Modularity, *Module Systems*

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

Didier Rémy

INRIA-Rocquencourt

January 2010

Note

This course is largely based on the description and the implementation of module the OCaml module system, and on the formalisation of ML modules in the litterature, especially in the following papers:

- ❷, ❸ Leroy (1994).
- ❹ Dreyer and Rossberg (2008); Hirschowitz and Leroy (2005).
- ❺ 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 ❷, ❸, ❹ contain formal definitions, but no formal result. For part ❺, complete formal definitions and all results can be found in Montagu and Rémy (2009).

Modular programming

What is it about?

- Split large monolithic programs into an assembling of smaller pieces, called *components*

Why?

- Understand components independently of one another (enforce their invariants, verify or prove them)
- Hide low-level details and implementations of components
- Maintain programs component by component
- Facilitate reusability of components
- Compile components separately

Modular programming

Modularity is not specific to computer science

- Compare with mechanics: Large systems in mechanics (airplanes, power plants, *etc.*) are also large and complex, and usually decomposed into small units for very similar reasons.

However, modular programming is peculiar in two ways

- Programs are fragile because their behavior is not continuous: a bogus program may crashe all at once without any prior notice.
- Programs can be dupplicated at (almost) no costs.
  This favors the generation, specialization, adaptation of components even further.

Modular programming

How?

- Poor man's modular programming.
  - No specific support, but a lot of discipline, which is not enforced by the language.
  - Limited expressiveness, may require acrobatics. (e.g. use base language records to group related definitions, emulate objects, etc.), especially if the language does not have good support for records, subtyping, type abstraction, etc.

Modular programming

How?

- Poor man's modular programming.
- Object-oriented paradigm.
  - Data-centric approach: data (objects) with similar structure and behavior are created from a class that groups all functions that operate on similar data.
  - Abstraction by hiding the representation of objects and exposing only some functions to operate on them.
  - Emphasis is put on reusability via inheritance, but it often lacks a good abstraction mechanism.
- Module systems
  - Group base-language definitions into modules
  - Provide a small calculus to combine modules together.

Hiding mecanism (parenthesis)

By lexical scoping

Keep some definitions local (private) and only export public definitions.

```
let monotonic_int_ref() =
  let r = ref 0 in
  let setter n = if n > !r then r := n in
  let getter () = !r in
  setter, getter
```

This is typically the object oriented style.

Hiding mecanism (parenthesis)

By lexical scoping

Keep some definitions local (private) and only export public definitions.

By type abstraction

Make types of critical parts of values abstract to prevent forgery.

```
module Unsafe_mref = struct
  type a mref = a ref
  let mref = ref
  let set r n = if !r > n then r := n
  let get r = !r
end
```

```
module Mref : sig
  type a mref
  val mref : a → a mref
  val set : a mref → a → unit
  val get : a mref  → a
 end = Unsafe_mref
```

This is more typical of modules systems.

Hiding mecanism (parenthesis)

By lexical scoping

Keep some definitions local (private) and only export public definitions.

By type abstraction

Make types of critical parts of values abstract to prevent forgery.
This is more typical of modules systems.

Comparison

- Hiding by abstraction is more flexible: it allows to return values of which some parts are private.
- Both forms can be combined, which is typical of module systems.

Modules The different eras

- Modular 3, CamlLight (later), Java packages (much later)
  Name space control *No type generativity*
- ML modules at their early stage (effectful *stamp* semantics)
  SML, functors, higher-order functors *Introduction of 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 (with dependent types and singleton kinds)
  + 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
- Modules may be hierarchical
- Modules may take other modules as arguments or return them as results (functors)

Already present in the base-language.

The essential features, specific to modules

- *Abstract types and type generativity*
- *Type definitions and their propagation*

Also the source of most difficulties... as it introduces

- type fields in record-like structures
- sharing of abstract types

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.

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.

Why not?

- Perhaps surprisingly, abstract types in modules systems are not primarily used for mixing data with different representation but accessible via a common interface, which is not permitted by second-class module systems.

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 model type abstraction but not in a modular way
- An *ad hoc* solution, based on paths, traces type identities

Abstract types Why not existential types?

Existential types

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

Typing rules

Exists

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

Exists

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

Abstract types — Why not existential types?

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

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

Problem

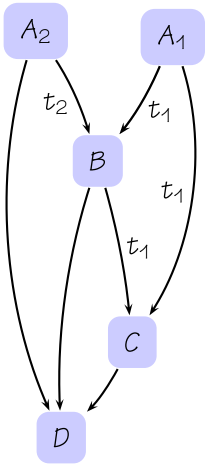
- Existential types only model abstract types in monolithic programs.
- Their uses cannot be spread in different program components—a key pattern of modular programming.

Abstract types Path-based approach

Modules (represented by boxes) are assembled in complex ways. Typically, types are 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 different module A_2 that just looks like A_1.

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*


$$\begin{tikzcd}
A_2 \arrow[dr, "t_2"] \arrow[dd, bend right] & & A_1 \arrow[dl, "t_1"'] \arrow[d, bend left, "t_1"] \\
& B \arrow[d, "t_1"] \arrow[dl] & C \\
D & & \\
\end{tikzcd}$$

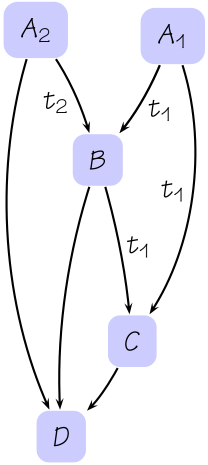
Abstract types Path-based approach

Paths are used in a critical way

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

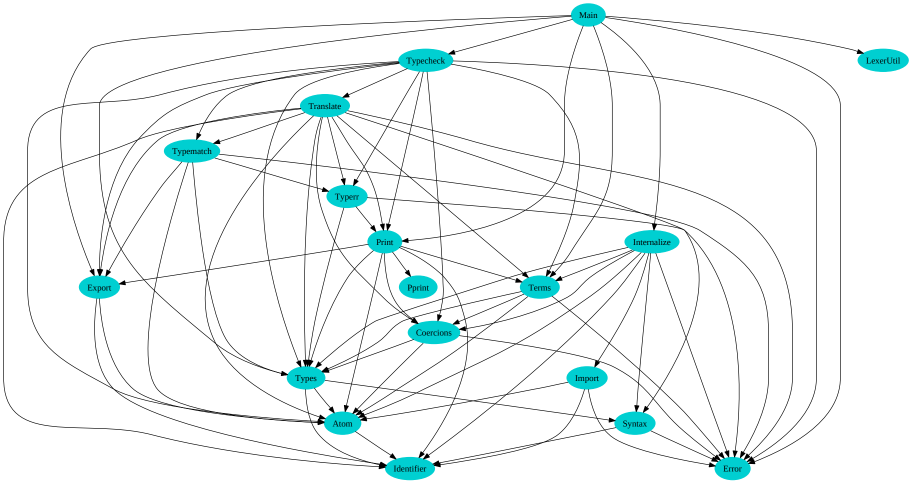
Paths are also used (in a not so critical way)

- to access other definitions (expressions, submodules).


$$\begin{tikzcd}
A_2 \arrow[dr, "t_2"] \arrow[ddd, bend right] & & A_1 \arrow[dl, "t_1"'] \arrow[dd, bend left, "t_1"] \\
 & B \arrow[dr, "t_1"] \arrow[dd] & \\
 & & C \arrow[dl] \\
 & D &
\end{tikzcd}$$

Abstract types Path-based approach

Dépendencies for the small programming project



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 represents two different currencies.

The application 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

$(\varphi)\{$
$\quad INT = (\psi)\{$
$\quad\quad t : *;$
$\quad\quad zero : \psi.t;$
$\quad\quad succ : \psi.t \to \psi.t$
$\quad \}$
$\quad Int_1 = (\psi)\{$
$\quad\quad t = int;$
$\quad\quad zero = 0;$
$\quad\quad succ = \lambda(x)\ x{+}1$
$\quad \} : \varphi.INT;$
$\quad Int_2 = \varphi.Int_1 : \varphi.INT;$
$\quad rejected = \varphi.Int_1.succ\ \ \varphi.Int_2.zero$
$\}$

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

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

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 $\ell : S = \mathcal{M}$ for $\ell = (\mathcal{M} : S)$ may be used for convenience.

Syntax

Paths

Identifiers

φ	$::=$	$\varphi_1 \mid \varphi_2 \mid \ldots$	Variables
ℓ	$::=$	$\ell_1 \mid \ell_2 \mid \ldots$	Labels
π	$::=$	$\varphi \mid \pi.\ell$	Paths

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

ρ	$::=$	$\pi \mid \rho \to \rho \mid \rho(\rho) \mid \mathit{int} \mid \ldots$	Simple types
τ	$::=$	$\pi \mid \lambda(\bar{a})\ \rho$	Type functions†
σ	$::=$	$\forall \bar{a}.\rho$	Type schemes†

Expressions

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

Kinds of types

$$\kappa \quad ::= \quad * \mid * \to \kappa$$

† We write ρ for $\lambda(\emptyset)\ \rho$ or $\forall\emptyset.\rho$ when $\bar{a}$ is empty.

