Modularity, *Module Systems*

MPRI course 2-4-2, Part 3, Lesson 1

Didier Rémy

INRIA-Rocquencourt

Janvier 6, 2009

Note

This course is largely based on the OCaml module system (implementation and description), and on the following papers from the literature:

1, 2 Leroy (1994).

3 Dreyer and Rossberg (2008); Hirschowitz and Leroy (2005).

4 Montagu and Rémy (2009).
(see also related work by Dreyer (2007))

Other pointers will also be provided along the course when necessary.

The formal treatment varies between the different parts. Parts 1, 2, 3 contain formal definitions, but no formal result. For part 4, complete formal definitions and all results can be found in Montagu and Rémy (2009).

Modular programming

Why?

- Split large programs into small pieces, called *components*
- Understand components independently
  (enforce their invariants, verify or proof them)
- Maintain programs component by component
- Increase reusability
- Separately compile components

Compare with mechanics

- Large systems in mechanics (airplanes, power plants, *etc.*) are also large and complex, and usually decomposed into small units.
- However, programs are more fragile because their behavior is not continuous: a program crashes all at once without any prior notice.

Modular programming

How?

- Poor man's modular programming.
  - No specific support, but a lot of discipline (which cannot be enforced).
  - Limited expressiveness, may require acrobatics. (e.g. use base language records to group related definitions, emulate objects, etc.)
- Object-oriented paradigm.
  - Data-centric approach: data (objects) with similar structure and behavior are created from a class that groups all operations that operate on similar data.
  - Abstraction by hiding the representation of objects and exposing only some operations to operate on them.
  - Focus is put on reusability via inheritance: see last lesson. But they lack a good abstraction mechanism.
- Module systems
  - Group base-language definitions into modules
  - Provide a small calculus to combine modules together.

Modules The different eras

- Modular 3, CamlLight, (and later, Java packages)
  Name space control *No type generativity*
- ML modules at their early stage, with the effectful *stamp* approach.
  SML, functors, higher-order functors *Introduced type generativity*
- Syntactic approach to type generativity
  SML (in theory), OCaml *Type checking, no subject reduction*
- Syntactic type soundness with heavy mathematics:
  A sophisticated core language (singleton kinds to model type equivalences)
  + Elaboration of the surface language into the core language.
  Subject reduction only for the core language
- Syntactic type soundness with lighter mathematics
  A simpler, more expressive core language with a reduction semantics +
  Elaboration *Subject reduction only for the core language*
- Expressive core language, first-class modules, mixins, no elaboration
  Goal, ongoing research

Modules — The key ingredients

Common features for assembling components

- Record-like structure of base-language values and type definitions
- Modules may be hierarchical.
- (Functor) modules may take other modules as arguments or return them as results.

Already present in the base-language, except for type definitions.

The essential feature, specific to modules

- *Support for abstract types and type generativity*

Also the source of most difficulties...

Modules — Abstract types

Why?

- Hide the differences between related components, showing them with a compatible interface and making them interchangeable.
- Allow for better access control to selected parts of data, which helps preserve finer invariants.
- Hide details of the implementation, which increases readability.

How?

- Several solutions, but no definite answer yet.
- Type abstraction is one of the main difficulties of module systems.
- Existential types models type abstraction but not in a modular way.
- An *ad hoc* solution, using paths to keep track of type identities.

Abstract types

Why not existential types?

Existential types

$$\begin{array}{lcll} \mathcal{M} & ::= & \ldots \mid \mathsf{pack}\ \tau, \mathcal{M}\ \mathsf{as}\ \exists\alpha.\tau' \mid \mathsf{unpack}\ \mathcal{M}\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ \mathcal{M}' & \text{Expressions} \\ \tau & ::= & \ldots \mid \exists\alpha.\tau & \text{Types} \end{array}$$

Typing rules

$$\textsc{Exists}\ \frac{\Gamma \vdash \mathcal{M} : \tau'[\tau/\alpha]}{\Gamma \vdash \mathsf{pack}\ \tau, \mathcal{M}\ \mathsf{as}\ \exists\alpha.\tau' : \exists\alpha.\tau'}$$

$$\textsc{Exists}\ \frac{\Gamma \vdash \mathcal{M} : \exists\alpha.\tau \qquad \Gamma, x : \alpha \vdash \mathcal{M}' : \tau' \qquad \alpha \notin \mathit{ftv}(\tau')}{\Gamma \vdash \mathsf{unpack}\ \mathcal{M}\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ \mathcal{M}' : \tau'}$$

Abstract types — Why not existential types?

unpack $\mathcal{M}$ as α, x in $\mathcal{M}'$

- Models type abstraction:
  Occurrences of x within $\mathcal{M}$ are seen *abstractly* with type α, of which nothing can be assumed.
- Lacks modular structure:
  Type variable α cannot occur free in the type τ' of $\mathcal{M}'$.

Problem

- Existential types only model abstract types in monolithic programs.
- Their uses cannot be spread in different program components—a key pattern of modular programming.

Abstract types

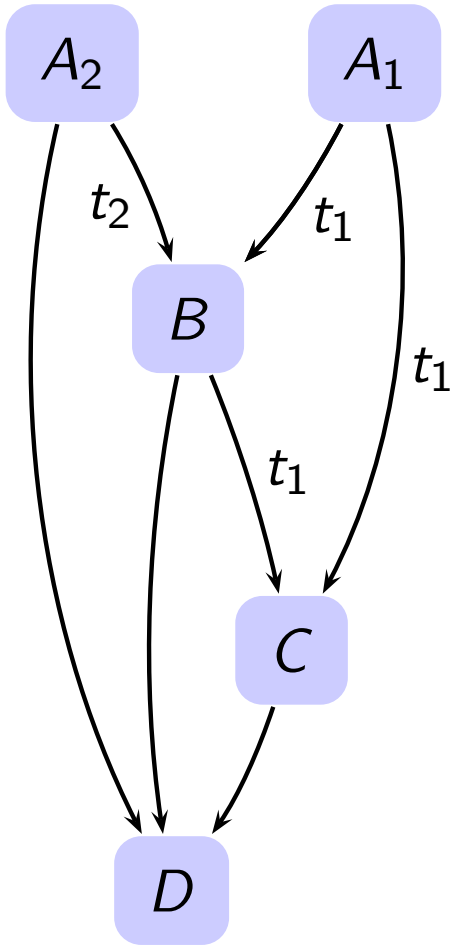
Path-based approach

Modules (represented by boxes) are assembled in complex ways. Types are typically defined in one module, and used in other ones. Imported types or modules may also be reexported (*e.g.* t_1 in B) to be used in other modules (*e.g.* in C). (These dependencies are represented by arrows, labeled with type identities.)

The whole program is well-typed if it can be checked that, *e.g.* the type t_1 defined in A seen from B and seen from C are actually the same type, *i.e.* that they originate from the same module A_1, and not from a copy of A_2.

Modules are bound to *variables* (*e.g.* X_A binds A). Their imports and exports are *named* with labels (*e.g.* t, M).

Type definition t_1 coming from A is seen in module B as the path $X_A.t$ where t is the named under which t_1 is exported from A. B may reexport module A under the name M. Then t_1 may be seen in C under the *path* $B.M.t$



Abstract types

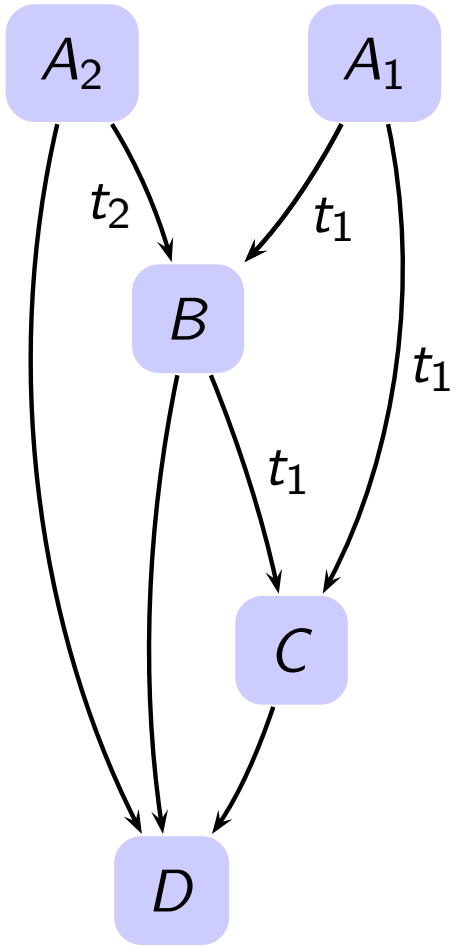
Path-based approach

Paths are used in a critical way

- to identify abstract types by *where* they have been defined.

Paths are also used (in a not so critical way)

- to access other definitions (expressions, submodules).



Syntax

Example (1) in OCaml syntax

```
module type INT = sig
    type t
    val zero : t;
    val succ : t → t
  end
module Int1 : INT = struct
    type t = int
    let zero = 0;
    let succ = λ(x) x+1;
  end
module Int2 : INT = Int1;
let rejected = Int1 . succ Int2 . zero
```

Abstract types preserve accidental merging of two identical concrete types that are semantically different.

For instance Int_1 and Int_2 implement accounting in two different currencies.

This is ill-typed because

$Int_1 . succ$ and $Int_2 . zero$

have types

$Int_1 . t \rightarrow Int_1 . t$ and $Int_2 . t$,

which are incompatible, because of the signature constrained

$Int_1 : INT$ and $Int_2 : INT$.

This is type generativity.

Syntax Simplified

$$
\begin{array}{l}
(\varphi)\{ \\
\quad INT = (\psi)\{ \\
\qquad t : 0; \\
\qquad zero : \psi.t; \\
\qquad succ : \psi.t \to \psi.t \\
\quad \} \\
\quad Int_1 = (\psi)\{ \\
\qquad t = int; \\
\qquad zero = 0; \\
\qquad succ = \lambda(x)\ x{+}1 \\
\quad \} : \varphi.INT; \\
\quad Int_2 = \varphi.Int_1 : \varphi.INT; \\
\quad rejected = \varphi.Int_1\,.\,succ\ \ \varphi.Int_2\,.\,zero \\
\}
\end{array}
$$

- Structures and signatures use record notation $(\varphi)\{\ldots\}$.
- Variable φ, which binds the structure (or signature) is used to refer to *previous* definitions of the same structure (or signature).
- We omit qualifiers (type, val, module) of field names. Field names cannot be renamed.
- Syntactic sugar may be used.

Syntax

Sugared

We use syntactic sugar to recover lighter syntax, while retaining a clear distinction between fields (which cannot be renamed) and variables:

$$\begin{array}{l}\{\ \mathit{INT} = \{\ t : 0;\ \mathit{zero} : t;\ \mathit{succ} : t \to t\ \};\\ \quad \mathit{Int}_1 : \mathit{INT} = \{\ t = \mathit{int};\ \mathit{zero} = 0;\ \mathit{succ} = \lambda(x)\ x{+}1\ \};\\ \quad \mathit{Int}_2 : \mathit{INT} = \mathit{Int}_1;\\ \quad \mathit{rejected} = \mathit{Int}_1\,.\,\mathit{succ}\ \mathit{Int}_2\,.\,\mathit{zero}\ \}\end{array}$$

The meaning is by desugaring

- $\{\bar{d}\}$ stands for $(\varphi)\{\bar{d}\}$ when φ does not appear in $\bar{d}$.
- A label ℓ stands for $\varphi.\ell$ where φ is the variable binding the enclosing structure or signature.
- For example, $\{\ell_1 = 0; \ell_2 = \ell_1 + 1\}$ means $(\varphi)\{\ell_1 = 0; \ell_2 = \varphi.\ell_1 + 1\}$.
- Additional syntactic sugar, such as $\ell : \mathcal{S} = \mathcal{M}$ for $\ell = (\mathcal{M} : \mathcal{S})$ may be used for convenience.

Syntax

Paths

Identifiers

$$
\begin{array}{lcll}
\varphi & ::= & \varphi_1 \mid \varphi_2 \mid \ldots & \text{Variables} \\
\ell & ::= & \ell_1 \mid \ell_2 \mid \ldots & \text{Labels} \\
\pi & ::= & \varphi \mid \pi.\ell & \text{Paths}
\end{array}
$$

Remarks

- For brevity, we use a single collection of variables and a single collection of labels for naming expressions, types, and modules.
- Bound variables can be freely renamed, but labels cannot.
- Paths are used to designate components of a structure bound to a (module) variable that is projected along a sequence of labels.

Syntax

Base language

Types

τ	$::=$	$\pi \mid \tau \to \tau \mid \mathit{int} \mid \ldots$	Types
σ	$::=$	$\forall \bar{\alpha}.\tau$	Type schemes

Expressions

v	$::=$	$\pi \mid \lambda(\varphi)\ a \mid 0 \mid 1 \mid \ldots$	Value forms
a	$::=$	$v \mid a(a)$	Expressions

Syntax — Module language

Definitions

$$
\begin{array}{rcll}
d & ::= & & \\
& | & \ell = \tau & \\
& | & \ell = a & \\
& | & \ell = \mathcal{M} &
\end{array}
$$

Specifications

$$
\begin{array}{rcll}
s & ::= & \ell : 0 & \text{Abstract type}^{\dagger} \\
& | & \ell = \tau & \text{Type definition} \\
& | & \ell : \sigma & \text{Expression} \\
& | & \ell : \mathcal{S} & \text{Module}
\end{array}
$$

Modules

$$
\begin{array}{rcll}
\mathcal{M} & ::= & \pi & \\
& | & (\varphi)\{d; \ldots d\} & \\
& | & \mathcal{M}\,(\mathcal{M}) & \\
& | & \lambda(\varphi : \mathcal{S})\,\mathcal{M} &
\end{array}
$$

Signatures

$$
\begin{array}{rcll}
\mathcal{S} & ::= & \pi & \\
& | & (\varphi)\{s; \ldots s\} & \text{Submodules} \\
& & & \text{Applications} \\
& | & (\varphi : \mathcal{S}) \to \mathcal{S} & \text{Functors}
\end{array}
$$

We assume that no label is repeated in submodules and their signatures.

†. We restrict to nullary abstract types, otherwise the arity n would replace 0.

Syntax

Naming convention

Although we formally have a single set of variables and of a single set of labels, in examples, we often distinguish the category of objects that they bind or name by using different letters for variables and labels as described below (or use extended names following the conventions of OCaml)

Category	expression	variable	label
type	τ	α, β	t, u
signature	$\mathcal{S}$	–	S, T, UPPERCASE
expression	a	x, y	m, n, v, f, lowercase
module	$\mathcal{M}$	X, Y	M, N, V, F, Capitalized
any		φ, ψ	ℓ

Typing rules

Contexts

Typing contexts

$$\Gamma \quad ::= \quad \emptyset \mid \Gamma, \pi : \sigma \mid \Gamma, \pi : \mathcal{S} \mid \Gamma, \pi : 0 \mid \Gamma, \pi = \tau$$

We assume that Γ never binds the same path twice.

If path π starts with variable φ and $\pi \in \mathit{dom}\,\Gamma$, then we write $\varphi \in \mathit{dom}\,\Gamma$.

