Non-Dependent Types For Modules

Claudio V. Russo

LFCS
Division of Informatics
University of Edinburgh
http://www.dcs.ed.ac.uk/~cvr
cvr@dcs.ed.ac.uk
(supported by EPSRC grant GR/K63795)

March 24, 1999

Introduction

What is the type structure of Standard ML Modules?

Background

- Accepted wisdom:
  "SML-like modules require dependent types."
- Standard dependent types:
  - violate the phase distinction,
  - preclude first class modules.
- Quasi-dependent types:
  - non-standard,
  - bad meta-theoretic properties.

Question Are dependent types really necessary?

Approach Re-assess the evidence:

- examine the static semantics of SML Modules.

Conclusion

- SML Modules has a purely second-order type theory.
- Dependent types can be avoided.

Review

Structures group related type and term definitions:

```
structure Int =
      struct type t = int;
             val z = 0;
             val s = λn:t. n + 1;
             val p = λn:t. n - 1
      end;
```

Components accessed by the **dot notation**:

- `Int.t` is a Core type,
- `Int.z` is a Core value.

"The type `Int.t` depends on the term `Int`."

. . . Review . . .

Signatures specify the types of structures:

```
signature INT =
       sig type t = int;       (* transparent type spec *)
           val z: t;
           val s: t->t;
           val p: t->t
       end;
```

- `INT` is a **complete** specification of `Int`.

```
signature NAT =  sig type t;    (* opaque type spec *)
                     val z: t;
                     val s: t->t
                 end
```

- `NAT` is a **partial** specification of `Int`:
  - type `t` specified, but **not defined**,
  - value `p` **omitted**.

. . . Review . . .

Signatures may be used to **constrain** structures:

```
structure IntNat =
      Int : NAT   (* transparent constraint *)
```

- `IntNat.p` is undefined,
- `IntNat.t` is **transparent**, thus equivalent to `int`:
  `IntNat.s(2)` well-typed.

```
structure AbsNat =
      Int :> NAT    (* opaque constraint *)
```

- `AbsNat.p` is undefined,
- `AbsNat.t` is "new", thus **abstract**:
  `AbsNat.s(2)` ill-typed.

. . . Review . . .

Structures and signatures may be **nested**:

```
structure X =
  struct structure N = Int :> NAT;
         val f = λn:N.t. N.s(N.s n)
  end;
```

- dot notation X.N accesses the sub-structure.
- definition of f refers to definition of N.

```
sig structure N : NAT;
    val f = N.t -> N.t
end;
```

- specification of f refers to specification of N.

“The type of f depends on the term N.”

. . . Review

Functors define functions on structures.

```
functor F(N:NAT)= struct val f = λn:N.t. N.s(N.s n)
                  end;
```

- `F(Int)` well-typed:
  - `N.t` realised by `int`;
  - even though `Int` defines additional component `p`.
- `F(AbsNat)` well-typed, `N.t` realised by `AbsNat.t`;
- `F(Int:>NAT)` well-typed, `N.t` realised by the new type generated by `Int:>NAT`.

Standard Dependent Types . . .

cf. MacQueen, Harper & Mitchell.

- **The type of a structure is a signature.**
- **Modules have dependent types.**

Opaque type specifications are **2nd-order** dependent record types:

```
sig type t;
    val z: t;     (* type of z depends on type t *)
    ...
end
```

Type definitions introduce **opaque types**.

$$\frac{\vdots}{\mathtt{C} \vdash \mathtt{struct\ type\ t\ =\ int;\ ...\ end} : \mathtt{sig\ type\ t;\ ...\ end}}$$

Transparency by **strong** interpretation of type projection:

$$\frac{\mathtt{C} \vdash \mathtt{Int} : \mathtt{sig\ type\ t;\ ...\ end} \qquad \mathtt{C} \vdash \mathtt{Int} = \mathtt{struct\ type\ t\ =\ int;\ ...\ end}}{\mathtt{C} \vdash \mathtt{Int.t} = \mathtt{int}}$$

- Type equivalence depends on term equivalence.

. . . Standard Dependent Types . . .

