Essentials of Standard ML Modules

Mads Tofte

Department of Computer Science
University of Copenhagen

Abstract. The following notes give an overview of Standard ML Modules system.[1]

Part 1 gives an introduction to ML Modules aimed at the reader who is familiar with a functional programming language but has little or no experience with ML programming.

Part 2 is a half-day practical intended to give the reader an opportunity to modify a small, but non-trivial piece of software using functors, signatures and structures.

PART 1

1 Introduction

It is now more than ten years ago that David MacQueen made his proposal for ML Modules[Mac84]. At the time, there was very little experience with large scale programming in ML. At the time the Modules were formally defined (1987-1989), there was still a certain amount of guesswork involved, still because of the limited practical experience. Today, hundreds of thousands of lines of ML later, ML programmers have a much clearer picture of what the most important aspects of the Standard ML modules are. In the experience of the author, there are certain features of the ML Modules system that are exploited again and again, while others play a strictly secondary rôle. Moreover, these essential features are actually surprisingly few in number and not hard to grasp. Finally, their scope is not limited to ML; for example, they are completely independent of the fact that ML is a strict (rather than an lazy) language. The purpose of these notes is to focus on these few essentials of ML Modules.

---

[1] By and large, these notes are consistent with *The Definition of Standard ML*[MTH90], as regards syntax, semantics and terminology. As it happens, the Definition is currently being revised, primarily in order to simplify the modules system. In these notes we concentrate on those aspects of the ML modules that will still be present in the revised language. For brevity, we refer to the old and the revised languages as SML 90 and SML 96, respectively, when the distinction matters.

The exercises in the first part can be solved without the use of a computer; the tutorial assumes that an SML 90 implementation with the Edinburgh Library preloaded is available. The ML code for Part 2 is available from the author's World-Wide Web home page.

2 Packaging Code using Structures

In programming languages the term *module* means a packaged program unit which can be combined with other modules to form a (possibly large) software system. The adjective *modular* is often used as a positive term, implying tidy design and good software hygiene.

Many functional languages already have a concept of type which is strong enough to allow the orderly organisation of *values*. For example, it is much easier to program with binary trees, if one uses recursive datatypes than if one uses pointers to represent trees. Also, function composition is a way of organising computations (each function is regarded as a computation); type checking can catch meaningless combinations of computations at compile time. Moreover, since the composition of two functions is again a function, which is itself a value, functional languages make it possible to "compute with computations" in an orderly manner. So why add language constructs for modularity?

The reason is that in a typed language one wants an orderly organisation not just of values but also of types. There is a useful distinction between a value and its type; the type is usually much simpler and reveals less detail than the value. Similarly, there is a useful distinction between a particular type (i.e., a choice of data type) and the specification that a type exist and have a certain arity, say. Both forms of information hiding are important in typed languages. In ML, values cannot contain types, so one cannot simply build a record containing a datatype together with some operations on that type. The separation of values and types makes static type checking possible. However, the price for this separation is that one needs separate language constructs for packaging types and values that belong together.

ML allows the programmer to package a collection of types and values into a single unit, called a *structure*. The corresponding notion in ADA is *package*.

Here is a structure, called *IntFn*, which implements finite maps on integers:

```
structure IntFn =
struct
 exception Apply
 type 'a intmap = int -> 'a
 fun e i = raise Apply
 fun app f x = f x
 fun extend (a,b) f i =
   if i=a then b else f i
end;
```

Having declared *IntFn*, we can refer to the types and values it contains using *qualified identifiers*. A qualified identifier starts with a structure name, then comes a period and at the end is a normal identifier.

```
val a: bool IntFn.intmap = IntFn.e
val b = IntFn.extend (3, true) a
val c = IntFn.app b 3
val d = IntFn.app b 4;
```

(The type constraint ": *bool IntFn.intmap*" isn't really needed, but it illustrates that one can refer to types as well as values.)

Exercise 1. What are the types of b, c and d?

Below is a signature which specifies the types and values of *IntFn* without revealing what they are:

```
signature INTMAP =
sig
 exception Apply
 type 'a intmap
 val e: 'a intmap
 val app: 'a intmap -> int -> 'a
 val extend: int*'a ->
         'a intmap -> 'a intmap
end;
```

To sum up, a collection of values and types can be packaged into a structure. In the above example, we had just one type in the structure; it is common to introduce several types in a single structure. A signature is a "structure type", i.e., it classifies structures in analogy with the fact that types classify values.

3 Using Signatures as Interfaces

In typed programming languages, a type checker can ensure that no value is used in a way which conflicts with its type. The same idea is clearly useful at the level of modules. For example, it should be a "type error" to place a structure M in some context where one actually needs a module which implements more operations than M does.

When one declares a structure in ML, one can get the compiler to check whether the structure matches a given signature. For example, if we assume that we have first declared signature *INTMAP* as above, but not yet *IntFn*, we can declare *IntFn* as follows:

```
structure IntFn: INTMAP =
struct
  exception Apply
  type 'a intmap = int -> 'a
  fun e i = raise Apply
  fun app f x = f x
  fun extend (a,b) f i =
    if i=a then b else f i
end;
```

The only change is in the first line: the ": *INTMAP*" is an example of a *signature constraint*; it makes the compiler check whether the declared structure really matches the signature. Roughly speaking, a structure matches a signature if it has all the types and values specified in the signature. In addition, the types in the structure must have the arities specified in the signature and the values in the structure must have the types specified in the signature. (The structure is allowed to have more values and types than the ones specified by the signature.)

If the structure does not match the signature, an error message is printed. Otherwise, the result of the constrained declaration is that the structure identifier (here *IntFn*) is bound to a structure which has *precisely* the values and types specified by the signature (here *INTMAP*). In other words, after the declaration, one can only refer to those components of the structure that are specified in the signature.

