

***Programming Language Concepts and Semantics
Part II***

***Lecture XIV
Abstractions and Recursion***

In this lecture we will look at procedural abstractions and recursive declarations involving such abstractions.

The lecture follows Chapter 8 of “Fundamental Concepts and Formal Semantics of Programming Languages, Lecture Notes by Peter Mosses. University of Aarhus”

Abstractions: Fundamental Concepts

Abstraction mechanisms allow us to hide implementation details of a program component from the *client* using it. How the client can use the component is specified by an *interface*.

A ***procedural abstraction*** $ABS:Abs$ encapsulates a part of a program (the *body*) which is executed each time the abstraction is *applied*.

Applications of abstractions with ***parameters*** $PAR:Par$ must supply appropriate *arguments*, usually evaluated *before* the application (other approaches are call-by-need and call-by-name).

Correspondence between parameter and arguments may be *positional* (following the order in which they are listed – as in tuples) or *by name* (thanks to parameter identifiers – as in records).

Parameters of abstractions in ML are single identifiers $Par ::= \text{bind}(I)$ **and tuples** $Par ::= \text{tup}(Par^*)$

Common associations between parameter identifiers and arguments are *direct* (call by reference: the parameter identifier is bound directly to the argument) or *indirect* (call by value: the parameter identifier is a local variable initialised to the value of the argument). Others variants are possible.

In ML parameters are bound directly to arguments (call-by-reference-simulated by having references as arguments).

Abstractions: Fundamental Concepts (Cont.)

A *procedure declaration* binds an identifier to a procedural abstraction. Functional languages allow procedural abstractions to be written as expressions without binding an identifier to it.

in ML, each match $\text{PAR} \Rightarrow E$ (in a case expression, a function, etc) involves a procedural abstraction $\text{Abs} ::= \text{abs}(\text{Par}, \text{Exp})$

Binding scopes are *static* if they start with the environment of the abstraction definition. They are *dynamic* when they start with the environment of the abstraction application.

Static scopes are modelled by embedding the current bindings in each abstraction to form a *closure* $\text{Abs} ::= \text{closure}(\text{Dec}, \text{Abs})$ where Dec is usually an environment. The expression then is *closed* $\text{Exp} ::= \text{close}(\text{Abs})$ by computing its closure.

We have seen applications for operators and identifiers. We now extend it for abstractions by including abstractions in expressions: $\text{Exp} ::= \text{Abs} \mid \text{app}(\text{Exp}, \text{Arg}) \mid \text{app-seq}(\text{Exp}, \text{Arg})$ Note that this makes the language higher order.

Parameters correspond to abstractions of declarations and can also be applied $\text{Dec} ::= \text{app}(\text{Par}, \text{Arg})$

Abstractions: Formal Semantics

Functions are expressible values.

Module *Cons/Exp/Abs*:

Value ::= Abs

Module *Cons/Exp/close*:

see Abs/closure

Label = {env:Env, ...}

close(ABS):Exp --{env=ENV, ---}-> closure(ENV, ABS)

Abstractions: Formal Semantics (Cont.)

Rules for applying procedural abstractions:

Module *Cons/Exp/app*:

$$E \text{ --}\{\dots\}\text{--> } E'$$

$$\text{app}(E, \text{ARG}) : \text{Exp} \text{ --}\{\dots\}\text{--> } \text{app}(E', \text{ARG})$$

$$\text{ARG} \text{ --}\{\dots\}\text{--> } \text{ARG}'$$

$$\text{app}(E, \text{ARG}) : \text{Exp} \text{ --}\{\dots\}\text{--> } \text{app}(E, \text{ARG}')$$

The above rules allow the evaluation of E and Arg to be interleaved. The following one requires them to be sequential.

