

Programmer avec des modules de première classe dans un langage noyau pourvu de sous-typage, sortes singletons et types existentiels ouverts

Soutenance de thèse

Benoît Montagu
sous la direction de Didier Rémy

École Polytechnique — INRIA Paris-Rocquencourt, projet Gallium

Mercredi 15 décembre 2010

Modularity in software development

Today's software: big and complex projects

Project	Lines of code
Unison (2.32.52)	27,000
Coq (8.3)	193,000
Objective Caml (3.12)	240,000
Emacs (23.2)	1,200,000
Mozilla Firefox (3.6)	3,101,000
GCC (4.4.5)	3,800,000
LibreOffice.org (3.2.99.3)	5,640,000
Linux Kernel (2.6.36.1)	8,900,000

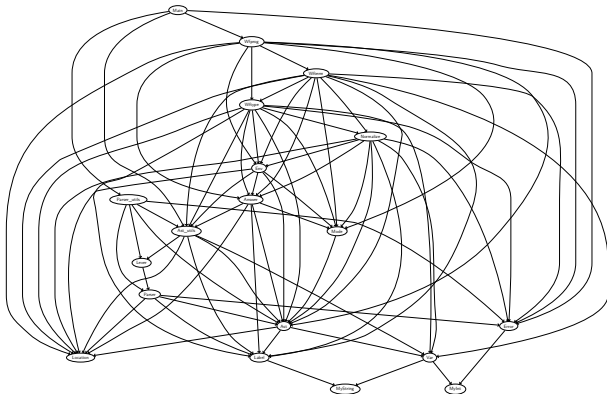
From SLOCCount, rounded.

► Big projects

Modularity in software development

Today's software: big and complex projects

- ▶ Big projects
- ▶ Complex projects



Dependency graph of the Fzip typechecker.

Modularity in software development

Today's software: big and complex projects

- ▶ Big projects
- ▶ Complex projects

Software developments needs:

- ▶ isolation of independent components;
- ▶ reusability of generic components.

Dependency graph of the Fzip typechecker.

ML modules

Key features

The ML module language

- ▶ Similar to *compilation units*, but more powerful
- ▶ A language on top of the ML language
- ▶ Can be adapted to any other language

ML modules

Key features

The ML module language

- ▶ Similar to *compilation units*, but more powerful
- ▶ A language on top of the ML language
- ▶ Can be adapted to any other language

Ingredients

- ▶ Functors: parametrized modules

Features

Generic libraries



ML modules

Key features

The ML module language

- ▶ Similar to *compilation units*, but more powerful
- ▶ A language on top of the ML language
- ▶ Can be adapted to any other language

Ingredients

- ▶ Functors: parametrized modules
- ▶ Abstract types

Features

- ▶ Generic libraries
- ▶ Isolation of components

ML modules

Key features

The ML module language

- ▶ Similar to *compilation units*, but more powerful
- ▶ A language on top of the ML language
- ▶ Can be adapted to any other language

Ingredients

- ▶ Functors: parametrized modules
- ▶ Abstract types
- ▶ Hierarchical composition

Features

- ▶ Generic libraries
- ▶ Isolation of components
- ▶ Namespace management

ML modules

Key features

The ML module language

- ▶ Similar to *compilation units*, but more powerful
- ▶ A language on top of the ML language
- ▶ Can be adapted to any other language

Ingredients

- ▶ Functors: parametrized modules
- ▶ Abstract types
- ▶ Hierarchical composition
- ▶ Extensible structures

Features

- ▶ Generic libraries
- ▶ Isolation of components
- ▶ Namespace management
- ▶ Ease of use

ML modules

Key features

The ML module language

- ▶ Similar to *compilation units*, but more powerful
- ▶ A language on top of the ML language
- ▶ Can be adapted to any other language

Ingredients

- ▶ Functors: parametrized modules
- ▶ Abstract types
- ▶ Hierarchical composition
- ▶ Extensible structures
- ▶ Type definitions

Features

- ▶ Generic libraries
- ▶ Isolation of components
- ▶ Namespace management
- ▶ Ease of use
- ▶ Concision

Problem...

Abstract types Phase distinction
Syntactic signatures Transparent functors
Avoidance problem Manifest types
Projectibility Higher-order functors
Generativity Elaboration semantics
Applicative functors Paths

Abstract types Phase distinction

Syntactic signatures Transparent functors

Avoidance of abstract types

Projection functors

Generativity Elaboration semantics

Applicative functors Paths

- ▶ Many concepts
- ▶ Large literature
- ▶ Complex and subtle systems

Why is it so complex?

A simpler approach

Polymorphism is enough

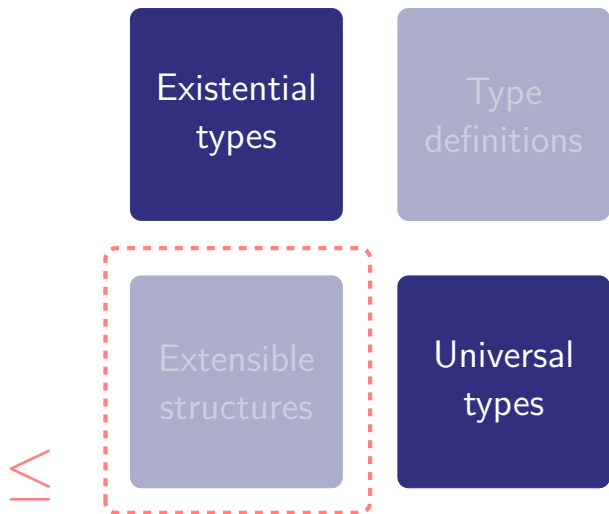
- ▶ [MP88]: Abstract types have existential types
- ▶ [Rus03]: Types for modules
Giving the types of F^ω to modules
- ▶ [RRD10]: F-ing modules
A simple translation from modules to F^ω , mechanically verified

In practice

- ▶ F^ω is too low-level
- ▶ F^ω constrains the structure of programs too much
- ▶ F^ω is too verbose
- ▶ F^ω cannot compete with modules

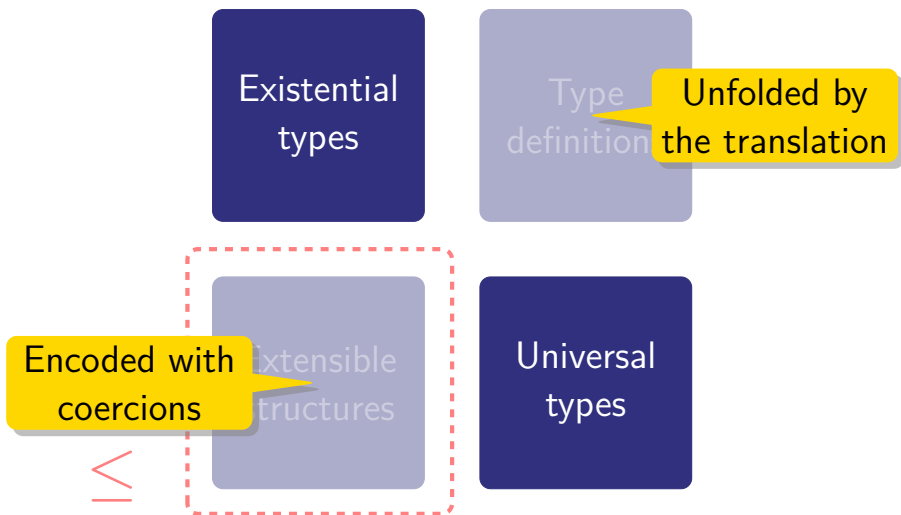
We propose a **better** F^ω . What should we add or change?