However, a signature constraint does not hide the identity of the types that appear both in the structure and in the signature. Hence, after the above declaration, one can exploit the fact that *IntFn* really is a function type, so one is allowed to write for example:

```
val x = IntFn.e 5;
```

(This will raise an exception, when evaluated, but the declaration is well-typed.) Contrast this with the politically correct:

```
val y = IntFn.app IntFn.e 5;
```

which is well-typed even if we only assume the type information which is given in the signature.

The form of matching just described is called *transparent* matching, since the true identity of types shines through the signature constraint. SML '96 also provides *opaque* matching, which results in a structure which has precisely the type information and components which are specified in the signature. It uses the keyword `:>` (read: coerced to) instead of `:`, so one can write for example:

```
structure IntFn:> INTMAP =
struct
 exception Apply
 type 'a intmap = int -> 'a
 fun e i = raise Apply
 fun app f x = f x
 fun extend (a,b) f i =
  if i=a then b else f i
end;
```

after which the declaration of x above would be illegal, whereas the declaration of y would still be legal.

4 An Analogy with Mathematics

The distinction we made in Section 2 (namely between, on the one hand, actual types and values and, on the other hand, the specification of types and values) is not in any way new. Indeed, mathematicians have been doing this sort of thing for centuries. A mathematician introduces the concept of a *group* roughly like this:

Definition 2. A *group* $(G, \bullet)$ is a set G equipped with a composition $\bullet : G \times G \rightarrow G$ which is associative, has a neutral element and satisfies that every element of G has an inverse.

Shortly after, one might find the following example:

> The integers $(\mathcal{Z}, +)$ is a group.

The point is that the definition of groups is independent of *which* set and *which* composition is chosen. To specify groups in SML, one declares:

```
signature GROUP =
sig
 type G
 val e : G
 val bullet: G * G -> G
 val inv: G -> G
end;
```

Admittedly, this specification would probably not satisfy a mathematician, since it does not specify the required properties of *e*, *bullet* and *inv*. However, the advantage of providing only relatively simple forms of specifications is that it is decidable whether a given structure matches a given signature — this is highly desirable when working with many modules and specifications. The group of integers is now declared as follows, where ~ means unary minus:

```
structure Z : GROUP =
struct
 type G = int
 val e = 0
 fun bullet(n: int, m) = n+m
 fun inv(n: int) = ~n
end;
```

5 Parameterised Modules

The reason group theory is group theory is that it applies to all groups. The mathematicians do not re-invent group theory each time a new group comes along. Using Computer Science jargon, the definition of groups is the interface to group theory. If we want to write code which works for all groups, it suffices to see how mathematicians refer to groups without considering a particular one. They simply say: "Let $(G, \bullet)$ be a group". This is a very compact way of saying several things at once. First, the statement fixes attention on a hypothetical group and gives it a name. Second, it says that, until further notice, all we may assume about $(G, \bullet)$ is that it is a group. It is an elementary logical mistake to use the members of G as integers, say, unless the set G has explicitly been constrained to be the integers.

The way to write an ML module which works for any group is to use a *functor*, e.g.,

```
functor Sq(Gr: GROUP) : GROUP =
struct
 type G = Gr.G * Gr.G
 val e =(Gr.e, Gr.e)
 fun bullet((a1,b1),(a2,b2)) =
         (Gr.bullet(a1,a2),
          Gr.bullet(b1,b2))
 fun inv(a,b) = (Gr.inv a, Gr.inv b)
end;
```

Here *Sq* is the name of the functor, *Gr* is the *formal parameter*, the first occurrence of *GROUP* is the *parameter signature*, the rightmost occurrence of *GROUP* is the *result* signature and the structure expression `struct` ... `end` is the *body* of the functor. Inside the body of the functor, all we may assume about structure *Gr* is that it matches the parameter signature. The scope of the specification of Gr is the result signature and the functor body. So in general, a functor declaration

$$\texttt{functor}\ f\ (X\texttt{:}\ \Sigma)\ \texttt{:}\ \Sigma'\ \texttt{=}\ body$$

is the ML programmer's way of saying: "let X be a structure which matches Σ".

If we want to write a module which works only for groups over the integers we have to constrain the type $Gr.G$ to *int* and this has to be done "up front", when we introduce Gr as a formal parameter (i.e., the body of the functor is not allowed to impose type equalities which are not specified in the parameter signatures). In SML 90 one uses a *type sharing constraint* in the signature:

```
functor Try(Gr:
       sig
         type G
         sharing type G = int
         val e: G
         val bullet: G*G->G
         val inv: G -> G
       end) =
struct
  val x = Gr.inv(Gr.bullet(7, 9))
end;
```

In SML 96 one can express the same thing more briefly using a `where type` qualifier on the signature *GROUP*:

```
functor Try(Gr: GROUP
             where type G = int) =
struct
  val x = Gr.inv(Gr.bullet(7, 9))
end;
```

or by using a *type abbreviation* in the signature:

```
functor Try(Gr:
       sig
         type G = int
         val e: G
         val bullet: G*G->G
         val inv: G -> G
       end) =
struct
  val x = Gr.inv(Gr.bullet(7, 9))
end;
```

Since the above functors are all closed — in the sense that they contain no free identifiers apart from identifiers which are available initially (e.g., *int* and $+$) — it is possible to compile the functors. When a functor has been successfully compiled, one knows that the body of the functor is well-typed assuming only what the parameter signature reveals about the parameter. Thus the parameter signature is not merely a comment about what structures the functor needs; it is a guarantee that whenever one provides an actual structure that matches the parameter signature, one can combine the functor and the argument structure without violating the type soundness of the functor body.

The result signature in a functor declaration is optional. Also, in SML 96 one can choose between specifying the result signature with opaque and transparent matching. (SML 90 provides only transparent matching.)