Type functions

We allow type functions $\lambda(\bar{a})\,\tau$ of kind $\bar{*} \longrightarrow *$ (*i.e.* $* \longrightarrow * \ldots \longrightarrow *$), as in F^{ω}. This is to model type definitions such as in

$$\textbf{struct type } a\ t = a\ \mathit{list}\ \textbf{end} : \textbf{sig type } a\ t = a\ \mathit{list}\ \textbf{end}$$

that defines a type module with a type component t of kind $* \longrightarrow *$ In our syntax, we write $(\varphi)\{t = \lambda(a)\ \mathit{list}(a)\} : (\varphi)\{t = \lambda(a)\ \mathit{list}(a)\}$ (for both the structure and its signature).

However, type functions only take types of kind $*$ as arguments and can only appear in signature definitions.

At first reading, one may ignore type functions by assuming that all type definitions are of kind $*$, which still retains most technicalities.

As an alternative, we could allow type functions anywhere, as in F^{ω}, and treat types up β-conversion in any context.

Syntax Module language

Definitions

$$
\begin{array}{lll}
d & ::= & \\
 & \mid & \ell : \kappa = \tau \\
 & \mid & \ell = a \\
 & \mid & \ell = \mathcal{M}
\end{array}
$$

Specifications

s	$::=$	$\ell : \kappa$	Abstract type
	$\mid$	$\ell : \kappa = \tau$	Type definition
	$\mid$	$\ell : \sigma$	Expression
	$\mid$	$\ell : S$	Module

Modules / Signatures

$\mathcal{M}$	$::=$	π	S	$::=$	π	
	$\mid$	$(\varphi)\{d; \ldots d\}$		$\mid$	$(\varphi)\{s; \ldots s\}$	Submodules
	$\mid$	$\mathcal{M}\ (\mathcal{M})$				Applications
	$\mid$	$\lambda(\varphi : S)\ \mathcal{M}$		$\mid$	$(\varphi : S) \to S$	Functors

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

For conciseness, we may write $\ell = \tau$ instead of $\ell : \kappa = \tau$ leaving κ implicit from context.

Syntax — Naming convention

Although we formally have a single set of variables and of a single set of labels, in examples, we often (but not always) 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	s, t, u	lowercase
signature	S	—	S, T, U	UPPERCASE
expression	a	x, y	m, n, v, f,	lowercase
module	$\mathcal{M}$	X, Y	M, N, V, F,	Capitalized
any		φ, ψ	ℓ	

Typechecking — Contexts

Typing contexts $\Gamma \quad ::= \quad \emptyset \mid \Gamma, \pi : \kappa \mid \Gamma, \pi : \kappa = \tau \mid \Gamma, \pi : \sigma \mid \Gamma, \pi : S$

- 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:

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

where b stands for κ, τ', or S

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 path π so as to preserve sharing of identities.
- Elininating the free references to φ inside b by unfolding φ as $(\varphi)\{\bar{s}_1; \ell : b; \bar{s}_2\}$ would loose sharing of abstract types—and could be ill-formed since submodule signatures cannot be projected.

Typechecking Example

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

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

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

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

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

However, this judgment holds by *equivalence*.

Typechecking

Type equivalence

Type equivalence $\approx$ is generated from type definitions that are directly or indirectly bound in Γ.

$$\frac{\pi : \kappa = \tau \in \Gamma}{\Gamma \vdash \pi \approx \tau : \kappa} \qquad \frac{\Gamma \vdash \pi : (\varphi)\{\bar{d}_1; \ell : \kappa = \tau; \bar{d}_2\}}{\Gamma \vdash \pi.\ell \approx \tau[\pi/\varphi] : \kappa} \qquad \frac{\Gamma \vdash \pi \approx \lambda(\bar{a})\ \rho' : \bar{*} \to *}{\Gamma \vdash \pi(\bar{\rho}) \approx \rho'[\bar{\rho}/\bar{a}]}$$

Type equivalence is congruent for all type and module type constructors

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

Type equivalence of type definitions is also extensional:

$$\frac{\Gamma \vdash \rho \approx \rho' \qquad \bar{a}\ \textit{disjoint} \qquad \bar{a} \notin \Gamma}{\Gamma \vdash \lambda(\bar{a})\ \rho \approx \lambda(\bar{a})\ \rho' : \bar{*} \to *}$$

Typechecking Type equivalence

Type equivalence is used with the following conversion rules

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

Example (continued)

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

Typechecking

Modules

Structures

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

where $\varphi.$ lifts a sequence mapping ℓ_i to q_i to one mapping $\varphi.\ell_i$ to q_i. We still write s and d for lifted declarations and specifications.

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

$$\frac{}{\Gamma \vdash \emptyset : \emptyset} \qquad \frac{\Gamma \vdash d : s \qquad \Gamma, s \vdash \bar{d} : \bar{s}}{\Gamma \vdash (d;\bar{d}) : (s;\bar{s})}$$

Typechecking — Declarations

Individual declarations

$$\Gamma \vdash (\varphi.\ell : \kappa) \;:\; (\varphi.\ell : \kappa) \qquad \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash (\varphi.\ell : \kappa = \tau) \;:\; (\varphi.\ell : \kappa = \tau)}$$

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

Typing rules for the base-language are omitted:

- *Well-formedness of types $\Gamma \vdash \rho$ is straightforward; we write $\Gamma \vdash \lambda(\bar{a})\, \rho : \bar{*} \rightarrow *$ if $\Gamma \vdash \rho$ and $\bar{a}$ is a sequence of distinct type variables of the same length as $\bar{*}$.*
- *Typing of expressions, $\Gamma \vdash a : \sigma$ is as in **ML** (see previous lessons).*

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

Typechecking Modules

Functors

$$\frac{\Gamma \vdash S_1 \qquad \varphi \notin dom\,\Gamma \qquad \Gamma, \varphi : S_1 \vdash \mathcal{M} : S_2}{\Gamma \vdash \lambda(\varphi : S_1)\mathcal{M} : (\varphi : S_1) \to S_2}$$

Applications

$$\frac{\Gamma \vdash \mathcal{M}_1 : (\varphi : S_2) \to S_1 \qquad \Gamma \vdash \mathcal{M}_2 : S_2}{\Gamma \vdash \mathcal{M}_1(\mathcal{M}_2) : S_1[\mathcal{M}_2/\varphi]}$$

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

$S_1[\mathcal{M}_2/\varphi]$ is ill-defined and $\mathcal{M}_1\ (\mathcal{M}_2)$ is ill-typed if $\mathcal{M}_2$ is not a path and φ occurs free in S_1.

Typechecking

Signatures

$$\frac{\pi \in \operatorname{dom}\Gamma}{\Gamma \vdash \pi} \qquad \frac{\Gamma \vdash S_1 \qquad \Gamma, \varphi : S_1 \vdash S_2}{\Gamma \vdash (\varphi : S_1) \to S_2} \qquad \frac{\Gamma \vdash \varphi.(\bar{s})}{\Gamma \vdash (\varphi)\{\bar{s}\}}$$

Declarations (we fold well-formedness of individual)

$$\Gamma \vdash \emptyset \qquad \frac{\Gamma \vdash s \qquad \Gamma, s \vdash \bar{s}}{\Gamma \vdash s; \bar{s}}$$

$$\Gamma \vdash (\varphi.\ell : \kappa) \qquad \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash (\varphi.\ell : \kappa = \tau)} \qquad \frac{\Gamma \vdash \sigma}{\Gamma \vdash (\varphi.\ell : \sigma)} \qquad \frac{\Gamma \vdash S}{\Gamma \vdash (\varphi.\ell : S)}$$

Subtyping **What is missing?**

$$(\varphi)\{\ t = int\ \ zero = 0;\ \ one = 1;\ \ succ = \lambda(x)\ x + \varphi.one;\ \}$$

has type?

$$(\varphi)\{\ t : *;\ \ zero : int;\ \ succ : int \longrightarrow int\ \}$$

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

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.

Subtyping

At the leaves

Abstract types.

$$\Gamma \vdash (\varphi.\ell : \kappa) <: (\varphi.\ell : \kappa)$$

(See following exercise)

$$\frac{\Gamma \vdash \varphi.\ell \approx \tau}{\Gamma \vdash (\varphi.\ell : \kappa) <: (\varphi.\ell : \kappa = \tau)}$$

Type definitions. Subtyping contains the equivalence of type definitions and allows to turn concrete type definitions into abstract ones.

$$\frac{\Gamma \vdash \tau_1 \approx \tau_2 : \kappa}{\Gamma \vdash (\varphi.\ell : \kappa = \tau_1) <: (\varphi.\ell : \kappa = \tau_2)} \qquad \Gamma \vdash (\varphi.\ell : \kappa = \tau) <: (\varphi.\ell : \kappa)$$

Value declarations.