The general picture



Russo's interpretation

The general picture



Russo's interpretation

The general picture

Open
existential
types

Singleton
kinds

Extensible
records

Universal
types



In this work

The general picture

More flexible
structure for
programs

Open
existential
types

Singleton
kinds

Definitions are
kept folded

Subtyping is
implicit

Extensible
records

Universal
types



In this work

Contributions

Open existential types: F^\forall

- ▶ Definitions and proof of safety
- ▶ Mechanized proof on a large subset
- ▶ Relation with System F

Contributions

Open existential types: F^\forall

- ▶ Definitions and proof of safety
- ▶ Mechanized proof on a large subset
- ▶ Relation with System F

Singleton kinds

- ▶ Small-step semantics
- ▶ Explicit constructors for η -expansion
- ▶ A simple and adequate characterization of type equivalence

Contributions

Open existential types: F^\forall

- ▶ Definitions and proof of safety
- ▶ Mechanized proof on a large subset
- ▶ Relation with System F

Singleton kinds

- ▶ Small-step semantics
- ▶ Explicit constructors for η -expansion
- ▶ A simple and adequate characterization of type equivalence

A kernel language: $F_{S \leq}^{\forall \omega}$

- ▶ Definition and implementation
- ▶ Encoding from a module system into this kernel

Outline

- ① Open existential types
- ② Singleton kinds
- ③ A kernel language
- ④ Conclusion

Outline

① Open existential types

② Singleton kinds

③ A kernel language

④ Conclusion

Existential types and abstract types

```
module type INT = sig
  type t
  val zero : t
  val succ : t → t
  val mult : t → t → t
end
(* definition *)
module Int : INT = struct
  type t = int
  let zero = 0
  let succ x = x + 1
  let mult x y = x * y
end
(* use *)
λ(x : Int.t) Int.mult x x
```

Existential types and abstract types

```
module type INT = sig
  type t
  val zero : t
  val succ : t → t
  val mult : t → t → t
end
(* definition *)
module Int : INT = struct
  type t = int
  let zero = 0
  let succ x = x + 1
  let mult x y = x * y
end
(* use *)
λ(x : Int.t) Int.mult x x
```

```
type INT =
  ∃α, {
    zero : α
    succ : α → α
    mult : α → α → α
  }
(* definition *)
let Int = pack ⟨int,
  {zero = 0
   succ (x : int) = x + 1
   mult (x y : int) = x * y
  }⟩ as INT in
(* use *)
unpack Int as ⟨α, Int'⟩ in
let square =
  λ(x : α) Int'.mult x x in
pack ⟨α, square⟩ as ∃β, β → β
```

Existential types and abstract types

```
module type INT = sig
  type t
  val zero : t
  val succ : t → t
  val mult : t → t → t
end
(* definition *)
module Int : INT = struct
  type t = int
  let zero = 0
  let succ x = x + 1
  let mult x y = x * y
end
(* use *)
λ(x : Int.t) Int.mult x x
```

Unpack

then repack!

```
type INT =
  ∃α, {
    zero : α
    succ : α → α
    mult : α → α → α
  }
(* definition *)
let Int = pack ⟨int,
  {zero = 0
  succ : α cannot escape
  mult : its scope y
  }⟩ as INT in
(* use *)
unpack Int as ⟨α, Int'⟩ in
let square =
  λ(x : α) Int'.mult x x in
pack ⟨α, square⟩ as ∃β, β → β
```

Existential types and modularity

```
module M = struct
  module A = ...
  module B = struct
    ...
    module N = e
    ...
  end
  module C = ...
end

... M.N ...
```

```
let M = {
  A = ...
  B = {
    ...
    N = e
    ...
  }
  C = ...
} in

... M.N ...
```

Existential types and modularity

```
module M = struct
  module A = ...
  module B = struct
    ...
    module N : S = e
    ...
  end
  module C = ...
end

... M.N ...
```

```
let M = {
  A = ...
  B = {
    ...
    N = e
    ...
  }
  C = ...
} in

... M.N ...
```

Abstraction in ML is modular.

Existential types and modularity

```
module M = struct
  module A = ...
  module B = struct
    ...
    module N : S = e
    ...
  end
  module C = ...
end

... M.N ...
```

```
unpack(pack < $\tau$ , e> as  $\exists\alpha.\tau'$ )
as < $\alpha$ , x> in
```

```
let M = {
  A = ...
  B = {
    ...
    N = x
    ...
  }
  C = ...
} in
```

Some code
is moved

```
... M.N ...
```

Abstraction in ML is modular.

Abstraction in F is **not** modular.

The usual rules for existential types

Elimination

$$\text{UNPACK} \quad \frac{\Gamma \vdash M_1 : \exists \alpha. \tau_1 \quad \Gamma, \alpha, x : \tau_1 \vdash M_2 : \tau_2 \quad \alpha \notin \text{ftv}(\tau_2)}{\Gamma \vdash \text{unpack } M_1 \text{ as } \langle \alpha, x \rangle \text{ in } M_2 : \tau_2}$$

- ▶ Condition necessary for wellformedness/binding reasons
- ▶ Constrains the structure of programs

Introduction

$$\text{PACK} \quad \frac{\Gamma \vdash M : \tau[\alpha \leftarrow \tau']}{\Gamma \vdash \text{pack } \langle \tau', M \rangle \text{ as } \exists \alpha. \tau : \exists \alpha. \tau}$$

- ▶ Very verbose: needs to repeat the whole type

F_λ: better primitives for existential types

Terms

$$\begin{aligned} M ::= & x \mid \text{let } x = M \text{ in } M \\ & \mid \lambda(x : \tau) M \mid M M \\ & \mid \Lambda\alpha. M \mid M \tau \\ & \mid \overline{\{l_i = M_i\}^{i \in 1..n}} \mid M.l \\ & \mid \exists\alpha. M \mid \text{open } \langle \alpha \rangle M \mid \nu\alpha. M \\ & \mid \Sigma \langle \beta \rangle (\alpha = \tau) M \mid (M : \tau) \end{aligned}$$

Types

$$\begin{aligned} \tau ::= & \alpha \\ & \mid \tau \rightarrow \tau \\ & \mid \forall\alpha. \tau \\ & \mid \overline{\{l_i : M_i\}^{i \in 1..n}} \\ & \mid \exists\alpha. \tau \end{aligned}$$

F_λ: better primitives for existential types

Terms

$M ::=$

- $x = M \text{ in } M$
- $M M$
- $\Lambda \alpha. M$
- $M \tau$
- $\overline{\{l_i = M_i\}}^{i \in 1..n}$
- $M.l$
- $\exists \alpha. M$
- $\text{open } \langle \alpha \rangle M$
- $\nu \alpha. M$
- $\Sigma \langle \beta \rangle (\alpha = \tau) M$
- $(M : \tau)$

