MLton SML Style Guide

Stephen Weeks

May 27, 2013

This section describes the programming style used in MLton. There are high-level structuring conventions for how the program is broken into modules and low-level formatting conventions for how syntactic constructs are written.

1 High-level structure

Code is structured in MLton so that signatures are closed. Thus, in MLton, one would never write the following.

```
signature SIG =
   sig
      val f: Foo.t -> int
   end
```

Instead, one would write the following.

```
signature SIG =
   sig
      structure Foo: FOO

      val f: Foo.t -> int
   end
```

The benefit of this approach is that one can first understand the specifications (i.e. signatures) of all of the modules in MLton before having to look at any implementations (i.e. structures or functors). That is, the signatures are self-contained.

We deviate from this only in allowing references to top level types (like `int`), basis library modules, and MLton library modules. So, the following signature is fine, because structure `Regexp` is part of the MLton library.

```
signature SIG =
   sig
      val f: Regexp.t -> int
   end
```

We also use signatures to express (some of) the dependencies between modules. For every module `Foo`, we write two signatures in a file named `foo.sig`. The signature `FOO` specifies what is implemented by `Foo`. The signature `FOO_STRUCTS` specifies the modules that are needed in order to specify `Foo`, but that are not implemented by `Foo`. As an example, consider MLton's closure conversion pass (in `mlton/closure-convert`), which converts from `Sxml`, MLton's higher-order simply-typed intermediate language, to `Cps`, MLton's first-order simply-typed intermediate language. The file `closure-convert.sig` contains the following.

```
signature CLOSURE_CONVERT_STRUCTS =
   sig
      structure Sxml: SXML
      structure Cps: CPS
      sharing Sxml.Atoms = Cps.Atoms
   end

signature CLOSURE_CONVERT =
   sig
      include CLOSURE_CONVERT_STRUCTS

      val closureConvert: Sxml.Program.t -> Cps.Program.t
   end
```

These signatures say that the `ClosureConvert` module implements a function `closureConvert` that transforms an `Sxml` program into a `Cps` program. They also say that `ClosureConvert` does not implement `Sxml` or `Cps`. Rather, it expects some other modules to implement these and for them to be provided to `ClosureConvert`. The sharing constraint expresses that the ILs must share some basic atoms, like constants, variables, and primitives.

Given the two signatures that specify a module, the module definition always has the same structure. A module `Foo` is implemented in a file named `foo.fun`, which defines a functor named `Foo` that takes as an argument a structure matching `FOO_STRUCTS` and returns as a result a structure matching `FOO`. For example, `closure-convert.fun` contains the following.

```
functor ClosureConvert (S: CLOSURE_CONVERT_STRUCTS): CLOSURE_CONVERT =
struct

open S

fun closureConvert ...

end
```

Although the signatures for `ClosureConvert` express the dependence on the `Sxml` and `Cps` ILs, they do not express the dependence on other modules that are only used internally to closure conversion. For example, closure conversion uses an auxiliary module `AbstractValue` as part of its higher-order control-flow analysis. Because `AbstractValue` is only used internally to closure conversion, it does not appear in the signatures that specify closure conversion. So, helper functors (like `AbstractValue`) are analogous to helper functions in that they are not visible to clients.

We do not put helper functors lexically in scope because SML only allows top level functor definitions and, more importantly, because files would become unmanageably large. Instead, helper functors get their own `.sig` and `.fun` file, which follow exactly the convention above.

2 General conventions

- A line of code never exceeds 80 columns.
- Use alphabetical order wherever possible.
  - record field names
  - datatype constructors
  - value specs in signatures
  - file lists in CM files
  - export lists in CM files

3 Signatures

We now enumerate the conventions we follow in writing signatures.

1. Signature identifiers are in all capitals, using "_" to separate words.
2. A signature typically contains a single type specification that defines a type constructor `t`, which is the type of interest in the specification. For oexample, here are signature fragments for integers, lists, and maps.