6 Functor Application

The way one uses a functor is to apply it to an actual argument which matches the parameter signature, e.g.,

```
structure S = Try(Z)
```

Hence combining modules is akin to combination (i.e., application) in the λ-calculus: a functor can be regarded as a map from structures to structures. In a functor application ($Try(Z)$) it is first checked that the argument structure (Z) matches the parameter signature ($GROUP$) of the functor (Try). If the match fails, an error message is printed. Otherwise, the body of the functor is evaluated, resulting in a structure. In our example, this structure is then bound to a structure identifier (S) but that is not part of the functor application *per se*.

Type information is propagated through functor application. For example, consider the application

```
structure SqZ = Sq(Z);
```

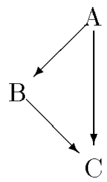
After the declaration we have $SqZ.G = int * int$, obtained as the result of simplifying the declaration `type` $G = Gr.G * Gr.G$ (which is part of the body of Sq), using that $Gr = Z$.

If the functor has been declared using an opaque result signature, the result structure will only have the type equalities which are specified in the result signature. Thus the equality $SqZ.G = int * int$ would not hold if we had used `:>` instead of `:` in the declaration of Sq. If one prefers using opaque signature constraints, one can retrieve the equality by imposing a `where type` qualification on the result signature when the functor is declared:

```
functor Sq(Gr: GROUP) :> GROUP
   where type G = Gr.G * Gr.G  =
struct
 type G = Gr.G * Gr.G
 val e =(Gr.e, Gr.e)
 fun bullet((a1,b1),(a2,b2)) =
      (Gr.bullet(a1,a2),
       Gr.bullet(b1,b2))
 fun inv(a,b) = (Gr.inv a, Gr.inv b)
end;
```

7 Building Systems

Suppose we want to create a system consisting of three structures, A, B and C, where B refers to A and C refers to both A and B. The situation can be drawn as follows:



Suppose that A, B and C have to match signatures *SIGA*, *SIGB* and *SIGC*, respectively. The simplest way to construct the system is to have three structure declarations after each other:

```
structure A: SIGA = strexp_A ;
structure B: SIGB = strexp_B ;
structure C: SIGC = strexp_C ;
```

where $strexp_A$, $strexp_B$ and $strexp_C$ are appropriate structure expressions, such that $strexp_B$ contains free occurrences of qualified identifiers starting with A and $strexp_C$ contains free occurrences of qualified identifiers starting with A or B. However, this organisation does not give a clear picture of the dependencies between the three modules. (To see whether C depends on A, one has to scan the entire declaration of C.)

To make the dependencies explicit (and to facilitate separate compilation) one can use functors instead:

```
functor mkA() = strexp_A;

functor mkB(A: SIGA): SIGB =
  strexp_B;

functor mkC(
  structure A: SIGA
  structure B: SIGB):SIGC =
  strexp_C;

structure A = mkA();
structure B = mkB(A);
structure C = mkC(
  structure A = A
  structure B = B);
```

Incidentally, this example illustrates how one writes nullary functors and functors with more than one structure parameter; in the latter case, one has to put the keyword `structure` in front of each structure parameter, and this is repeated when the functor is applied.

The signature *SIGB* may specify a type which really stems from *SIGA* (an example will be given below). It may then be necessary for *mkC* to assume that the two types *A.t* or *B.t* are actually the same type. For example, consider:[2]

```
signature SIGA =
  sig
    type t
    val mk: int-> t
    val p: t*t-> t
  end;

signature SIGB =
  sig
    type b
    val b0: b
    type t
    val f: b -> t
  end;
```

[2] In SML 96, the example has to be modified slightly, since `chr` and `ord` are changing type.

```
signature SIGC =
 sig
  type t
  val test: t
 end;

functor mkA(): SIGA =
struct
 type t = string
 fun mk(i:int):string =
     chr((i + ord "a") mod 128)
 fun p(n,m:string) = n^m  (* ^ means string concatenation *)
end;

functor mkB(A: SIGA): SIGB =
struct
 type b = string
 val b0 = "abc"
 type t = A.t
 fun f(s:string) = A.mk(size s)
end;

functor mkC(
 structure A: SIGA
 structure B: SIGB):SIGC =
struct
 type t = A.t
 val test = A.p(A.mk 16,A.p(A.mk 4,B.f(B.b0)))
end;

structure A = mkA();
structure B = mkB(A);
structure C = mkC(
 structure A = A
 structure B = B);
```

The declarations up to and including *mkB* are all fine; but the declaration of *mkC* is ill-typed. Indeed, the ML Kit complains:

```
       A.p(A.mk 16,A.p(A.mk 4,B.f(B.b0)))
                   ^^^^^^^^^^^^^^^^^^^^

Type clash,
   operand suggests operator type: t * t->t
   but I found operator type:      t * t<162>->t
```

The mysterious `<162>` in the last line indicates that the type differs from the corresponding type in the line above. Indeed, `B.f(B.b0)` has type $B.t$ and `A.mk 4` has type $A.t$ and nowhere did we state that those two types be the same (as is required by the type of $A.p$). In some cases such a type error is an indication that the functor is wrong, i.e., that one has confused two types. But in this case, we really want to say that $A.t$ and $B.t$ are the same type, which we achieve by inserting a type sharing constraint in the start of *mkC*:

```
functor mkC(
  structure A: SIGA
  structure B: SIGB
  sharing type A.t = B.t) : SIGC =
   ··· as before ···
```

After this correction, the declaration of *mkC* is well-typed. Moreover, the application of *mkC* is well-typed: it is automatically checked that the sharing constraint is satisfied, which it is with $A.t = B.t = string$. The result is a system which consists of three structures, as depicted earlier.