Existential
introduction

Types

$\tau ::=$

- α
- $\tau \rightarrow \tau$
- $\forall \alpha. \tau$
- $\overline{\{l_i : M_i\}}^{i \in 1..n}$
- $\exists \alpha. \tau$

F_∃: better primitives for existential types

Terms

M	::	$\Lambda\alpha. M$		$\exists\alpha. M$		$\Sigma \langle\beta\rangle (\alpha = \tau) M$		$M\tau$		$\text{open } \langle\alpha\rangle M$		$\nu\alpha. M$
		$\{\overline{l_i} = M_i^{i \in 1..n}\}$						$M.l$				

Existential introduction points to $\Lambda\alpha. M$
Existential elimination points to $\text{open } \langle\alpha\rangle M$

Types

τ	::=	α
		$\tau \rightarrow \tau$
		$\forall\alpha. \tau$
		$\{\overline{l_i : M_i^{i \in 1..n}}\}$
		$\exists\alpha. \tau$

F_λ: better primitives for existential types

Terms

M	$::$	$\Lambda\alpha. M$	$ $	$M\tau$	$ $	$\overline{\{l_i = M_i\}}^{i \in 1..n}$	$ $	$M.l$	$ $	$\exists\alpha. M$	$ $	$\text{open } \langle\alpha\rangle M$	$ $	$\nu\alpha. M$	$ $	$\Sigma \langle\beta\rangle (\alpha = \tau) M$	$ $	$(M : \tau)$
-----	------	--------------------	-----	---------	-----	---	-----	-------	-----	--------------------	-----	---------------------------------------	-----	----------------	-----	--	-----	--------------

Existential introduction

Existential elimination

Name restriction

Types

τ	$=$	α
τ	\rightarrow	τ
$\forall\alpha. \tau$		
$\overline{\{l_i : M_i\}}^{i \in 1..n}$		
$\exists\alpha. \tau$		

F_{\forall} : better primitives for existential types

Terms

M	$::$	$\Lambda \alpha. M$	$ $	$M \tau$	$ $	$\{ \overline{l_i} = M_i^{i \in 1..n} \}$	$ $	$M.l$	$ $	$\exists \alpha. M$	$ $	$\text{open } \langle \alpha \rangle M$	$ $	$\nu \alpha. M$	$ $	$\Sigma \langle \beta \rangle (\alpha = \tau) M$	$ $	$(M : \tau)$
-----	------	---------------------	-----	----------	-----	---	-----	-------	-----	---------------------	-----	---	-----	-----------------	-----	--	-----	--------------

Existential introduction

Existential elimination

Name restriction

Open witness definition

Types

τ	$=$	α
$\tau \rightarrow \tau$	$=$	$\tau \rightarrow \tau$
$\forall \alpha. \tau$	$=$	$\forall \alpha. \tau$
$\{ \overline{l_i} : M_i^{i \in 1..n} \}$	$=$	$\{ \overline{l_i} : M_i^{i \in 1..n} \}$
$\exists \alpha. \tau$	$=$	$\exists \alpha. \tau$

F_λ: better primitives for existential types

Terms

M	$::$	$\Lambda \alpha. M$	$ $	$M \tau$	$ $	$\exists \alpha. M$	$ $	$\text{open } \langle \alpha \rangle M$	$ $	$\nu \alpha. M$	$ $	$\Sigma \langle \beta \rangle (\alpha = \tau) M$	$ $	$(M : \tau)$
		$\frac{\Lambda \alpha. M}{\Lambda \alpha. M}$				$\frac{\{l_i = M_i^{i \in 1..n}\}}{\exists \alpha. M}$		$\frac{M \tau}{\text{open } \langle \alpha \rangle M}$		$\frac{M.l}{\nu \alpha. M}$		$\frac{\{l_i = M_i^{i \in 1..n}\}}{\Sigma \langle \beta \rangle (\alpha = \tau) M}$		$\frac{(M : \tau)}{(M : \tau)}$

Existential introduction

Existential elimination

Name restriction

Open witness definition

Explicit coercion

Types

τ	$=$	α
		$\tau \rightarrow \tau$
		$\forall \alpha. \tau$
		$\frac{\{l_i : M_i^{i \in 1..n}\}}{\forall \alpha. \tau}$
		$\exists \alpha. \tau$

F_{\forall} : better primitives for existential types

Terms

M	$::$	$\lambda\alpha. M$	$ $	$M \tau$	$ $	$\{l_i = M_i^{i \in 1..n}\}$	$ $	$M.l$	$ $	$\exists\alpha. M$	$ $	$\text{open } \langle\alpha\rangle M$	$ $	$\nu\alpha. M$	$ $	$\Sigma \langle\beta\rangle (\alpha = \tau) M$	$ $	$(M : \tau)$
-----	------	--------------------	-----	----------	-----	------------------------------	-----	-------	-----	--------------------	-----	---------------------------------------	-----	----------------	-----	--	-----	--------------

Existential introduction

Existential elimination

Name restriction

Open witness definition

Explicit coercion

Types

τ	$=$	α
$\tau \rightarrow \tau$	$=$	$\tau \rightarrow \tau$
$\forall\alpha. \tau$	$=$	$\forall\alpha. \tau$
$\{l_i : M_i^{i \in 1..n}\}$	$=$	$\{l_i : M_i^{i \in 1..n}\}$
$\exists\alpha. \tau$	$=$	$\exists\alpha. \tau$

Two ideas

- ▶ Separate the opening of scope from the opening of existentials
- ▶ Use equations to define and use witnesses

Recovering unpack

Typing rules

OPEN

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

- ▶ Opens up an existential package
- ▶ Consumes an existential resource

NU

$$\frac{\Gamma, \exists \alpha \vdash M : \tau \quad \alpha \notin \text{ftv}(\tau)}{\Gamma \vdash \nu \alpha. M : \tau}$$

- ▶ Introduces a resource in the context
- ▶ Implements the restriction of scope

Recovering unpack

Typing rules

OPEN

$\Gamma \vdash M : \exists \alpha. \tau$

► Opens up an existential package

$\Gamma, \exists \alpha \vdash \text{open } \langle \alpha \rangle M : \tau$

► Consumes an existential resource

$\text{unpack } M_1 \text{ as } \langle \alpha, x \rangle \text{ in } M_2 \triangleq \nu \alpha. \text{let } x = \text{open } \langle \alpha \rangle M_1 \text{ in } M_2$

$\frac{\Gamma \vdash M_1 : \exists \alpha. \tau \quad \alpha \notin \text{ftv}(\tau)}{\Gamma \vdash \nu \alpha. M : \tau}$

► Introduces a resource in the context

$\Gamma \vdash \nu \alpha. M : \tau$

► Implements the restriction of scope

Recovering unpack

Typing rules

OPEN

$\Gamma \vdash M : \exists\alpha.\tau$

► Opens up an existential package

$\text{unpack } M_1 \text{ as } \langle \alpha, x \rangle \text{ in } M_2 \triangleq \nu\alpha. \text{let } x = \text{open } \langle \alpha \rangle M_1 \text{ in } M_2$