Subtyping contains the equivalence and instantiation of type schemes:

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

Subtyping For structures

Subtyping allows to omit and reorder components

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

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 (even if they are not retained).
- components that are referred from retained components must also be retained (so that the resulting signature is well-formed).

Subtyping

Exercise

Exercise

Show that $(\varphi)\{\mathsf{t} : *; \mathsf{u} = \mathsf{t}\} <: (\varphi)\{\mathsf{u} : *; \mathsf{t} = \mathsf{u}\}$ Answer □

Subtyping Submodules and functors

Propagation

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

$$\frac{\Gamma \vdash S_1 <: S_2}{\Gamma \vdash (\varphi.\ell : S_1) <: (\varphi.\ell : S_2)} \qquad \frac{\Gamma \vdash S_1' <: S_1 \qquad \Gamma, \varphi : S_1' \vdash S_2 <: S_2'}{\Gamma \vdash (\varphi : S_1) \to S_2 <: (\varphi : S_1') \to S_2'}$$

Subtyping for non-dependent types

If φ appears neither in S_2 nor in S_2', the subtyping rules looks familiar:

$$\frac{\Gamma \vdash S_1' <: S_1 \qquad \Gamma \vdash S_2 <: S_2'}{\Gamma \vdash S_1 \to S_2 <: S_1' \to S_2'}$$

Subtyping

Comparison with $\mathsf{F}_{:>}$

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 (and not be in a subtype relationship with) the signature S_1 or S'_1.

In particular, we cannot reason under subtyping assumptions as in $\mathsf{F}_{:>}$.

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

Subtyping

Subtyping is implicit and can be used anywhere:

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

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

Therefore, we introduce a construct to enforce subtyping, called *sealing*:

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

Do not be fooled: implicit subtyping permits the principal signature of $\mathcal{M}$ *to be a proper subsignature* S' *of the principal signature* S *of* $(\mathcal{M} : S)$.

Subtyping

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

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

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

Subtyping Subsumption

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

$$\frac{\Gamma \vdash \mathcal{M}_1 : (\varphi : S_2) \to S_1 \qquad \Gamma \vdash \mathcal{M}_2 : S_2' \qquad \Gamma \vdash S_2' <: S_2}{\Gamma \vdash \mathcal{M}_1(\mathcal{M}_2) : S_1[\mathcal{M}_2/\varphi]}$$

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

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 limitation is also needed for type inference.

Typing rules

Example (2)

Give the best type to the following declarations

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

Typing rules

Example (2)

Give the best type to the following declarations

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

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

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

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

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

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

where:

$$\begin{aligned} (\varphi)\{s_1, \ldots s_n\}/\pi &\triangleq (\varphi)\{s_1/\pi, \ldots s_n/\pi\} \\ \pi'/\pi &\triangleq \pi' \\ (\varphi : S_1) \to S_2/\pi &\triangleq (\varphi : S_1) \to S_2 \end{aligned}$$

$$\begin{aligned} \ell : \sigma/\pi &\triangleq \ell : \sigma \\ \ell : \mathcal{M}/\pi &\triangleq \ell : (\mathcal{M}/\pi.\ell) \\ \ell : \kappa/\pi &\triangleq \ell : \kappa = \pi.\ell \\ \ell : \kappa = \tau/\pi &\triangleq \ell : \kappa = \pi.\ell \end{aligned}$$

Strengthening

Exercise

Exercise

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

Why is this important?

Strengthening

Exercise

Exercise

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

Why is this important?

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

This is used by type inferrence to infer principal signatures.

Strengthening

Exercise

Exercise

Consider the program:

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

Which of the applications are well-typed without strengthening?

With strengthening? □

Strengthening

Exercise

Exercise

Consider the program:

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

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?* □

Strengthening

Exercise

Exercise

Consider the program:

$\{\ F = \lambda(X : \{t : *\})\ \lambda(Y : \{t = X.t\})\ \{\};$
$\quad A : \{\ t : *\ \};$
$\quad B : \{\ t = A.t\ \};$
$\quad M_0 = F\ (A)\ (B);$ *ok*
$\quad M_1 = F\ (B)\ (A);$ *ok*
$\quad M_2 = F\ (A)\ (A);$ *ok*
$\}$

Which of the applications are well-typed without strengthening?

With strengthening? They all succeed. □

Strengthening just strong enough

It preserves equalities for aliases:

$$(\varphi)\left\{\begin{array}{l} \mathsf{M} = (\mathcal{M}_{\mathbb{N}} : \mathcal{S}_{\mathbb{N}}); \\ \\ \mathsf{N} \ = \varphi.\mathsf{M}; \\ \mathsf{m} \ = (\varphi.\mathsf{M}.\mathsf{m} = \varphi \mathsf{N}.\mathsf{m}); \end{array}\right\}$$

It preserves generativity for sealing:

$$(\varphi)\left\{\begin{array}{l} \mathsf{M} = (\mathcal{M}_{\mathbb{N}} : \mathcal{S}_{\mathbb{N}}); \\ \\ \mathsf{N} \ = (\varphi.\mathsf{M} : \mathcal{S}_{\mathbb{N}}); \\ \\ \mathsf{m} \ = (\varphi.\mathsf{M}.\mathsf{m} = \varphi \mathsf{N}.\mathsf{m}); \end{array}\right\}$$

Strengthening just strong enough

It preserves equalities for aliases:

$$(\varphi)\left\{\begin{array}{l} M = (\mathcal{M}_N : \mathcal{S}_N); \\ \\ N = \varphi.M; \\ m = (\varphi.M.m = \varphi N.m); \end{array}\right\} : \left\{\begin{array}{l} M \;\; = (\psi)\{t = *; m : \psi.t\}; \\ \varphi.M : (\psi)\{t = \varphi.M.t; M : \psi.t\} \\ N \;\; = (\psi)\{t = \varphi.M.t; M : \psi.t\} \\ m \; : \;\; \mathit{bool} \end{array}\right\}$$

It preserves generativity for sealing:

$$(\varphi)\left\{\begin{array}{l} M = (\mathcal{M}_N : \mathcal{S}_N); \\ \\ N = (\varphi.M : \mathcal{S}_N); \\ \\ m = (\varphi.M.m = \varphi N.m); \end{array}\right\} : \left\{\begin{array}{l} M \;\; = (\psi)\{t = *; m : \psi.t\}; \\ \varphi.M : (\psi)\{t = \varphi.M.t; M : \psi.t\} \\ N \;\; = (\psi)\{t : *; m : \psi.t\}; \\ \varphi.N : (\psi)\{t = \varphi.N.t; M : t\} \\ \mathit{fails} \end{array}\right\}$$

Strengthening Exercise

Consider again the example (2): what is the best type of

$$\left\{\begin{array}{lcl} \mathsf{F} & = & \lambda(X : S_{\mathbb{N}})\; X; \\ \mathsf{M} & = & \mathcal{M}_{\mathbb{N}}; \\ \mathsf{N} & = & \mathsf{F}\;(\mathsf{M}); \\ \mathsf{m} & = & \mathsf{N.m} + 1; \end{array}\right\}$$

Why is this a justification of sealing?

- Without strengthening, an application of $\lambda(X : S)\; X$ to $\mathcal{M}$ would be equivalent to sealing $\mathcal{M}$ with 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

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

Strengthening

Exercise

Consider again the example (2): what is the best type of

$$\left\{\begin{array}{ll} \mathsf{F} & = \lambda(X : S_{\mathcal{N}})\, X; \\ \mathsf{M} & = \mathcal{M}_{\mathcal{N}}; \\ \mathsf{N} & = \mathsf{F}\ (\mathsf{M}); \\ \mathsf{m} & = \mathsf{N.m} + 1; \end{array}\right\} : \left\{\begin{array}{ll} \mathsf{F} & = (X : S_{\mathcal{N}}) \longrightarrow (\varphi)\{\mathsf{t} = X.\mathsf{t}; \mathsf{M} : \varphi.\mathsf{t}\}; \\ \mathsf{M} & = \{\mathsf{t} = \mathit{int}; \mathsf{m} : \mathit{int}\}; \\ \mathsf{N} & = (\varphi)\{\mathsf{t} = \mathsf{M.t}; \mathsf{M} : \varphi.\mathsf{t}\}; \\ \mathsf{m} & : \ \mathit{int} \end{array}\right\}$$

Why is this a justification of sealing?

- Without strengthening, an application of $\lambda(X : S)\, X$ to $\mathcal{M}$ would be equivalent to sealing $\mathcal{M}$ with 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

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

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.
- If subtyping were used instead of subsumption.

It is *conjectured* that type inference returns a principal signature when it succeeds. However, type inference might fail when sometimes more specific type annotations would make it succeed.

Type inference Non-regular datatypes

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

```
type a term =
  Var of a |  App (a term * a term) | Abs of (a bind) term
and a bind = Zero | Succ of a
```

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 $\{id : a \rightarrow a\}$ but not with $\{id : \forall a.a \rightarrow a\}$.

