
o:Obj⟨let c = record   inc    = action⟨o1⟩ val := val + 1
                       read   = request⟨o1⟩ return val)
      action⟨o1⟩ val := val + 1
      c.inc
      v <- c.read
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 0]
```

Invoke the first action, creating a new message

```
o:Obj⟨let c = record   inc    = action⟨o1⟩ val := val + 1
                       read   = request⟨o1⟩ return val)
      c.inc
      v <- c.read
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 0]

Msg⟨o1,  val := val + 1,   τ,   _ ⟩
```

Dispatch the message (this is just one of many possible schedules)

```
o:Obj⟨let c = record   inc    = action⟨o1⟩ val := val + 1
                       read   = request⟨o1⟩ return val)
       c.inc
      v <- c.read
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨val := val + 1, τ, _ ⟩ [val := 0]
```

Invoke the second action (this is just one of many possible schedules)

```
o:Obj⟨let c = record   inc    = action⟨o1⟩ val := val + 1
                       read   = request⟨o1⟩ return val)
      v <- c.read
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨val := val + 1, τ, _ ⟩ [val := 0]

Msg⟨o1,  val := val + 1,  τ,   _ ⟩
```

Evaluate c.read (this is just one of many possible schedules), garbage collect c

```
o:Obj⟨v <- request⟨o1⟩ return val
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨val := val + 1, τ, _ ⟩ [val := 0]

Msg⟨o1,  val := val + 1, τ,   _ ⟩
```

Invoke request (this is just one of many possible schedules)

```
o:Obj⟨v <- ⟨blocked⟩
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨val := val + 1, τ, _ ⟩ [val := 0]

Msg⟨o1,  val := val + 1,  τ,  _ ⟩

Msg⟨o1, return val, τ, o⟩
```

Execute assignment, complete action

```
o:Obj⟨v <- ⟨blocked⟩
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 1]

Msg⟨o1,  val := val + 1, τ, _ ⟩

Msg⟨o1,  return val,  τ,  o⟩
```

Dispatch message (scheduling requirements state that this is the only possible message to dispatch for o1)

```
o:Obj⟨v <- ⟨blocked⟩
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨val := val + 1, τ, _ ⟩ [val := 1]

Msg⟨o1,  return val, τ, o⟩
```

Execute assignment, complete action

```
o:Obj⟨v <- ⟨blocked⟩
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 2]

Msg⟨o1, return val, τ, o⟩
```

Dispatch message

```
o:Obj⟨v <- ⟨blocked⟩
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨return val, τ, o⟩ [val := 2]
```

Evaluate local state variable

```
o:Obj⟨v <- ⟨blocked⟩
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨return 2, τ, o⟩ [val := 2]
```

Complete request, unblock invoking object

```
o:Obj⟨v <- return 2
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 2]
```

Bind v to returned expression, garbage collect o1

```
o:Obj⟨   let v = 2
         env.putStr (show v), τ, _ ⟩
```

Evaluate expression, garbage collect v

```
o:Obj⟨env.putStr "2", τ, _ ⟩
```

7. Additional Features

Timber also provides a number of minor, mostly syntactic extensions to the Haskell base, which we will briefly review in this section.

Extended do-syntax

The **do** -syntax of Haskell already contains an example of an expression construct lifted to a corresponding role as a command: the **let**-command, illustrated in “Expressions vs. commands” on page 23. Timber defines commands corresponding to the **if**- and **case**-expressions as well, using the following syntax.

```
do   if e then
        cmds
     else
        cmds
     if e then
        cmds
     case e of
        p1 -> cmds
        p2 -> cmds
```

In addition, Timber provides syntactic support for recursive generator bindings, and iteration.

```
do   fix x <- cmd y
         y <- cmd x
     forall i <- e do
         cmds
     while e do
         cmds
```

Array updates

To simplify programming with the primitive Array type, Timber supports a special array-update syntax for arrays declared as state variables. Assuming a is such an array, an update to a at index i with expression e can be done as follows. (The array indexing operator in Haskell is !)

