3.8 Decorating Functors

Dynamically, the body of a functor is not evaluated when the functor is declared but it is evaluated once for each time the functor is applied.

```
functor StackFct() =
struct
 datatype stack = ST of int list ref
 val data = ST(ref [ ])
 ...
end

structure Stack1 = StackFct()

structure Stack2 = StackFct()
```

Since the two applications of *StackFct* create two distinct references, *Stack1* and *Stack2* are different and must not be seen to share.

Now let us consider the problem of decorating *StackFct* with names. We start out by decorating the body in the usual way. However, each time we need a fresh name, we record it at the = in the first line. The names that hence are accumulated are called the GENERATIVE NAMES of the functor. The generative names are bound in the sense that they stand as place holders for fresh names which we choose when we eventually apply the functor. (Like the bound names in signatures, we write generative names between parenthesis; unlike the bound names of signatures, generative names are written on the right — because they concern the right side only.)

In the case of nullary functors, i.e. functors that take an empty argument, the structure resulting from a functor application is decorated by taking the decoration of the functor body with each generative name replaced by a fresh name.

`functor` *StackFct*`()` $=_{(m1,s1)}$
`struct`$_{m1}$
 `datatype` *stack*$_{s1}$ `=` *ST* `of` *int list ref*
 `val` *data*$_{s1}$ `=` *ST*`(`*ref* `[ ])`
 ...
`end`

`structure` *Stack1*$_{n7}$ `=` *StackFct*`()`

`structure` *Stack2*$_{n8}$ `=` *StackFct*`()`

Exercise 21 The decorations of *Stack1* and *Stack2* show the top-most structure name only. Complete the decorations.

Notice that *Stack1* and *Stack2* do not share; not even the types *Stack1.stack* and *Stack2.stack* share. Consequently, the variables *Stack1.data* and *Stack2.data* have different types and so the type checker prevents one from mistaking the one for the other.

3.9 External Sharing

Within the body of a functor one may refer to identifiers (of any kind) declared in the context of the functor. Such identifiers are said to occur FREE in the functor. This results in EXTERNAL SHARING, i.e. a decoration in which some of the names stem from outside the functor.

```
structure MyPervasives =
struct_{n1}
 datatype num_{t1} = NUM of int
 ...
end

functor StackFct'() =_{(m2)}
struct_{m2}
 structure MyPer_{n1} = MyPervasives
 type stack_{t1 list ref} =
        MyPer.num list ref
 val data_{t1 list ref}:stack = ref [ ]
end

structure Stack1'_{n8} = StackFct'()

structure Stack2'_{n9} = StackFct'()
```

Notice that external names are not generative; they are left unchanged when the functor is applied.

Exercise 22 Which of the following sharing equations hold?
Stack1' = *Stack2'*;
Stack1'.*MyPer* = *Stack2'*.*MyPer*;
type *Stack1'*.*stack* = *Stack2'*.*stack*.

3.10 Functors with Arguments

```
signature SymSig_{(m1,s1)} =
sig_{m1}
 eqtype sym_{s1}
end

functor SymDir(Sym: SymSig) =_{(m2,s2)}
struct_{m2}
 datatype dir_{s2} = DIR of
                        Sym.sym->int
 fun update ...
end
```

When decorating the body of a functor which has an argument, we assume that we have a structure (by the name of the formal parameter) which *precisely* matches the parameter signature. We assume neither more components nor more sharing than is specified in the signature, for we want the functor to be applicable to all actual argument structures that match the formal parameter signature.

In the above example we simply assume that the name of *Sym* is $m1$ and that the name of *Sym.sym* is $s1$ (as those names are not used of free structures elsewhere; in general, one might have to rename some of the bound names. Having used $m1$ and $s1$ we simply start generating names from $m2$ and $s2$ in the body.

```
structure Actual_{n1} =
struct type sym_{string} = string
end

structure Result_{n2} = SymDir(Actual)
```

Result receives a fresh structure name and *Result.dir* a fresh type name. Note that *Actual* matches *SymSig*.

3.11 Sharing Between Argument and Result

```
signature SymSig_(m1,s1) =
sig_m1
 eqtype sym_s1
end

functor SymDir(Sym: SymSig) =_(m2)
struct_m2
 type dir_s1→int = Sym.sym->int
 fun update ...
end
```

The type name $s1$ is shared between the argument and the body. When the functor is applied, this sharing must be translated into sharing between the actual argument and the actual result.

```
structure Actual_n1 =
struct type sym_string = string
end

structure Result_n2 = SymDir(Actual)
```

Exercise 23 Complete the decoration of *Result*.

Exercise 24 (1) Using the latest definition of *SymDir*, is the following expression legal?

```
fn (d:Result.dir) => d "abc"
```

(2) Same question, but for the earlier definition of *SymDir*.

A full decoration of the result of applying a functor with one argument can be obtained as follows:

1. Match the actual argument against the formal parameter signature yielding a realisation which maps bound names to formal signature to names in the actual argument;

2. Apply this realisation to the decoration of the functor body;

3. also substitute fresh names for the generative names of the functor body.

3.12 Explicit Result Signatures

When a functor declaration contains a result signature, the decoration of the functor declaration proceeds as follows:

1. decorate the functor without the result signature;

2. decorate the result signature. If one can get an instance of the result signature by removing components and polymorphicm from the decorated body, then this instance is used as a formal result instead of the decorated body; otherwise the declaration is rejected.

This has the effect that upon application of the functor, sharing is propagated as before, but only the components and polymorphism of the result signature are visible in the actual result.

Exercise 25 Why is it not always the case that obtaining R by

```
functor F(S: SIG): SIG'=
struct...end

structure R = F(...)
```

is equivalent to obtaining R by

```
functor F(S: SIG) =
struct...end

structure R: SIG' = F(...)
```

Give a condition on *SIG'* under which this difference disappears.

4 Implementing an Interpreter in ML

The purpose of this lecture is to show a worked example of program development using ML modules. We shall tackle the problem of implementing a small ML system. The system is of course going to considerable simplified compared to a real ML implementation.

We implement only a few of the language constructs found in real ML. The user of our system will not get the ability to declare new types and data types; however, there will be arithmetic on the build-in integers, `if`...`then`...`else` expressions, and indeed lists, higher order functions and recursion, so it is far from a trivial language. We shall refer to this language as Mini ML.

Moreover, the system will be an interpreter rather than a compiler. It still has a typechecker, indeed we shall see how one can implement a restricted form of polymorphism.

The system is actually running and you can modify and extend it provided you have access to an implementation and to the files listed in Appendix B. To make life easier for you, we provide a parse functor which can parse a string (the Mini ML source expression) into an ABSTRACT SYNTAX TREE, the shape of which will be defined below. The rest of the interpreter works on abstract syntax trees.

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. In this lecture I shall show how one in successive steps can extend the typechecker to handle polymorphic lists, variables and `let` expressions. In the practical sessions you can extend the evaluator in the same manner (it is easier than extending the typechecker).

The typechecker and the evaluator can be developed independently as long as you do not change the signatures we provide. This will allow you to take the typechecker functors I have written and plug into your own system as you improve the power of your evaluator. Alternatively, you might want to modify or extend my typechecker functors, and take over evaluator functors that other people write.

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

The development of the typechecker and the evaluator need not be in step. You can disable either by assigning false to one of the variables `tc` and `eval`.