Exercise 3. What is the value of $C.test$?

The preceding examples (excluding the SML 96 examples) can be found in the file `examples.sml` . To run them, start an ML session in the same directory as the examples file. Then type: `use "examples.sml";`.

PART 2: PRACTICAL

8 Implementing a Polymorphic Type-Checker

The purpose of this practical is to allow you to work through a slightly larger example of program development using ML modules. You are given a collection of modules that implement a type checker and interpreter for Mini ML, a tiny subset of the SML Core language.

The system can be executed and you can modify and extend it provided you have access to an implementation and to the files listed in Appendix B. We provide a parse functor which can parse a Mini ML source expression (represented as a string) into an abstract syntax tree. The rest of the interpreter works on abstract syntax trees. Unlike most real ML systems, the Mini ML system is an interpreted system. Your job will be to work on the polymorphic type checker.

Here is the grammar for Mini ML:

---

exp	::=	*exp* `+` *exp*
		exp `-` *exp*
		exp `*` *exp*
		`true`
		`false`
		exp `=` *exp*
		`if` *exp* `then` *exp* `else` *exp*
		exp `::` *exp*
		`[` exp_1 `,` $\cdots$ `,` exp_n `]` $(n \geq 0)$
		`let` x `=` *exp* `in` *exp* `end`
		`let rec` x `=` *exp* `in` *exp* `end`
		x
		`fn` x `=>` *exp*
		exp `(` *exp* `)` (function application)
		n (natural numbers)
		`(` *exp* `)`

The abstract syntax of Mini ML is defined as a datatype in the signature `EXPRESSION`.

Exercise 4. Find and read this signature. What is the constructor corresponding to `let` expressions?

The interpreter uses a *typechecker* to check the validity of input expressions and an *evaluator* to evaluate them. Initially, the typechecker and evaluator handle only a tiny subset of Mini ML.

The typechecker and the evaluator can be developed independently as long as you do not change the signatures. The development of the typechecker and the evaluator need not be in step. You can disable either by assigning false to one of the references `tc` and `eval`.

The source of the bare interpreter is in Appendix A. An overview of how to run the systems is provided in Appendix B.

Exercise 5. Find and read the signature of the interpreter (it is called *INTERPRETER*).

We program with signatures and functors only. After the signatures, which we shall not yet study, the first functor is the interpreter itself.

Exercise 6. Find this functor. Find the application of *Ty.prType*. Find it's type. What do you think *Ty.prType* is supposed to do? What is the type of *abstsyn*? What do you think the evaluator is supposed to do when asked to evaluate something which has not yet been implemented?

We shall now describe Version 1, the bare typechecker, and then proceed to the extensions.

9 Version 1: The bare Typechecker

The first version is just able to type check integer constants and `+`. As signature *TYPE* reveals, the type *Type* of types is abstract (in the sense that the constructors are hidden), but there are functions we can use to build basic types and decompose them. *unTypeInt* is one of the latter; it is supposed to raise exception *Type* if applied to any Mini ML type different from the Mini ML integer type.[3] This is a common way of hiding implementation details and it might be helpful to take a look at *functor Type*, which can produce a structure which matches the signature *Type*. [4]

As revealed by the signature *TYPECHECKER*, the typechecker is going to depend on the abstract syntax and a *Type* structure. Notice that it is possible to specify structures in signatures as well as values and types.[5] Similarly, it is possible to declare structures inside structures; such structures are called *substructures*.[6] As you can see from the declaration of functor *TypeChecker*, all the typechecker knows about the implementation of types is what is specified by the signature *TYPE*. This allows us to experiment with the implementation of types to obtain greater efficiency without changing the typechecker, as we shall see in the later stages.

[3] In SML it is legal to use the same identifier as an exception constructor and a type constructor — the position of the identifier occurrence uniquely determines the identifier class.

[4] It is also legal to use the same identifier as a signature identifier, a functor identifier and a structure identifier — the position of the identifier occurrence uniquely determines the identifier class.

[5] However, it is not possible to specify functors or signatures in signatures.

[6] However, it is not possible to declare functors or signatures inside structures.

Exercise 7. Functor *TypeChecker* is hostile to any expression which is not an integer constant or a sum expression. Modify the typechecker to handle `true`, `false`, and multiplication of integers. Make sure the revised functor compiles and runs. Assuming that your revised version of Appendix A is stored in file `myversion1.sml`, type:

```
map use ["myversion1.sml", "parser.sml", "build1.sml"];
```

Once the parser has been compiled once, you can omit it from the list. However, you have to compile the build file after each modification of your code, since the build file contains all the functor applications that build the system.

10 Version 2: Adding lists and polymorphism

The first extension is to implement the type checking of lists. In Version 1 the type of an expression could be inferred either directly (as in the case of `true` and `false`), or from the type of the subexpressions (as in the case of the arithmetic operations). When we introduce list, this is no longer the case. For example, consider the expression

```
if ([] = [9]) then 5 else 7
```

Suppose we want to type check `([] = [9])` by first type checking the left subexpression `[]`, then the right subexpression `[9]` and finally checking that the left and right-hand sides are of the same type before returning the type `bool`. The problem now is that when we try to type check `[]` we cannot know that this empty list is supposed to be an integer list. The typechecker therefore just ascribes the type `'a list` to `[]`, where `'a` is a *(Mini ML) type variable.* The `[9]` of course turns out to be an `int list`. The typechecker now unifies the two types `'a list` and `int list` resulting in the substitution that maps `'a` to `int`. Hence the type of the expression `[]` depends not just on the expression itself, but also on the context of the expression. The context can force the type inferred for the expression to become more specific.

To implement all this, we first extend the *TYPE* signature and introduce a new signature, *UNIFY*, as shown in Figure 1.