We allow projection on paths when reading Γ. If b stands for 0, τ, or $\mathcal{S}$,

$$\textsc{Hyp}\ \frac{\pi : b \in \Gamma}{\Gamma \vdash \pi : b} \qquad \textsc{Proj}\ \frac{\Gamma \vdash \pi : (\varphi)\{\bar{s}_1; \ell : b; \bar{s}_2\}}{\Gamma \vdash \pi.\ell : b[\pi/\varphi]}$$

Remarks

- Signature variable φ in b has no *identity*, but path π originating from a value variable has one.
- Substitution $b[\pi/\varphi]$ forces references to previous components to go via π so as to preserve sharing of identities.
- Inlining these references inside b would loose sharing of abstract types (and also be ill-formed for abstract types)

Typing rules

Example

Let $\mathcal{M}$ be $(\varphi)\{\mathsf{t} = \mathit{int}; \mathsf{u} = \varphi.\mathsf{t}; \mathsf{m} = 1\}$. A possible signature $\mathcal{S}$ for for $\mathcal{M}$ is $(\varphi)\{\mathsf{t} = \mathit{int}; \mathsf{u} = \varphi.\mathsf{t}; \mathsf{m} : \varphi.\mathsf{u}\}$,

Assume that $X : \mathcal{S} \in \Gamma$. We then have $\Gamma \vdash X : \mathcal{S}$.

By projection we also have $\Gamma \vdash X.\mathsf{m} : X.\mathsf{u}$.

Notice that the signature of $X.\mathsf{m}$ still refers to X, but not to φ. That is, occurrences of X have not been recursively eliminated.

In particular, we do not have $\Gamma \vdash X.\mathsf{m} : \mathit{int}$ by projection alone.

However, this judgment holds by equivalence.

Typing rules — Type equivalence

Type equivalence $\approx$ is generated from type definitions

that are directly or indirectly bound in Γ.

$$\frac{\pi = \tau \in \Gamma}{\Gamma \vdash \pi \approx \tau} \qquad \frac{\Gamma \vdash \pi : (\varphi)\{\bar{d}_1; \ell = \tau; \bar{d}_2\}}{\Gamma \vdash \pi.\ell \approx \tau[\pi/\varphi]}$$

Type equivalence is congruent for all type and module type constructors

(Equivalence (Ref, Sym, Trans) and congruence rules are omitted)

Type equivalence is used with the following conversion rules

$$\frac{\Gamma \vdash a : \sigma \qquad \Gamma \vdash \sigma \approx \sigma'}{\Gamma \vdash a : \sigma'} \qquad \frac{\Gamma \vdash \mathcal{M} : \mathcal{S} \qquad \Gamma \vdash \mathcal{S} \approx \mathcal{S}'}{\Gamma \vdash \mathcal{M} : \mathcal{S}'}$$

Example (continued)

We had $\Gamma \vdash X.\mathrm{m} : X.\mathrm{u}$. By type equivalence rules, we have $\Gamma \vdash X.\mathrm{u} \approx X.t$ and $\Gamma \vdash X.\mathrm{t} \approx \mathit{int}$, thus $\Gamma \vdash X.\mathrm{u} \approx \mathit{int}$. Finally, $\Gamma \vdash X.\mathrm{m} : \mathit{int}$ follows by the conversion rule.

Typing rules

Structures

$$\frac{\Gamma \vdash_\varphi \bar{d} : \bar{s} \qquad \varphi \notin dom\,\Gamma}{\Gamma \vdash (\varphi)\{\bar{d}\} : (\varphi)\{\bar{s}\}}$$

Functors

$$\frac{\Gamma \vdash \mathcal{S}_1 \qquad \varphi \notin dom\,\Gamma \qquad \Gamma, \varphi : \mathcal{S}_1 \vdash \mathcal{M} : \mathcal{S}_2}{\Gamma \vdash \lambda(\varphi : \mathcal{S}_1)\mathcal{M} \ : (\varphi : \mathcal{S}_1) \to \mathcal{S}_2}$$

Applications

$$\frac{\Gamma \vdash \mathcal{M}_1 : (\varphi : \mathcal{S}_2) \to \mathcal{S}_1 \qquad \Gamma \vdash \mathcal{M}_2 : \mathcal{S}_2}{\Gamma \vdash \mathcal{M}_1(\mathcal{M}_2) : \mathcal{S}_1[\mathcal{M}_2/\varphi]}$$

Typing rule for module application is the standard rule for elimination of dependent types, but restricted to path-dependent types. Thus:

$\mathcal{S}_1[\mathcal{M}_2/\varphi]$ is ill-defined and $\mathcal{M}_1\ (\mathcal{M}_2)$ ill-typed if $\mathcal{M}_2$ is not a path and φ occurs free in $\mathcal{S}_1$.

Typing rules — Declarations

Sequences of declarations are typed by folding typing of individual declarations, making previous declarations visible to the current one.

$$\frac{}{\Gamma \vdash_\varphi \emptyset : \emptyset} \qquad \frac{\Gamma \vdash \tau \qquad \Gamma, \varphi.\ell = \tau \vdash_\varphi \bar{d} : \bar{s}}{\Gamma \vdash_\varphi (\ell = \tau; \bar{d}) \ : \ (\ell : \tau; \bar{s})}$$

$$\frac{\Gamma \vdash a : \sigma \qquad \Gamma, \varphi.\ell : \sigma \vdash_\varphi \bar{d} : \bar{s}}{\Gamma \vdash_\varphi (\ell = a; \bar{d}) : (\ell : \sigma; \bar{s})} \qquad \frac{\Gamma \vdash \mathcal{M} : \mathcal{S} \qquad \Gamma, \varphi.\ell : \mathcal{S} \vdash_\varphi \bar{d} : \bar{s}}{\Gamma \vdash_\varphi (\ell = \mathcal{M}; \bar{d}) \ : \ (\ell : \mathcal{S}; \bar{s})}$$

Typing rules for the base-language are omitted:

- *Well-formedness of types $\Gamma \vdash \tau$ are straightforward;*
- *Typing of expressions, $\Gamma \vdash a : \sigma$ are as in ML (see previous lessons) .*

In fact, the module language can be parametrized by typing rules for the base language. See Leroy (2000)

Typing rules

Signatures

$$\frac{\pi \in dom\,\Gamma}{\Gamma \vdash \pi} \qquad \frac{\Gamma \vdash \mathcal{S}_1 \qquad \Gamma, \varphi : \mathcal{S}_1 \vdash \mathcal{S}_2}{\Gamma \vdash (\varphi : \mathcal{S}_1) \to \mathcal{S}_2} \qquad \frac{\Gamma \vdash_\varphi \overline{s}}{\Gamma \vdash (\varphi)\{\overline{s}\}}$$

Declarations (we fold well-formedness of individual)

$$\Gamma \vdash_\varphi \emptyset \qquad \frac{\Gamma, \varphi.\ell : 0 \vdash \overline{s}}{\Gamma \vdash_\varphi \ell : 0; \overline{s}} \qquad \frac{\Gamma \vdash \tau \qquad \Gamma, \varphi.\ell : \tau \vdash \overline{s}}{\Gamma \vdash_\varphi \ell : 0; \overline{s}}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma, \varphi.\ell = \tau \vdash \overline{s}}{\Gamma \vdash_\varphi \ell = \tau; \overline{s}} \qquad \frac{\Gamma \vdash \mathcal{S} \qquad \Gamma, \ell : \mathcal{S} \vdash \overline{s}}{\Gamma \vdash_\varphi \ell : \mathcal{S}; \overline{s}}$$

We omit well-formedness of types $\Gamma \vdash \tau$, which is as in the base language.

Subtyping

What is missing?

$$
\begin{array}{l}
(\varphi)\{ \\
\quad t = \mathit{int} \\
\quad \mathit{zero} = 0; \\
\quad \mathit{one} = 1; \\
\quad \mathit{succ} = \lambda(x)\ x + \varphi.\mathit{one}; \\
\}
\end{array}
\qquad \text{has type?} \qquad
\begin{array}{l}
(\varphi)\{ \\
\quad t\ :\ 0; \\
\quad \mathit{zero}\ :\ \mathit{int}; \\
\\
\quad \mathit{succ}\ :\ \mathit{int} \rightarrow \mathit{int} \\
\}
\end{array}
$$

So far, we do not have subtyping, hence module components cannot be hidden and cannot be given abstract types[1].

The solution is to permit hiding by subtyping:

- a structure with more components can always be seen as if it had fewer components.
- a concrete type definition can be seen as an abstract type.

1. except for polymorphic functions

Subtyping

At the leaves

Subtyping contains the equivalence.

$$\frac{\Gamma \vdash \tau_1 \approx \tau_2}{\Gamma \vdash (\ell = \tau_1) <: (\ell = \tau_2)}$$

Subtyping allows to make concrete type definitions abstract.

$$\Gamma \vdash (\ell = \tau) <: (\ell : 0) \qquad \Gamma \vdash (\ell : 0) <: (\ell : 0)$$

Subtyping contains equivalence and instantiation of type schemes

$$\frac{\Gamma \vdash \sigma_1 \approx \sigma_2}{\Gamma \vdash \sigma_1 <: \sigma_2} \qquad \frac{\Gamma \vdash \sigma <: \forall \bar{\alpha}.\tau_0 \qquad \bar{\beta} \notin \mathit{fv}(\forall \bar{\alpha}.\tau_0)}{\Gamma \vdash \sigma <: \forall \bar{\beta}.\tau_0[\bar{\tau}/\bar{\alpha}]} \qquad \frac{\Gamma \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash (\ell : \sigma_1) <: (\ell : \sigma_2)}$$

Subtyping

For structures

Subtyping allows to omit component of structures.

$$\frac{\Gamma \vdash (\varphi)\{\bar{s}_1\} \qquad \Gamma \vdash (\varphi)\{\bar{s}_2\} \qquad \forall s_2 \in \bar{s}_2, \exists s_1 \in \bar{s}_1, \quad \Gamma, \overline{\varphi.s_1} \vdash_\varphi s_1 <: s_2}{\Gamma \vdash (\varphi)\{\bar{s}_1\} <: (\varphi)\{\bar{s}_2\}}$$

Key ideas

- each component of the result signature must be also be defined in the original signature, but it may be a subtype of the original.
- type definitions of the original signature may be used to check subtyping of retained components.
- components that are referred from retained components must also be retained (so that the resulting signature is well-formed).

Subtyping — Submodules and functors

Propagation

Subtyping of signatures propagates contravariantly on the left of functors and covariantly everywhere else:

$$\frac{\Gamma \vdash \mathcal{S}_1 <: \mathcal{S}_2}{\Gamma \vdash (\ell : \mathcal{S}_1) <: (\ell : \mathcal{S}_2)} \qquad \frac{\Gamma \vdash \mathcal{S}'_1 <: \mathcal{S}_1 \qquad \Gamma, \varphi : \mathcal{S}'_1 \vdash \mathcal{S}_2 <: \mathcal{S}'_2}{\Gamma \vdash (\varphi : \mathcal{S}_1) \to \mathcal{S}_2 <: (\varphi : \mathcal{S}'_1) \to \mathcal{S}'_2}$$

Subtyping for non-dependent types

If φ appears neither in $\mathcal{S}_2$ nor in $\mathcal{S}'_2$, the subtyping rules looks familiar:

$$\frac{\Gamma \vdash \mathcal{S}'_1 <: \mathcal{S}_1 \qquad \Gamma \vdash \mathcal{S}_2 <: \mathcal{S}'_2}{\Gamma \vdash \mathcal{S}_1 \to \mathcal{S}_2 <: \mathcal{S}'_1 \to \mathcal{S}'_2}$$

Subtyping — Comparison with $F_{:>}$

Remark

The subtyping rule for functor looks similar to the subtyping rule for bounded quantification in the language $F_{:>}$ (read Fsub), but it is in fact quite different: φ is a module variable and not a signature variable which is assumed to have exactly (as opposed to a subtype of) signature $\mathcal{S}_1$ or $\mathcal{S}'_1$.

In particular, we cannot reason under subtyping assumptions as in $F_{:>}$.

Checking subtyping for modules remains decidable (and relatively easy) while checking subtyping for (the most permissive version of) $F_{:>}$ is not.

Subtyping

Sealing

Subtyping is implicit and can be used anywhere:

$$\frac{\Gamma \vdash \mathcal{M} : \mathcal{S} \qquad \Gamma \vdash \mathcal{S} <: \mathcal{S}'}{\Gamma \vdash \mathcal{M} : \mathcal{S}'}$$

Although this may turn manifest type definitions into abstract ones, abstraction is not performed unless explicitly required, since principal signatures are always inferred.

We introduce a construct to enforce subtyping, called *sealing*:

$$\mathcal{M} \quad ::= \quad \ldots \mid (\mathcal{M} : \mathcal{S}) \qquad\qquad \frac{\Gamma \vdash \mathcal{M} : \mathcal{S}}{\Gamma \vdash (\mathcal{M} : \mathcal{S}) : \mathcal{S}}$$

Subtyping — Sealing

Sealing is generative

If $\mathcal{S}$ is a module signature with an abstract type t, then $\mathcal{M}$ and $(\mathcal{M} : \mathcal{S})$ have incompatible views of t. (See example (1))

Example

Let $\mathcal{M}_N$ be $\{\mathsf{t} = \mathit{int}; \mathsf{m} = 1\}$ and $\mathcal{S}_N$ be $(\varphi)\{\mathsf{t} : 0; \mathsf{m} : \varphi.\mathsf{t}\}$.
Then, the following definition fails:

$$(\varphi)\left\{\begin{array}{l} \mathsf{M} = \mathcal{M}_N; \\ \mathsf{N} = (\varphi.\mathsf{M} : \mathcal{S}_N); \\ \mathsf{m} = (\varphi.\mathsf{M}.\mathsf{m} = \varphi.\mathsf{N}.\mathsf{m}) \end{array}\right\} \quad \longleftarrow \text{ here}$$

because φ.M.m and φ.N.m have different abstract types φ.M.t and φ.N.t

Subtyping — Subsumption

Instead of implicit subtyping which may float anywhere, we may restrict uses of subtyping at functor applications and sealings:

$$\frac{\Gamma \vdash \mathcal{M}_1 : (\varphi : \mathcal{S}_2) \to \mathcal{S}_1 \qquad \Gamma \vdash \mathcal{M}_2 : \mathcal{S}_2' \qquad \Gamma \vdash \mathcal{S}_2' <: \mathcal{S}_2}{\Gamma \vdash \mathcal{M}_1(\mathcal{M}_2) : \mathcal{S}_1[\mathcal{M}_2/\varphi]}$$

$$\frac{\Gamma \vdash \mathcal{M} : \mathcal{S} \qquad \Gamma \vdash \mathcal{S} <: \mathcal{S}'}{\Gamma \vdash (\mathcal{M} : \mathcal{S}') : \mathcal{S}'}$$

Note that subtyping is not performed on the result of functor application.

This is in fact more restrictive as it disallows the use of subtyping to avoid ill-formed applications (see the avoidance problem below).

This is also used for type inference.

Typing rules

Example (2)

Give the best type to the following declarations

$$\left\{\begin{array}{l} \mathsf{F} = \lambda(X : \mathcal{S}_N)\ X; \\ \mathsf{M} = \mathcal{M}_N; \\ \mathsf{N} = \mathsf{F}\ (\mathsf{M}); \\ \mathsf{m} = \mathsf{N.m} + 1; \end{array}\right\}$$

Typing rules

Example (2)

Give the best type to the following declarations

$$\left\{\begin{array}{l} \mathsf{F} = \lambda(X : \mathcal{S}_N)\ X; \\ \mathsf{M} = \mathcal{M}_N; \\ \mathsf{N} = \mathsf{F}\ (\mathsf{M}); \\ \mathsf{m} = \mathsf{N.m} + 1; \end{array}\right\} \quad : \quad \left\{\begin{array}{l} \mathsf{F} = (X : \mathcal{S}_N) \to \mathcal{S}_N; \\ \mathsf{M} = \{\mathsf{t} = \mathit{int}; \mathsf{m} : \mathit{int}\}; \\ \mathsf{N} = \mathcal{S}_N \\ \mathsf{m}\ \ \mathit{is}\ \ \mathit{ill\text{-}typed} \end{array}\right\}$$

(The typing of applications will be improved later with strengthening)

Strengthening

What is missing?