► Introduces a resource $\exists\alpha.\tau_1$ context

► Implements the restriction of scope

$\Gamma \vdash \nu\alpha. M : \tau$

Recovering unpack

Typing rules

OPEN

$\Gamma \vdash M : \exists \alpha. \tau$

► Opens up an existential package

$\text{unpack } M_1 \text{ as } \langle \alpha, x \rangle \text{ in } M_2 \triangleq \nu \alpha. \text{let } x = \underbrace{\text{open } \langle \alpha \rangle M_1}_{\tau_1} \text{ in } M_2$

► Consumes an existential resource

► Introduces a resource τ_1 in the context

► Implements the restriction of scope

Recovering unpack

Typing rules

OPEN

$\Gamma \vdash M : \exists \alpha. \tau$

► Opens up an existential package

$\text{unpack } M_1 \text{ as } \langle \alpha, x \rangle \text{ in } M_2 \triangleq \nu \alpha. \text{let } x = \underbrace{\text{open } \langle \alpha \rangle M_1}_{\tau_1} \text{ in } \underbrace{M_2}_{\tau_2}$

$\Gamma \vdash \nu \alpha. M : \tau$

Recovering unpack

Typing rules

OPEN

$\Gamma \vdash M : \exists \alpha. \tau$

► Opens up an existential package

$\text{unpack } M_1 \text{ as } \langle \alpha, x \rangle \text{ in } M_2 \triangleq \nu \alpha. \underbrace{\text{let } x = \text{open } \langle \alpha \rangle M_1 \text{ in } M_2}_{\tau_2}$

$\Gamma \vdash \nu \alpha. M : \tau$

Recovering unpack

Typing rules

OPEN

$\Gamma \vdash M : \exists \alpha. \tau$

► Opens up an existential package

$\text{unpack } M_1 \text{ as } \langle \alpha, x \rangle \text{ in } M_2 \triangleq \nu \alpha. \text{let } x = \text{open } \langle \alpha \rangle M_1 \text{ in } M_2$

τ_2

$\Gamma \vdash \nu \alpha. M : \tau$

Soundness concerns

Problem

Assume M_1 has type $\exists\alpha.\alpha \rightarrow \alpha$ and M_2 has type $\exists\alpha.\alpha$:

$$\begin{array}{l} \nu\alpha. \text{ let } f = \text{open } \langle\alpha\rangle M_1 \text{ in } \quad (* f : \alpha \rightarrow \alpha *) \\ \quad \text{let } x = \text{open } \langle\alpha\rangle M_2 \text{ in } \quad (* x : \alpha *) \\ \quad \quad f x \end{array}$$

Soundness concerns

Problem

Assume M_1 has type $\exists\alpha.\alpha \rightarrow \alpha$ and M_2 has type $\exists\alpha.\alpha$:

$$\begin{array}{l} \nu\alpha. \text{let } f = \text{open } \langle\alpha\rangle M_1 \text{ in } \quad (* f : \alpha \rightarrow \alpha *) \\ \quad \text{let } x = \text{open } \langle\alpha\rangle M_2 \text{ in } \quad (* x : \alpha *) \\ \quad \quad f x \end{array}$$

If the two witnesses are different, this is unsound!

Soundness concerns

Problem

Assume M_1 has type $\exists\alpha.\alpha \rightarrow \alpha$ and M_2 has type $\exists\alpha.\alpha$:

$$\begin{array}{l} \nu\alpha. \text{let } f = \text{open } \langle\alpha\rangle M_1 \text{ in } \quad (* f : \alpha \rightarrow \alpha *) \\ \quad \text{let } x = \text{open } \langle\alpha\rangle M_2 \text{ in } \quad (* x : \alpha *) \\ \quad \quad f \ x \end{array}$$

If the two witnesses are different, this is unsound!

Solution

Impose a **linear discipline** on openings:

- ▶ distinguish existential variables $\exists\alpha$ from other ones $\forall\beta$
- ▶ use a zipping operator on contexts $\Gamma_1 \curlywedge \Gamma_2$ such that $\exists\alpha$ is **never** duplicated along the typing derivation:

$$\exists\alpha \curlywedge \forall\alpha = \exists\alpha$$

$$\forall\alpha \curlywedge \exists\alpha = \exists\alpha$$

Recovering pack

Typing rules

EXISTS

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

- ▶ Introduces an existential quantifier

SIGMA

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

- ▶ Introduces an equation for the witness
- ▶ Inside, the witness is known
- ▶ Outside, the witness is unknown

COERCE

$$\frac{\Gamma \vdash M : \tau' \quad \Gamma \vdash \tau' \sim \tau}{\Gamma \vdash (M : \tau) : \tau}$$

- ▶ Uses the equations
- ▶ Coercibility = equality up to equations

Recovering pack

Typing rules

EXISTS

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

- ▶ Introduces an existential quantifier

SIGMA

$$\text{pack } \langle \tau', M \rangle \text{ as } \exists \alpha. \tau \triangleq \exists \beta. \Sigma \langle \beta \rangle (\alpha = \tau') (M : \tau)$$

$$\Gamma \vdash \exists \beta \vdash \Sigma \langle \beta \rangle (\alpha = \tau') M : \tau[\alpha \leftarrow \beta]$$

- ▶ Introduces an equation for the witness
- ▶ Inside, the witness is known
- ▶ Outside, the witness is unknown

COERCE

$$\frac{\Gamma \vdash M : \tau' \quad \Gamma \vdash \tau' \sim \tau}{\Gamma \vdash (M : \tau) : \tau}$$

- ▶ Uses the equations
- ▶ Coercibility = equality up to equations

Recovering pack

Typing rules

EXISTS

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

- ▶ Introduces an existential quantifier

SIGMA

$$\text{pack } \langle \tau', M \rangle \text{ as } \exists \alpha. \tau \triangleq \exists \beta. \Sigma \langle \beta \rangle (\alpha = \tau') (\underbrace{M : \tau}_{\tau[\alpha \leftarrow \tau']})$$

- ▶ Introduces an equation for the witness
- ▶ Inside, the witness is known
- ▶ Outside, the witness is unknown

COERCE

$$\frac{\Gamma \vdash M : \tau' \quad \Gamma \vdash \tau' \sim \tau}{\Gamma \vdash (M : \tau) : \tau}$$

- ▶ Uses the equations
- ▶ Coercibility = equality up to equations

Recovering pack

Typing rules

$$\frac{\text{EXISTS} \quad \Gamma, \exists \alpha \vdash M : \tau}{\Gamma \vdash \exists \alpha. M : \exists \alpha. \tau}$$

- ▶ Introduces an existential quantifier

$$\text{pack } \langle \tau', M \rangle \text{ as } \exists \alpha. \tau \quad \triangleq \quad \exists \beta. \Sigma \langle \beta \rangle (\alpha = \tau') \underbrace{(M : \tau)}_{\tau}$$

- ▶ Introduces an equation for the witness
- ▶ Inside, the witness β is known
- ▶ Outside, the witness is unknown

$$\frac{\text{COERCE} \quad \Gamma \vdash M : \tau' \quad \Gamma \vdash \tau' \sim \tau}{\Gamma \vdash (M : \tau) : \tau}$$