Nested signatures are **1st-order** dependent record types:

```
sig structure N: NAT;
    val f : N.t → N.t      (* type of f depends on term N *)
end
```

Structure definitions introduce structure specifications:

$$\frac{\vdots}{\begin{array}{ll}\mathtt{C} & \vdash \mathtt{struct\ structure\ N\ =\ Int\ :>\ NAT;\ ...\ end} \\ & :\ \ \mathtt{sig\ structure\ N:\ NAT;\ ...\ end}\end{array}}$$

$$\frac{\mathtt{C} \vdash \mathtt{X} : \mathtt{sig\ structure\ N{:}NAT;\ val\ f{:}\ \ N.t} \to \mathtt{N.t\ end}}{\mathtt{C} \vdash \mathtt{X.f} : [\mathtt{X.N}/\mathtt{N}]\,(\mathtt{N.t} \to \mathtt{N.t})}$$

- Typing a projection involves **1st-order** substitution.

. . . Standard Dependent Types . . .

Functor types are **1st-order** dependent function spaces:

```
funsig(N:NAT)sig val f: N.t → N.t end;
```
(range depends on term* `N` **)*

Functor definitions introduce dependent function spaces:

$$\frac{\vdots}{\mathtt{C} \vdash \mathtt{F} : \mathtt{funsig(N:NAT)sig\ val\ f:\ N.t} \rightarrow \mathtt{N.t\ end}}$$

$$\frac{\begin{array}{l}\mathtt{C} \vdash \mathtt{F} : \mathtt{funsig(N:NAT)sig\ val\ f:\ N.t} \rightarrow \mathtt{N.t\ end} \\ \mathtt{C} \vdash \mathtt{IntNat} : \mathtt{NAT}\end{array}}{\mathtt{C} \vdash \mathtt{F(IntNat)} : [\mathtt{IntNat/N}]\,(\mathtt{sig\ val\ f:\ N.t}{\rightarrow}\mathtt{N.t\ end})}$$

- Typing an application involves **1st-order** substitution.

. . . Standard Dependent Types

- No phase distinction.
- No account of sharing (transparent type specifications).
- No abstract types — all type components are transparent.
- No support for separate compilation — must have implementation to derive all type equalities.
- Strong interpretation of opaque types precludes first-class modules.

Harper, Mitchell & Moggi rescue the phase distinction with a **non-standard** interpretation of dependent types — other caveats still apply.

Non-Standard Dependent Types . . .

cf. Leroy, Harper & Lillibridge, Lillibridge

- **Adopt weak interpretation of opaque type specifications.**
- **Add transparent type specifications as primitives.**

Transparent type specifications publish static type equalities:

$$\frac{\mathtt{C} \vdash \mathtt{Int} : \mathtt{sig\ type\ t\ =\ int;\ ...end}}{\mathtt{C} \vdash \mathtt{Int.t} = \mathtt{int}}$$

For opaque types, use **strengthening**:

$$\frac{\mathtt{C} \vdash \mathtt{AbsNat} : \mathtt{sig\ type\ t;\ ...end}}{\mathtt{C} \vdash \mathtt{AbsNat} : \mathtt{sig\ type\ t\ =\ AbsNat.t;\ ...end}}$$

Then $\mathtt{AbsNat.t} = \mathtt{AbsNat.t}$ but $\mathtt{AbsNat.t} \neq \mathtt{int}$.

Problem: Can't distinguish (Int:>NAT).t and (Int:>NAT).t:

$$\frac{\dfrac{\mathtt{C} \vdash \mathtt{Int{:}{>}NAT} : \mathtt{sig\ type\ t;\ ...end}}{\mathtt{C} \vdash \mathtt{Int{:}{>}NAT} : \mathtt{sig\ type\ t\ =\ Int{:}{>}NAT;\ ...end}}}{\mathtt{C} \vdash \mathtt{(Int{:}{>}NAT).t} = \mathtt{(Int{:}{>}NAT).t}}$$

Projection Restriction:

- Restrict projections to paths (values).