Problem

The following example fails, as before,

$$(\varphi)\left\{\begin{array}{l} \mathsf{M} = (\mathcal{M} : \mathcal{S}); \\ \mathsf{N} = \varphi.\mathsf{M}; \\ \mathsf{m} = (\varphi.\mathsf{M}.\mathsf{m} = \varphi.\mathsf{N}.\mathsf{m}) \end{array}\right\} \quad \longleftarrow \text{ here}$$

because $\varphi.\mathsf{M}.\mathsf{m}$ and $\varphi.\mathsf{N}.\mathsf{m}$ have different abstract types $\varphi.\mathsf{M}.\mathsf{t}$ and $\varphi.\mathsf{N}.\mathsf{t}$.

Solution

- What is the type of $\varphi.\mathsf{N}$?
- Asume $\varphi.\mathsf{M}$ can be given type $\{\mathsf{t} = \varphi.\mathsf{M}; \mathsf{m} : \varphi.\mathsf{t}\}$. Is $\varphi.m$ well-typed?

Strengthening — Solution

Intuitively, we add

$$\frac{\Gamma \vdash \pi : (\varphi)\{\bar{d}_1; \ell : 0; \bar{d}_2\}}{\Gamma \vdash \pi : (\varphi)\{\bar{d}_1; \ell = \pi.\ell; \bar{d}_2\}}$$

More generally, we allow strengthening of type definitions and submodules as follows:

$$\frac{\Gamma \vdash \pi : \mathcal{S}}{\Gamma \vdash \pi : \mathcal{S}/\pi}$$

where

$$\begin{array}{rcl} (\varphi)\{d_1, \ldots d_n\}/\pi & = & (\varphi)\{d_1/\pi, \ldots d_n/\pi\} \\ \pi'/\pi & = & \pi' \\ (\varphi : \mathcal{M}_1) \to \mathcal{M}_2/\pi & = & (\varphi : \mathcal{M}_1) \to \mathcal{M}_2 \end{array}$$

$$\begin{array}{rcl} \ell : \sigma/\pi & = & \ell : \sigma \\ \ell : \mathcal{M}/\pi & = & \ell : (\mathcal{M}/\pi.\ell) \\ \ell : 0/\pi & = & \ell = \pi.\ell \\ \ell = \tau/\pi & = & \ell = \pi.\ell \end{array}$$

Strengthening

Exercise

Exercise

Explain (informally) why $\Gamma \vdash p : \mathcal{S}$ *implies* $\Gamma \vdash \mathcal{S}/p <: \mathcal{S}$. Answer

Why is this important? Answer □

Strengthening

Exercise

Exercise

Consider the program:

$$
\begin{array}{l}
\{\ F = \lambda(X : \{t : 0\})\ \lambda(Y : \{t = X.t\})\ \{\}; \\
\quad A : \{\ t\ :\ 0\ \}; \\
\quad B : \{\ t\ :\ A.t\ \}; \\
\quad M_0 = F\ (A)\ (B); \\
\quad M_1 = F\ (B)\ (A); \\
\quad M_2 = F\ (A)\ (A); \\
\}
\end{array}
$$

Which of the applications are well-typed without strengthening?

With strengthening? □

Strengthening

Exercise

Exercise

Consider the program:

$$
\begin{array}{l}
\{\ F = \lambda(X : \{t : 0\})\ \lambda(Y : \{t = X.t\})\ \{\}; \\
\quad A : \{\ t\ :\ 0\ \}; \\
\quad B : \{\ t\ :\ A.t\ \}; \\
\quad M_0 = F\ (A)\ (B); \qquad \textit{ok} \\
\quad M_1 = F\ (B)\ (A); \qquad \textit{fails} \\
\quad M_2 = F\ (A)\ (A); \qquad \textit{fails} \\
\}
\end{array}
$$

Which of the applications are well-typed without strengthening? *In both cases, f requires its second argument to be concrete, but it is abstract.*

With strengthening? *They all succeed.* □

Strengthening

Exercise

Consider again the example (2)

Give the best type the following declarations:

$$\left\{\begin{array}{l} \mathsf{F} = \lambda(X : \mathcal{S}_N)\ X; \\ \mathsf{M} = \mathcal{M}_N; \\ \mathsf{N} = \mathsf{F}\ (\mathsf{M}); \\ \mathsf{m} = \mathsf{N.m} + 1; \end{array}\right\}$$

Why is this a justification of sealing?

- Without strengthening, an application of $\lambda(X : \mathcal{S})\ X$ to $\mathcal{M}$ would be equivalent to sealing $\mathcal{M}$ with $\mathcal{S}$ and sealing would be useless.
- With strengthening, an abstract type in a functor parameter signature is only seen abstract in the body of the functor, but is not made abstract in the result of a functor application. Informally, it is as if

$$\lambda(X : \mathcal{S})\ \mathcal{M} \qquad \text{meant} \qquad \Lambda(\varphi <: \mathcal{S})\ \lambda(X : \varphi)\ \mathcal{M}$$

Strengthening

Exercise

Consider again the example (2)

Give the best type the following declarations:

$$\left\{\begin{array}{l} \mathsf{F} = \lambda(X : \mathcal{S}_N)\ X; \\ \mathsf{M} = \mathcal{M}_N; \\ \mathsf{N} = \mathsf{F}\ (\mathsf{M}); \\ \mathsf{m} = \mathsf{N.m} + 1; \end{array}\right\} : \left\{\begin{array}{l} \mathsf{F} = (X : \mathcal{S}_N) \to (\varphi)\{\mathsf{t} = X.\mathsf{t}; \mathsf{M} : \varphi.\mathsf{t}\}; \\ \mathsf{M} = \{\mathsf{t} = \mathit{int}; \mathsf{m} : \mathit{int}\}; \\ \mathsf{N} = (\varphi)\{\mathsf{t} = \mathsf{M.t}; \mathsf{M} : \varphi.\mathsf{t}\}; \\ \mathsf{m} : \mathit{int} \end{array}\right\}$$

Why is this a justification of sealing?

- Without strengthening, an application of $\lambda(X : \mathcal{S})\ X$ to $\mathcal{M}$ would be equivalent to sealing $\mathcal{M}$ with $\mathcal{S}$ and sealing would be useless.
- With strengthening, an abstract type in a functor parameter signature is only seen abstract in the body of the functor, but is not made abstract in the result of a functor application. Informally, it is as if

$$\lambda(X : \mathcal{S})\ \mathcal{M} \qquad \text{meant} \qquad \Lambda(\varphi <: \mathcal{S})\ \lambda(X : \varphi)\ \mathcal{M}$$

Strengthening

Summary

Strengthening plays a key role in typing of modules

- It is at the very heart of the propagation of type equalities.
- It enhances functor application in an essential way, by specializing the abstract signature of the formal parameters to that of the actual arguments, performing a form of implicit type instantiation.
- Thanks to strengthening, functors are parametric in all specialized versions of the signature of the arguments.

However

- Strengthening proceeds by replacing type definitions (concrete or abstract) by new type aliases (indirections) to previous definitions rather than *adding* new type equations to already existing ones.
- Strengthening remains somewhat an *ad hoc* treatment of some underlying equational theory on paths.

Type inference

Signatures for modules may be inferred

- In ML, base-language definitions have principal types which may be inferred.
- Sequences of definitions may also be inferred.
- Signatures of functors must be provided, indeed.

Potential problems for inference (discussed next)

- Non-regular datatype definitions.
- Value restriction and non closed signatures.
- Local module definitions and the avoidance problem.
- Subtyping instead of subsumption.

It is *conjectured* that type inference returns a principal signatures when it succeeds. However, it might fail when perhaps more specific type annotations would make it accepted.

Type inference

Non-regular datatypes

If the host language has non-regular type definitions, checking for equivalence of type definitions becomes undecidable.

This is not a problem for the host language since, there is no need to test for the equality of type definitions.

However, this is a potential problem for a module language.

A solution is to compare datatype definitions syntactically instead of semantically.

Type inference — Value restriction

In OCaml, well-formed signatures must be closed, *i.e.* have no free type variables. This is usually fine with ML style polymorphism since expressions can be generalized at *toplevel*, hence at the module level.

However, the value-restriction prohibits generalization of non value forms. For example, $\{id = (\lambda(x)\ x)\ (\lambda(x)\ x)\}$ can be typed with $\{id : \alpha \to \alpha\}$ but not with $\{id : \forall\alpha.\alpha \to \alpha\}$.

The common solution (also followed in OCaml) is to reject such programs, although they could also be ascribed legal but non-principal signatures.

In our example, $\{id : \tau \to \tau\}$ would be a correct signature for any ground type τ, such as *int*, *bool* $\to$ *bool*, *etc.*

Note: there are solutions that allow signatures with free type variables, but this implies mixing base-language level and module level type inference; they are also more complex. See Dreyer and Blume (2007)

Type inference — The avoidance problem

This is a general problem in a language with subtyping and abstract types.

It is always incorrect for an abstract type to escape its scope. When subtyping is allows, it is sometimes possible to hide by subtyping components that would otherwise lead to ill-formed types. The question is whether this can be done in a principal manner.

The problem arises with local module definitions For example, if module expressions can be of the form let $m = \mathcal{M}$ in $\mathcal{M}$. Then, the module

$$\begin{array}{l}\text{let } X : (\varphi)\{\mathsf{M} : \{\mathsf{t} = \mathit{int}\}; \mathsf{N} : \{\mathsf{m} : \varphi.\mathsf{M.t}\}\} = \\ \qquad\qquad \{\mathsf{M} = \{\mathsf{t} = \mathit{int}\}; \mathsf{N} = \{\mathsf{m} = 1\}\} \\ \text{IN } X.\mathsf{N.m}\end{array}$$

has principal signature $\{\mathsf{m} : X.\mathsf{M.t}\}$, which is ill formed since X is not in scope.

In this case, it can also be given the equivalent signature $\{\mathsf{m} : \mathit{int}\}$ that *avoids* X, by conversion.

Type inference — The avoidance problem

This is no more the case if X.N.t is abstract, as in:

$$\begin{array}{l}\text{let } X : (\varphi)\{\mathsf{M} : \{\mathsf{t} = 0\}; \mathsf{N} : \{\mathsf{m} : \varphi.\mathsf{M.t}\}\} = \\ \qquad\qquad \{\mathsf{M} = \{\mathsf{t} = \mathit{int}\}; \mathsf{N} = \{\mathsf{m} = 1\}\} \\ \text{IN } X.\mathsf{N.m}\end{array}$$

The principal signature is still $\{\mathsf{m} : X.\mathsf{M.t}\}$ but now m cannot be avoided, except by subtyping, leading to the empty signature $\{\}$. In this case, this is a principal type for $\mathcal{M}$.

Unfortunately, this is not always always the case.

With subsumption instead of subtyping (see above) this example is rejected (because subtyping cannot be used implicitly).

Type inference

Can make the difference!

With all explicit type information as in System F, *i.e.* all type abstraction and type applications explicitly written, programs may be quite verbose.

In fact, type information may even be in $O(n^2)$ the size of the underlying program stripped of all type information.

(Consider for example, large tuples encoded with pairs.)

Type inference allows for not writing all type information and in some cases keep the source program manageable.

There are also examples where a small change in the source program may induce a much larger change in the typing derivation, hence in the explicitly typed term, while the type erasure of the program will change very little. In such cases, type inference may increase modularity.

Signature definitions

A module may also contain (concrete) signature definitions:

$$d \quad ::= \quad \ldots \mid \ell = \mathcal{S} \qquad\qquad s \quad ::= \quad \ldots \mid \ell = \mathcal{S}$$

Typing rules:

$$\frac{\Gamma \vdash_\varphi \mathcal{S} \qquad \Gamma, \varphi.\ell = \mathcal{S} \vdash \mathcal{M} : \mathcal{S}'}{\Gamma \vdash_\varphi (\ell = \mathcal{S}; \mathcal{M}) : (\ell = \mathcal{S}; \mathcal{S}')} \qquad\qquad \frac{\pi = \mathcal{S} \in \Gamma}{\Gamma \vdash \pi \approx \mathcal{S}}$$

This does not increase expressiveness, since as type definitions alone, signature definitions could always be expanded.

However, this increases conciseness and clarity by avoiding repeating the same signature many times.

Signature definitions — The with notation

The construction is used to change the type definitions of a signature:

$$\mathcal{S} \quad ::= \quad \ldots \mid \mathcal{S} \text{ with } \bar{\ell} = \tau$$

Informally, this refines the type definition ℓ in $\mathcal{S}$ to be equal to τ (which must be compatible with the old definition τ).

The with notation is often used when $\mathcal{S}$ is signature definition π.S. It can always be eliminated by inlining π.S and replacing the component $\bar{\ell}$ by τ.

For example, the last line of

$$\left\{ \begin{array}{l} \mathsf{S} = \{\mathsf{t} : 0; m : \mathsf{t}\}; \\ \mathsf{M} = \lambda(X : \mathsf{S})\ \lambda(Y : \mathsf{S} \text{ with } \mathsf{t} = X.\mathsf{t})\ \mathcal{M} \end{array} \right\}$$

could be also be written $\mathsf{M} = \lambda(X : \mathsf{S})\ \lambda(Y : \{\mathsf{t} : X.\mathsf{t}; m : \mathsf{t}\})\ \mathcal{M}$.

The with notation does not increases expressiveness, *alone*.
However, it avoids dupplication of code increases maintainability.

Signature definitions — The with notation

It may be formalized using the equivalence relation:

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash \mathcal{S} \approx (\varphi)\{\bar{d}_1, \ell : 0, \bar{d}_2\}}{\Gamma \vdash (\mathcal{S} \text{ with } \ell = \tau) \approx (\varphi)\{\bar{d}_1, \ell = \tau, \bar{d}_2\}}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash \mathcal{S} \approx (\varphi)\{\bar{d}_1, \ell = \tau, \bar{d}_2\}}{\Gamma \vdash (\mathcal{S} \text{ with } \ell = \tau) \approx (\varphi)\{\bar{d}_1, \ell = \tau, \bar{d}_2\}}$$

The with notation may also operate in submodules:

$$\frac{\Gamma \vdash \mathcal{S} \approx (\varphi)\{\bar{d}_1, \ell_1 = \mathcal{S}_1, \bar{d}_2\} \qquad \Gamma \vdash (\mathcal{S}_1 \text{ with } \bar{\ell} = \tau) \approx \mathcal{S}_1'}{\Gamma \vdash (\mathcal{S} \text{ with } \ell_1.\bar{\ell} = \tau) \approx (\varphi)\{\bar{d}_1, \ell_1 = \mathcal{S}_1', \bar{d}_2\}}$$

Abstract signatures

We allow abstract signatures in specifications:

$$s \quad ::= \quad \ldots \mid \ell$$

A type component of a functor parameter may be an abstract signature, which gets instantiated to a concrete signature when the functor is applied.

Additional typing rules:

$$\Gamma \vdash_\varphi (\ell = \mathcal{S}) <: (\ell) \qquad \frac{\pi \in \Gamma}{\Gamma \vdash \pi}$$

For example, the application functor *App* may be written:

$$\lambda(\varphi : \{\mathsf{S}; \mathsf{T}\})\ \lambda(F : (X : \varphi.\mathsf{S}) \to \varphi.\mathsf{T})\ \lambda(X : \varphi.\mathsf{S})\ F\ (X)$$

It allows to reach all of F^ω (higher-order polymorphism), but programming with polymorphic signatures is a bit heavy.