- ▶ Uses the equations
- ▶ Coercibility = equality up to equations

Recovering pack

Typing rules

EXISTS

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

- ▶ Introduces an existential quantifier

$$\text{pack } \langle \tau', M \rangle \text{ as } \exists \alpha. \tau \triangleq \exists \beta. \underbrace{\Sigma \langle \beta \rangle (\alpha = \tau') (M : \tau)}_{\tau[\alpha \leftarrow \beta]}$$

- ▶ Introduces an equation for the witness
- ▶ Inside, the witness is known
- ▶ Outside, the witness is unknown

COERCE

$$\frac{\Gamma \vdash M : \tau' \quad \Gamma \vdash \tau' \sim \tau}{\Gamma \vdash (M : \tau) : \tau}$$

- ▶ Uses the equations
- ▶ Coercibility = equality up to equations

Recovering pack

Typing rules

EXISTS

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

- ▶ Introduces an existential quantifier

$$\text{pack } \langle \tau', M \rangle \text{ as } \exists \alpha. \tau \triangleq \underbrace{\exists \beta. \Sigma \langle \beta \rangle (\alpha = \tau') (M : \tau)}_{\exists \beta. \tau[\alpha \leftarrow \beta]}$$

- ▶ Introduces an equation for the witness
- ▶ Inside the pack, the witness is known
- ▶ Outside, the witness is unknown

COERCE

$$\frac{\Gamma \vdash M : \tau' \quad \Gamma \vdash \tau' \sim \tau}{\Gamma \vdash (M : \tau) : \tau}$$

- ▶ Uses the equations
- ▶ Coercibility = equality up to equations

Small-step semantics

Problem

How can we reconcile substitution and linearity?

Answer

By **extruding** the sources of linearities:

- ▶ Call-by-value semantics
- ▶ Usual β -reduction on values: $(\lambda(x : \tau) M) v \longrightarrow M[x \leftarrow v]$
- ▶ Extrusion of Σ s: $v_1 (\Sigma \langle \beta \rangle (\alpha = \tau) v_2) \longrightarrow \Sigma \langle \beta \rangle (\alpha = \tau) v_1[\beta \leftarrow \alpha] v_2$
- ▶ Collapsing of open-exists: $\text{open } \langle \alpha \rangle (\exists \alpha. M) \longrightarrow M$
- ▶ Elimination of restricted variables:
$$\nu \beta. \Sigma \langle \beta \rangle (\alpha = \tau) M \longrightarrow M[\beta \leftarrow \alpha][\alpha \leftarrow \tau]$$

F_v: soundness

Type soundness

- ▶ Subject reduction
- ▶ Progress
- ▶ A mechanized proof (on a large subset) in Coq
 - ▶ \approx 10000 lines
 - ▶ uses locally nameless techniques
 - ▶ issues with nested binders, extrusions

Some particularities

- ▶ Weakening lemma: does not hold in general (due to linearity)
- ▶ Substitution lemma: only pure terms can safely be substituted
Pure term = typeable in a \exists -free environment

F^\forall and System F: a very tight relation

Two encodings

- ▶ from F to F^\forall : easy, presented in the previous slides
- ▶ from F^\forall to F: reorganization of the whole program
 - ▶ introduces intermediate let-bindings
 - ▶ F^\forall provides new ways to structure programs

A static correspondence

The two translations preserve types.

A dynamic correspondence

The two translations preserve meanings of programs:

- ▶ from F to F^\forall : the untyped skeletons are identical
- ▶ from F^\forall to F: the untyped skeleton of the target reduces to the one of the source

Extensions of F^\forall

Recall:

Two simple ingredients:

- ▶ linearity
- ▶ extrusion

The system extends well to:

- ▶ recursively defined witnesses
- ▶ recursive types
- ▶ recursive values

Related and future work

Related work

- ▶ Dreyer's RTG [Dre07]
 - ▶ slightly different constructs
 - ▶ imperative flavor in typing rules
 - ▶ imperative flavor in reduction rules

Future goals

- ▶ Interaction with references
- ▶ Parametricity, preservation of abstraction
- ▶ Strong reduction?
- ▶ Inference for existential types?

Outline

- ① Open existential types
- ② Singleton kinds
- ③ A kernel language
- ④ Conclusion

Singleton kinds in a nutshell

Finer types for F^ω

$$\begin{array}{l} \tau ::= \alpha \mid \lambda(\alpha :: \kappa)\tau \mid \tau\tau \mid (\tau, \tau) \mid \tau.1 \mid \tau.2 \\ \kappa ::= \star \mid \Pi(\alpha : \kappa)\kappa \mid \Sigma(\alpha : \kappa)\kappa \mid \mathcal{S}(\tau) \end{array}$$

Singleton kinds in a nutshell

Finer types for F^ω

With finer typing rules (excerpts)

$$\text{SINGLE} \quad \frac{\Gamma \vdash \tau :: \star}{\Gamma \vdash \tau :: \mathcal{S}(\tau)}$$

$$\text{SUB} \quad \frac{\Gamma \vdash \tau :: \kappa_1 \quad \Gamma \vdash \kappa_1 \leq \kappa_2}{\Gamma \vdash \tau :: \kappa_2}$$

$$\text{SUBSTAR} \quad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \star \leq \star}$$

$$\text{SUBFORGET} \quad \frac{\Gamma \vdash \tau :: \star}{\Gamma \vdash \mathcal{S}(\tau) \leq \star}$$

$$\text{SUBSINGLE} \quad \frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \star}{\Gamma \vdash \mathcal{S}(\tau_1) \leq \mathcal{S}(\tau_2)}$$

$$\text{EQEXTSINGLE} \quad \frac{\Gamma \vdash \tau_1 :: \mathcal{S}(\tau_2)}{\Gamma \vdash \tau_1 \equiv \tau_2 :: \mathcal{S}(\tau_2)}$$

$$\text{EQSUB} \quad \frac{\Gamma \vdash \tau \equiv \tau' :: \kappa_1 \quad \Gamma \vdash \kappa_1 \leq \kappa_2}{\Gamma \vdash \tau \equiv \tau' :: \kappa_2}$$

Singleton kinds in a nutshell

Finer types for F^ω

With finer typing rules (excerpts)

SINGLE

$$\frac{\Gamma \vdash \tau :: \star}{\Gamma \vdash \tau :: \mathcal{S}(\tau)}$$

SUB

$$\frac{\Gamma \vdash \tau :: \kappa_1 \quad \Gamma \vdash \kappa_1 \leq \kappa_2}{\Gamma \vdash \tau :: \kappa_2}$$

SUBSTAR

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \star \leq \star}$$

SUBFORGET

$$\frac{\Gamma \vdash \tau :: \star}{\Gamma \vdash \mathcal{S}(\tau) \leq \star}$$

SUBSINGLE

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \star}{\Gamma \vdash \mathcal{S}(\tau_1) \leq \mathcal{S}(\tau_2)}$$

EQEXTSINGLE

$$\frac{\Gamma \vdash \tau_1 :: \mathcal{S}(\tau_2)}{\Gamma \vdash \tau_1 \equiv \tau_2 :: \mathcal{S}(\tau_2)}$$