The nice thing is that we can extend the typechecker without knowing anything about the inner workings of unification, simply by including a formal parameter of signature *UNIFY* in the typechecker functor. The complete functor is in the file `version2.sml`, but the most important bits are shown in Figure 2.

Here we see a new form of sharing constraint, namely sharing between structures. In SML 90 this specifies that when the functor is applied to actual structures *Ty* and *Unify*, it must be the case that *Ty* is the same substructure as the *Type*-substructure of *Unify*. This of course implies that types that are specified in both *Ty* and *Unify.Type* are shared as well, e.g., we have the type equality *Ty.Type* = *Unify.Type.Type*. In SML 96, structure sharing has a weaker semantics: there is no notion of identity of structure; structure sharing constraints are still allowed, but they just abbreviate a sequence of type sharing constraints.

```
signature TYPE =
  sig
    eqtype tyvar
    val freshTyvar: unit -> tyvar
    (*... components omitted ... *)
    val mkTypeTyvar: tyvar -> Type
       and unTypeTyvar: Type -> tyvar

    val mkTypeList: Type -> Type
       and unTypeList: Type -> Type


    type subst
    val Id: subst
     (*the identify substitution; *)
    val mkSubst: tyvar*Type -> subst
     (*make singleton substitution; *)
    val on : subst * Type -> Type
     (*application *)

    val prType: Type->string
     (*printing *)
  end


signature UNIFY=
  sig
    structure Type: TYPE
    exception NotImplemented of string
    exception Unify
    val unify: Type.Type * Type.Type ->
            Type.subst
  end;
```

Fig. 1. Signatures *TYPE* and *UNIFY*

We also have to extend the *Type* functor to meet the enriched *TYPE* signature, see Figure 3.

Exercise 8. Extend the typechecker of Version 2 to handle equality.

```
functor TypeChecker
  ( (*... *)
   structure Ty: TYPE
   structure Unify: UNIFY
    sharing Unify.Type = Ty
  )=
struct
  infix on
  val (op on) = Ty.on
  (*... *)

  fun tc (exp: Ex.Expression): Ty.Type =
   (case exp of
      (*... *)
    | Ex.LISTexpr [] =>
        let val new = Ty.freshTyvar()
         in Ty.mkTypeList(
              Ty.mkTypeTyvar new)
        end
    | Ex.CONSexpr(e1,e2) =>
      let
        val t1 = tc e1
        val t2 = tc e2
        val new = Ty.freshTyvar ()
        val newt= Ty.mkTypeTyvar new
        val t2' = Ty.mkTypeList newt
        val S1 =
             Unify.unify(t2, t2')
             handle Unify.Unify =>
             raise TypeError(e2,
               "expected list type")

        val S2 =
             Unify.unify(S1 on newt,
                         S1 on t1)
             handle Unify.Unify =>
             raise TypeError(exp,
  "element and list have different types")
      in S2 on (S1 on t2)
      end

   )handle Unify.NotImplemented msg =>
    raise NotImplemented msg

end; (*TypeChecker*)
```

Fig. 2. The *TypeChecker* functor

```
functor Type():TYPE =
struct
  type tyvar = int
  val freshTyvar =
     let val r= ref 0
     in fn()=>(r:= !r +1; !r)
     end
  datatype Type = INT
              | BOOL
              | LIST of Type
              | TYVAR of tyvar
  fun mkTypeTyvar tv = TYVAR tv
  and unTypeTyvar(TYVAR tv) = tv
    | unTypeTyvar _ = raise Type
  fun mkTypeList(t)=LIST t
  and unTypeList(LIST t)= t
    | unTypeList(_)= raise Type

  type subst = Type -> Type

  fun Id x = x

  fun mkSubst(tv,ty)=
   let
     fun su(TYVAR tv')=
          if tv=tv' then ty
          else TYVAR tv'
       | su(INT) = INT
       | su(BOOL)= BOOL
       | su(LIST ty') =
          LIST (su ty')
   in su
   end

  fun on(S,t)= S(t)

  fun prType = (*... *)
end;
```

Fig. 3. The *Type* functor

11 Version 3: A different implementation of types

Version 3 arises from Version 2 by replacing the *Type* functor by a different implementation of types. Instead of representing substitutions as functions, Version 3 implements type variables by references (pointers) so that it can perform substitutions very efficiently, by assignments. Here is an outline of the code:[7]

```
functor ImpType():TYPE =
struct
  datatype 'a option =
    NONE
  | SOME of 'a
  datatype Type =
    INT | BOOL
  | LIST of Type
  | TYVAR of tyvar
  withtype tyvar =
    Type option ref
  fun freshTyvar() = ref (NONE)
  exception Type
  fun mkTypeInt() = INT
  and unTypeInt(INT)=()
    | (*... *)
    | unTypeInt(TYVAR(ref(SOME t)))=
          unTypeInt t
    | unTypeInt _ = raise Type
  (*...*)
  type subst = unit
  val Id = ();
  exception MkSubst;
  fun mkSubst(tv,ty)=
     case tv of
       ref(NONE) => tv:= (SOME ty)
     | ref(SOME t) => raise MkSubst
  fun on(S,t)= t
  fun prType = (*... *)
end;
```

Exercise 9. You will find the *prType* operation in *ImpType* in Version 3 rather unsatisfactory; make modifications to correct this. (Hint: do not change anything but the functor.)

[7] The `withtype` construct declares a type abbreviation within a `datatype` declaration.

12 Version 4: Introducing variables and `let`

We now extend Version 3 by implementing the type checking of `let` expressions and of identifiers.

The typechecker function `tc` now has to take two arguments,

$$tc(TE,\ \ e)$$

where e is an expression and TE is a *type environment*, which maps variables occurring free in e to *type schemes*. The definition of what a type scheme is will be given below; for now it suffices to know that every type can be regarded as a type scheme.