Abstract signatures inscrease expressiveness (and with them, the with notation now also increases expressiveness.)

(We still cannot reason under subtyping assumptions and reach all of $\mathsf{F}_{:>}$.)

Abstract signatures

Exercise

Rewrite App in OCaml syntax. Write the identity Id functor in a similar style. Verify that App can be applied to the identity Id specialized at any signature $\mathcal{S}$.
Rewrite the example by passing App and Id to an appropriately specialized version of App instead of using the primitive application. Answer □

Sharing in signatures

With type equalities

Consider again the example:

$$\left\{\begin{array}{l} \mathsf{S} = \{\mathsf{t} : 0; m : \mathsf{t}\}; \\ \mathsf{F} = \lambda(X : \mathsf{S})\ \lambda(Y : \mathsf{S}\ \text{with}\ \mathsf{t} = X.\mathsf{t})\ \mathcal{M} \\ \mathsf{t} = \mathit{int} \\ \mathsf{M}_1 = \mathcal{M}_1[\mathsf{t}] \\ \mathsf{M}_2 = \mathcal{M}_2[\mathsf{t}] \\ \mathsf{N} = \mathsf{M}\ (\mathsf{M}_1)\ (\mathsf{M}_2) \end{array}\right\}$$

Can we eliminate the *sharing* constraint $\mathsf{t} = X.\mathsf{t}$ between the arguments (which makes the type of Y depend on the value X)?

Sharing in signatures — By parametrization

The standard way of doing abstraction for both values and types is by parametrization.
We could instead abstract the shared part of types in both arguments, and provide the exact type later when applying the functor:

$$\left\{\begin{array}{l} \mathsf{S} = \lambda(\alpha)\ \{\mathsf{t} = \alpha; m : \mathsf{t}\}; \\ \mathsf{F} = \lambda(\alpha)\ \lambda(X : \mathsf{S}\ (\alpha))\ \lambda(Y : \mathsf{S}\ (\alpha))\ \mathcal{M} \\ \mathsf{t} = \mathit{int} \\ \mathsf{M}_1 = \mathcal{M}_1[\mathsf{t}] \\ \mathsf{M}_2 = \mathcal{M}_2[\mathsf{t}] \\ \mathsf{N} = \mathsf{F}\ (\mathsf{t})\ (\mathsf{M}_1)\ (\mathsf{M}_2) \end{array}\right\}$$

(Using type abtraction and type applications that could be encoded)

This is called type *sharing by parametrization*.

Unfortunately, sharing by parametrization does not scale well to large programs.

Applicative functors

Why?

Functors are generative: if the body of a functor contains type definitions, each application of the functor to different arguments creates new incompatible type definitions. This is always the desired behavior.

For example, each application may create a new database with its own invariants: type generativity ensures that two databases will not interfere.

However, in some cases, one might wish that two applications with the *same* arguments be compatible. If this is the case we say that functors are applicative.

For example, a *MakeMap* functor may create a *Map* module when given an ordering structure as argument. Then, two applications of *MakeMap* with the very same ordering produce compatible maps that can be merged together.

Applicative functors

How?

To model applicative functors, we allow functor applications in paths. Then, two applications of the same functor path to the same argument path are equal paths.

For example, let φ be bound to the following module:

$$\begin{array}{lll} \{ & \mathsf{F} = \lambda(X : \{\})\ (\{\mathsf{t} = \mathit{int}; \mathsf{m} = 1\} : \{\mathsf{t} : 0; \mathsf{m} : \mathsf{t}\}); & \\ & \mathsf{V}_1 = \{\}; \quad \mathsf{M}_{11} = \mathsf{F}\ (\mathsf{V}_1); & \mathsf{M}_{12} = \mathsf{F}\ (\mathsf{V}_1); \\ & \mathsf{V}_2 = \{\}; \quad \mathsf{M}_{12} = \mathsf{F}\ (\mathsf{V}_2); & \\ & \mathsf{V}_3 = \mathsf{V}_2; \quad \mathsf{M}_{12} = \mathsf{F}\ (\mathsf{V}_3); & \} \end{array}$$

Then:

- $\varphi.\mathsf{M}_{11}$ and $\varphi.\mathsf{M}_{12}$ are compatible, both of type $\varphi.\mathsf{F}(\mathsf{V}_1).\mathsf{t}$,
- they are incompatible with $\varphi.\mathsf{M}_{21}$, of type $\varphi.\mathsf{F}(\mathsf{V}_2).\mathsf{t}$.
- $\varphi.\mathsf{F}(\mathsf{V}_2).\mathsf{t}$ and $\varphi.\mathsf{F}(\mathsf{V}_3).\mathsf{t}$ are also incompatible even though V_3 is just a rebinding of V_2.

See Leroy (1995)

Applicative functors — Does it matter?

Should functors be applicative or generative?

Applicative functors can type more programs. Moreover, with applicative functors, a functor application can still be made generative by rebinding the argument before the application.

Conversely, if functors are generative, two applications of the functor can never be make compatible a posteriori. Instead, the program must be reorganized. For instance, the application may be performed only once. if it has no side effect. Otherwise, the reorganization may be more complex, but it is also likely that the functor should not be applicative in this case.

In fact generativity/applicativity should rather be a property of the functor the result of a global choice and of naming of arguments.

In summary, it is not essential that module systems provide support for applicative functors, while they cannot avoid dealing with type generativity.

Type soundness — Subject reduction does not hold!

The syntax is not even stable by reduction!

$$(\lambda(m : (\varphi)\{\mathsf{t} : 0; \mathsf{m} : \varphi.\mathsf{t}\})\ \ m.\ell_x)\ \ (\{\mathsf{t} = \mathit{int}; \mathsf{m} = 1\})$$

reduces to the ill-formed projection:

$$\{\mathsf{t} = \mathit{int}; \mathsf{m} = 1\}.\mathsf{m}$$

This syntactic limitation is essential: only paths can be used in projection or functor application positions, so as to keep track of type identities. Hence, paths bound to modules are not substitutable. For example, if $(\mathcal{M} : \mathcal{S})$ is

$$(\{\mathsf{t} = \mathit{int}; \mathsf{m} = 1; \mathsf{n} = \mathit{succ}\} : (\varphi)\{\mathsf{t} : 0; \mathsf{m} : \varphi.t; \mathsf{n} : \varphi.t \to \varphi.t\})$$

then $(\lambda(X : \mathcal{S})\ \ X.\mathsf{n}\ X.\mathsf{m})\ \ (\mathcal{M} : \mathcal{S})$ would reduce to $(\mathcal{M} : \mathcal{S}).\mathsf{n}\ (\mathcal{M} : \mathcal{S}).\mathsf{m}$ in which the function and the argument have incompatible abstract types.

Type soundness — By translation to System F

Modules are second class

Modules cannot be manipulated as ordinary values. For instance, it is not possible to choose between two modules with the same abstract interface but different implementations, as in *if* a then $\mathcal{M}_1$ else $\mathcal{M}_2$.
This is in fact a real limitation of expressiveness.

Types abstraction is never necessary

This is a beneficial consequence of modules beeing second class.
In particular, revealing abstract types preserves well-typedness.
(It may of course expose type invariants, at the programmer's own risk).

Formally, ML modules can be translated into System F (with records), which ensures type soundness. Of course, as type abstraction is lost during the translation, type soundness does not ensure that values with compatible representation but incompatible semantics are never merged.

Type soundness — By translation to System F^ω

Proceed as follows (we assume no abstract signatures for simplicity)

1) Transform any sealing that turns a type definition (that does not contrain an abtract type) into an abstract type so that it reveals the type definition. All sealing can be transformed this way, in some appropriate order.
2) The remaining abstract types only appear in signatures of functor parameters. These can be made concrete by adding explicit parametrization, transforming $\lambda(X : \{\mathrm{t}; \overline{s}\})\ \mathcal{M}$ into $\lambda(\alpha)\ \lambda(X : \{\mathrm{t} = \alpha; \overline{s}\})\ \mathcal{M}$ (and transforming functors applications accordingly).
3) Type definitions may be inlined and become unused. Once all type definitions are unused, they can be removed altogether. This turns modules into records and functors into functions,
4) Replace uses of subtyping (at sealing and functor applications) by explicit coercions, much as the compiler does, returning a program in System F^ω.

The result is actually in System F if all type components are nullary (and no abstract signature has been used).

Type soundness — Elaboration into a richer language

An indirect solution to type soundness

Use a calculus of dependent types to model modules, enriched with singleton kinds to abstract type equalities, where subject reduction holds and elaborate ML modules into this core language

Problems

These approaches are technically involved.

The semantics is given by elaboration, a global translation that does not provide a good intuition of what modules are.

For instance, see Dreyer et al. (2003).

Recursive modules

Example

Two recursive modules

```
module rec A : sig
  type t = Leaf of int | Node of ASet.t
  val compare : t → t → int
end = struct
  type t = Leaf of int | Node of ASet.t
  let compare t1 t2 =
    match t1, t2 with
    | Leaf i1, Leaf i2 → i2 − i1
    | Node n1, Node n2 → ASet.compare n1 n2
    | Leaf _, Node _ → 1 | Node _, Leaf _ → −1
end

and ASet : Set.S with type elt = A.t = Set.Make(A)
```

Recursive modules

Example

Their recursive signatures

```
module rec A : sig
    type t = Leaf of int | Node of ASet.t
    val compare : t → t → int
end

and ASet : sig
    type elt = A.t
    type t
    val empty : t
    val elem : elt → t → bool
    ...
end
```

Recursive modules — Difficulties

Typechecking

- Signatures are recursive and depend on a module that has not yet been typechecked.
- Modules are recursive, but may be generative, *i.e.* is the recursive occurrence of the module type the same as the module type itself?
- Double vision problem: a type definition of $\mathcal{M}_1$ hidden in the signature of $\mathcal{M}_1$ should be seen as abstract from a recursively defined module $\mathcal{M}_2$, but as concrete within $\mathcal{M}_1$.

Compilation

- When are recursive definitions well-formed?
- What is their semantics?
- How can they be compiled? efficiently?

Recursive modules — Typechecking

Syntax

$$\mathcal{M} \quad ::= \quad \ldots \mid \mu(\psi)\{\overline{\mathsf{M} : \mathcal{S} = \mathcal{M}}\} \qquad\qquad \mathcal{S} \quad ::= \quad \ldots \mid \mu(\psi)\{\overline{\mathsf{M} : \mathcal{S}}\}$$

By comparison with modules, the construction forces all recursive fields to be submodules.
Variable ψ may now appear in any field before they have been defined.

Typechecking signatures

$$\frac{\psi \notin \mathit{dom}\,\Gamma \qquad \Gamma, \overline{\psi.\mathsf{M} : \mathcal{S}} \vdash \mathcal{S}_i \qquad \mathit{acyclic}(\mu(\psi)\{\overline{\mathsf{M} : \mathcal{S}}\})}{\Gamma \vdash \mu(\psi)\{\overline{\mathsf{M} : \mathcal{S}}\}}$$

Some extended occur check $\mathit{acyclic}(\mu(\psi)\{\overline{\mathsf{M} : \mathcal{S}}\})$ is used to avoid ill-formed recursive type definitions such as $\mu(\psi)\{\mathsf{t} = \psi.\mathsf{t}\}$ where the definition of t is not contractive.

Recursive modules — Typechecking

The signature of a recursive module is provided and thus never inferred.

Naive typing rule

$$\frac{\Gamma \vdash \mu(\psi)\{\overline{\mathsf{M} : \mathcal{S}}\} \qquad \psi \notin \mathit{dom}\,\Gamma \qquad \Gamma, \overline{\psi.\mathsf{M} : \mathcal{S}} \vdash \mathcal{M}_i : \mathcal{S}_i}{\Gamma \vdash \mu(\psi)\{\overline{\mathsf{M} : \mathcal{S} = \mathcal{M}}\}}$$

Limitation

This rule is too weak, since it does not take into account that $\psi.\mathsf{M}_i$ and $\mathcal{M}_i$ are eventually the very same module.

This is known as the double vision problem.

Double vision problem — An example

Example

Let $\mathcal{S}_1$ be $(\varphi)\{\mathsf{t} : 0; \mathsf{m} : \varphi.\mathsf{t}\}$ and let $\mathcal{M}_1$ be $(\{\mathsf{t} = \mathit{int}; \mathsf{m} = 1\} : \mathcal{S}_1)$.

Let $\mathcal{S}_\psi$ be $\{\mathsf{t} : 0; \mathsf{m} : \psi.\mathsf{M}.\mathsf{t}\}$.

Then, $\mu(\psi)\{\mathsf{M} : \mathcal{S}_\psi = \mathcal{M}_1\}$ is ill-typed because under $\psi.\mathsf{M} : \mathcal{S}_\psi$, the signature $\mathcal{S}_1$ of $\mathcal{M}_1$ is not a subtype of $\mathcal{S}_\psi$.

Solution

The strengthened signature, $\mathcal{S}_1/\psi.\mathsf{M}$, which is equal to $(\varphi)\{\mathsf{t} = \psi.\mathsf{M}.\mathsf{t}; \mathsf{m} : \varphi.\mathsf{t}\}$ is a subtype of $\mathcal{S}_\psi$.

Indeed, in the context $\varphi.\mathsf{t} = \psi.M.\mathsf{t}; \varphi.m : \varphi.\mathsf{t}$ (used for checking subtyping of signature declarations), we have $\varphi.\tau \approx \psi.M.\mathsf{t}$.

Double vision problem — Strengthening

Strengthening for recursive modules

A solution is to use strengthening in combination with subtyping:

$$\Gamma \vdash \mu(\psi)\{\overline{\mathsf{M} : \mathcal{S}}\} \qquad \psi \notin \mathit{dom}\,\Gamma$$

$$\frac{\Gamma, \overline{\psi.\mathsf{M} : \mathcal{S}} \vdash \mathcal{M}_i : \mathcal{S}'_i \qquad \Gamma \vdash \mathcal{S}'_i/\varphi.\mathsf{M}_i <: \mathcal{S}_i}{\Gamma \vdash \mu(\psi)\{\overline{\mathsf{M} : \mathcal{S} = \mathcal{M}}\}}$$

This solves the previous problem.

However, this form of strengthening breaks the previous nice property that strengthening can always be canceled by subtyping.

In this case, we have $\Gamma' \vdash \mathcal{M}_i : \mathcal{S}'_i$ (where Γ' is $\Gamma, \overline{\psi.\mathsf{M} : \mathcal{S}}$), but there is no reason for $\Gamma' \vdash \mathcal{S}'_i/\varphi.\mathsf{M}_i <: \mathcal{S}'_i$.

This creates a problem for type inference, since applying strengthening immediately may enable new derivations but also prevent other valid ones.

Double vision problem

Example

The following definition is rejected

$$\mu(\psi)\left\{\begin{array}{l} \mathsf{M} : \{\mathsf{N} : \mathcal{S}_1\} = (\varphi)\{\mathsf{N} = \mathcal{M}_1; \mathsf{m} = \psi.\mathsf{F.f}\ (\varphi.\mathsf{N.m})\} \\ \mathsf{F} \ : \{\mathsf{f} : \psi.\mathsf{M.N.t} \to \mathit{int}\} = \{\mathsf{f} = \lambda(x)\ 0\} \end{array}\right\}$$

The problem is the following:

- Field ψ.F is typed with the abstract view $\{\mathsf{N} : \mathcal{S}_1\}$ of ψ.M.
- Hence, the domain of ψ.F.f has type the external type ψ.M.N.t in ψ.M while the argument φ.N.*m* has the internal type φ.N.t.
- Hence, the application fails.