   ```
   signature INTEGER =
      sig
         type t

         val + : t * t -> t
         ...
      end

   signature LIST =
      sig
         type 'a t
   ```

```
      val map: 'a t * ('a -> 'b) -> 'b t
      ...
   end

signature MAP
   sig
      type ('a, 'b) t

      val extend: ('a, 'b) t * 'a * 'b -> ('a, 'b) t
      ...
   end
```

Although at first it might appear confusing to name every type `t`, in fact there is never ambiguity, because at any point in the program there is at most one unqualified `t` in scope, and all other types will be named with long identifiers (like `Int.t` or `Int.t List.t`). For example, the code for a function `foo` within the `Map` module might look like the following.

```
fun foo (l: 'a List.t, n: Int.t): ('a, Int.t) t = ...
```

In practice, for pervasive types like `int`, `'a list`, we often use the standard pervasive name instead of the `t` name.

3. Signatures should not contain free types or structures, other than pervasives, basis library modules, or MLton library modules. This was explained in Section 1.

4. If additional abstract types (other than pervasive types) are needed to specify operations, they are included as substructures of the signature, and have a signature in their own right. For example, the following signature is good.

```
signature FOO =
   sig
      structure Var: VAR

      type t
      val fromVar: Var.t -> t
      val toVar: t -> Var.t
   end
```

5. Signatures do not use substructures or multiple structures to group different operations on the same type. This makes you waste energy remembering where the operations are. For exmample, the following signature is bad.

```
signature REAL =
   sig
     type t

     val + : t * t -> t

     structure Trig:
        sig
           val sin: t -> t
           val cos: t -> t
        end
   end
```

6. Signatures usually should not contain datatypes. This exposes the implementation of what should be an abstract type. For example, the following signature is bad.

   ```
   signature COMPLEX =
      sig
         datatype t = T of real * real
      end
   ```

   A common exception to this rule is abstract syntax trees.

7. Use structure sharing to express type sharing. For example, in `closure-convert.sig`, a single structure sharing equation expresses a number of type sharing equations.

3.1 Value specifications

Here are the conventions that we use for individual value specifications in signatures. Of course, many of these conventions directly impact the way in which we write the core language expressions that implement the specifications.

1. In a datatype specification, if there is a single constructor, then that constructor is called `T`.

   ```
   datatype t = T of int
   ```

2. In a datatype specification, if a constructor carries multiple values of the same type, use a record to name them to avoid confusion.

   ```
   datatype t = T of {length: int, start: int}
   ```

3. Identifiers begin with and use small letters, using capital letters to separate words.

```
val helloWorld: unit -> unit
```

4. There is no space before the colon, and a single space after it. In the case of operators (like `+`), there is a space before the colon to avoid lexing the colon as part of the operator.

5. Pass multiple arguments as tuple, not curried.

```
val eval: Exp.t * Env.t -> Val.t
```

6. Currying is only used when there staging of a computation, i.e., if precomputation is done on one of the arguments.

```
val match: Regexp.t -> string -> bool
```

7. Functions which take a single element of the abstract type of a signature take the element as the first argument, and auxiliary arguments after.

```
val push: t * int -> unit
val map: 'a t * ('a -> 'b) -> 'b t
```

8. n-ary operations take the n elements first, and auxilary arguments after.

```
val merge: 'a t * 'a t * ('a * 'a -> 'b) -> 'b t
```

9. If two arguments to a function are of the same type, and the operation is not commutative, pass them using a record. This names the arguments and ensures they are not confused. Exceptions are the standard numerical and algebraic operators.

```
val fromTo: {start: int, step: int, stop: int} -> int list
val substring: t * {length: int, start: int} -> t
val - : t * t -> t
```

10. Field names in record types are written in alphabetical order.

11. Return multiple results as a tuple, or as a record if there is the potential for confusion.

```
val parse: string -> t * string
val quotRem: t * t -> t * t
val partition: 'a t * ('a -> bool) -> {no: 'a t, yes: 'a t}
```