EQSUB

$$\frac{\Gamma \vdash \tau \equiv \tau' :: \kappa_1 \quad \Gamma \vdash \kappa_1 \leq \kappa_2}{\Gamma \vdash \tau \equiv \tau' :: \kappa_2}$$

Singleton kinds and type definitions

Singletons as definitions

- ▶ $\mathcal{S}(\tau)$ represents the kind of all types that are equivalent to τ
- ▶ if τ' has kind $\mathcal{S}(\tau)$, then τ and τ' are interchangeable
- ▶ let type $\alpha = \tau$ in $M \triangleq (\lambda(\alpha :: \mathcal{S}(\tau))M) \tau$

Singleton kinds and type definitions

Singletons as definitions

Already used in works on modules

- ▶ The TILT compiler [PCHS01]
- ▶ The mechanized definition of SML [LCH07]

Type equivalence in the singleton kinds system

Equivalence is sensitive to the context

- ▶ $\alpha :: \star, \beta :: \star \quad \vdash \alpha \equiv \beta :: \star$ does **not** hold
- ▶ $\alpha :: \star, \beta :: \mathcal{S}(\alpha) \quad \vdash \alpha \equiv \beta :: \star$ holds

Equivalence is sensitive to the kind

- ▶ $\alpha :: \star \vdash \lambda(\beta :: \star) \alpha \equiv \lambda(\beta :: \star) \beta :: \star \rightarrow \star$ does **not** hold
- ▶ $\alpha :: \star \vdash \lambda(\beta :: \star) \alpha \equiv \lambda(\beta :: \star) \beta :: \mathcal{S}(\alpha) \rightarrow \star$ holds

Equivalence is decidable

... but algorithms are subtle.

Singleton kinds: challenges

Extensions

How can we extend the theory and the algorithms with richer types?

- ▶ recursive types
- ▶ records with subtyping
- ▶ kind polymorphism

A reduction semantics?

Is it possible to characterize type equivalence using a small-step reduction relation, with strong normalization and confluence?

- ▶ In an intensional setting? yes! (Judicaël Courant, [Cou03])
- ▶ In an extensional setting?

Singleton kinds: challenges

Extensions

How can we extend the theory and the algorithms with richer types?

- ▶ recursive types
- ▶ records with subtyping
- ▶ kind polymorphism

A reduction semantics?

Is it possible to characterize type equivalence using a small-step reduction relation, with strong normalization and confluence?

- ▶ In an intensional setting? yes! (Judicaël Courant, [Cou03])
- ▶ In an **extensional** setting? **yes!** (this work)

Small-step extensional equivalence for singleton kinds

The general idea

From Crary [Cra07]

Unfolding a definition is η -equivalence at a singleton kind:

$$\frac{\text{EQEXTSINGLE} \quad \Gamma \vdash \tau :: \mathcal{S}(\tau')}{\Gamma \vdash \tau \equiv \tau' :: \mathcal{S}(\tau')}$$

Problem

- ▶ β -reduction + η -reduction: terminating, but **not confluent**
- ▶ β -reduction + η -expansion: confluent, but **not terminating**

Solution

Use explicit constructs to control η -expansion.

Explicit expanders

Syntax and semantics

$$\tau ::= \dots \mid \eta_{\kappa}$$

$$\eta_{\star} \xrightarrow{\eta_{\bullet}} \lambda(\alpha :: \star) \alpha$$

$$\eta_{\mathcal{S}(\tau)} \xrightarrow{\eta_{\bullet}} \lambda(\alpha :: \star) \tau$$

$$\eta_{\Pi(\alpha:\kappa_1) \kappa_2} \xrightarrow{\eta_{\bullet}} \lambda(f :: \Pi(\alpha : \kappa_1) \kappa_2) \lambda(\alpha :: \kappa_1) \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1} \alpha]} (f (\eta_{\kappa_1} \alpha))$$

$$\eta_{\Sigma(\alpha:\kappa_1) \kappa_2} \xrightarrow{\eta_{\bullet}} \lambda(p :: \Sigma(\alpha : \kappa_1) \kappa_2) (\eta_{\kappa_1} p.1, \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1} p.1]} p.2)$$

$$(\lambda(\alpha :: \kappa) \tau_1) \tau_2 \xrightarrow{\beta} \tau_1[\alpha \leftarrow \tau_2]$$

$$(\tau_1, \tau_2).i \xrightarrow{\pi} \tau_i$$

Explicit expandors

Syntax and semantics

$$\tau ::= \dots \mid \eta_{\kappa}$$

$$\begin{aligned} \eta_{\star} & \xrightarrow{\eta_{\bullet}} \lambda(\alpha :: \star) \alpha \\ \eta_{\mathcal{S}(\tau)} & \xrightarrow{\eta_{\bullet}} \lambda(\alpha :: \star) \tau \\ \eta_{\Pi(\alpha:\kappa_1) \kappa_2} & \xrightarrow{\eta_{\bullet}} \lambda(f :: \Pi(\alpha : \kappa_1) \kappa_2) \lambda(\alpha :: \kappa_1) \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1} \alpha]} (f (\eta_{\kappa_1} \alpha)) \\ \eta_{\Sigma(\alpha:\kappa_1) \kappa_2} & \xrightarrow{\eta_{\bullet}} \lambda(p :: \Sigma(\alpha : \kappa_1) \kappa_2) (\eta_{\kappa_1} p.1, \eta_{\kappa_2[\alpha \leftarrow \eta_{\kappa_1} p.1]} p.2) \end{aligned}$$

$$(\lambda(\alpha :: \kappa) \tau_1) \tau_2 \xrightarrow{\beta} \tau_1[\alpha \leftarrow \tau_2] \qquad (\tau_1, \tau_2).i \xrightarrow{\pi} \tau_i$$

Example

$$\begin{aligned} & \eta_{\Pi(\alpha:\mathcal{S}(\tau)) (\mathcal{S}(\alpha) \times \star)} \tau_0 \\ \xrightarrow{\eta^+} & \lambda(\alpha :: \star) \eta_{\mathcal{S}(\eta_{\mathcal{S}(\tau)} \alpha) \times \star} (\tau_0 (\eta_{\mathcal{S}(\tau)} \alpha)) \\ \xrightarrow{\eta^+} & \lambda(\alpha :: \star) \eta_{\mathcal{S}(\tau) \times \star} (\tau_0 \tau) \\ \xrightarrow{\eta^+} & \lambda(\alpha :: \star) (\eta_{\mathcal{S}(\tau)} (\tau_0 \tau).1, \eta_{\star} (\tau_0 \tau).2) \\ \xrightarrow{\eta^+} & \lambda(\alpha :: \star) (\tau, \eta_{\star} (\tau_0 \tau).2) \end{aligned}$$

Confluence and strong normalization

Facts

- ▶ Subject reduction holds for $\xrightarrow{\beta\pi\eta\bullet}$
- ▶ The subrelation $\xrightarrow{\beta\pi}$ is:
 - ▶ confluent
 - ▶ strongly normalizing for wellformed types
- ▶ The subrelation $\xrightarrow{\eta\bullet}$ is:
 - ▶ confluent
 - ▶ strongly normalizing

What about their combination?

confluence? strong normalization?