Solution type the body ψ.M with the knowledge that the external and internal types are equal, *i.e.* with the additional equality $\varphi.\mathsf{N.t} = \psi.\mathsf{M.N.t}$.

Double vision problem — Solution

(Informal)

When $\mathcal{M}_i$ is a structure $\{\bar{d}\}$ and Γ' is $\Gamma, \overline{\psi.\mathrm{M} : \mathcal{S}}$, then typecheck $\Gamma' \vdash \mathcal{M}_i : \mathcal{S}'_i$ as $\Gamma' \vdash_{\varphi=\psi.\mathcal{M}_i} \bar{d} : \bar{s}$. where $\vdash_{\psi.\mathcal{M}_i=\varphi}$ replaces $\vdash_\varphi$ in typing declarations, and is used to push $\varphi.\mathrm{t} = \psi.\mathcal{M}_i.\mathrm{t} = \tau$ in the context instead of $\varphi.\mathrm{t} = \tau$.

Comment

This is somewhat ad hoc and at the limits of the path-based approach in its current formulation.

We should instead really treat paths φ and $\psi.\mathrm{M}$ as equal and propagate such equalities on paths to equalities on types, instead of adding only *some* equalities on *only* types to (a sort of) *canonical forms* for type definitions.

Recursive definitions — Problem

This is not a problem specific to the module language.

Examples or well-formed recursive definitions

$$\texttt{let } s = \texttt{let rec } z = 0 :: z \texttt{ in } 3 :: 2 :: 1 :: z$$

is the infinite stream starting with 3, 2, 1, followed by infinitely many 0.

$$\texttt{type } r = \{\ l : \mathit{int};\ r :\ r\ \};;$$
$$\texttt{let rec } \mathit{ok} = \{\ l = 1;\ r = \{\ l = 2;\ r = \mathit{ok}\}\}$$

Recursive values are only used to build other values during recursion, but never accessed.

Ill-formed ones

$$\texttt{let rec } \mathit{ko} = \{\ l = 1;\ r = \{\ l = 1 + \mathit{ko}\,.\,l;\ r = \mathit{ko}\}\}$$

The recursive value is accessed before being defined.

Recursive definitions — Solutions?

Rejected useful forms of recursion

```
let rec decay x r = if x = 0 then x :: r else x :: decay (x−1) r
let s = let rec z = decay 0 z in decay 3 (decay 2 (decay 1 z));;
```

Although this is safe, this example is rejected because it is difficult to detect that *decay* does not access its second argument.

(Replacing *decay* by (::) gives back the previous example.)

Can we allow more well-formed recursive definitions?

- Use more sophisticated type systems.

  See Dreyer (2004); Hirschowitz and Leroy (2005)

- Use runtime detection of ill-formed recursion.

  See Hirschowitz et al. (2003)

Recursion — Compilation

A matter of compromise between

- Extra indirections and/or tests at module access.
- Easier compilation and dynamic detection of ill-formed recursions. Larger class of recursive definitions accepted.

The backpatching semantics and compilation schema

- A partial record-value Ω is allocated with initially all fields undefined.
- Accesses to undefined fields of Ω must be detected and raise an error.
- The recursive definition is evaluated, using Ω for recursive references.
- Fields are evaluated in order of definition.
- When a field ℓ is evaluated to a value v, $\Omega.\ell$ is backpatched with v, and $\Omega.\ell$ can now be accessed without an error.

Mixin modules

Limitations of modules

Modules

- Split programs into components,
- But components must be well identified, and created as a whole.
- Functors allow to program the assembling of components, but partial components are not permitted, or must explicitly be represented as functors.
- Recursive modules allow smaller grain components that recursively depend on one another, but components must still be created atomically.

Mixin modules

More flexibility

Mixins

- Mixins are *partial components* that may be incomplete. That is, they may define and export values that depend on missing imports.
- Exports of mixins can be recursively defined.
- Incomplete mixins cannot be used.
- Mixins can be extended by adding new definitions, which may fill in missing imports or just provide additional exports.
- Complete mixins can be used as modules.

Mixin modules

More difficulties

Typechecking and compilation of mixins raise problems that are similar to but even harder than recursive modules, because recursive definitions are assembled incrementally as opposed to built atomically.

Below, we only give a brief flavor of what mixin modules could look like.

For design and typechecking issues, see Dreyer and Rossberg (2008).
For compilation issues, see Hirschowitz and Leroy (2005).

Mixin modules

Example

Consider $\mathcal{M}$ defined as:

$$\{(\psi)\langle|$$
$$\quad \mathit{Even} = (\varphi)\langle \mathit{odd} : \mathit{int} \to \mathit{bool} \mid \mathit{even} = \lambda(x)\ (x = 0)\ \mathit{or}\ \varphi.\mathit{odd}(x-1)\rangle$$
$$\quad \mathit{Odd} = (\varphi)\langle \mathit{even} : \mathit{int} \to \mathit{bool} \mid \mathit{odd} = \lambda(x)\ (x > 0)\ \mathit{and}\ \varphi.\mathit{even}(x-1)\rangle$$
$$\quad \mathit{Nat} = \{\psi.\mathit{Even} \otimes \psi.\mathit{Odd}\}$$
$$|\rangle\}$$

- $\psi..\mathit{Even}$ is a mixin with en import *odd* and an export *even*
- $\psi.\mathit{Odd}$ is a mixin with en import *even* and an export *odd*
- $\mathit{Even} \otimes \mathit{Odd}$ is the mixin composition of $\psi..\mathit{Even}$ and $\psi..\mathit{Odd}$
- $\{\mathit{Even} \otimes \mathit{Odd}\}$ is the module obtained by closing the mixin.
- $\mathcal{M}$ is itself a module obtained by closing the mixin with no import and *Even*, *Odd*, and *Nat* as exports.

Mixin modules

Basic ideas

Mixin modules $(\varphi)\langle \mathcal{I} \mid \mathcal{D} \rangle$

They are incomplete modules where $\mathcal{D}$ is a sequence of declarations $\bar{d}$ that may refer to yet undefined, but declared imports $\mathcal{I}$. Import $\mathcal{I}$, *i.e.* a sequence $\bar{s}$ where each s_i is an abstract type, a value or a submodule declaration.

Mixin signatures $(\varphi)\langle \mathcal{I} \mid \mathcal{E} \rangle$

They are as module specifications, except that they separate import from export specifications. An import specification $\mathcal{E}$ is a sequence $\bar{s}$ where each s_i is a type definition, a value, or a submodule declaration.

As for modules and signatures, fields are referred to one another via the bound variable φ. However, as with recursive modules, fields may recursively depend on one another.

If subtyping is enabled, mixin signatures are covariant in exports and contravariant in imports.

Mixin modules

Main operations

Mixins composition $\mathcal{M}_1 \otimes \mathcal{M}_2$

This has signature $(\varphi)\langle \mathcal{I} \mid \mathcal{E} \rangle$ whenever

- $\mathcal{I}$ and $\mathcal{E}$ are $(\mathcal{I}_1 \cup \mathcal{I}_2) \setminus \mathcal{E}$ and $\mathcal{E}_1 \cup \mathcal{E}_2$
  (where $(\varphi)\langle \mathcal{I}_i \mid \mathcal{E}_i \rangle$ is the signature of $\mathcal{M}_i$).
- Fields in $(\mathcal{I}_1 \cup \mathcal{E}_1) \cap (\mathcal{I}_1 \cup \mathcal{E}_2)$ must be compatible.
  (If subtyping is enabled, it may have been used to strengthen imports on both sides prior to composition).
- Only type definitions can appear in the intersection of exports.

Closing $\{\mathcal{M}\}$

Assuming that $\mathcal{M}$ has no import, *i.e.* a signature $(\varphi)\langle \mid \mathcal{E} \rangle$, this returns a module, of signature $(\varphi)\{\mathcal{E}\}$

Mixin modules — Other operations

Binding and access

Only components of closed mixins can be accessed.

Deletion

$\mathcal{M} \setminus \ell$ removes the definition ℓ from $\mathcal{M}$ and instead makes field ℓ an import of the corresponding type.
This is only possible if field ℓ has not been subtyped (either subtyping is disable, or the type system keeps track of where subtyping has been used.)
This allows for overriding, as in object-oriented languages.

Renaming

$\mathcal{M}[\ell_1 \leftarrow \ell_2]$ replaces the label ℓ_1 by a (new) label ℓ_2 in $\mathcal{M}$.
This might be useful to avoid conflicting names before composition.

Mixin modules

Functors are encodable

Functors can be encoded as mixins

$$\lambda(X : \mathcal{S})\ \mathcal{M} \stackrel{\triangle}{=} (\varphi)\langle \mathsf{M} : \mathcal{S} \mid \mathsf{F} = \mathcal{M}[\varphi.\mathsf{M}/X]\rangle$$

Then functor application is replaced by composition followed by closing and projection.

$$\mathcal{M}_1\ (\mathcal{M}_2) \stackrel{\triangle}{=} \{\mathcal{M}_1 \otimes \langle \mid \mathsf{M} = \mathcal{M}_2\rangle\}.\mathsf{F}$$

(Auxiliary bindings can be used to avoid the projection on non variables).

Mixin modules — Typechecking difficulties

Type generativity

As with modules, we need to keep track of type identities. This is the reason for closing, which besides verifying the absence of imports generate fresh type components (as a functor application would do).

Components of a mixin cannot be accessed before it is closed.

Recursion and well-foundedness

Recursion is the default. Mixins are open recursive definitions, which may be ill-founded.

Worse, the composition of independently well-founded recursive definitions may become ill-founded.

Mixin modules — Hierarchical composition

Example

Is the following composition

$$\langle |\ \mathsf{M} = (\varphi)\langle \mathsf{n} : \mathit{int} \mid \mathsf{m} = 1\rangle\rangle \otimes \langle |\ \mathsf{M} = (\varphi)\langle \mathsf{m} : \mathit{int} \mid \mathsf{n} = 2\rangle\rangle$$

well-defined and equal to $\langle |\ \mathsf{M} = (\varphi)\langle |\ \mathsf{m} = 1; \mathsf{n} = 2\rangle\rangle$?

Interest

This operation allows to organize the name space more freely.
In particular, definitions may all be *shifted* under some prefix to avoid conflicting with other definitions.

Mixin modules — Summary

Powerful

- Many new possibilities: mixin composition, renaming, overriding...
- Many resemblances with objects and classes (plus type components)
- Many possible variants in the design.
  (More precise, but more complex types allow for more operations)

Difficult

- Type generativity
- Recursion at the type level and
- Recursion at the value level
- Incrementality of recursive definitions makes it much harder

Need for strong theoretical basis

Open Existential Types

Why?

Path-based syntactic approaches

Reveal a contradiction (and a persistent tension) between apparent simplicity and actual complexity, on both theoretical and practical levels:

- At first glance, they are intuitively simple, but this is only a lure...
  - Technically they are cumbersome, with ad hoc, unintuitive corners.
  - Practically, they may also become harder to use and heavy weight.
- Theoretically, type soundness and subject reduction require involved technical machinery and does not yet explain recursive modules.

Sources of problems

- Type abstraction and sharing is an obvious source of difficulties.
- Technically, putting type components inside structures depart from the usual approach for core languages, where expressions have (and may depend on) types but types do not depend on expressions.

Open Existential Types

How?

Most ingredients for modules are already in System $F_{:>}$

- Records and functors can model modules and functors.
- Subtyping at the level of types.
- Existential types for type abstraction
- What is missing is a *modular* treatment of type abstraction.

Can type abstraction be made modular?

- Avoid type components, hence path-dependent types.
- Use a first-class rather than a stratified approach.

Approach

- Start with system F with existential types.
- Break existential types into more atomic constructs.

System F

Core language

Core system F

$$\begin{array}{lcl} \mathcal{M} & ::= & x \mid \lambda(x:\tau)\,\mathcal{M} \mid \mathcal{M}\,(\mathcal{M}) \mid \lambda(\alpha)\,\mathcal{M} \mid \mathcal{M}\,(\tau) \\ \tau & ::= & \alpha \mid \tau \to \tau \mid \forall\alpha.\tau \end{array}$$

Typing rules

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau} \qquad \frac{\Gamma \vdash \mathcal{M}:\tau \qquad \alpha \notin \Gamma}{\Gamma \vdash \lambda(\alpha)\,\mathcal{M}:\forall\alpha.\tau} \qquad \frac{\Gamma \vdash \mathcal{M}:\forall\alpha.\tau_0 \qquad \Gamma \vdash \tau}{\Gamma \vdash \mathcal{M}\,(\tau):\tau_0[\tau/\alpha]}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma, x:\tau_0 \vdash \mathcal{M}:\tau}{\Gamma \vdash \lambda(x:\tau_0)\,\mathcal{M}:\tau_0 \to \tau} \qquad \frac{\Gamma \vdash \mathcal{M}_1:\tau_2 \to \tau_1 \qquad \Gamma \vdash \mathcal{M}_2:\tau_2}{\Gamma \vdash \mathcal{M}_1(\mathcal{M}_2):\tau_1}$$

Plus existential types

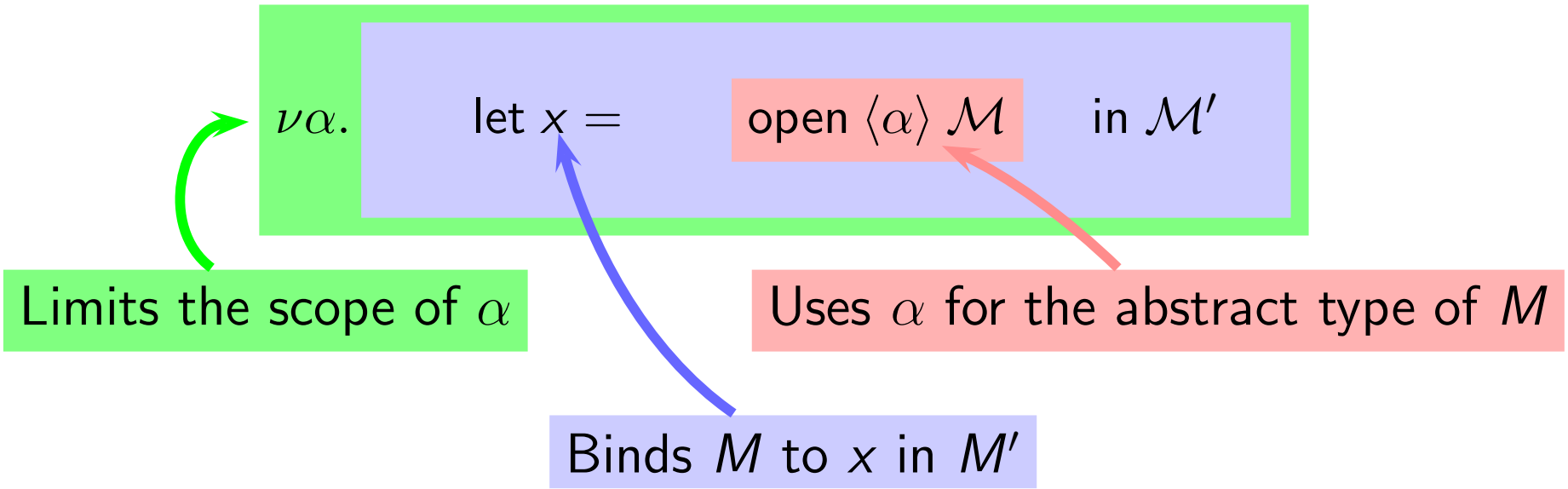
Plus records

Splitting unpack

Into three pieces

$$\text{unpack } \mathcal{M} \text{ as } \alpha, x \text{ in } \mathcal{M}'$$

$$\triangleq$$



Splitting unpack

Into three pieces

$$\nu\alpha.\ \text{let } x = \text{open } \langle\alpha\rangle\ \mathcal{M} \text{ in } \mathcal{M}'$$

Splitting unpack

Gain in expressiveness

$$\nu\alpha.\ \text{let } x = D\{\ \text{open } \langle\alpha\rangle\ \mathcal{M}\ \}\ \text{in } \mathcal{M}'$$

$\mathcal{M}$ need not be at toplevel.