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

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

Type inference The avoidance problem

This is a general problem when mixing subtyping and abstract types.

It is incorrect for an abstract type to escape its scope. When subtyping is allowed, it is sometimes possible to use subtyping to hide 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 $\mathsf{let}\ m = \mathcal{M}\ \mathsf{in}\ \mathcal{M}$. Then, the module

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

has principal but ill-formed signature $\{\mathsf{m} : X.\mathsf{M}.\mathsf{t}\}$ as X is not in scope.

Here, the module can also be given the equivalent signature $\{\mathsf{m} : \mathit{int}\}$ that *avoids* X, by conversion. However, this is not always possible.

Type inference — The avoidance problem

For example, this is no more the case if X.N.t is abstract, as in:

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

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

Exercise

Consider the signature S *equal to* $(\psi)\{\mathsf{t} : *; M : \{\mathsf{u} = \lambda(a)\, \psi.\mathsf{t}; \mathsf{s} = \psi.\mathsf{t}\}\}$. *Asume that a module* $\mathcal{M}$ *bound to* X *has signature* S*. What is the type of* $X.M$*?* Answer

What could be the signature of $\mathbf{let}\, X = \mathcal{M}\ \mathbf{in}\ X.M$ *if this expression were allowed and subtyping were implicit?* Answer □

Type inference — Can make the difference!

Reduces verbosity

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

In ML, the size of principal types may grow up exponentially.

Even if the size of types remains bounded by k, type information may be k times larger than the program. (Consider for example, large tuples encoded with pairs.)

Increases maintainability

Not writing *all* type information often keeps the source program more manageable.

It also avoids duplicating type information which increases maintanability

Type inference Can make the difference!

Reduces verbosity

Increases maintainability

Increases modularity

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 need only a very small change. In such cases, type inference increases modularity.

Type inference

Can make the difference!

Reduces verbosity

Increases maintainability

Increases modularity

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 need only a very small change. In such cases, type inference increases modularity.

May increase reusability

Inferring principal types lead to principal signatures, i.e. which may be more general than the signature the user had in mind (this is perhaps more true when programming in the small than when programming in the large).

To remember

The basic ingredients

- Path in types.
- Type definitions in signatures (module types)
- Equivalence to propagate type definitions through types
- Subtyping to forget and reorder components, allow abstraction.
- Strengthening to propagate sharing of abstract types
- Sealing to enforce abstraction, *i.e.* break sharing of abstract types

Type inference

- Application and projection restricted to paths
  (all types can be named)
- Use subsumption instead of subtyping (*avoids the avoidance problem*)
- Restrict to closed signatures (no free type variables in signatures)
- *This ensures principal signatures, but*
  subsumption and strengthening have an algorithmic flavor.

Signature definitions

A module may also contain (concrete) signature definitions:

$$d ::= \ldots \mid \ell = S \qquad\qquad s ::= \ldots \mid \ell = S$$

Typing rules:

$$\frac{\Gamma \vdash S}{\Gamma \vdash (\varphi.\ell = S) : (\varphi.\ell = S)} \qquad\qquad \frac{\pi = S \in \Gamma}{\Gamma \vdash \pi \approx S}$$

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 several times.

Signature definitions — The with notation

The construction changes some type definition of a signature:

$$S \quad ::= \quad \ldots \mid S \; \textsf{with} \; \bar{\ell} = \tau$$

Informally, this refines the type definition (at path) $\bar{\ell}$ in S, that must exist and be compatible with τ, to make it equal to τ.
The with notation is mostly used when S is a path $\pi.\mathsf{S}$.

It can always be eliminated by inlining S and replacing the component $\bar{\ell}$ by τ. For example, the last line of

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

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 increase expressiveness.
However, it avoids dupplicating code, hence it increases maintainability.

Signature definitions — The with notation

It may be formalized using the equivalence relation:

$$\frac{\Gamma \vdash S \approx (\varphi)\{\bar{d}_1; \ell : \kappa; \bar{d}_2\} \qquad \Gamma \vdash \tau : \kappa \qquad \Gamma \vdash (\varphi)\{\bar{d}_1; \ell : \kappa = \tau; \bar{d}_2\} <: S}{\Gamma \vdash (S \textbf{ with } \ell = \tau) \approx (\varphi)\{\bar{d}_1; \ell : \kappa = \tau; \bar{d}_2\}}$$

$$\frac{\begin{array}{c}\Gamma \vdash S \approx (\varphi)\{\bar{d}_1; \ell : \kappa = \tau'; \bar{d}_2\} \\ \Gamma \vdash \tau : \kappa \qquad \Gamma \vdash (\varphi)\{\bar{d}_1; \ell : \kappa = \tau; \bar{d}_2\} <: S\end{array}}{\Gamma \vdash (S \textbf{ with } \ell = \tau) \approx (\varphi)\{\bar{d}_1; \ell : \kappa = \tau; \bar{d}_2\}}$$

The with notation may also operate in submodules:

$$\frac{\Gamma \vdash S \approx (\varphi)\{\bar{d}_1; \ell_1 = S_1; \bar{d}_2\}}{\Gamma \vdash (S \textbf{ with } \ell_1.\bar{\ell} = \tau) \approx (\varphi)\{\bar{d}_1; \ell_1 = (S_1 \textbf{ with } \bar{\ell} = \tau); \bar{d}_2\}}$$

Sharing in signatures of functor arguments

With equality constraints

Consider the example:

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

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 of functor arguments

By parametrization

This is the standard way of doing abstraction for base language values and types. In modules, we could as well abstract the yet unknown part of types and provide the exact type later when applying the functor:

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

(We used type abstraction $\Lambda(a)\, S$ and type application $S(\tau)$ in signatures, which can be encoded.)

Sharing in signatures of functor arguments

In principle

All sharing in signatures of functor arguments could be by parametrization.

In practice

Unfortunately, sharing by parametrization *does not scale up very well* to large programs, as the number of type parameters quickly blows up.

It also forces programming in a more functorial manner, often using functors and higher-order functors, *were otherwise structures and* first-order functors would suffice.

See also this *exercise*.

Hence, sharing by *equality constraints* is the common practice.

Applicative functors What are they?

Generative/Applicative functors

A functor is *generative* if two applications of the functor to the same argument creates two modules with incompatible signatures.
It is *applicative* if two applications of the functor to the same argument always create two modules with compatible signatures.

Generative functors

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

The following functor alway returns a new incompatible version of integers by sealing its argument:

$$\lambda(X : S_{\mathbb{N}})\ (X : S_{\mathbb{N}})$$

It is generative.

Applicative functors What are they?

Applicative functors

Applicative functors are sometimes also desired.

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 could produce compatible maps that can be merged together.

For instance, the following functor renames the labels of its argument

$$Copy \triangleq \lambda(X : S_{\mathbb{N}})\ \{\mathbf{u} = X.\mathbf{t}; \mathbf{n} = X.\mathbf{m}\}$$

but shares the types of the result with those of the argument. Hence, two applications of the functor to the same argument have the same, compatible signature type.

Applicative functors Higher-order functors

Problem

Assume:

$$\begin{aligned} S &\stackrel{\triangle}{=} \{\mathsf{t} : *\} \\ X_{\mathbb{N}} &\stackrel{\triangle}{=} \{\mathsf{t} = int\} \\ Id &\stackrel{\triangle}{=} \lambda(X : S)\, X : (X : S) \longrightarrow (S \textbf{ with } \mathsf{t} = X.\mathsf{t}) \end{aligned}$$

What is the best type of

$$(\varphi) \left\{ \begin{array}{l} \mathsf{Apply} = \lambda(F : (X : S) \longrightarrow S)\ \lambda(Y : S)\ F(Y) : \\ \\ \mathsf{F} = \mathsf{Apply}(Id) : \\ \mathsf{N} = \varphi.\mathsf{F}(X_{\mathbb{N}}) : \end{array} \right.$$

Question Can we give type $\varphi.\mathsf{Apply}$ so that it is "transparent", *i.e.* $\varphi.\mathsf{Apply}(F)(Y)$ is typed as $F(Y)$ and $Z.\mathsf{N}.\mathsf{t}$ be compatible with *int*?

Applicative functors Higher-order functors

Problem

Assume:

$$\begin{aligned} S &\stackrel{\triangle}{=} \{\mathsf{t} : *\} \\ X_{\mathbb{N}} &\stackrel{\triangle}{=} \{\mathsf{t} = int\} \\ Id &\stackrel{\triangle}{=} \lambda(X : S)\, X : (X : S) \to (S \;\mathbf{with}\; \mathsf{t} = X.\mathsf{t}) \end{aligned}$$

What is the best type of

$$(\varphi) \left\{ \begin{array}{l} \mathsf{Apply} = \lambda(F : (X : S) \to S)\, \lambda(Y : S)\, F(Y) : \\ \qquad\qquad (F : (X : S) \to S) \to (Y : S) \to S \\ \mathsf{F} = \mathsf{Apply}(Id) : \\ \mathsf{N} = \varphi.\mathsf{F}(X_{\mathbb{N}}) : \end{array} \right.$$

Question Can we give type $\varphi.\mathsf{Apply}$ so that it is “transparent”, *i.e.* $\varphi.\mathsf{Apply}(F)(Y)$ is typed as $F(Y)$ and $Z.\mathsf{N}.\mathsf{t}$ be compatible with *int*?