```
a!i := e
```

Semantically, this form of assignment is equivalent to

```
a := a // [(i,e)]
```

where // is Haskell's pure array update operator. But apart from being intuitively simpler, the former syntax has the merit of making it clear that normal use of an encapsulated array is likely to be single-threaded, *i.e.*, implementable by destructive update. The rare cases where a is used for a purpose other than indexing become easily identifiable, and hence conservative of the array can be reserved for these occasions. Ordinary updates to a can be performed in place, which is also exactly what the array-update syntax above suggests.

Record stuffing

Record expressions may optionally be terminated by a type constructor name, as in the following examples:

```
record ..S

record a = exp; b = exp; ..S
```

These expressions utilize *record stuffing*, a syntactic device for completing record definitions with equations that just map a selector name to an identical variable already in scope. The missing selectors in such an expression are determined by the appended type constructor S, which must stand for a record type, on condition that corresponding variables are defined in the enclosing scope. So if S is a (possibly parameterized) record type with selectors a, b, and c, the two record values above are actually

```
record a = a; b = b; c = c
```

and

```
record a = exp; b = exp; c = c
```

where c, and in the first case even a and b, must already be bound. Record stuffing is most useful in conjunction with **let**-expressions, as we will see in the examples.

8. An Autonomous Vehicle Controller

We present here a complete Timber program. The example is idealized for brevity, but illustrates Timber's reactive style of programming and many of the features of the language. This example also shows the separation between the calculations performed by a program and the interactions in which it is involved. Since it is an implementation of an interrupt-driven system with parallel processes that also performs significant computation, it captures many of the characteristics of an embedded system.

The environment that this program assumes is as follows:

```
record Register where
    load    :: Cmd Int
    store   :: Int -> Cmd ()

record EmbeddedEnv where
    register_at    :: Int -> Template Register
    reset          :: Action

record EmbeddedProgram where
    start      :: Action
    interrupts :: [(Int,Action)]
```

Here is the controller program itself:

```
module AGV where

type Angle     = Float
type Speed     = (Angle,Float)
type Pos       = (Float,Float)

calcpos  :: [Angle] -> [Pos] -> Pos
regulate :: Pos -> Pos -> Speed -> Speed
room     :: [Pos]

calcpos     = undefined
regulate    = undefined
room        = undefined

-----------------------------------------------------------------

record Driver where
    new_scan    :: [Angle] -> Action
    new_path    :: [Pos] -> Action
```

```
driver :: Servo -> Template Driver
driver servo =
    template
        speed := (0.0,0.0)
        path  := repeat (0.0,0.0)
    in record
        new_scan angles = action
            let is_pos                = calcpos angles room
                should_pos:path'      = path
                speed'                = regulate is_pos should_pos speed
            speed := speed'
            path  := path'
            servo.set_speed speed'
        new_path p = action
            path := p

-------------------------------------------------------------------

record Scanner where
    detect      :: Action
    zero_cross  :: Action

tick_period  = 100*milliseconds
reg_change= 10*microseconds

scanner :: Register -> Driver -> Template Scanner
scanner angle_reg driver =
    template
        angles := [ ]
    in record
        detect = beforereg_change action
            a <- angle_reg.load
            angles := 2*pi*(fromIntegral a)/4000 : angles
    zero_cross = action
            before tick_period driver.new_scan angles
            angles := []

-------------------------------------------------------------------

record Servo where
    set_speed  :: Speed -> Action

servo :: Register -> Register -> Template Servo
servo  = undefined

-------------------------------------------------------------------

record Radio where
    incoming :: Action

radio :: Register -> Driver -> Template Radio
radio  = undefined
```

```
-----------------------------------------------------------------

main :: EmbeddedEnv -> Template EmbeddedProgram
main env =
    template
        thrust_reg    <- env.register_at 0xFFFF0001
        steer_reg     <- env.register_at 0xFFFF0002
        angle_reg     <- env.register_at 0xFFFF0003
        radio_reg     <- env.register_at 0xFFFF0004

        serv          <- servo thrust_reg steer_reg
        driv          <- driver serv
        scan          <- scanner angle_reg driv
        comm          <- radio radio_reg driv
    in record
        start = actiondone
        interrupts =  [
                          (0x80, scan.detect),
                          (0x81, scan.zero_cross),
                          (0x82, comm.incoming)
                      ]
```

Appendix: A Context-Free Grammar for Timber

Module Header

```
module   : 'module' CONID 'where' body

body     : '{' topdecls '}'
         | topdecls                          -- using layout
```

Top-level declarations