Splitting unpack

Gain in expressiveness

$$\nu\alpha.\ C\left\{\ \text{let } x = \boxed{\text{open } \langle\alpha\rangle\ \mathcal{M}}\ \text{ in } \mathcal{M}'\ \right\}$$

α need not be hidden immediately.

Splitting unpack

Gain in expressiveness

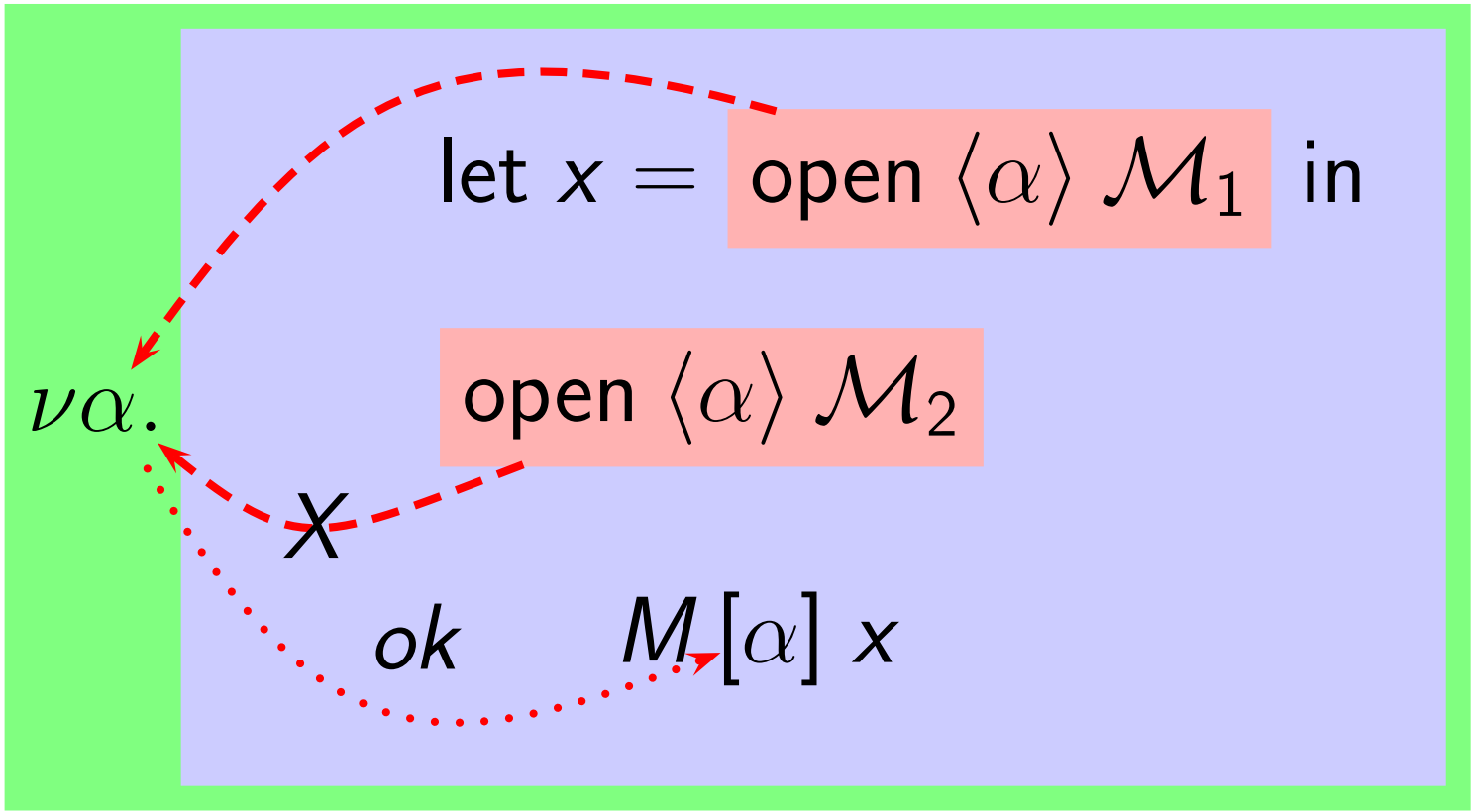
$$C\left\{\text{let } x = \boxed{\text{open } \langle\alpha\rangle\ \mathcal{M}} \text{ in } \mathcal{M}'\right\}$$

α need not be hidden at all in program *components*

Splitting unpack

Typechecking

Must forbid incorrect programs such as

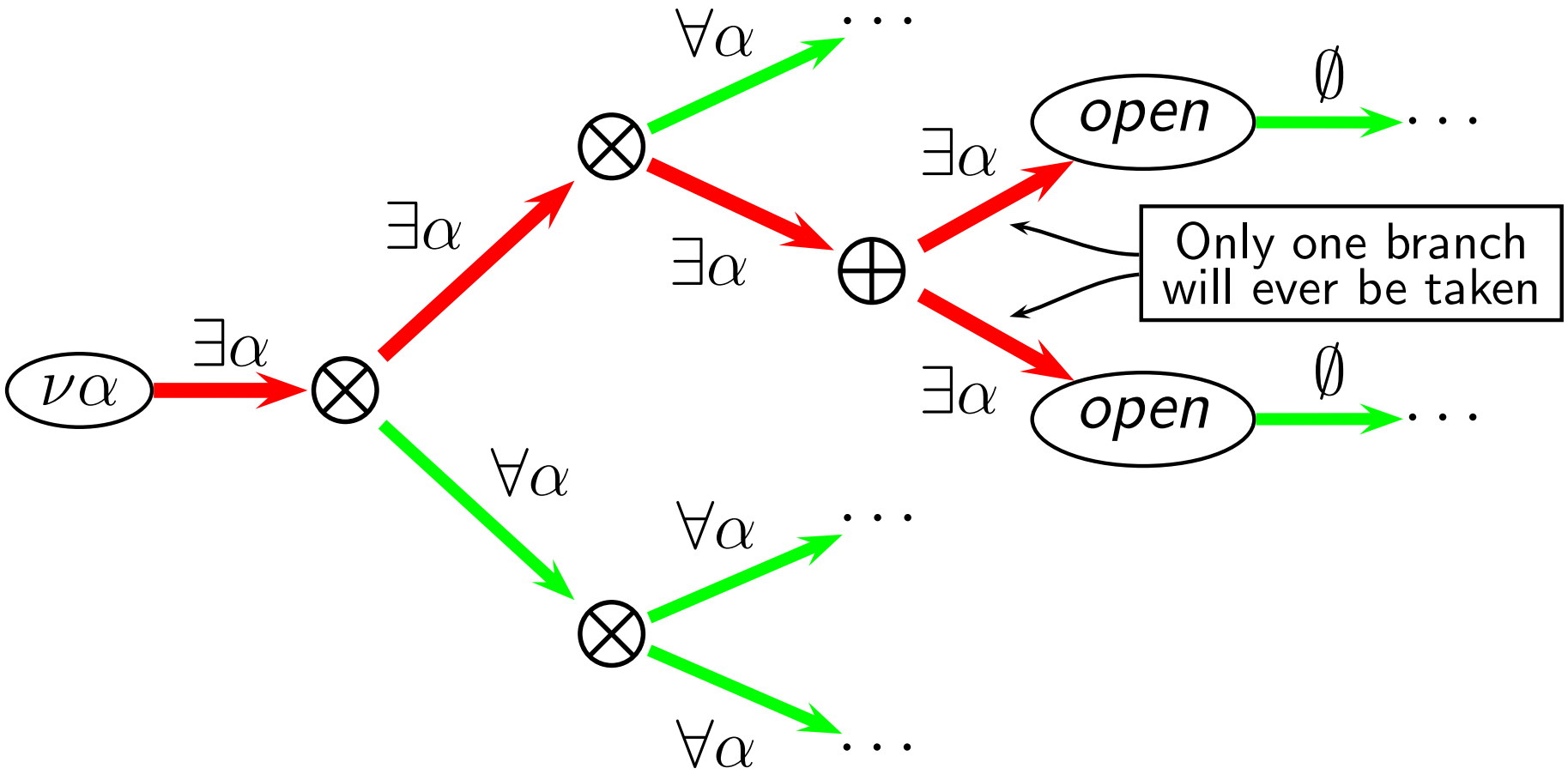


- There must be at most one opening with the same variable α.
- There may be any uses of α (after or before the opening).

Splitting unpack

Typechecking

Evaluation contexts:



Splitting unpack

$$\textsc{Nu}\ \frac{\Gamma, \exists\alpha \vdash \mathcal{M} : \tau \qquad \alpha \notin \mathit{ftv}(\tau)}{\Gamma \vdash \nu\alpha.\,\mathcal{M} : \tau}$$

The typing environment keeps track of open existentials and enforces they linear use

Splitting unpack

OPEN

$$\frac{\Gamma \vdash \mathcal{M} \ : \ \exists\alpha.\,\tau}{\Gamma, \exists\alpha \vdash \mathsf{open}\ \langle\alpha\rangle\ \mathcal{M} \ : \ \tau}$$

NU

$$\frac{\Gamma, \exists\alpha \vdash \mathcal{M} \ : \ \tau \qquad \alpha \notin \mathit{ftv}(\tau)}{\Gamma \vdash \nu\alpha.\,\mathcal{M} \ : \ \tau}$$

The typing environment keeps track of open existentials and enforces they linear use

Splitting unpack

Typechecking

$$\textsc{Open}\quad \frac{\Gamma \vdash \mathcal{M} \,:\, \exists\alpha.\,\tau}{\Gamma, \exists\alpha \vdash \mathsf{open}\ \langle\alpha\rangle\ \mathcal{M} \,:\, \tau}$$

$$\textsc{Let}\quad \frac{\Gamma_1 \vdash \mathcal{M}_1 \,:\, \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash \mathcal{M}_2 \,:\, \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash \mathsf{let}\ x = \mathcal{M}_1\ \mathsf{in}\ \mathcal{M}_2 \,:\, \tau_2}$$

$$\textsc{Nu}\quad \frac{\Gamma, \exists\alpha \vdash \mathcal{M} \,:\, \tau \qquad \alpha \notin \mathit{ftv}(\tau)}{\Gamma \vdash \nu\alpha.\,\mathcal{M} \,:\, \tau}$$

The typing environment keeps track of open existentials and enforces they linear use with zipping.

Splitting unpack

Typechecking

$$\textsc{Open}\ \frac{\Gamma \vdash \mathcal{M} \ : \ \exists\alpha.\,\tau}{\Gamma, \exists\alpha \vdash \mathsf{open}\ \langle\alpha\rangle\ \mathcal{M} \ : \ \tau} \qquad \frac{\vdots}{\Gamma, \forall\alpha \vdash \mathcal{M}'\,[\alpha] \ : \ \tau'}$$

$$\textsc{Let}\ \frac{\Gamma_1 \vdash \mathcal{M}_1 \ : \ \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash \mathcal{M}_2 \ : \ \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash \mathsf{let}\ x = \mathcal{M}_1\ \mathsf{in}\ \mathcal{M}_2 \ : \ \tau_2}$$

$$\textsc{Nu}\ \frac{\Gamma, \exists\alpha \vdash \mathcal{M} \ : \ \tau \qquad \alpha \notin \mathit{ftv}(\tau)}{\Gamma \vdash \nu\alpha.\,\mathcal{M} \ : \ \tau}$$

The typing environment keeps track of open existentials and enforces they linear use with zipping.

Splitting unpack

Zipping of typing assumptions

$$b ::= \exists\alpha \mid \forall\alpha \mid x : \tau \mid \forall(\alpha = \tau)$$

Zipping of two bindings ensures that every existential type appears in exactly one of the two.

$$\begin{array}{rcl}
\forall\alpha \curlyvee \exists\alpha & = & \exists\alpha \\
\exists\alpha \curlyvee \forall\alpha & = & \exists\alpha \\
\forall\alpha \curlyvee \forall\alpha & = & \forall\alpha \\
x : \tau \curlyvee x : \tau & = & x : \tau \\
\forall(\alpha = \tau) \curlyvee \forall(\alpha = \tau) & = & \forall(\alpha = \tau)
\end{array}$$

Zipping extends to typing contexts in the obvious way:

$$\emptyset \curlyvee \emptyset = \emptyset \qquad (\Gamma_1, b_1) \curlyvee (\Gamma_2, b_2) = (\Gamma_1 \curlyvee \Gamma_2), (b_1 \curlyvee b_2)$$

Splitting pack

In two pieces

$$\text{pack } \tau, \mathcal{M} \text{ as } \exists\alpha.\,\tau' \triangleq \exists(\alpha = \tau)\ (\mathcal{M} : \tau')$$

makes α abstract with witness τ

converts the type of $\mathcal{M}$ using the equation(s)

Splitting pack

In three pieces

$$\text{pack } \tau, \mathcal{M} \text{ as } \exists\alpha.\,\tau' \triangleq$$

$$\exists\beta.\ \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\ (\mathcal{M} : \tau')$$

closes the abstract type β

converts the type of $\mathcal{M}$

defines the open abstract type β
with internal name α and witness τ

Splitting pack

Gain in expressiveness

$$\exists\beta.\ C\Big\{\ \Sigma\,\langle\beta\rangle\,(\alpha=\tau)\ D\{\ (\mathcal{M}:\tau')\ \}\ \Big\}$$

The opening may be deeper under C, which sees β abstractly.

The annotation may be deeper (hence shorter) at the leaves of D.

Splitting pack

Gain in expressiveness

$$\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\;\; D\{\;(\mathcal{M} : \tau')\;\}$$

A module with an open abstract type β.

Splitting pack

Gain in expressiveness

$$C\left\{ \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\; D\{\; (\mathcal{M} : \tau')\; \}\right\}$$

A submodule with an open abstract type β.