Applicative functors Higher-order functors

Problem

Assume:

$$\begin{aligned} S &\triangleq \{\mathsf{t} : *\} \\ X_{\mathbb{N}} &\triangleq \{\mathsf{t} = int\} \\ Id &\triangleq \lambda(X : S)\, X : (X : S) \to (S \textbf{ with } \mathsf{t} = X.\mathsf{t}) \end{aligned}$$

What is the best type of

$$(\varphi) \left\{ \begin{array}{l} \mathsf{Apply} = \lambda(F : (X : S) \to S)\, \lambda(Y : S)\, F(Y) : \\ \qquad\qquad (F : (X : S) \to S) \to (Y : S) \to S \\ \mathsf{F} = \mathsf{Apply}(Id) : (Y : S) \to S \qquad \Leftarrow \text{Sharing of result type is lost} \\ \mathsf{N} = \varphi.\mathsf{F}(X_{\mathbb{N}}) : \end{array} \right.$$

Question Can we give type $\varphi.\mathsf{Apply}$ so that it is "transparent", *i.e.* $\varphi.\mathsf{Apply}(F)(Y)$ is typed as $F(Y)$ and $Z.\mathsf{N}.\mathsf{t}$ be compatible with *int*?

Applicative functors Higher-order functors

Problem

Assume:

$$\begin{aligned} S &\triangleq \{\mathsf{t} : *\} \\ X_{\mathbb{N}} &\triangleq \{\mathsf{t} = \mathit{int}\} \\ \mathit{Id} &\triangleq \lambda(X : S)\, X : (X : S) \rightarrow (S \;\mathbf{with}\; \mathsf{t} = X.\mathsf{t}) \end{aligned}$$

What is the best type of

$$(\varphi) \left\{ \begin{array}{l} \mathsf{Apply} = \lambda(F : (X : S) \rightarrow S)\, \lambda(Y : S)\, F(Y) : \\ \qquad\qquad (F : (X : S) \rightarrow S) \rightarrow (Y : S) \rightarrow S \\ \mathsf{F} = \mathsf{Apply}(\mathit{Id}) : (Y : S) \rightarrow S \quad \Longleftarrow \text{Sharing of result type is lost} \\ \mathsf{N} = \varphi.\mathsf{F}(X_{\mathbb{N}}) : S \quad \Longleftarrow \text{type } \mathsf{t} \text{ is abstract} \end{array} \right.$$

Question Can we give type $\varphi.\mathsf{Apply}$ so that it is "transparent", *i.e.* $\varphi.\mathsf{Apply}(F)(Y)$ is typed as $F(Y)$ and $Z.\mathsf{N}.\mathsf{t}$ be compatible with *int*?

Applicative functors

Solution: extended paths

To model applicative functors, we allow functor applications in paths.

$$\pi ::= \ldots | \pi(\pi)$$

Then, two applications of the same functor path to the same argument path are equal paths.

Strengthening strengthening

We also change strengthening for functor types:

$$(\varphi : S_1) \to S_2 / \pi \overset{\triangle}{=} (\varphi : S_1) \to S_2 / \pi(\varphi)$$

$$\textit{was} \quad (\varphi : S_1) \to S_2$$

Applicative functors

Example

Does this typecheck?

$$(\varphi)\left\{\begin{array}{l} \mathsf{Apply} = \lambda(F : (X : S) \to S)\ \lambda(Y : S)\ F(Y) \\ \\ \\ \mathsf{F} = \mathsf{Apply}(Id) \\ \mathsf{N} = \varphi.\mathsf{F}(X_{\mathbb{N}}) \end{array}\right.$$

Applicative functors Example

Does this typecheck?

—Yes! $F(Y)$ is a path and can be strengthenned

$$(\varphi)\begin{cases} \mathsf{Apply} = \lambda(F : (X : S) \to S)\ \lambda(Y : S)\ F(Y) : \\ \qquad (F : (X : S) \to S) \to (Y : S) \to (S \textbf{ with } \mathsf{t} = F(Y).\mathsf{t}) \\ \qquad (F(Y) : S, \text{ hence } F(Y) : S/F(Y) = (S \textbf{ with } \mathsf{t} = F(Y).\mathsf{t})) \\ \mathsf{F} = \mathsf{Apply}(Id)(Y : S) \to S \textbf{ with } \mathsf{t} = Id(Y).\mathsf{t} \\ \mathsf{N} = \varphi.\mathsf{F}(X_{\mathbb{N}}) : (S \textbf{ with } \mathsf{t} = Id(X_{\mathbb{N}}).\mathsf{t}) \end{cases}$$

Applicative functors — Example

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

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

Then:

- $\varphi.\mathsf{F}$ has type $(X : \{\}) \to \{\mathsf{t} : *; \mathsf{m} : \mathsf{t}\}$
  By strengthening, it also has type $(X : \{\}) \to \{\mathsf{t} : *; \mathsf{m} : \mathsf{t}\}/\varphi.\mathsf{F}$
  i.e. $(X : \{\}) \to \{\mathsf{t} = \varphi.\mathsf{F}(X).t; \mathsf{m} : \mathsf{t}\}$
- $\varphi.\mathsf{M}_{11}$ and $\varphi.\mathsf{M}_{12}$ are compatible, both of type $\varphi.\mathsf{F}(\varphi.\mathsf{V}_1).\mathsf{t}$,
- they are incompatible with $\varphi.\mathsf{M}_{21}$, of type $\varphi.\mathsf{F}(\varphi.\mathsf{V}_2).\mathsf{t}$.
- $\varphi.\mathsf{F}(\varphi.\mathsf{V}_2).\mathsf{t}$ and $\varphi.\mathsf{F}(\varphi.\mathsf{V}_3).\mathsf{t}$ are also incompatible even though V_3 is just a rebinding of V_2.

See Leroy (1995)

Applicative functors

Does it matter?

Should higher-order functors be applicative?

Applicative higher-order functors can type more programs. Moreover, the application can still be made generative by η-expansion and sealing or by rebinding the argument before the application.

Conversely, if functors are generative, two applications of the functor can never be made 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.

Applicative functors Does it matter?

Conclusions

It is not essential that module systems provide support for applicative higher-order functors, while they cannot avoid dealing with type generativity.

Applicative higher-order functors are more expressive than generative functors. They give higher-order functors a more transparent semantics.

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

Abstract signatures Language Extension

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:

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

Abstract signatures Example

The application functor *App* may be written:

$$Apply \stackrel{\triangle}{=} \lambda(Z : \{\mathsf{A}; \mathsf{B}\})\ \lambda(F : (X : Z.\mathsf{A}) \to Z.\mathsf{B})\ \lambda(Y : Z.\mathsf{A})\ F\ (Y)$$

Abstract signatures inscrease expressiveness—without them we can only write specific versions of the application functor.

We may then add signature abstraction and signature application as syntactic sugar:

$$\begin{aligned} \Lambda(\mathsf{A})\ \mathcal{M} &\stackrel{\triangle}{=} \lambda(Z_{\mathsf{A}} : \{\mathsf{A}\})\ \mathcal{M}[Z_{\mathsf{A}}.\mathsf{A}/\mathsf{A}] \\ \mathcal{M}[S] &\stackrel{\triangle}{=} \mathcal{M}(\{\mathsf{A} = S\}) \end{aligned}$$

Then

$$Apply \stackrel{\triangle}{=} \Lambda(\mathsf{A})\ \Lambda(\mathsf{B})\ \lambda(F : (X : \mathsf{A}) \to \mathsf{B})\ \lambda(Y : \mathsf{A})\ F\ (Y)$$

as in System F.

Abstract signatures Limitation

Higher-order functors with abstract signatures are not applicative

$$Apply : \Lambda(\mathsf{A})\ \Lambda(\mathsf{B})\ (F : (X : \mathsf{A}) \to \mathsf{B}) \to (Y : \mathsf{A}) \to \mathsf{B}$$

Strengthening has no effect on abstract signatures.

$$\ell/\pi \overset{\triangle}{=} \ell \qquad\qquad (\text{Compare with} \quad \ell : \kappa/\pi \overset{\triangle}{=} \ell : \kappa = \pi.\ell)$$

Hence, the apply functor cannot tell that the result type B is in fact the type of application F to X.

This should then be told in the signature of F by making B depend on $(X : \mathsf{A})$. Unfortunately, abstract signatures cannot be higher-order.