To take an example, if TE maps `x` to `int` and `y` to `int`, then tc will deduce the Mini ML type `int` for the expression `x+y`. However, if TE mapped `y` to `bool`, there would be a type error.

The fact that we can bind variables to expressions whose types have been inferred to contain type variables means that we get type variables in the type environment. For instance, to type check

```
let x = [] in 4 :: x end
```

we first check `[]` yielding the type `'a1 list`, say. Then we bind `x` to the type scheme ∀`'a1.'a1 list`. Here the binding ∀`'a1` of `'a1` indicates that when we look up the the type of `x` in the type environment, we return a type obtained from the type scheme ∀`'a1.'a1 list` by instantiating the bound variables (here just `'a1`) by fresh type variables. In our example, when we look up `x` in the type environment during the checking of `4 :: x`, we instantiate `'a1` to a fresh type variable `'a2`, say, yielding the type `'a2 list` for `x`. Thus we get to unify `int list` against `'a2 list`, yielding the substitution of `int` for `'a2`.

Throughout the body of the `let`, `x` will be bound to ∀`'a1.'a1 list` in the type environment. Since we take a fresh instance of this type scheme each time we look up `x`, we can use `x` both as an `int list` and as an `int list list`, say:

```
let x = [] in   (4::x)::x end
```

Exercise 10. Assuming that you instantiate the bound `'a1` to `'a3` when you meet the last occurrence of `x`, what two types should be unified, and what is the resulting substitution on `'a3` ?

In ML, a type scheme always takes the form $\forall\, \alpha_1 \cdots \alpha_n.\tau$, $(n \geq 0)$, where $\alpha_1, \ldots, \alpha_n$ are type variables and τ is a type not containing quantifiers. In the fragment of Mini ML considered so far, all type schemes inferred by the algorithm will be closed (i.e., any type variable occurring in τ is amongst the $\alpha_1, \ldots, \alpha_n$), but when one introduces functions and application, this no longer is the case.

Exercise 11. Extend the type checker (Version 4) to handle conditionals and equality.

Exercise 12 For the extra keen. Extend Version 4 to cope with lambda abstraction (`fn`) and application. First, you have to introduce arrow types with constructors and destructors. Then you have to change the type of `close` so that it takes two arguments, namely a type environment and a type. It should return the type scheme that is obtained by quantifying all the type variables that occur in the type but do not occur free in the type environment.

Then you can modify the type checker. When you type check a lambda abstraction, you just bind the formal parameter to the trivial type scheme which is just a fresh type variable (no quantified variables). Thus the type environment can now contain type schemes with free type variables.

An application `tc(TE,e)` now yields two arguments, namely a type t and a substitution S; the idea is that if you apply the substitution S to the type environment `TE`, which now can contain free type variables, the expression `e` has the type t. When an expression consists of more than one subexpression, the type environment gradually becomes more and more specific by applying the substitutions produced by the checking of the subexpressions one by one. Moreover, the substitution returned from the whole expression is the composition of these individual substitutions. (You have to extend the *TYPE* signature (and the *Type* functor) with composition of substitutions.

Finally, you can extend the unification algorithm to cope with arrow types. (This will also use composition of substitutions.)

Exercise 13. Finally, extend type type checker (Version 4) to handle recursive functions. In `let rec` f `=` e_1 `in` e_2 `end`, e_1 must be a lambda abstraction and the typing rule is

$$\frac{\begin{array}{c} TE + \{f \mapsto \tau\} \vdash e_1 : \tau \\ TE + \{f \mapsto \texttt{close}(TE, \tau)\} \vdash e_2 : \tau' \end{array}}{TE \vdash \texttt{let rec } f \texttt{ = } e_1 \texttt{ in } e_2 \texttt{ end} : \tau'}$$

13 Acknowledgements

The parser and evaluator are due to Nick Rothwell.

14 Further Reading

The Definition of Standard ML[MTH90] defines Standard ML formally. It is accompanied by a Commentary[MT91]. Milner's report on the Core Language[Mil84], MacQueen's modules proposal[Mac84] and Harper's I/O proposal were unified in[RHM86].

Several books on Computer Programming, using Standard ML as a programming language, are available[kW87,Rea89,Pau91,Sta92,CMP93]. In addition, there are medium-length introductions[Har86,Tof89].

Compilation techniques are treated by Appel[App92]. In this note we have used bits of The Edinburgh Standard ML Library[Ber91].

There is a large body of research papers related to ML, none of which we will cite on this occasion.

References

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[Ber91] Dave Berry. The Edinburgh SML Library. Technical Report ECS-LFCS-91-148, Laboratory for Foundations of Computer Science, Department of Computer Science, Edinburgh University, April 1991.

[CMP93] Chris Clarck Colin Myers and Ellen Poon. *Programming with Standard ML.* Prentice Hall, 1993.

[Har86] Robert Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, Dept. of Computer Science, University of Edinburgh, 1986.

[kW87] Åke Wikström. *Functional Programming Using Standard ML.* Series in Computer Science. Prentice Hall, 1987.

[Mac84] D. MacQueen. Modules for Standard ML. In *Conf. Rec. of the 1984 ACM Symp. on LISP and Functional Programming*, pages 198–207, Aug. 1984.

[Mil84] Robin Milner. The Standard ML Core language. Technical Report CSR-168-84, Dept. of Computer Science, University Of Edinburgh, October 1984. Also in[RHM86].

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Pau91] Laurence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Rea89] C. Reade. *Elements of Functional Programming.* Addison-Wesley, 1989.

[RHM86] David MacQueen Robert Harper and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Dept. of Computer Science, University Of Edinburgh, March 1986.

[Sta92] Ryan Stansifer. *ML Primer.* Prentice Hall, 1992.