. . . Non-Standard Dependent Types

Dependent application can violate the projection restriction:

```
F(Int:>NAT) : sig val f: (Int:>NAT).t → (Int:>NAT).t
              end
```

Application Restriction:

- A functor may only be applied at a non-dependent supertype.

$$\frac{\begin{array}{l}\mathtt{C} \vdash \mathtt{f} : \mathtt{funsig(X{:}D)R} \\ \mathtt{C} \vdash \mathtt{funsig(X{:}D)R} \subseteq \mathtt{funsig(X{:}D')R'} \\ \mathtt{X} \notin \mathcal{V}(\mathtt{R'}) \\ \mathtt{C} \vdash \mathtt{a} : \mathtt{D'}\end{array}}{\mathtt{C} \vdash \mathtt{f(a)} : \mathtt{R'}}$$

```
F(IntNat)    :   sig val f :  IntNat.t → IntNat.t end

F(Int:>NAT)  :   sig end (* f is removed by subtyping *)
```

Doesn't scale to higher-order: an application

- may be ill-typed even when the argument is in the domain.
- may not have a principal type.

Static Semantics of SML Modules . . .

SML types terms using **semantic objects**:

$$
\begin{array}{rll}
\alpha & \in \mathit{Var} \stackrel{\mathrm{def}}{=} \{\alpha, \beta, \delta, \gamma, \ldots\} \\
P, Q & \in \mathit{VarSet} \stackrel{\mathrm{def}}{=} \mathrm{Fin}(\mathit{Var}) \\
u & \in \mathit{Type} ::= \alpha \\
& \qquad\qquad\; \mid \; u \to u' \\
& \qquad\qquad\; \mid \; \mathrm{int} \\
\mathcal{S} & \in \mathit{Str} ::= \{\mathrm{t} \triangleright u, \ldots, \mathrm{x} : u, \ldots, \mathrm{X} : \mathcal{S}\} \\
\mathcal{L} & \in \mathit{Sig} ::= \Lambda P.\mathcal{S} \\
\mathcal{X} & \in \mathit{ExStr} ::= \exists P.\mathcal{S} \\
\mathcal{F} & \in \mathit{Fun} ::= \forall P.\mathcal{S} \to \mathcal{X} \\
\mathcal{C} & \in \mathit{Context} ::= \{\mathrm{t} \triangleright u, \ldots, \mathrm{x} : u, \ldots, \mathrm{X} : \mathcal{S}, \ldots, \mathrm{F} : \mathcal{F}\}
\end{array}
$$

- α, β, . . . are **type variables**
- **2nd-order** type parameterisation and quantification (Λ,$\forall$, $\exists$)
- **no 1st-order dependencies of types on terms**

Subtyping relation:

$$
\frac{\begin{array}{l}
\mathcal{D}(\mathcal{S}) \supseteq \mathcal{D}(\mathcal{S}') \\
\forall \mathrm{t} \in \mathcal{D}(\mathcal{S}').\mathcal{S}(\mathrm{t}) = \mathcal{S}'(\mathrm{t}) \\
\forall \mathrm{x} \in \mathcal{D}(\mathcal{S}').\mathcal{S}(\mathrm{x}) = \mathcal{S}'(\mathrm{x}) \\
\forall \mathrm{X} \in \mathcal{D}(\mathcal{S}').\mathcal{S}(\mathrm{X}) \succeq \mathcal{S}'(\mathrm{X})
\end{array}}{\mathcal{S} \succeq \mathcal{S}'}
$$

. . . Static Semantics of SML Modules . . .

Denotation judgements relate syntactic types to semantic objects:

$$\frac{}{\mathcal{C} \vdash \mathrm{t} \triangleright \mathcal{C}(\mathrm{t})} \qquad \frac{\mathcal{C} \vdash \mathrm{u} \triangleright u \quad \mathcal{C} \vdash \mathrm{u}' \triangleright u'}{\mathcal{C} \vdash \mathrm{u} \to \mathrm{u}' \triangleright u \to u'} \qquad \frac{}{\mathcal{C} \vdash \mathbf{int} \triangleright \mathrm{int}}$$

$$\frac{\mathcal{C} \vdash \mathrm{sp} : \mathcal{S}}{\mathcal{C} \vdash \mathrm{sp.t} \triangleright \mathcal{S}(\mathrm{t})}$$