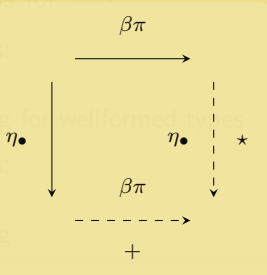
We would like to prove them without starting from scratch...

Confluence and strong normalization

Decreasing diagrams to the rescue!

Taken from [Cos96], proved again in Coq.

Since $\xrightarrow{\beta\pi}$ is terminating and:



We get:

- ▶ $\xrightarrow{\beta\pi\eta\bullet}$ is confluent (from the confluence of $\xrightarrow{\beta\pi}$ and $\xrightarrow{\eta\bullet}$)
- ▶ $\xrightarrow{\beta\pi\eta\bullet}$ is strongly normalizing (from the normalization of $\xrightarrow{\beta\pi}$ and $\xrightarrow{\eta\bullet}$ and the fact that $\xrightarrow{\beta\pi}$ preserves $\xrightarrow{\eta\bullet}$ -normal forms)

Insertion of expandors

η -expansion happens only in the presence of expandors

Definition (excerpts)

$$[\alpha]^\Gamma \triangleq \begin{cases} \eta_{\Gamma}(\alpha) \alpha & \text{if } \alpha \in \text{dom } \Gamma \\ \alpha & \text{otherwise} \end{cases}$$

variables are annotated with their expandors

$$[\lambda(\alpha :: \kappa) \tau]^\Gamma \triangleq \lambda(\alpha :: [\kappa]^\Gamma) [\tau]^\Gamma, \alpha :: [\kappa]^\Gamma$$

also bound variables

$$[\tau_1 \tau_2]^\Gamma \triangleq [\tau_1]^\Gamma [\tau_2]^\Gamma$$

congruence everywhere else

$$[\star]^\Gamma \triangleq \star$$

$$[\mathcal{S}(\tau)]^\Gamma \triangleq \mathcal{S}([\tau]^\Gamma)$$

$$[\Pi(\alpha : \kappa_1) \kappa_2]^\Gamma \triangleq \Pi(\alpha : [\kappa_1]^\Gamma) [\kappa_2]^\Gamma, \alpha :: [\kappa_1]^\Gamma$$

$$[\varepsilon] \triangleq \varepsilon$$

$$[\Gamma, \alpha :: \kappa] \triangleq [\Gamma], \alpha :: [\kappa]^{[\Gamma]}$$

pointwise extension for environments

Insertion of expandors

η -expansion happens only in the presence of expandors

Definition (excerpts)

$$[\alpha]^\Gamma \triangleq \begin{cases} \eta_{\Gamma(\alpha)} \alpha & \text{if } \alpha \in \text{dom } \Gamma \\ \alpha & \text{otherwise} \end{cases}$$

variables are annotated with their expandors

$[\lambda(\alpha :: \tau) \beta]^\Gamma \triangleq \lambda(\alpha :: [\kappa]^\Gamma) [\tau]^\Gamma [\beta]^\Gamma$ also bound variables

Example

$$\begin{aligned} & [\lambda(\gamma : \star \rightarrow \star) \gamma \beta]^{[\alpha :: \star, \beta :: \mathcal{S}(\alpha)]} \\ &= \lambda(\gamma : \star \rightarrow \star) (\eta_{\star \rightarrow \star} \gamma) (\eta_{\mathcal{S}(\alpha)} \beta) \end{aligned}$$

congruence
anywhere else

$$[\mathcal{S}(\tau)]^\Gamma \triangleq \mathcal{S}([\tau]^\Gamma)$$

$$[\Pi(\alpha : \kappa_1) \kappa_2]^\Gamma \triangleq \Pi(\alpha : [\kappa_1]^\Gamma) [\kappa_2]^\Gamma, \alpha :: [\kappa_1]^\Gamma$$

$$[\varepsilon] \triangleq \varepsilon$$

$$[\Gamma, \alpha :: \kappa] \triangleq [\Gamma], \alpha :: [\kappa]^{[\Gamma]}$$

pointwise extension for environments

The adequacy theorem

Theorem

$$\Gamma \vdash \tau_1 \equiv \tau_2 :: \kappa \quad \text{iff} \quad \left\{ \begin{array}{l} \Gamma \vdash \tau_1 :: \kappa \quad \Gamma \vdash \tau_2 :: \kappa \\ [\eta_\kappa \tau_1]^{\Gamma} \xleftrightarrow{\beta\pi\eta} [\eta_\kappa \tau_2]^{\Gamma} \end{array} \right.$$

Note: the head expansion is imposed by the presence of subtyping.

Proof: based on a logical relation similar to [SH06]

What is new?

- ▶ Compared with Harper-Stone [SH06]:
 - ▶ a subtle algorithm vs. a classic convertibility test
 - ▶ might be easier to implement
 - ▶ can possibly lead to more efficient algorithms
 - ▶ kind-indexed normal forms $[\eta_\kappa \tau]^{\Gamma}$ vs. kind-free normal form $[\tau]^{\Gamma}$
- ▶ Compared with Crary [Cra07]:
 - ▶ expanding free variables vs. expanding all variables
 - ▶ large-step η -expansion vs. small-step η -expansion
 - ▶ $\xrightarrow{\eta}^*$; $\xrightarrow{\beta}^*$ vs. $(\xrightarrow{\eta} \cup \xrightarrow{\beta})^*$

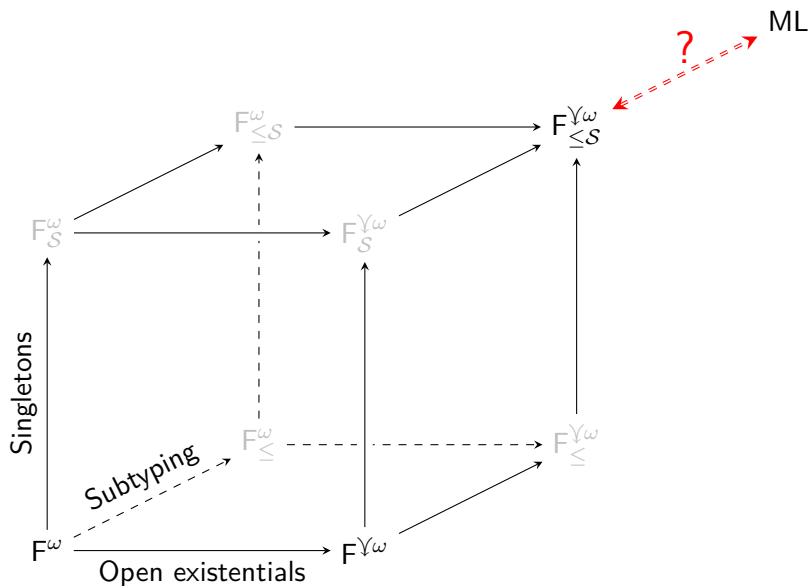
Future work

- ▶ Verify proofs in Coq
- ▶ Extend the system with records and subtyping on records
- ▶ Extend with (a restricted form of) recursive types
- ▶ Study a system where equivalence is *defined* using convertibility

Outline

- ① Open existential types
- ② Singleton kinds
- ③ A kernel language
- ④ Conclusion