A specialization of $Apply\ \lambda(F : (X : S) \to S)\ \lambda(Y : S)\ F\ (Y)$ where A and B are some signature S_{t} containing some abstract type t can be typed more precisely if typed directly:

$$(F : (X : S) \to S) \to (Y : S) \to (S \ \mathbf{with}\ \mathsf{t} = F(Y).\mathsf{t})$$

Abstract signatures Limitation

This provides a hint why programming with modules in F^ω style does not scale up well:

- strengthening, which plays a key role in transparency works for concrete signatures but does not permit abstraction.
- value-dependent signatures, which also play a key role in transparency, can only be concrete.

Hence, to keep transparency, one cannot use abstract signatures and must instead use long concrete signatures which may be quite verbose.

Abstract signatures Exercises

Exercise

Write Apply in OCaml. Write the identity functor Id in a similar style. Verify that Apply can be applied to Id specialized at any signature S. Answer

Bootstrap the example, by writing a second version of Apply that uses Apply internally instead of the primitive functor application and recheck the application of Apply to Id. Answer □

Exercise

Specialize Apply to the case where Z.A and Z.B are a same signature containing some abstract type t. *Comment.* □

Type soundness Subject reduction does not hold!

The syntax is not even stable by reduction!

$$(\lambda(m : (\varphi)\{\mathbf{t} : *; \mathbf{m} : \varphi.\mathbf{t}\})\ m.\ell_x)\ (\{\mathbf{t} = int; \mathbf{m} = 1\})$$

reduces to the ill-formed projection

$$\{\mathbf{t} = int; \mathbf{m} = 1\}.\mathbf{m}$$

since only path can be projected! This syntactic limitation (and the similar one for applications) is essential to trace type identities. Paths bound to modules are not substitutable.

Otherwise, for instance, if $(\mathcal{M} : S)$ were

$$(\{\mathbf{t} = int; \mathbf{m} = 1; \mathbf{n} = succ\} : (\varphi)\{\mathbf{t} : *; \mathbf{m} : \varphi.t; \mathbf{n} : \varphi.t \to \varphi.t\})$$

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

Type soundness — By translation to System F^{ω}

Modules are second-class citizen

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.

Revealing abstract types preserves well-typedness

Because modules are second class, an implementation of a signature cannot be choseen dynamically: any abstract types has a unique statically known concrete type associated with, which may be safely revealed, or it appears in the argument of a functor, which is polymophic in the corresponding type.

(Of course, it may expose type invariants, at the programmer's risk).

Type soundness By translation to System F^ω

Translation

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 *by construction* that values with compatible representation but incompatible semantics are never merged, but this invariant will be preserved after translation.

Type soundness By translation to System F^ω

Proceed as follows (assuming no abstract signatures)

1) Transform any sealing that turns a type definition (that does not contain an abstract 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 $\lambda(X : \{\bar{s}_1; \mathsf{t}; \bar{s}_2\})\ \mathcal{M}$ into $\lambda(a)\ \lambda(X : \{\bar{s}_1; \mathsf{t} = a; \bar{s}_2\})\ \mathcal{M}$ and functor applications correspondingly.
3) Type definitions may be inlined and so 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 a 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

This approach is technically involved.

For instance, see Dreyer et al. (2003)

An abstraction-preserving translation to F^ω

In the translation to F^ω, existential types may be used to preserve abstraction. This also preserves well-typedness with first-class modules.

See Rossberg et al. (2010)

Problems

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

To remember

The path-based approach works fine for the core language, but shows its limitations for more advanced features:

- The ad-hoc algorithmic aspects of strengthening is emphasized by applicative functors.
- Abstract signatures are of very limited uses becomes they cannot be strengthened.
- No direct accound of type soundness.

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 problems

- 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 problems

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

Recursive modules

Typechecking

Syntax

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

Notice that all fields of a recursive module must be submodules.
All fields may now refer to one another.

Typechecking signatures

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

Some extended occur check $\mathit{acyclic}(\mu(\psi)\{\overline{\mathsf{M} : S}\})$ 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

Signatures are provided and not inferred

Naive typing rule

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

Limitation

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

This is known as the double vision problem.

Double vision problem Example

Problem

Let S_1 be $(\varphi)\{\mathsf{t} : *; \mathsf{m} : \varphi.\mathsf{t}\}$ and let $\mathcal{M}_1$ be $(\{\mathsf{t} = int; \mathsf{m} = 1\} : S_1)$.

Let S_ψ be $\{\mathsf{t} : *; \mathsf{m} : \psi.\mathsf{M}.\mathsf{t}\}$.

Then, $\mu(\psi)\{\mathsf{M} : S_\psi = \mathcal{M}_1\}$ is ill-typed because under $\psi.\mathsf{M} : S_\psi$, the signature S_1 of $\mathcal{M}_1$ is not a subtype of S_ψ.

Observe

The strengthened signature $S_1/\psi.\mathsf{M}$, which is equal to $(\varphi)\{\mathsf{t} = \psi.\mathsf{M}.\mathsf{t}; \mathsf{m} : \varphi.\mathsf{t}\}$, is a subtype of S_ψ.

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 $\varphi.\mathsf{t} \approx \psi.M.\mathsf{t}$.

Double vision problem First attempt

Combine strengthening and subsumption

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

That is, S is simultaneously a strengthening (stronger than) and a subtyping of (weaker than) the inferred signature S' for $\mathcal{M}$.

Unfortunately...

This form of strengthening breaks the property that strengthening can always be canceled by subtyping, because S', the signature of $\mathcal{M}$ is strengthened by $\psi.\mathsf{M}$, which is only assigned a supertype of S'.

This is a problem for type inference, since applying strengthening immediately may enable new derivations but also disable valid ones. This solution is too weak.

Double vision problem

Example

The following definition is rejected

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

The problem is the following:

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

Solution

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

Notice: while strengthening must chose one view (internal or external), this equation keeps the two views and makes them locally coincide.

Double vision problem — Solution

Informally

Let Γ' be $\Gamma, \overline{\psi.\mathsf{M} : S}$. Remind that premisses are $\Gamma' \vdash \mathcal{M} : S$ for all recursive definitions $\mathcal{M} : S$.

When $\mathcal{M} : S$ is a structure definition $(\varphi)\{\bar{d}\} : (\varphi)\{\bar{s}\}$, this would amount to typechecking $\Gamma' \vdash \varphi.\bar{d} : \varphi.\bar{s}$.

Instead, typecheck $\Gamma' \vdash \varphi.\bar{d} : \varphi.\bar{s}/\psi.\mathsf{M}$ which is decomposed as follows

$$\frac{\Gamma' \vdash d : s \qquad \Gamma', d : s/\psi.\mathsf{M} \vdash (\bar{d}) : (\bar{s})/\psi.\mathsf{M}}{\Gamma' \vdash (d, \bar{d}) : (s; \bar{s})/\psi.\mathsf{M}}$$

where

- $d : s/\psi.\mathsf{M}$ is $\varphi.\mathsf{t} : \kappa = \psi.\mathsf{M}.\mathsf{t} = \tau$ (resp. $\varphi.\mathsf{t} : \kappa = \psi.\mathsf{M}.\mathsf{t}$) when d is a type declaration $\varphi.\mathsf{t} : \kappa = \tau$ (resp. $\varphi.\mathsf{t} : \kappa$) and just $d : s$ otherwise;
- rules $\Gamma, \pi : \kappa = \pi' = \tau, \Gamma' \vdash \pi \approx \pi' : \kappa$ and $\Gamma, \pi = \pi' = \tau, \Gamma' \vdash \pi \approx \tau : \kappa$ are added to exploit these double-vision asumptions.

Double vision problem Solution

Comment

This is becoming quite ad hoc and algorithmic, reaching the limits of the path-based approach.

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

Recursive definitions Problem

Examples of 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 and followed by infinitely many 0.

$$\texttt{type } \mathit{loop} = \{ \text{ left } : \mathit{int};\ \text{right} : \mathit{loop} \ \}$$
$$\texttt{let rec } \mathit{ok} = \{ \text{ left} = 1;\ \text{right} = \{ \text{ left} = 2;\ \text{right} = \mathit{ok}\}\}$$

During recursion, recursive values can be used to build other values, but should never be accessed.