$$\frac{\mathcal{C} \vdash \mathrm{u} \triangleright u \quad \mathcal{C}, \mathrm{t} \triangleright u \vdash \mathrm{B} \triangleright \Lambda P.\mathcal{S}}{\mathcal{C} \vdash \mathbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{B} \triangleright \Lambda P.\mathrm{t} \triangleright u, \mathcal{S}}$$

$$\frac{\mathcal{C}, \mathrm{t} \triangleright \alpha \vdash \mathrm{B} \triangleright \Lambda P.\mathcal{S}}{\mathcal{C} \vdash \mathbf{type}\ \mathrm{t}; \mathrm{B} \triangleright \Lambda\{\alpha\} \cup P.\mathrm{t} \triangleright \alpha, \mathcal{S}}$$

$$\frac{\mathcal{C} \vdash \mathrm{S} \triangleright \Lambda P.\mathcal{S} \qquad \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{B} \triangleright \Lambda Q.\mathcal{S}'}{\mathcal{C} \vdash \mathbf{structure}\ \mathrm{X} : \mathrm{S}; \mathrm{B} \triangleright \Lambda P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}'}$$

$$\vdots$$

$$\frac{\mathcal{C} \vdash \mathrm{B} \triangleright \Lambda P.\mathcal{S}}{\mathcal{C} \vdash \mathbf{sig}\ \mathrm{B}\ \mathbf{end} \triangleright \Lambda P.\mathcal{S}}$$

- signatures denote parameterised types.
- dependent projection $\mathrm{sp.t}$ denotes a non-dependent object.
- opaque type specifications introduce type parameters.
- **reduction of 1st-order to 2nd-order dependencies.**

. . . Static Semantics of SML Modules . . .

Typing judgements relate structures to existential types:

$$\frac{\mathcal{C} \vdash \mathrm{u} \triangleright u \quad \mathcal{C}, \mathrm{t} \triangleright u \vdash \mathrm{b} : \exists P.\mathcal{S}}{\mathcal{C} \vdash \textbf{type}\ \mathrm{t} = \mathrm{u}; \mathrm{b} : \exists P.\mathrm{t} \triangleright u, \mathcal{S}}$$

$$\frac{\mathcal{C} \vdash \mathrm{e} : u \quad \mathcal{C}, \mathrm{x} : u \vdash \mathrm{b} : \exists P.\mathcal{S}}{\mathcal{C} \vdash \textbf{val}\ \mathrm{x} = \mathrm{e}; \mathrm{b} : \exists P.\mathrm{x} : u, \mathcal{S}}$$

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \qquad \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{b} : \exists Q.\mathcal{S}'}{\mathcal{C} \vdash \textbf{structure}\ \mathrm{X} \ = \ \mathrm{s;b} : \exists P \cup Q.\mathrm{X} : \mathcal{S}, \mathcal{S}'}$$

$$\frac{}{\mathcal{C} \vdash \epsilon_{\mathrm{b}} : \exists \emptyset.\epsilon_{\mathcal{S}}}$$

- Components of the existential type are independent.
- Existential quantifiers are silently eliminated and re-introduced.

. . . Static Semantics of SML Modules . . .

Typing constraints . . .

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \quad \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda Q.\mathcal{S}' \qquad \mathcal{S} \succeq \varphi\left(\mathcal{S}'\right) \quad \mathcal{D}(\varphi) = Q}{\mathcal{C} \vdash \mathrm{s} : \mathrm{S} : \exists P.\varphi\left(\mathcal{S}'\right)}$$

- eliminate the existential type
- verify it is a subtype of an instance of the signature
- preserve the instantiation
- forget components but **preserve** types

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \quad \mathcal{C} \vdash \mathrm{S} \triangleright \Lambda Q.\mathcal{S}' \qquad \mathcal{S} \succeq \varphi\left(\mathcal{S}'\right) \quad \mathcal{D}(\varphi) = Q}{\mathcal{C} \vdash \mathrm{s} :> \mathrm{S} : \exists Q.\mathcal{S}'}$$

- existentially quantify over the instantiation
- forget components and **hide** types

. . . Static Semantics of Modules . . .