Module *Cons/Exp/app-seq*:

$$E \text{ --}\{\dots\}\text{--> } E'$$

$$\text{app-seq}(E, \text{ARG}) : \text{Exp} \text{ --}\{\dots\}\text{--> } \text{app-seq}(E', \text{ARG})$$

$$\text{ARG} \text{ --}\{\dots\}\text{--> } \text{ARG}'$$

$$\text{app-seq}(V, \text{ARG}) : \text{Exp} \text{ --}\{\dots\}\text{--> } \text{app-seq}(V, \text{ARG}')$$

$$\text{app-seq}(V, \text{PV}) : \text{Exp} \text{ ---> } \text{app}(V, \text{PV})$$

see *Exp/app*

Abstractions: Formal Semantics (Cont.)

Module *Cons/Dec/app*:

$$\text{ARG} \text{ --}\{\dots\}\text{--> ARG'}$$

 $\text{app}(\text{PAR}, \text{ARG}) : \text{Dec} \text{ --}\{\dots\}\text{--> app}(\text{PAR}, \text{ARG}')$

Module *Cons/Abs/abs-Exp*:

see *Dec/local*, *Dec/app*

$$\text{app}(\text{abs}(\text{PAR}, \text{E}), \text{PV}) : \text{Exp} \text{ ----> local}(\text{app}(\text{PAR}, \text{PV}), \text{E})$$

Module *Cons/Abs/closure*:

see *Dec/local*, *Abs*

$$\text{app}(\text{closure}(\text{D}, \text{ABS}), \text{PV}) : \text{Exp} \text{ ----> local}(\text{D}, \text{app}(\text{ABS}, \text{PV}))$$

Abstractions: Formal Semantics (Cont.)

Rules for applying a parameter to a value to compute bindings:

Module *Cons/Par/bind*:

see Dec/app

$$\text{app}(\text{bind}(I), BV) : \text{Dec} \dashrightarrow \{I \mid \rightarrow BV\}$$

Module *Cons/Par/tup*:

see Dec/app, Dec/simult

$$\text{PAR}^+ = (\text{PAR}, \text{PAR}^*), \text{BV}^+ = (\text{BV}, \text{BV}^*)$$

$$\begin{aligned} \text{app}(\text{tup}(\text{PAR}^+), \text{tup}(\text{BV}^+)) : \text{Dec} \dashrightarrow \\ \text{simult}(\text{app}(\text{PAR}, \text{BV}), \text{app}(\text{tup}(\text{PAR}^*), \text{tup}(\text{BV}^*))) \end{aligned}$$
$$\text{void} = \text{ENV}$$

$$\text{app}(\text{tup}(), \text{tup}()) : \text{Dec} \dashrightarrow \text{ENV}$$

Recursive Abstractions: Fundamental Concepts

An abstraction is *recursive* when executing its body results in an application of itself.

Dynamic scoping allows declarations which binds an identifier to an abstraction to be recursive (occurrences of the identifier will refer to the abstraction itself).

For static scoping the current environment does not yet include the mapping of the identifier to the abstraction, so it must refer to some previous binding. Thus we need to either:

- find binding in advance (so we can reuse it)
- declare a special construct which has the desired effect

The first approach is problematic when dealing with unknown-length computations. We will take the second approach `Dec :: rec(Dec)`. We will only consider `bind`, `simult` and `simult-seq` declarations.

Recursive Abstractions: Formal Semantics

Module *Cons/Dec/rec*:

`Dec ::= reclose(Dec, Dec)`

see `Dec/bind`, `Dec/simult`, `Dec/simult-seq`,
`Exp/close`, `Abs/closure`

`rec(D) : Dec ---> reclose(rec(D), D)`

`reclose(rec(D), bind(I, close(ABS))) : Dec --->`
`bind(I, close(closure(rec(D), ABS)))`

`reclose(rec(D), simult-seq(D1, D2)) : Dec --->`
`simult-seq(reclose(rec(D), D1),`
`reclose(rec(D), D2))`

`reclose(rec(D), simult(D1, D2)) : Dec --->`
`simult(reclose(rec(D), D1),`
`reclose(rec(D), D2))`

For example: how would you represent `fun f x = f x`

`rec(bind(id(f), close(abs(bind(id(x)), app(id(f), id(x))))))`