Ill-formed recursive definitions *(not specific to modules)*

$$\texttt{let rec } \mathit{ko} = \{\text{left} = 1;\ \text{right} = \{ \text{ left} = 1 + \mathit{ko}.\text{left};\ \text{right} = \mathit{ko}\}\}$$

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 record Ω is allocated with all fields undefined, initially.
- Accesses to undefined fields of Ω are detected and raise an error
- The 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 are 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 all recursive 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:

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

- $\psi..Even$ is a mixin with an import *odd* and an export *even*
- $\psi.Odd$ is a mixin with an import *even* and an export *odd*
- $Even \otimes Odd$ is the mixin composition of $\psi..Even$ and $\psi..Odd$
- $\{Even \otimes Odd\}$ 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 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

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

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

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

(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 which 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 $F_{<:}$

- 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

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

Typing rules

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

$$\frac{\Gamma \vdash \tau \qquad \Gamma, x:\tau_0 \vdash M:\tau}{\Gamma \vdash \lambda(x:\tau_0)\, M : \tau_0 \to \tau} \qquad \frac{\Gamma \vdash M_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash M_1(M_2) : \tau_1}$$

Plus *existential types*

Plus records

Splitting unpack

Into three pieces

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

$$\triangleq$$

$$\underbrace{\nu a.}_{\text{Limits the scope of } a}\ \text{let } \underbrace{x}_{\text{Binds } \mathcal{M} \text{ to } x \text{ in } \mathcal{M}'} = \underbrace{\text{open } \langle a \rangle\, \mathcal{M}}_{\text{Uses } a \text{ for the abstract type of } \mathcal{M}} \text{ in } \mathcal{M}'$$

Splitting unpack Into three pieces

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

Splitting unpack

Gain in expressiveness

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

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

Splitting unpack Gain in expressiveness

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

a need not be hidden immediately.

Splitting unpack

Gain in expressiveness

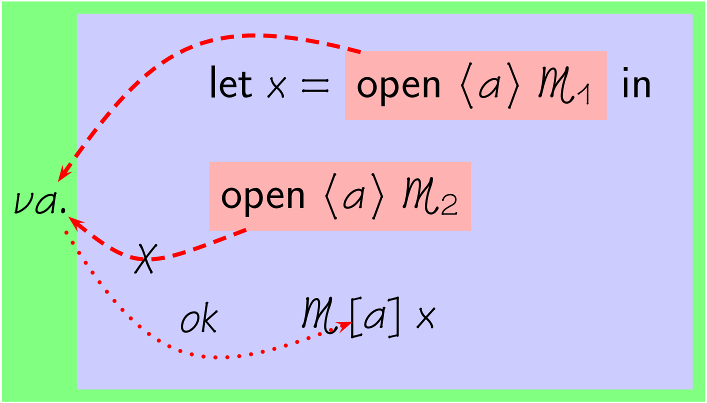
$$C\left\{ \text{let } x = \quad \text{open } \langle a \rangle\ \mathcal{M} \quad \text{in } \mathcal{M}' \right\}$$

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

Splitting unpack

Typechecking

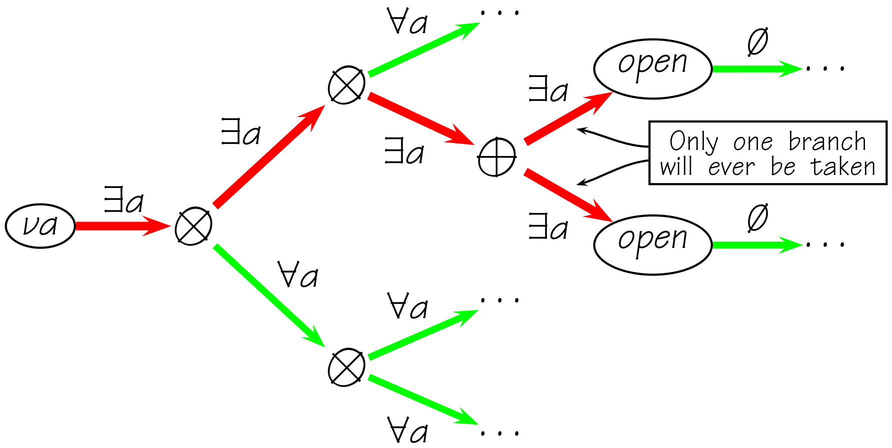
Must forbid incorrect programs such as


$\nu a.$
let x = open $\langle a \rangle$ M_1 in
open $\langle a \rangle$ M_2
ok
$M [a]$ x

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

Splitting unpack

Evaluation contexts:



Splitting unpack

Typechecking

Nu

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

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

Splitting unpack

Typechecking

Open

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

Nu

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

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

Splitting unpack

Typechecking

Open

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

Let

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

Nu

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

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

Splitting unpack

Typechecking

Open

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

$$\frac{\vdots}{\Gamma, \forall a \vdash \mathcal{M}'\,[a] : \tau'}$$

Let

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

Nu

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

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

Splitting unpack — Zipping of typing assumptions

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

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

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

but $\exists a \curlyvee \exists a$ is ill-formed

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

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

$$\triangleq$$

$$\exists (a = \tau)\ (\mathcal{M} : \tau')$$

makes a abstract with witness τ

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

Splitting pack

In three pieces

$$\textsf{pack}\ \tau, \mathcal{M}\ \textsf{as}\ \exists a.\tau'$$

$$\triangleq$$

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

closes the abstract type β

converts the type of $\mathcal{M}$

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

Splitting pack — Gain in expressiveness

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

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

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

.

Splitting pack

Gain in expressiveness

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

A module with an open abstract type β.

Splitting pack Gain in expressiveness

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

A submodule with an open abstract type β.

Splitting pack

Typechecking

Exists

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

Splitting pack

Typechecking

Exists

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

Open

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

Splitting pack

Typechecking

Sigma

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

Exists

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

Open

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

Splitting pack

Coerce

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

uses

Sigma

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

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

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

Examples

Abstract type

In ML:

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

In Fzip:

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

Examples

Abstract type

In ML:

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

In Fzip:

$$\textbf{let } x = \exists(a = \mathit{int}) \left(\left\{\begin{array}{l} z = 0\,; \\ s = \lambda(x:\mathit{int})x+1 \end{array}\right\} : \left\{\begin{array}{l} z : a\,; \\ s : a \to a \end{array}\right\}\right) \textbf{ in}$$
$$\textbf{open } \langle\beta\rangle\ x$$

Examples

Type generativity

In ML:

$$M_1 = \left(\left\{ \begin{array}{l} t = int \\ z = 0 \\ s = \lambda(x : int) x{+}1 \end{array} \right\} : \left\{ \begin{array}{l} t : * \\ z : t \\ s : t \to t \end{array} \right\} \right)$$

$$M_2 = \left(\quad M \quad : \left\{ \begin{array}{l} t : * \\ z : t \\ s : t \to t \end{array} \right\} \right)$$

In Fzip:

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

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

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

$\ldots$

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

$$\begin{array}{l} \text{let } \mathit{MakeSet} = \\ \qquad \Lambda \alpha.\, \lambda(\mathit{cmp} : \alpha \to \alpha \to \mathit{bool})\ \exists(\beta = \mathit{set}(\alpha))\,(\ldots : \mathit{set}(\beta))\ \textbf{in} \\ \text{let } s_1 = \textbf{open}\ \langle \beta_1 \rangle\ \mathit{MakeSet}\,[\mathit{int}]\,(<) \text{ in} \\ \text{let } s_2 = \textbf{open}\ \langle \beta_2 \rangle\ \mathit{MakeSet}\,[\beta_1]\,(s_1.\mathit{cmp}) \text{ in} \\ \ldots \end{array}$$

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 block).
- This safely enlarges the scope of identities,
- moving the Σ's outside of redexes, and
- Allowing further reduction to proceed.

Reduction — Example

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

$$\Big\downarrow \text{(extrude)}$$

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

$$\Big\downarrow \text{(reduce)}$$

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

$$\Big\downarrow \text{(reduce)}$$

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

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

$$\begin{array}{rcllllll} v & ::= & u & \mid & (u : \tau) \\ u & ::= & x & \mid & \lambda(x:\tau)\mathcal{M} & \mid & \Lambda a.\,\mathcal{M} & \mid & \exists\beta.\,\Sigma\,\langle\beta\rangle\,(a=\tau)\,v \end{array}$$

Results

$$\begin{array}{rclll} w & ::= & v & \mid & \Sigma\,\langle\beta\rangle\,(a=\tau)\,w \end{array}$$

Note

- Abstractions λ's and Λ's are always values because they are pure, *i.e.* typechecked in a context Γ without $\exists a$'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,

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

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

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

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

+ 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

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

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

Keep this information in the context and use it whenever needed

Sigma

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

Sim

$$\frac{\Gamma \vdash \mathcal{M} : \tau' \qquad \Gamma \vdash \tau \triangleleft \tau'}{\Gamma \vdash \mathcal{M} : \tau}$$

Rules for $\Gamma \vdash \cdot \triangleleft \cdot$ 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:

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

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

reduces to: $\textbf{open } \langle \beta \rangle\, \exists(a = \beta \to \beta)\, \mathcal{M}$
which leads to the recursive equation $\beta = \beta \to \beta$.

External recursion, through open witness definitions:

$$\begin{aligned} \{ \ell_1 &= \Sigma\, \langle \beta_1 \rangle\, (a_1 = \beta_2 \to \beta_2)\, \mathcal{M}_1\ ; \\ \ell_2 &= \Sigma\, \langle \beta_2 \rangle\, (a_2 = \beta_1 \to \beta_1)\, \mathcal{M}_2\ \} \end{aligned}$$

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

Why may we wish to reject these examples?