Functor definitions introduce **polymorphic** functions:

$$\frac{\mathcal{C} \vdash \mathrm{S} \rhd \Lambda P.\mathcal{S} \qquad\qquad \mathcal{C}, \mathrm{X} : \mathcal{S} \vdash \mathrm{s} : \mathcal{X} \qquad \mathcal{C}, \mathrm{F} : \forall P.\mathcal{S} \to \mathcal{X} \vdash \mathrm{b} : \mathcal{X}'}{\mathcal{C} \vdash \textbf{functor}\ \mathrm{F}\ (\mathrm{X} : \mathrm{S})\ =\ \mathrm{s};\ \mathrm{b} : \mathcal{X}'}$$

- The argument signature determines the functor's type parameters and domain.

Functor applications eliminate polymorphic functions:

$$\frac{\mathcal{C} \vdash \mathrm{s} : \exists P.\mathcal{S} \qquad \mathcal{C}(\mathrm{F}) > \mathcal{S}' \to \exists Q.\mathcal{S}'' \quad \mathcal{S} \succeq \mathcal{S}'}{\mathcal{C} \vdash \mathrm{F}(\mathrm{s}) : \exists P \cup Q.\mathcal{S}''}$$

- eliminate the existential quantifier from the argument
- choose an appropriate instance of the functor
- check that the argument's type is a subtype of the domain
- existentially quantify over the hypothetical types.

. . . Static Semantics of Modules. . .

$$\texttt{Int} : \exists\emptyset.\{\texttt{t} \triangleright \mathrm{int}, \texttt{z} : \mathrm{int}, \texttt{s} : \mathrm{int}{\rightarrow}\mathrm{int}, \texttt{p} : \mathrm{int}{\rightarrow}\mathrm{int}\}$$

$$\texttt{NAT} \triangleright \Lambda\alpha.\{\texttt{t} \triangleright \alpha, \texttt{z} : \alpha, \texttt{s} : \alpha{\rightarrow}\alpha\}$$

$$(\texttt{Int:NAT}) : \exists\emptyset.\{\texttt{t} \triangleright \mathrm{int}, \texttt{z} : \mathrm{int}, \texttt{s} : \mathrm{int}{\rightarrow}\mathrm{int}\}$$

$$(\texttt{Int:>NAT}) : \exists\beta.\{\texttt{t} \triangleright \beta, \texttt{z} : \beta, \texttt{s} : \beta{\rightarrow}\beta\}$$

$$\texttt{F} : \forall\gamma.\{\texttt{t} \triangleright \gamma, \texttt{z} : \gamma, \texttt{s} : \gamma{\rightarrow}\gamma\} \rightarrow \exists\emptyset.\{\texttt{f} : \gamma{\rightarrow}\gamma\}$$

$$\texttt{F(Int)} : \exists\emptyset.\{\texttt{f} : \mathrm{int}{\rightarrow}\mathrm{int}\}$$

$$\texttt{F(Int:>NAT)} : \exists\beta.\{\texttt{f} : \beta{\rightarrow}\beta\}$$

. . . Static Semantics of Modules

```
struct  structure N = Int:>NAT;
        val f = λn:N.t.N.s(N.s n)
end
```

$: \exists\beta.\{\quad \mathtt{N} : \{\mathtt{t} \triangleright \beta, \mathtt{z} : \beta, \mathtt{s} : \beta \to \beta\},$
$\quad\quad \mathtt{f} : \beta \to \beta\}$

```
sig  structure N = NAT;
     val f = N.t→N.t
end
```

$\triangleright \Lambda\beta.\{\quad \mathtt{N} : \{\mathtt{t} \triangleright \beta, \mathtt{z} : \beta, \mathtt{s} : \beta \to \beta\},$
$\quad\quad \mathtt{f} : \beta \to \beta\}$

Summary & Conclusions

What is the type structure of SML Modules?

Accounts based on standard dependent types:

- violate the phase distinction
- preclude first-class modules

Alternatives based on quasi-dependent types have:

- non-standard rules
- bad properties (no principal types for h.o. modules)

Static semantics of SML shows no dependent types, instead:

- non-dependent record types
- subtyping
- type parameterisation
- type quantification

This semantics extends naturally to:

- generalised projections
- higher-order functors that admit principal types
- first-class modules

The complications of dependent types can be avoided.