```
topdecls : topdecls ';' topdecl
         | topdecl

topdecl  : 'type' CONID tyvars '=' type
         | 'data' CONID tyvars optsubs optcs
         | 'record' CONID tyvars optsups optbs
         | 'class' CONID tyvars optsups optbs
         | 'instance' qtype optbs
         | bind

tyvars   : tyvars VARID
         | {- empty -}

optsups  : '<' types
         | '<' type
         | {- empty -}

optsubs  : '>' types
         | '>' type
         | {- empty -}
```

Datatype declarations

```
optcs     : '=' constrs
          | {- empty -}

constrs   : constrs '|' qconstr
          | qconstr

qconstr   : context '=>' constr
          | constr

constr    : constr atype
          | CONID
```

Bindings

```
optbs     : 'where' bindlist
          | {- empty -}

bindlist  : '{' binds '}'
          | binds                    -- using layout
          | '..' CONID               -- only in a record expression

binds     : binds ';' bind
          | bind

bind      : vars '::' qtype
          | pat rhs                  -- unless inside a record declaration

vars      : vars ',' var
          | var

rhs       : '=' exp
          | gdrhss
          | rhs 'where' bindlist

gdrhss    : gdrhss gdrhs
          | gdrhs

gdrhs     : '|' quals '=' exp
```

Qualified types

```
qtype     : context '=>' type
          | type

context   : '(' preds ')'
          | pred

preds     : preds ',' pred
          | pred

pred      : classpred
          | type '<' type

classpred : classpred atype
          | CONID
```

Types

```
type      : btype '->' type
          | btype
```

```
btype    : btype atype
         | atype

atype    : CONID
         | VARID
         | '[' ']'
         | '(' '->' ')'
         | '(' commas ')'
         | '(' ')'
         | '(' type ')'
         | '(' types ')'
         | '[' type ']'

types    : types ',' type
         | type  ',' type

commas   : commas ','
         | ','
```

Expressions

```
exp      : '\' apats '->' exp
         | 'let' bindlist 'in' exp
         | 'if' exp 'then' exp 'else' exp
         | 'case' exp 'of' altlist
         | 'record' bindlist
         | 'do' stmtlist
         | 'action' stmtlist
         | 'request' stmtlist
         | 'template' stmtlist 'in' exp
         | 'template' 'in' exp
         | 'after' aexp exp
         | 'before' aexp exp
         | exp '::' qtype
         | infixexp

infixexp : infixexp op infixexp
         | '-' fexp
         | fexp

fexp     : fexp aexp
         | aexp

aexp     : aexp SELID
         | bexp

bexp     : var
         | 'self'
         | con
         | lit
         | '(' ')'
         | '(' exp ')'
         | '(' exps ')'
         | '[' list ']'
         | '(' infixexp op ')'
```

```
                | '(' op infixexp ')'
                | '(' commas ')'
lit             : INT
                | RATIONAL
                | CHAR
                | STRING
```

List expressions

```
list            : {- empty -}
                | exp
                | exps
                | exp '|' quals
exps            : exps ',' exp
                | exp ',' exp
quals           : quals ',' qual
                | qual
qual            : pat '<-' exp
                | exp
                | 'let' bindlist
```

Case alternatives

```
altlist         : '{' alts '}'
                | alts                          -- using layout
alts            : alts ';' alt
                | alt
alt             : pat rhs1
rhs1            : '->' exp
                | gdrhss1
                | rhs1 'where' bindlist
gdrhss1         : gdrhss1 gdrhs1
                | gdrhs1
gdrhs1          : '|' quals '->' exp
```

Statement sequences

```
stmtlist        : '{' stmts '}'
                | stmts                         -- using layout
stmts           : stmts ';' stmt
                | stmt
stmt            : pat '<-' exp
                | exp
                | pat ':=' exp
                | 'let' bindlist
                | 'if' exp 'then' stmtlist 'else' stmtlist
                | 'if' exp 'then' stmtlist
                | 'case' exp 'of' altlist2
                | 'forall' quals 'do' stmtlist
```