- Without recursive types, their evaluation would block.
- They cannot be translated to System F.
- Implicit recursive types may hide users type errors.

Avoiding recursive types

Why?

Origin of the problem

Sigma

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

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

Solutions

1. Remove $\boxed{\forall \beta}$ 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 $\forall a$ or $\exists a$.

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 $\exists a$, $\forall a$, $\forall(a = \tau)$
- 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

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

where $\{\exists\bar\beta\} = dom\,\mathcal{E}_1 \,\Delta\, dom\,\mathcal{E}_2$

(weakening to remove unnecessary dependencies)

Avoiding recursive types

How? ◀ ▶

Sigma

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

In particular,

- Free variables of the witness type τ' are in $\mathcal{D}'$ (by well-formedness).
- Bindings that $\mathcal{D}'$ depends on are also in $\mathcal{D}'$ (by transitivity of $\prec$).
- Bindings of $\mathcal{D}'$ (the witness type τ' depends on) must be in $\mathcal{D}$ (bindings β depends on).

Avoiding recursive types

How? ◀ ▶

Sigma

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

The prevents typechecking:

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

But allows typechecking:

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

Avoiding recursive types

How? ◀ ▶

Open

$$\frac{\Gamma \vdash \mathcal{M} : \exists \beta . \tau \qquad \{(\exists a) \in \Gamma\} \cup \{(\forall a) \in \Gamma\} \subseteq \mathcal{D}}{\Gamma, (\exists \beta \prec \mathcal{D}) \vdash \textbf{open } \langle \beta \rangle\, \mathcal{M} : \tau}$$

Let

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

Open:

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

Let:

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

Avoiding recursive types

How? ◀ ▶

Open

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

Let

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

Prevents typechecking:

$\mathbf{let}\ x = \exists(\alpha = \beta \to \beta)\, \mathcal{M}\ \mathbf{in}$ *implies* $x \prec \beta$, *since* $\exists \beta \in \Gamma_2 \wedge \forall \beta \in \Gamma_1$

$\mathbf{open}\ \langle \beta \rangle\ x$ *requires* $x \not\prec \beta$ *since* $x \in \mathit{dom}\, \Gamma$

Avoiding recursive types — Restriction to $\mathsf{F}^{\curlyvee-}$ ◀

Why a further restriction?

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

Avoiding recursive types — Restriction to $\mathsf{F}^{\curlyvee^{\downarrow}-}$

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

🟩 $\forall a$	$\curlyvee^{\nearrow}$	🟥 $\exists a$	$=$	🟥 $\exists a$
🟥 $\exists a$	$\curlyvee^{\nearrow}$	$\cdot$	$=$	🟥 $\exists a$
🟩 $\forall a$	$\curlyvee^{\nearrow}$	🟩 $\forall a$	$=$	🟩 $\forall a$

$\cdot$	$\curlyvee^{\nwarrow}$	🟥 $\exists a$	$=$	🟥 $\exists a$
🟥 $\exists a$	$\curlyvee^{\nwarrow}$	$\cdot$	$=$	🟥 $\exists a$
🟩 $\forall a$	$\curlyvee^{\nwarrow}$	🟩 $\forall a$	$=$	🟩 $\forall a$

- 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 this zipping).

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 ($\mathsf{F}^{\forall-}$) enforces abstract types to follow the scope of type variables:

- System F is still a subset of $\mathsf{F}^{\forall-}$ (using the syntactic sugar).
- Pure expressions of $\mathsf{F}^{\forall-}$ can be translated to System F such that
  - Semantics, type abstraction, and typings are preserved
  - 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 $\Sigma\,\langle\beta\rangle\,(a = \tau')\,(\mathcal{M} : \tau)$ and $\exists a.\,(\mathcal{M} : \tau)$.
2. Replace existential quantifiers by uses of **pack**, according to the rule: $\exists a.\,(\mathcal{M} : \tau) \rightharpoonup \nu a.\,\mathbf{let}\ x = \mathcal{M}\ \mathbf{in}\ \mathbf{pack}\ a, x\ \mathbf{as}\ \exists a.\,\tau$
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:
   $\nu a.\,\mathbf{let}\ x = \mathbf{open}\ \langle a\rangle\ \mathcal{M}\ \mathbf{in}\ \mathcal{M}' \rightharpoonup \mathbf{unpack}\ \mathcal{M}\ \mathbf{as}\ a, x\ \mathbf{in}\ \mathcal{M}'$
   $\nu a.\,\mathbf{let}\ x = \Sigma\,\langle a\rangle\,(a = \tau_0)\,(\mathcal{M} : \tau)\ \mathbf{in}\ \mathcal{M}'$
   $\rightharpoonup \mathbf{unpack}\ (\mathbf{pack}\ \tau_0, \mathcal{M}[a \leftarrow \tau_0]\ \mathbf{as}\ \exists a.\,\tau)\ \mathbf{as}\ a, x\ \mathbf{in}\ \mathcal{M}'$
5. Finally, remove all remaining coercions.

Expressiveness Translation to System F

Extrusion of Open's and Σs

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

Intrusion of νs

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

Context with open existentials

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

Expressiveness

Relation to System F

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

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

Expressiveness

Adding recursion

Type level recursion

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

  Then, recursive types always originate from an $\approx$-form of Σ's (and not accidentally from the other form.

Term level recursion

- Allow restricted fixpoints 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

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

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!

Extension to higher-order kinds is needed but not a problem

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 41 Both signatures are well-formed. It remains to show that $\Gamma \vdash (\varphi.\mathsf{t} : *) <: (\varphi.\mathsf{t} : * = \varphi.\mathsf{u})$ (**1**)
$\Gamma \vdash (\varphi.\mathsf{u} = \varphi.\mathsf{t}) <: (\varphi.\mathsf{u} : *)$ (**2**). where Γ is $\varphi.\mathsf{t} : *, \varphi.\mathsf{u} = \mathsf{t}$. (2) is by subtyping of type definitions. (1) is by concretization of abstract types: the premisse $\Gamma \vdash \varphi.\mathsf{t} \approx \varphi.\mathsf{u}$ holds by hypothesis and commutativity of $\approx$.

Answers to exercises II

Exercise 2, page 51 By induction on the definition of $\Gamma \vdash \pi : S$, 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 $\Gamma \vdash S/p <: S$ by induction on the definition of 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 $\Gamma \vdash \pi : \kappa = \pi.\ell <: \pi : \kappa$.)

Answers to exercises III

Exercise 2 (continued) By induction on the definition of $\Gamma \vdash \pi : S$, 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 $\Gamma \vdash S/p <: S$ by induction on the definition of 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 $\Gamma \vdash \pi : \kappa = \pi.\ell <: \pi : \kappa$.)
Exercise 4, page 66

$$\{\mathsf{u} = \lambda(a)\ X.\mathsf{t}; \mathsf{s} = X.\mathsf{t}\}$$

Answers to exercises IV

Exercise 4 (continued) The signature $\{\mathsf{u} = \lambda(a)\, X.\mathsf{t}; \mathsf{s} = X.\mathsf{t}\}$ of $X.\mathsf{M}$ should avoid X. Well-formed sub-signatures are either $\{\mathsf{u} : * \longrightarrow *; \mathsf{s} = *\}$ or $(\varphi)\{\mathsf{u} : * \longrightarrow *; \mathsf{s} = \varphi.\mathsf{u}(\tau)\}$ for any type τ. However, each of the latter forms are incomparable and the former is a strict subtype of any of the latter forms.
So there would be no principal well-formed signature for this expression.

(Notice, that the signature $(\varphi)\{\mathsf{u} : \lambda(a)\, \varphi.\mathsf{s}; \mathsf{s} = *\}$ is ill-formed because field $\varphi.\mathsf{s}$ would be used before being defined.)

Answers to exercises V

Exercise 5, page 94

```
module Apply =
  functor(Z: sig module type A module type B end) →
    functor(F: functor (X : Z.A) → Z.B) → functor(Y:Z.A)→ F(Y)
module Id =
  functor(Z: sig module type A end) → functor(X:Z.A)→ X
module WithApply
    (Apply : functor(Z: sig module type A module type B end) →
      functor(F: functor (X : Z.A) → Z.B) → functor(Y:Z.A)→ Z.B) =
  functor (Z : sig module type A end) →
    Apply
      (struct module type A = Z.A module type B = Z.A end)
      (Id (struct module type A = Z.A end))
module T1 (Z : sig module type A end) = WithApply (Apply) (Z)
```

Answers to exercises VI

Exercise 5 (continued)

```
module Apply2 =
  functor(Z: sig module type A module type B end) →
    functor(F: functor (X : Z.A) → Z.B) → functor(Y:Z.A)→
      Apply (struct module type A = Z.A module type B = Z.B end)
        (F)(Y)

module T2 (Z : sig module type A end) = WithApply (Apply) (Z)
```

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.

▷ Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing Modules. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI2010)*, January 2010.