[Tof89] Mads Tofte. Four lectures on Standard ML. LFCS Report Series ECS-LFCS-89-73, Laboratory for Foundations of Computer Science, Department of Computer Science, Edinburgh University, Mayfield Rd., EH9 3JZ Edinburgh, U.K., March 1989.

Appendix A: The bare Interpreter (Version 1)

```
(* interp1.sml:   Mini ML interpreter, VERSION 1 *)

signature INTERPRETER=
   sig
      val interpret: string -> string
      val eval: bool ref
      and tc  : bool ref
   end;

                  (* syntax *)

signature EXPRESSION =
   sig
      datatype Expression =
         SUMexpr of Expression * Expression   |
         DIFFexpr of Expression * Expression   |
         PRODexpr of Expression * Expression   |
         BOOLexpr of bool   |
         EQexpr of Expression * Expression   |
         CONDexpr of Expression * Expression * Expression   |
         CONSexpr of Expression * Expression   |
         LISTexpr of Expression list   |
         DECLexpr of string * Expression * Expression   |
         RECDECLexpr of string * Expression * Expression   |
         IDENTexpr of string   |
         LAMBDAexpr of string * Expression   |
         APPLexpr of Expression * Expression   |
         NUMBERexpr of int
   end


              (* parsing *)

signature PARSER =
   sig
      structure E: EXPRESSION

      exception Lexical of string
      exception Syntax of string

      val parse: string -> E.Expression
   end
```

```
                        (* environments *)

signature ENVIRONMENT =
   sig
      type 'object Environment

      exception Retrieve of string

      val emptyEnv: 'object Environment
      val declare: string * 'object * 'object Environment ->
                                'object Environment
      val retrieve: string * 'object Environment -> 'object
   end

                        (* evaluation *)
signature VALUE =
   sig
      type Value
      exception Value

      val mkValueNumber: int -> Value
          and unValueNumber: Value -> int

      val mkValueBool: bool -> Value
          and unValueBool: Value -> bool

      val ValueNil: Value
      val mkValueCons: Value * Value -> Value
          and unValueHead: Value -> Value
          and unValueTail: Value -> Value

      val eqValue: Value * Value -> bool
      val printValue: Value -> string
   end


signature EVALUATOR =
   sig
      structure Exp: EXPRESSION
      structure Val: VALUE
      exception Unimplemented
      val evaluate: Exp.Expression -> Val.Value
   end

                  (* type checking *)
```

```
signature TYPE =
   sig
      type Type

(*constructors and decstructors*)
      exception Type
      val mkTypeInt: unit -> Type
          and unTypeInt: Type -> unit

      val mkTypeBool: unit -> Type
          and unTypeBool: Type -> unit

      val prType: Type->string
   end



signature TYPECHECKER =
   sig
      structure Exp: EXPRESSION
      structure Type: TYPE
      exception NotImplemented of string
      exception TypeError of Exp.Expression * string
      val typecheck: Exp.Expression -> Type.Type
   end;

                  (* the interpreter*)

functor Interpreter
   (structure Ty: TYPE
    structure Value : VALUE
    structure Parser: PARSER
    structure TyCh: TYPECHECKER
    structure Evaluator:EVALUATOR
      sharing Parser.E = TyCh.Exp = Evaluator.Exp
          and TyCh.Type = Ty
          and Evaluator.Val = Value
   ): INTERPRETER=

struct
  val eval= ref false   (* toggle for evaluation *)
  and tc  = ref true    (* toggle for type checking *)
  fun interpret(str)=
    let val abstsyn= Parser.parse str
        val typestr= if !tc then Ty.prType(TyCh.typecheck abstsyn)
```

```
                    else "(disabled)"
             val valuestr= if !eval then
                              Value.printValue(Evaluator.evaluate abstsyn)
                           else "(disabled)"

         in  valuestr ^ " : " ^ typestr
         end
         handle Evaluator.Unimplemented =>
                    "Evaluator not fully implemented"
              | TyCh.NotImplemented msg =>
                    "Type Checker not fully implemented " ^ msg
              | Value.Value   => "Run-time error"
              | Parser.Syntax msg => "Syntax Error: " ^ msg
              | Parser.Lexical msg=> "Lexical Error: " ^ msg
              | TyCh.TypeError(_,msg)=> "Type Error: " ^ msg
end;

                    (* the evaluator *)

functor Evaluator
  (structure Expression: EXPRESSION
   structure Value: VALUE):EVALUATOR=

   struct
      structure Exp= Expression
      structure Val= Value
      exception Unimplemented

      local
         open Expression Value
         fun evaluate exp =
            case exp
              of BOOLexpr b => mkValueBool b
               | NUMBERexpr i => mkValueNumber i
               | SUMexpr(e1, e2) =>
                    let val e1' = evaluate e1
                        val e2' = evaluate e2
                    in
                       mkValueNumber(unValueNumber e1'
                                   + unValueNumber e2')
                    end

               | DIFFexpr(e1, e2) =>
                    let val e1' = evaluate e1
                        val e2' = evaluate e2
```

```
                    in
                       mkValueNumber(unValueNumber e1'
                                   - unValueNumber e2')
                    end

               | PRODexpr(e1, e2) =>
                    let val e1' = evaluate e1
                        val e2' = evaluate e2
                    in
                       mkValueNumber(unValueNumber e1'
                                   * unValueNumber e2')
                    end

               | EQexpr _ => raise Unimplemented