Splitting pack

Typechecking

Exists

$$\frac{\Gamma, \exists\beta \vdash \mathcal{M} : \tau}{\Gamma \vdash \exists\beta.\, \mathcal{M} : \exists\beta.\, \tau}$$

Splitting pack

Typechecking

$$\textsc{Exists}\quad \frac{\Gamma, \exists\beta \vdash \mathcal{M} : \tau}{\Gamma \vdash \exists\beta.\,\mathcal{M} : \exists\beta.\,\tau}$$

$$\textsc{Open}\quad \frac{\Gamma \vdash \mathcal{M} : \exists\beta.\,\tau}{\Gamma, \exists\beta \vdash \mathsf{open}\ \langle\beta\rangle\ \mathcal{M} : \tau}$$

Splitting pack

Typechecking

SIGMA

$$\frac{\Gamma, \forall\beta, \Gamma', \forall(\alpha = \tau) \vdash \mathcal{M} : \tau'}{\Gamma, \exists\beta, \Gamma' \vdash \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,\mathcal{M} : \tau'[\alpha \leftarrow \beta]}$$

EXISTS

$$\frac{\Gamma, \exists\beta \vdash \mathcal{M} : \tau}{\Gamma \vdash \exists\beta.\,\mathcal{M} : \exists\beta.\,\tau}$$

OPEN

$$\frac{\Gamma \vdash \mathcal{M} : \exists\beta.\,\tau}{\Gamma, \exists\beta \vdash \mathsf{open}\,\langle\beta\rangle\,\mathcal{M} : \tau}$$

Splitting pack

Typechecking

Coerce

$$\frac{\Gamma \vdash \mathcal{M} : \tau' \qquad \Gamma \vdash \tau' \equiv \tau}{\Gamma \vdash (\mathcal{M} : \tau) : \tau}$$

uses

Sigma

$$\frac{\Gamma, \forall\beta, \Gamma', \forall(\alpha = \tau) \vdash \mathcal{M} : \tau'}{\Gamma, \exists\beta, \Gamma' \vdash \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,\mathcal{M} : \tau'[\alpha \leftarrow \beta]}$$

Exists

$$\frac{\Gamma, \exists\beta \vdash \mathcal{M} : \tau}{\Gamma \vdash \exists\beta.\,\mathcal{M} : \exists\beta.\,\tau}$$

Summary

Types are unchanged (as in System F with existentials)

$$\tau \;::=\; \alpha \;\mid\; \tau \to \tau \;\mid\; \forall\alpha.\,\tau \;\mid\; \exists\alpha.\,\tau$$

Exressions are

$$\begin{array}{rcl} \mathcal{M} & ::= & \ldots \\ & \mid & \exists\alpha.\,\mathcal{M} \;\mid\; \Sigma\,\langle\beta\rangle\,(\alpha=\tau)\,\mathcal{M} \;\mid\; (\mathcal{M}:\tau) \\ & \mid & \nu\alpha.\,\mathcal{M} \;\mid\; \mathsf{open}\,\langle\alpha\rangle\,\mathcal{M} \end{array}$$

Examples

Abstract type

In ML:

$$\left(\left\{\begin{array}{l} \mathsf{t} = \mathit{int} \\ \mathsf{z} = 0 \\ \mathsf{s} = \lambda(x : \mathit{int})x{+}1 \end{array}\right\} : \left\{\begin{array}{l} \mathsf{t} : 0 \\ \mathsf{z} : t \\ \mathsf{s} : t \to t \end{array}\right\}\right)$$

In Fzip:

$$\Sigma\,\langle\beta\rangle\,(\alpha = \mathit{int})\left(\left\{\begin{array}{l} z = 0\,; \\ s = \lambda(x : \mathit{int})x{+}1 \end{array}\right\} : \left\{\begin{array}{l} z : \alpha\,; \\ s : \alpha \to \alpha \end{array}\right\}\right)$$

Examples

Abstract type

In ML:

$$\left(\left\{\begin{array}{l} \mathsf{t} = \mathit{int} \\ \mathsf{z} = 0 \\ \mathsf{s} = \lambda(x : \mathit{int})x{+}1 \end{array}\right\} : \left\{\begin{array}{l} \mathsf{t} : 0 \\ \mathsf{z} : t \\ \mathsf{s} : t \to t \end{array}\right\}\right)$$

In Fzip:

$$\text{let } x = \exists(\alpha = \mathit{int}) \left(\left\{\begin{array}{l} z = 0\,; \\ s = \lambda(x : \mathit{int})x{+}1 \end{array}\right\} : \left\{\begin{array}{l} z : \alpha\,; \\ s : \alpha \to \alpha \end{array}\right\}\right) \text{ in}$$

$$\text{open } \langle\beta\rangle\ x$$

Examples

Type generativity

In ML:

$$\mathsf{M}_1 = \left(\left\{ \begin{array}{l} \mathsf{t} = \mathit{int} \\ \mathsf{z} = 0 \\ \mathsf{s} = \lambda(x : \mathit{int})x{+}1 \end{array} \right\} : \left\{ \begin{array}{l} \mathsf{t} : 0 \\ \mathsf{z} : t \\ \mathsf{s} : t \to t \end{array} \right\} \right)$$

$$\mathsf{M}_2 = \left(\mathsf{M} \quad : \left\{ \begin{array}{l} \mathsf{t} : 0 \\ \mathsf{z} : t \\ \mathsf{s} : t \to t \end{array} \right\} \right)$$

In Fzip:

$$\text{let } x = \exists(\alpha = \mathit{int}) \left(\left\{ \begin{array}{l} z = 0\,; \\ s = \lambda(x : \mathit{int})x{+}1 \end{array} \right\} : \left\{ \begin{array}{l} z : \alpha\,; \\ s : \alpha \to \alpha \end{array} \right\} \right) \text{ in}$$

$$\text{let } x_1 = \text{open } \langle \beta_1 \rangle\ x \text{ in}$$

$$\text{let } x_2 = \text{open } \langle \beta_2 \rangle\ x \text{ in}$$

$$\ldots$$

Examples

Functors

- Functions must be pure (*i.e.* not create open abstract types)
- Thus, body of functors are *closed* abstract types
- that are opened after each application of the functor.

Example

let $MakeSet =$
$\quad \Lambda\alpha.\, \lambda(cmp : \alpha \to \alpha \to bool)\ \exists(\beta = set(\alpha))\,(\ldots : set(\beta))$ in
let $s_1 =$ open $\langle\beta_1\rangle$ $MakeSet\,[int]\,(<)$ in
let $s_2 =$ open $\langle\beta_2\rangle$ $MakeSet\,[\beta_1]\,(s_1.cmp)$ in
$\ldots$

Reduction

Problem (well-known)

- Expressions that create open abstract types can't be substituted.
- This would duplicate—hence break—the use of linear resources.
- The reduct would thus be ill-typed.

Solution

- Extrude Σ's whenever needed (when reduction would blocked).
- This safely enlarges the scope of identities,
- moving the Σ's outside of redexes, and
- Allowing further reduction to proceed.

Reduction

Example

$$\text{let } x = \Sigma\,\langle\beta\rangle\,(\alpha = \mathit{int})\ (1:\alpha)\ \text{ in } \{\ell_1 = x\,;\ \ell_2 = (\lambda(y:\beta)y)\ x\}$$

$\downarrow$ (extrude)

$$\Sigma\,\langle\beta\rangle\,(\alpha = \mathit{int})\ \text{let } x = (1:\alpha)\ \text{ in } \{\ell_1 = x\,;\ \ell_2 = (\lambda(y:\beta)y)\ x\}$$

$\downarrow$ (reduce)

$$\Sigma\,\langle\beta\rangle\,(\alpha = \mathit{int})\ \{\ell_1 = (1:\alpha)\ ;\ \ell_2 = (\lambda(y:\beta)y)\ (1:\alpha)\}$$

$\downarrow$ (reduce)

$$\Sigma\,\langle\beta\rangle\,(\alpha = \mathit{int})\ \{\ell_1 = (1:\alpha)\ ;\ \ell_2 = (1:\alpha)\}$$

Reduction — Results and Values

Informally

- Results are non erroneous expressions that cannot be reduced.
- Some results cannot be duplicated and are not values.
- Values are results that can be duplicated.

Formally

Values

$$v ::= u \mid (u : \tau)$$
$$u ::= x \mid \lambda(x : \tau)\mathcal{M} \mid \Lambda\alpha.\,\mathcal{M} \mid \exists\beta.\,\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,v$$

Results

$$w ::= v \mid \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,w$$

Note

- Abstractions λ's and Λ's are always values because they are pure, *i.e.* typechecked in a context Γ without $\exists\alpha$'s.
- Otherwise, impure abstractions should be treated linearly.

Reduction — Call-by-value small-step reduction semantics

Elimination rules: Usual reduction rules (for λ and Λ, records) plus,

$$\text{open } \langle\beta\rangle\ \exists\alpha.\ \mathcal{M} \rightsquigarrow \mathcal{M}[\alpha \leftarrow \beta]$$

$$\nu\beta.\ \Sigma\ \langle\beta\rangle\ (\alpha = \tau)\ w \rightsquigarrow w[\beta \leftarrow \alpha][\alpha \leftarrow \tau]$$

+ Extrusion rule applies for all extrusion contexts E (definition omitted)

$$E\left[\ \Sigma\ \langle\beta\rangle\ (\alpha = \tau)\ w\ \right] \rightsquigarrow \Sigma\ \langle\beta\rangle\ (\alpha = \tau)\ E[w]$$

+ Propagation of coercions (uninteresting reduction rules, see sample)

Reduction Type soundness

Theorem (Subject reduction)

If $\Gamma \vdash \mathcal{M} : \tau$ *and* $\mathcal{M} \rightsquigarrow \mathcal{M}'$, *then* $\Gamma \vdash \mathcal{M}' : \tau$.

Theorem (Progress)

If $\Gamma \vdash \mathcal{M} : \tau$ *and* Γ *does not contain value variable bindings, then either* $\mathcal{M}$ *is a result, or it is reducible.*

Double vision

This example is rejected

$$\text{let } f = \lambda(x : \beta)x \text{ in } \Sigma\,\langle\beta\rangle\,(\alpha = \mathit{int})\,f\;(1 : \alpha)$$

We do not know that the external type β in the type of f is equal to the internal view α also equal to int.

Keep this information in the context and use it whenever needed

$$\textsc{Sigma}\quad \frac{\Gamma, \forall\alpha, \Gamma', \forall(\alpha \lhd \beta = \tau') \vdash \mathcal{M} : \tau}{\Gamma, \exists\beta, \Gamma' \vdash \Sigma\,\langle\beta\rangle\,(\alpha = \tau')\,\mathcal{M} : \tau[\alpha \leftarrow \beta]}$$

$$\textsc{Sim}\quad \frac{\Gamma \vdash \mathcal{M} : \tau' \qquad \Gamma \vdash \tau \lhd \tau'}{\Gamma \vdash \mathcal{M} : \tau}$$

Rules for $\Gamma \vdash \cdot \lhd \cdot$ are omitted—it is a congruence generated by the equalities between internal and external names in Γ.

Avoiding recursive types

Why?

Internal recursion, through openings:

$$\text{let } x = \exists(\alpha = \beta \to \beta)\,\mathcal{M} \text{ in open } \langle\beta\rangle\, x$$

reduces to:

$$\text{open } \langle\beta\rangle\, \exists(\alpha = \beta \to \beta)\,\mathcal{M}$$

$\exists(\alpha = \tau)\,\mathcal{M}$ stands for $\exists\gamma.\,\Sigma\,\langle\gamma\rangle\,(\alpha = \beta \to \beta)\,\mathcal{M}$

which leads to the recursive equation $\beta = \beta \to \beta$.

External recursion, through open witness definitions:

$$\begin{aligned}\{\ell_1 &= \Sigma\,\langle\beta_1\rangle\,(\alpha_1 = \beta_2 \to \beta_2)\,\mathcal{M}_1\ ;\\ \ell_2 &= \Sigma\,\langle\beta_2\rangle\,(\alpha_2 = \beta_1 \to \beta_1)\,\mathcal{M}_2\ \}\end{aligned}$$

already contains the recursive equations $\beta_1 = \beta_2 \to \beta_2$ and $\beta_2 = \beta_1 \to \beta_1$

Cannot occur in System F.

Avoiding recursive types

Why?

Origin of the problem

$$\textsc{Sigma}\quad \frac{\Gamma, \forall\beta, \Gamma', \forall(\alpha = \tau) \vdash \mathcal{M} : \tau'}{\Gamma, \exists\beta, \Gamma' \vdash \Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,\mathcal{M} : \tau'[\alpha \leftarrow \beta]}$$

β may appear in τ which is later meant to be equated with β.

Solutions

1. Remove $\forall\beta$ from the premise:
   - requires that Γ' does not depend on β either.
   - too strong:
     - at least requires some special case for let-bindings.
     - some useful cases would still be eliminated.
2. Keep a more precise track of dependencies.

Avoiding recursive types

How? 

Traditional view

- Γ is a mapping together with a total ordering on its domain.

Generalization

- Organize the context as a strict partial order, where bindings b depends on type variable bindings $\forall\alpha$ or $\exists\alpha$.

Avoiding recursive types

How? ◀ ▶

Traditional view

- Γ is a mapping together with a total ordering on its domain.

Generalization

- Organize the context as a strict partial order.
- A binding b may depend on type variables bindings $\exists\alpha$, $\forall\alpha$, $\forall(\alpha = \tau)$
- Γ is a pair $(\mathcal{E}, \prec)$ where $\mathcal{E}$ is a set of bindings ordered by $\prec$.
- We write $\Gamma, (b \prec \mathcal{D}), \Gamma'$ when
  - $dom\,\Gamma \not\prec b$ and $b \not\prec dom\,\Gamma'$ and $\mathcal{D}$ is the set b depends on.

Zipping of contexts is redefined

- $(\mathcal{E}_1, \prec_1) \curlyvee (\mathcal{E}_2, \prec_2) = \big((\mathcal{E}_1 \curlyvee \mathcal{E}_2), (\prec_1 \cup \prec_2)^+\big)$
- $\mathcal{E}_1 \curlyvee \mathcal{E}_2 = \{b_1 \curlyvee b_2 \mid b_1 \in \mathcal{E}_1, b_2 \in \mathcal{E}_2, dom\,b_1 = dom\,b_2\}$
  $\cup\, \{\exists\beta \mid \beta \in dom\,\mathcal{E}_1 \,\Delta\, dom\,\mathcal{E}_2\}$

(weakening to remove unnecessary dependencies)

Avoiding recursive types

How? ◀ ▶

SIGMA

$$\frac{\mathcal{D}' \setminus \mathit{dom}\,\Gamma' \subseteq \mathcal{D} \qquad \Gamma, (\forall\beta \prec \mathcal{D}), \Gamma', (\forall(\alpha = \tau') \prec \mathcal{D}') \vdash \mathcal{M} \,:\, \tau}{\Gamma, (\exists\beta \prec \mathcal{D}), \Gamma' \vdash \Sigma\,\langle\beta\rangle\,(\alpha = \tau')\,\mathcal{M} \,:\, \tau[\alpha \leftarrow \beta]}$$

In particular,

- Free variables of the witness type τ' are in $\mathcal{D}'$ (by well-formedness).
- Variables $\mathcal{D}'$ depends on are also in $\mathcal{D}'$ (by transitivity of $\prec$).
- Variables of $\mathit{dom}\,\Gamma$ that the witness type τ' depends on must be in $\mathcal{D}$ (as requested by the side condition)
- The witness may not depend on β.