12. If a function returns multiple results, at least two of which are of the same type, and the name of the function does not clearly indicate which result is which, use a record to name the results.

```
val vars: t -> {frees : Vars.t, bound : Vars.t}
val partition: 'a t * ('a -> bool) -> {yes : 'a t, no : 'a t}
```

13. Use the same names and argument orders for similar functions in different signatures. This is especially common in the MLton library.

```
val < : t * t -> bool
val equals: t * t -> bool
val forall: 'a t * ('a -> bool) -> bool
```

14. Use `is`, `are`, `can`, etc. to name predicates. One exception is `equals`.

```
val isEven: int -> bool
val canRead: t -> bool
```

3.2 Example

Here is the complete specification of a simple interpreter. This demonstrates the `t`-convention, the closed-signature convention, and the use of sharing constraints.

```
signature VAR =
   sig
      type t
   end

signature EXP =
   sig
      structure Var: VAR

      datatype t =
         Var of Var.t
       | Lam of Var.t * t
       | App of t * t
   end

signature VAL =
   sig
      structure Var: VAR

      type t

      val var: Var.t -> t
      val lam: Var.t * t -> t
      val app: t * t -> t
   end

signature INTERP =
   sig
      structure Exp: EXP
      structure Val: VAL
      sharing Exp.Var = Val.Var

      val eval: Exp.t -> Val.t
   end

signature ENV =
   sig
      structure Var: VAR

      type 'a t

      val lookup: 'a t * Var.t -> 'a
      val extend: 'a t * Var.t * 'a -> 'a t
   end
```

4 Functors and structures

We now enumerate the conventions we follow in writing functors and structures. There is some repetition with Section 1.

1. Functor identifiers begin with capital letters, use mixed case, and use capital letters to separate words.
2. Functor definitions look like the following.

   ```
   functor Foo (S: FOO_STRUCTS): FOO =
   struct

   open S

   ...

   end
   ```

3. The name of the functor is the same as the name of the signature describing the structure it produces.

4. The functor result is constrained by a signature.
5. A functor takes as arguments any structures that occur in the signature of the result that it does not implement.
6. Structure identifiers begin with capital letters, and use capital letters to separate words.
7. The name of the structure is the same as the name of the functor that produces it.
8. A structure definition looks like one of the following.

   ```
   structure Foo = Foo (S)

   structure Foo =
      struct
         ...
      end
   ```

9. Avoid the use of `open` except within tightly constrained scopes. The use of `open` makes it hard to look at code later and understand where things come from.

5 Core expressions

We now enumerate the conventions we follow in writing core expressions. We do not repeat the conventions of Section **??**, although many of them apply here.

1. Tuples are written with spaces after commas, like `(a, b, c)`.
2. Records are written with spaces on both sides of equals and with spaces after commas, like `{bar = 1, foo = 2}`.
3. Record field names are written in alphabetical order, both in expressions and types.
4. Function application is written with a space between the function and the argument. If there is one untupled argument, it looks like `f x`. If there is a tupleg argument, it looks like `f (x, y, z)`.
5. When you want to mix declarations with side-effecting statements, use a declaration like `val _ = sideEffectingProcedure()`.
6. In sequence expressions `(e1; e2)` that span multiple lines, place the semicolon at the beginning of lines.

   ```
   (e1
    ; e2
    ; e3)
   ```

7. Never write nonexhaustive matches. Always handle the default case and raise an error message. Your error message will be better than the compiler's. Also, if you have lots of uncaught cases, then you are probably not using the type system in a strong enough way - your types are not expressing as much as they could.

8. Never use the syntax for declaring functions that repeats the function name. Use `case` or `fn` instead. That is, do not write the following.

   ```
   fun f 0 = 1
     | f n = n + 1
   ```

   Instead, write the following.

   ```
   val f =
      fn 0 => 1
       | n => n + 1
   ```

   Or, write the following.

   ```
   fun f n =
      case n of
         0 => 1
       | _ => n + 1
   ```

References