               | CONDexpr _ => raise Unimplemented
               | CONSexpr _ => raise Unimplemented
               | LISTexpr _ => raise Unimplemented
               | DECLexpr _ => raise Unimplemented
               | RECDECLexpr _ => raise Unimplemented
               | IDENTexpr _ => raise Unimplemented
               | LAMBDAexpr _ => raise Unimplemented
               | APPLexpr _ => raise Unimplemented

      in
         val evaluate = evaluate
      end
   end;

                        (* the type checker *)

functor TypeChecker
  (structure Ex: EXPRESSION
   structure Ty: TYPE)=
struct
  structure Exp = Ex
  structure Type = Ty
  exception NotImplemented of string
  exception TypeError of Ex.Expression * string

  fun tc (exp: Ex.Expression): Ty.Type =
    case exp of
      Ex.BOOLexpr b => raise NotImplemented "(bool const)"
    | Ex.NUMBERexpr _ => Ty.mkTypeInt()
    | Ex.SUMexpr(e1,e2)  => checkIntBin(e1,e2)
    | Ex.DIFFexpr(e1,e2) => raise NotImplemented "(minus)"
```

```
    | Ex.PRODexpr(e1,e2) => raise NotImplemented "(multiplication)"
    | Ex.LISTexpr _ => raise NotImplemented "(lists)"
    | Ex.CONSexpr _ => raise NotImplemented "(lists)"
    | Ex.EQexpr _ => raise NotImplemented "(equality)"
    | Ex.CONDexpr _ => raise NotImplemented "(conditional)"
    | Ex.DECLexpr _ => raise NotImplemented "(declaration)"
    | Ex.RECDECLexpr _ => raise NotImplemented "(rec decl)"
    | Ex.IDENTexpr _   => raise NotImplemented "(identifier)"
    | Ex.LAMBDAexpr _  => raise NotImplemented "(function)"
    | Ex.APPLexpr _ => raise NotImplemented    "(application)"


  and checkIntBin(e1,e2) =
    let val t1 = tc e1
        val _  = Ty.unTypeInt t1
                 handle Ty.Type=> raise TypeError(e1,
                                       "expected int")
        val t2 = tc e2
        val _  = Ty.unTypeInt t2
                 handle Ty.Type=> raise TypeError(e2,
                                       "expected int")
     in Ty.mkTypeInt()
    end;

  val typecheck = tc

end; (*TypeChecker*)



                     (* the basics -- nullary functors *)

functor Type():TYPE =
struct
  datatype Type = INT
                | BOOL

  exception Type

  fun mkTypeInt() = INT
  and unTypeInt(INT)=()
    | unTypeInt(_)= raise Type

  fun mkTypeBool() = BOOL
  and unTypeBool(BOOL)=()
```

```
    | unTypeBool(_)= raise Type

  fun prType INT = "int"
  |   prType BOOL= "bool"
end;



functor Expression(): EXPRESSION =
   struct
      type 'a pair = 'a * 'a

      datatype Expression =
         SUMexpr of Expression pair   |
         DIFFexpr of Expression pair   |
         PRODexpr of Expression pair   |
         BOOLexpr of bool   |
         EQexpr of Expression pair   |
         CONDexpr of Expression * Expression * Expression   |
         CONSexpr of Expression pair   |
         LISTexpr of Expression list   |
         DECLexpr of string * Expression * Expression   |
         RECDECLexpr of string * Expression * Expression   |
         IDENTexpr of string   |
         LAMBDAexpr of string * Expression   |
         APPLexpr of Expression * Expression   |
         NUMBERexpr of int
   end;

functor Value(): VALUE =
   struct
      type 'a pair = 'a * 'a

      datatype Value = NUMBERvalue of int   |
                       BOOLvalue of bool   |
                       NILvalue   |
                       CONSvalue of Value pair

      exception Value

      val mkValueNumber = NUMBERvalue
      val mkValueBool = BOOLvalue

      val ValueNil = NILvalue
      val mkValueCons = CONSvalue
```

```
      fun unValueNumber(NUMBERvalue(i)) = i   |
          unValueNumber(_) = raise Value

      fun unValueBool(BOOLvalue(b)) = b   |
          unValueBool(_) = raise Value

      fun unValueHead(CONSvalue(c, _)) = c   |
          unValueHead(_) = raise Value

      fun unValueTail(CONSvalue(_, c)) = c   |
          unValueTail(_) = raise Value

      fun eqValue(c1, c2) = (c1 = c2)

(* Pretty-printing *)

      fun intToString(i:int)=  (if i<0 then " -" else "")^
                                     natToString (abs i)
      and natToString(n:int)=
          let val d = n div 10 in
            if d = 0 then chr(ord"0" + n)
            else natToString(d)^ chr(ord"0" + (n mod 10))
          end
      fun printValue(NUMBERvalue(i)) = intToString(i)   |
          printValue(BOOLvalue(true)) = "true"   |
          printValue(BOOLvalue(false)) = "false"   |
          printValue(NILvalue) = "[]"   |
          printValue(CONSvalue(cons)) =
                         "[" ^ printValueList(cons) ^ "]"
          and printValueList(hd, NILvalue) = printValue(hd)   |
              printValueList(hd, CONSvalue(tl)) =
                 printValue(hd) ^ ", " ^ printValueList(tl)   |
              printValueList(_) = raise Value
   end;
```

Appendix B: Files

The following files are available:

Part 1

- `examples.sml` The examples from Part 1

Part 2

- `interp1.sml` Version 1 (as included in Appendix A).
- `interp2.sml` ··· `interp4.sml` The other versions.
- `build1.sml` the structure declarations needed to build Version 1.
- `build2.sml` ··· `build4.sml` Similarly for the other versions.
- `parser.sml` The parser functor.

To build Version 3, say, you type the following (assuming you have copied the files to your directory):

```
use "interp3.sml";
use "parser.sml";
use "build3.sml";
```

Since the parser functor is completely closed, you don't have to include it more than once in every session, although you will probably want to build your system several times while you experiment with the extensions.

This article was processed using the LaTeX macro package with LLNCS style