Avoiding recursive types

How? ◀ ▶

SIGMA

$$\frac{\mathcal{D}' \setminus dom\,\Gamma' \subseteq \mathcal{D} \qquad \Gamma, (\forall\beta \prec \mathcal{D}), \Gamma', (\forall(\alpha = \tau') \prec \mathcal{D}') \vdash \mathcal{M} \,:\, \tau}{\Gamma, (\exists\beta \prec \mathcal{D}), \Gamma' \vdash \Sigma\,\langle\beta\rangle\,(\alpha = \tau')\,\mathcal{M} \,:\, \tau[\alpha \leftarrow \beta]}$$

Prevents typechecking:

$$\begin{array}{rll} \{\ell_1 = \Sigma\,\langle\beta_1\rangle\,(\alpha_1 = \beta_2 \to \beta_2)\,\mathcal{M}_1\ ; & & \textit{implies } \beta_1 \prec \beta_2 \\ \ell_2 = \Sigma\,\langle\beta_2\rangle\,(\alpha_2 = \beta_1 \to \beta_1)\,\mathcal{M}_2\ \} & & \textit{implies } \beta_2 \prec \beta_1 \end{array}$$

But allows typechecking:

$$\begin{array}{rll} \{\ell_1 = \Sigma\,\langle\beta_1\rangle\,(\alpha_1 = \mathit{int})\,\mathcal{M}_1\ ; & & \textit{implies nothing} \\ \ell_2 = \Sigma\,\langle\beta_2\rangle\,(\alpha_2 = \beta_1 \to \beta_1)\,\mathcal{M}_2\ \} & & \textit{implies } \beta_2 \prec \beta_1 \end{array}$$

Avoiding recursive types

How? ◀ ▶

$$\textsc{Open}\quad \frac{\Gamma \vdash \mathcal{M} \,:\, \exists\beta.\,\tau \qquad \{(\exists\alpha) \in \Gamma\} \cup \{(\forall\alpha) \in \Gamma\}}{\Gamma, (\exists\beta \prec \mathcal{D}) \vdash \mathsf{open}\ \langle\beta\rangle\ \mathcal{M} \,:\, \tau}$$

$$\textsc{Let}\quad \frac{\{(\exists\alpha) \in \Gamma_2 \mid (\forall\alpha) \in \Gamma_1\} \subseteq \mathcal{D} \qquad \Gamma_1 \vdash \mathcal{M}_1 \,:\, \tau_1 \qquad \Gamma_2, (x : \tau_1 \prec \mathcal{D}) \vdash \mathcal{M}_2 \,:\, \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash \mathsf{let}\ x = \mathcal{M}_1\ \mathsf{in}\ \mathcal{M}_2 \,:\, \tau_2}$$

Open:

- Γ must not depend on β.
- β depends on every existential or univeral bindings in Γ.

Let:

- x depends on all existential bindings $(\exists\alpha \in \Gamma_2)$ that are determined in $\mathcal{M}_2$ and universally used $(\forall\alpha \in \Gamma_1)$ in M_1.

Avoiding recursive types

How? ◀ ▶

$$\textsc{Open}\quad \frac{\Gamma \vdash \mathcal{M} \,:\, \exists\beta.\,\tau \qquad \{(\exists\alpha) \in \Gamma\} \cup \{(\forall\alpha) \in \Gamma\}}{\Gamma, (\exists\beta \prec \mathcal{D}) \vdash \mathsf{open}\ \langle\beta\rangle\ \mathcal{M} \,:\, \tau}$$

$$\textsc{Let}\quad \frac{\{(\exists\alpha) \in \Gamma_2 \mid (\forall\alpha) \in \Gamma_1\} \subseteq \mathcal{D} \qquad \Gamma_1 \vdash \mathcal{M}_1 \,:\, \tau_1 \qquad \Gamma_2, (x : \tau_1 \prec \mathcal{D}) \vdash \mathcal{M}_2 \,:\, \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash \mathsf{let}\ x = \mathcal{M}_1\ \mathsf{in}\ \mathcal{M}_2 \,:\, \tau_2}$$

Prevents typechecking:

let $x = \exists(\alpha = \beta \to \beta)\,\mathcal{M}$ in — *implies $x \prec \beta$, since $\beta \in dom\,\Gamma_2$*

open $\langle\beta\rangle\ x$ — *requires $x \not\prec \beta$ since $x \in dom\,\Gamma$*

Avoiding recursive types

Restriction to $F^{\downarrow -}$

Why?

- Enforces abstract types to follow the scope of value variables.
- Programs can be translated to System F.
- Dependencies reduces to well-formedness dependencies, as usual.

Avoiding recursive types

Restriction to $F^{\curlyvee -}$

1) Replace $\Gamma_1 \curlyvee \Gamma_2$ in typing rules by more restrictive versions, $\Gamma_1 \barwedge \Gamma_2$ for let-bindings and $\Gamma_1 \bar{\curlyvee} \Gamma_2$ for applications and products.

$$\forall\alpha \barwedge \exists\alpha = \exists\alpha \qquad \exists\alpha \barwedge \cdot = \exists\alpha \qquad \forall\alpha \barwedge \forall\alpha = \forall\alpha$$

$$\cdot \bar{\curlyvee} \exists\alpha = \exists\alpha \qquad \exists\alpha \bar{\curlyvee} \cdot = \exists\alpha \qquad \forall\alpha \bar{\curlyvee} \forall\alpha = \forall\alpha$$

- Side condition of rule LET becomes a tautology and can be removed.
- Dependencies on rules SIGMA and OPEN become useless (acyclicity check cannot fail as a result of these).

2) Restrict rule Sigma so that β does not depend on Γ'.

- Dependencies are thus reduced to well-formedness dependencies.

Expressiveness — Relation to System F

Open existential types are more expressive than System F

System F is a special case, using the syntactic sugar.

Conversely, open existential types do not enforce abstract types to follow the scope of type variables. This is useful in practice, but goes beyond what can be done in System F.

Open existential types with more restrictive dependencies

Using more restrictive dependencies ($\mathrm{F}^{\curlyvee-}$) enforces abstract types to follow the scope of type variables:

- System F is a still a subset of $\mathrm{F}^{\curlyvee-}$ (using the syntactic sugar).
- Pure expressions of $\mathrm{F}^{\curlyvee-}$ can be translated to System F such that
  - Semantics, type abstraction, and typings are preserves
  - β-reduction steps are preserved, but new *let*-reduction steps are introduced.

Expressiveness — Translation to System F

Algorithm

1. From the typing derivation, insert coercions around Σs and $\exists$s in order to get $\Sigma\,\langle\beta\rangle\,(\alpha = \tau')\,(\mathcal{M} : \tau)$ and $\exists\alpha.\,(\mathcal{M} : \tau)$.
2. Replace existential quantifiers by uses of pack, according to the rule: $\exists\alpha.\,(\mathcal{M} : \tau) \rightarrowtriangle \nu\alpha.\,\mathsf{let}\ x = \mathcal{M}\ \mathsf{in}\ \mathsf{pack}\ \alpha, x\ \mathsf{as}\ \exists\alpha.\,\tau$
3. Extrude open's and Σ's using let-bindings and intrude ν's so that they get closer to each other.
4. Recover System F constructs:
   $\nu\alpha.\,\mathsf{let}\ x = \mathsf{open}\ \langle\alpha\rangle\ \mathcal{M}\ \mathsf{in}\ \mathcal{M}' \rightarrowtriangle \mathsf{unpack}\ \mathcal{M}\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ \mathcal{M}'$
   $\nu\alpha.\,\mathsf{let}\ x = \Sigma\,\langle\alpha\rangle\,(\alpha = \tau_0)\,(\mathcal{M} : \tau)\ \mathsf{in}\ \mathcal{M}'$
   $\rightarrowtriangle \mathsf{unpack}\ (\mathsf{pack}\ \tau_0, \mathcal{M}[\alpha \leftarrow \tau_0]\ \mathsf{as}\ \exists\alpha.\,\tau)\ \mathsf{as}\ \alpha, x\ \mathsf{in}\ \mathcal{M}'$
5. Finally, remove all remaining coercions.

Expressiveness

Translation to System F

Extrusion of Open's and Σs

$$\nu\alpha.\,\mathsf{let}\ x = Q^\alpha\ \mathcal{M}\ \mathsf{in}\ \mathcal{M}' \rightarrowtriangle \nu\alpha.\,\mathsf{let}\ y = Q^\alpha\ \mathsf{in}\ \mathsf{let}\ x = y\ \mathcal{M}\ \mathsf{in}\ \mathcal{M}'$$
$$\nu\alpha.\,\mathsf{let}\ x = \mathcal{M}\ Q^\alpha\ \mathsf{in}\ \mathcal{M}' \rightarrowtriangle \nu\alpha.\,\mathsf{let}\ y = Q^\alpha\ \mathsf{in}\ \mathsf{let}\ x = \mathcal{M}\ y\ \mathsf{in}\ \mathcal{M}'$$

Intrusion of νs

$$\nu\alpha.\,(Q^\alpha\ \mathcal{M}) \rightarrowtriangle (\nu\alpha.\,Q^\alpha)\ \mathcal{M}$$
$$\nu\alpha.\,(\mathcal{M}\ Q^\alpha) \rightarrowtriangle \mathcal{M}\ (\nu\alpha.\,Q^\alpha)$$
$$\nu\alpha.\,(\mathsf{let}\ x = \mathcal{M}\ \mathsf{in}\ Q^\alpha) \rightarrowtriangle \mathsf{let}\ x = \mathcal{M}\ \mathsf{in}\ \nu\alpha.\,Q^\alpha$$

Context with open existentials

$$\begin{array}{rcl} Q^\alpha & ::= & \mathsf{open}\ \langle\alpha\rangle\ \mathcal{M} \mid \Sigma\ \langle\alpha\rangle\ (\beta = \tau)\ \mathcal{M} \mid Q^\alpha\ \mathcal{M} \mid \mathcal{M}\ Q^\alpha \\ & \mid & Q^\alpha\ [\tau] \mid \mathsf{pack}\ \tau, Q^\alpha\ \mathsf{as}\ \exists\beta.\,\tau' \mid \nu\beta.\,Q^\alpha \mid Q^\alpha.\ell \\ & \mid & \mathsf{open}\ \langle\beta\rangle\ Q^\alpha \mid \{(\ell_i = \mathcal{M}_i)^{i\in I}\ ;\ \ell = Q^\alpha\ ;\ (\ell_j = \mathcal{M}_j)^{j\in J}\} \\ & \mid & \Sigma\ \langle\beta\rangle\ (\gamma = \tau)\ Q^\alpha \mid \mathsf{let}\ x = \mathcal{M}\ \mathsf{in}\ Q^\alpha \mid \mathsf{let}\ x = Q^\alpha\ \mathsf{in}\ \mathcal{M} \end{array}$$

Expressiveness

Relation to System F

Reading through the Curry-Howard isomorphism for $F^{\curlyvee -}$

- The formulae are the same as in System F.
- The provable formulae are the same as in System F.
- They are more proofs in $F^{\curlyvee -}$, which can be assembled in mode modular ways.

Expressiveness — Adding recursion

Type level recursion

- Add equi-recursive types
- Let recursive types appear from $\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,\mathcal{M}$ (by not tracking dependencies), or
- Add an expression $\Sigma\,\langle\beta\rangle\,(\alpha \approx \tau)\,\mathcal{M}$ that behaves as $\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,\mathcal{M}$ but does not make β depend on what τ' depends on.

  Then, recursive types always originate from an $\approx$-form of Σ's.

Term level recursion

- allow restricted fixpoints so that are guaranteed to be well-formed.
- allow more fixpoints and raise an exception when ill-founded recursion is detected at runtime.

The combination can model recursive modules

Expressiveness — Adding recursion

Example

$$
\begin{array}{l}
\nu\beta_1.\,\nu\beta_2. \\
\quad \text{let rec } A : \{\mathit{compare} : \beta_1 \to \beta_1 \to \mathit{bool}; \ldots\} = \\
\quad\quad \Sigma\,\langle\beta_1\rangle\,(\alpha \approx \text{Leaf of } \mathit{int} \mid \text{Node of } \beta_2)\,\{ \\
\quad\quad\quad \mathit{compare} = \lambda(t_1 : \alpha)\,\lambda(t_2 : \alpha)\ \text{ match } t_1, t_2 \text{ with} \\
\quad\quad\quad\quad \mid \mathit{Leaf}\ i_1, \mathit{Leaf}\ i_2 \to i_2 - i_1 \\
\quad\quad\quad\quad \mid \mathit{Node}\ n_1, \mathit{Node}\ n_2 \to \mathit{ASet.compare}(n_1)(n_2) \\
\quad\quad\quad\quad \mid \mathit{Leaf}\ \_, \mathit{Node}\ \_ \to 1 \mid \mathit{Node}\ \_, \mathit{Leaf}\ \_ \to -1 \\
\quad\quad\quad \mathit{leave} = \lambda(x_1)\ \mathit{Leave}\ x_1 \\
\quad\quad\quad \ldots \\
\quad\quad \} \\
\quad \mathit{and}\ \mathit{ASet} : \mathit{SET}(\beta_1, \beta_2) = \\
\quad\quad \text{open}\,\langle\beta_2\rangle\,(\mathit{Set.Make}[\beta_1](A)) \\
\quad \mathit{in}\ \ldots
\end{array}
$$

Summary (open existential types)

Type generativity can be explained by open existential types

- Standard small step reduction semantics.
  Scope extrusion is a good, fine grain explanations of type abstraction
- Linearity provides a good explanation of type generativity.
- Close connection to logic with new ways of assembling proofs.

Easy modeling of double-vision

Accommodate recursive type and value definitions

Explains modules as first-class records

- whose components have abstract types,
- but without type components!

Summary (open existential types)

However,

- Sharing is by parametrization,
- Which does not scale up.
- This needs to be solved—without bringing back type components. (on going work)

Ideas to bring back home — Summary

Modules with type components

- Common approach to generativity with path-dependent types.
- Not so easy as it appears.

Open existentials keep types out of modules

- No need for dependent types; more intuitive; modules *are* records.
- Sharing is by parametrization. Still need support for scalability.

Mixins modules

- More expressive and more flexible; closer to object-oriented languages.
- Need good static semantics, perhaps with open existential types.
- Tracking down ill-formed recursion is hard.

Should we accept some dynamic errors, here?

Ideas to bring back home — State of the art

Pessimistic view

- Despite man years of use, the state of the art is still far behind what one could expect.
- There remain differences between the theory and the implementations.

Optimistic view

- Modules have been an area of continuous research.
- Recently, there have been significant advances.
- There are still *places* and *needs* for new results...

Appendix

Answers to exercises I

Exercise 1, page 39 By induction on the definition of $\Gamma \vdash \pi : \mathcal{S}$, one may show that π or a prefix of π is in Γ.
In Γ, type definitions are either abstract or concrete, but not strengthened (which would be an ill-formed recursive definition).
One may then build a derivation of $\Gamma \vdash \mathcal{S}/p <: \mathcal{S}$ by induction on the definition of $\mathcal{S}$ where the only non trivial case is for concrete type definitions, which should then be found in the context.
(For abstract type definition, it suffices to use subtyping axiom $\Gamma \vdash \pi : \pi.\ell <: \pi : 0$.)

Exercise 1 (continued) This ensures that the strengthening rule can be applied aggressively, since the weaker type may always be recovered by subtyping.

Answers to exercises II

Exercise 3, page 56

```
module App =
  functor(T: sig module type A module type B end) →
    functor(F: functor (X : T.A) → T.B) → functor(Y:T.A)→ F(Y)
module Id = functor(T: sig module type A end) → functor(X:T.A)→ X
module Test(Q: sig module type A end) = struct
  module A = App (struct module type A = Q.A module type B = Q.A end)
  module I = Id (struct module type A = Q.A end)
  module R = A (I)
  module T =
    App (struct
      module type A = functor (Y : Q.A) → Q.A
      module type B = functor (Y : Q.A) → Q.A
    end) (A) (I)
end;;
```

Answers to exercises III

Bibliography I

- Derek Dreyer. Recursive type generativity. *Journal of Functional Programming*, pages 433–471, 2007.
- Derek Dreyer. A type system for well-founded recursion. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 293–305, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X.
- Derek Dreyer and Matthias Blume. Principal type schemes for modular programs. In *ESOP*, number 4421 in LNCS, pages 441–457. Springer Verlag, 2007.
- Derek Dreyer and Andreas Rossberg. Mixin' up the ml module system. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 307–320, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7.

Bibliography II

- Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 236–249, 2003.
- Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. *ACM Transactions on Programming Languages and Systems*, 27(5):857–881, 2005.
- Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Compilation of extended recursion in call-by-value functional languages. In *International Conference on Principles and Practice of Declarative Programming*, pages 160–171. ACM Press, 2003.
- Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

Bibliography III

- Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.
- Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 109–122. ACM Press, 1994.
- Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL'09)*, pages 63–74, Savannah, Georgia, USA, January 2009.