```
                | 'while' exp 'do' stmtlist
                | 'fix' stmtlist
altlist2        : '{' alts2 '}'
                | alts2                                  -- using layout
alts2           : alts2 ';' alt2
                | alt2
alt2            : pat rhs2
rhs2            : '->' stmtlist
                | gdrhss2
                | rhs2 'where' bindlist
gdrhss2         : gdrhss2 gdrhs2
                | gdrhs2
gdrhs2          : '|' quals '->' stmtlist
```

Patterns

```
pat             : pat op pat
                | apats
apats           : apats apat
                | apat
apat            : '_'
                | var
                | con
                | lit
                | '-' INT
                | '-' RATIONAL
                | '(' ')'
                | '(' pat ')'
                | '(' pats ')'
                | '[' pats ']'
                | '(' commas ')'
pats            : pats ',' pat
                | pat ',' pat
```

Variables, Constructors and Operators

```
var             : VARID
                | '(' VARSYM ')'
con             : CONID
                | '(' CONSYM ')'
varop           : VARSYM
                | '`' VARID '`'
conop           : CONSYM
                | '`' CONID '`'
op              : varop
                | conop
```

Terminal symbols

Rather than providing full definitions for the terminals, we illustrate them by example.

Variable Identifiers

VARID: abc | aBC | ab_c | abc1 | ...

Constructor Identifiers.

CONID: Abc | ABC | Ab_c | Abc1 | ...

Selector Identifiers:

SELID: .abc | .aBC | .ab_c | .abc1 | ...

Variable Symbols

VARSYM: + | < | <= | ...

Constructor Symbols

CONSYM: : | :+ | :< | :<= | ...

Integers

INT: 0 | 123 | 0x123ABC | ...

Rational Numbers

RATIONAL: 0.12 | 0.12E4 | 0.12E-4 | ...

Character Constants

CHAR: 'a' | 'X' | '\n' | ...

String Constants

STRING: "abc" | "abc\n" | ...

References

1. Haskell—A Purely Functional Language. Web site, http://www.haskell.org/
2. TiX: Tk interface eXtension. Web site, http://tix.sourceforge.net/
3. Gérard Berry, *The Foundations of Esterel*, in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Editors. 1998, MIT Press.
4. Gerard Berry and Gerard Boudol. *The Chemical Abstract Machine*. In *Seventeenth annual ACM symposium on Principles of programming languages*, 1990, San Francisco, CA, USA: ACM Press, .
5. Paul V. Biron and Ashok Malhotra, XML Schema Part 2: Datatypes. Stable W3C Recommendation 02 May 2001, World Wide Web Consortium (W3C), 2001. http://www.w3.org/TR/xmlschema-2/
6. Andrew P. Black. *Object Identity.* In *Proceedings 3rd International Workshop on Object Orientation in Operating Systems*, 1993, Asheville, NC: IEEE Computer Society Press, .
7. Luca Cardelli and Peter Wegner, *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, 1985. **17**(4): pp 471-522.
8. Benedict R. Gaster. *Polymorphic Extensible Records for Haskell*. In *Haskell Workshop*, 1997, Amsterdam, The Netherlands.

9. Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones and Philip Wadler. *Type classes in Haskell*. In *5th European Symposium on Programming*, 1994, Edinburgh, Scotland: Springer Verlag, Lecture Notes in Computer Science vol. 788.
10. ISO, Representations of dates and times. 1988-06-15, International Organization for Standardization, 1988.
11. Mark P. Jones. *A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism*. In *Functional Programming and Computer Architecture*, 1993, Copenhagen, Denmark: ACM Press, .
12. Simon Peyton Jones, John Hughes, Lennart Augustsson, *et al.*, Report on the Programming Language Haskell 98: A Non-strict, Purely Functional Language. , , 1999. http://www.haskell.org
13. Leslie Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM, 1978. **21**(7): pp 558-565.
14. R. Milner, M. Tofte and R. Harper, *The Definition of Standard ML*. 1990, Cambridge, MA: MIT Press.
15. Johan Nordlander. Reactive Objects and Functional Programming [Ph.D. Dissertation]. Chalmers University of Technology, Göteborg, Sweden:1999.
16. R. K. Raj, E. D. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson and E. Jul, *Emerald: A General Purpose Programming Language.* Software—Practice & Experience, 1991. **21**(1): pp 91-118.
17. David Ungar and Randall B. Smith. *Self: The Power of Simplicity.* In *OOPSLA'87*, 1987.