The general picture



Ingredients of $F_{S \leq}^{\omega}$

- ▶ An explicitly typed language
- ▶ Open existential types
- ▶ Singleton kinds
- ▶ Records of terms and *records of types*
- ▶ Subtyping and subkinding on records

Ingredients of $F_{\mathcal{S} \leq}^{\omega}$

- ▶ An explicitly typed language
- ▶ Open existential types
- ▶ Singleton kinds
- ▶ Records of terms and *records of types*
- ▶ Subtyping and subkinding on records

Example

- ▶ Subtyping on records:
 $\vdash \{l_1 : \tau_1 ; l_2 : \tau_2 ; l_3 : \tau_3\} \leq \{l_3 : \tau_3 ; l_1 : \tau'_1\}$ when $\vdash \tau_1 \leq \tau'_1$
- ▶ Subkinding on records of types:
 $\vdash \langle l_1 \text{ as } \alpha_1 :: \star ; l_2 :: \mathcal{S}(\alpha_1) ; l_3 :: \star \rangle \leq \langle l_2 \text{ as } \alpha_2 :: \star ; l_1 :: \mathcal{S}(\alpha_2) \rangle$
- ▶ Subtyping and existentials: covariance in the bound
 $\beta :: \star \vdash \exists(\alpha :: \mathcal{S}(\beta))\tau \leq \exists(\alpha :: \star)\tau$
- ▶ Subtyping and universals: contravariance in the bound
 $\beta :: \star \vdash \forall(\alpha :: \star)\tau \leq \forall(\alpha :: \mathcal{S}(\beta))\tau$

Ingredients of $F_{\mathcal{S} \leq}^{\downarrow \omega}$

- ▶ An explicitly typed language
- ▶ Open existential types
- ▶ Singleton kinds
- ▶ Records of terms and *records of types*
- ▶ Subtyping

A prototype implementation in OCaml:

- ▶ proof of concept, rather inefficient
- ▶ uses the Sato-Pollack representation of binders
- ▶ about 4000 lines of code
- ▶ available from
<https://bitbucket.org/esope/fzip/>

Example

- ▶ Subtyping and existentials: covariance in the bound
 $\vdash \{l_1 \vdash \exists(\alpha :: \mathcal{S}(\beta))\tau \leq \exists(\alpha :: *)\tau$
- ▶ Subtyping and existentials: covariance in the bound
 $\vdash \langle l_1 \vdash \exists(\alpha :: \mathcal{S}(\beta))\tau \leq \exists(\alpha :: *)\tau$
- ▶ Subtyping and existentials: covariance in the bound
 $\beta :: * \vdash \exists(\alpha :: \mathcal{S}(\beta))\tau \leq \exists(\alpha :: *)\tau$
- ▶ Subtyping and universals: contravariance in the bound
 $\beta :: * \vdash \forall(\alpha :: *)\tau \leq \forall(\alpha :: \mathcal{S}(\beta))\tau$

From ML modules to $F_{S \leq}^{\omega}$

A translation

- ▶ Inspired from the F-ing modules translation [RRD10]
- ▶ Using open existential types
 - ▶ less unpack/repack
- ▶ Using singleton kinds
 - ▶ uniform treatment of type components
 - ▶ type definitions are not unfolded
- ▶ Using subtyping/subkinding
 - ▶ avoids insertions of coercions
- ▶ Implements the same discipline as in ML:
 - ▶ Type components of a module are reachable by a *single* access path

A comparison with ML

Differences

ML modules

- ▶ type and value fields are mixed
- ▶ automatic handling of type fields
- ▶ dependent value fields
- ▶ more structured
- ▶ —

 $F_{S \leq}^{\omega}$

- ▶ phase-split style
- ▶ manual handling of existentials
- ▶ no dependencies inside records
- ▶ more flexible
- ▶ some kinds are duplicated

Similarities

- ▶ about the same level of verbosity
- ▶ about the same structure of programs (after type erasure)

Future work

- ▶ Metatheory of $F_{S \leq}^{\forall \omega}$
 - ▶ Challenging part: singletons with records of types
- ▶ Prototype
 - ▶ Partial type inference: bidirectional? implicit arguments?
 - ▶ Partial inference for open existential types?
- ▶ Better constructs to program in a phase-split style?

Outline

- ① Open existential types
- ② Singleton kinds
- ③ A kernel language
- ④ Conclusion

Achievements

Open existential types

- ▶ better constructs for existentials
- ▶ simple ingredients: linearity + extrusion
- ▶ a mechanized soundness proof
- ▶ tightly connected to System F

Singleton kinds

- ▶ a small-step reduction semantics
- ▶ explicit η -expansion
- ▶ equivalence as conversion

A better System F ^{ω}

- ▶ subtyping + singletons kinds + open existential types
- ▶ rather close to ML
- ▶ a prototype typechecker
- ▶ a step towards modular programming with F ^{ω}

To conclude

Decomposing modules. . .

To conclude

Decomposing modules...

- ▶ to better understand them (hopefully)

To conclude

Decomposing modules. . .

- ▶ to better understand them (hopefully)
- ▶ to rebuild a new language (eventually)

To conclude

Decomposing modules...

- ▶ to better understand them (hopefully)
- ▶ to rebuild a new language (eventually)

Thank you!

Extra material

⑤ Decreasing diagrams

⑥ The logical relation for completeness of convertibility

⑦ References

Outline

5 Decreasing diagrams

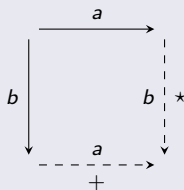
6 The logical relation for completeness of convertibility

7 References

Commutation by decreasing diagrams

DPG-commutation

A relation \rightarrow_a DPG-commutes with \rightarrow_b when:



Commutation condition

Assume that \rightarrow_a is a terminating relation. If \rightarrow_a DPG-commutes with \rightarrow_b , then \rightarrow_a^* commutes with \rightarrow_b^* .

Combination of confluence and normalization

Enhanced Hindley-Rosen's lemma

Assume that \rightarrow_a and \rightarrow_b are two confluent relations. If \rightarrow_a DPG-commutes with \rightarrow_b , and if \rightarrow_a is terminating, then $\rightarrow_a \cup \rightarrow_b$ is confluent.

Enhanced Akama's lemma

Let \rightarrow_a and \rightarrow_b be two confluent and terminating relations. If \rightarrow_a DPG-commutes with \rightarrow_b , and if \rightarrow_a preserves \rightarrow_b -normal forms, then $\rightarrow_a \cup \rightarrow_b$ is also a confluent and terminating relation.

Outline

5 Decreasing diagrams

6 The logical relation for completeness of convertibility

7 References

The logical relation

\mathcal{G} , \mathcal{T} and \mathcal{K} are non-empty finite sets

Definition

By induction on the sizes of kinds:

- ▶ $\mathcal{G} \models \mathcal{K}$ holds if:
 - ▶ $\mathcal{K} = \{\star\}$
 - ▶ or, $\mathcal{K} = \mathcal{S}(\mathcal{U})$ and $\mathcal{G} \models \mathcal{U} :: \{\star\}$ holds;
 - ▶ or, $\mathcal{K} = \Pi(\alpha : \mathcal{K}_1) \mathcal{K}_2$ and $\mathcal{G} \models \mathcal{K}_1$ and for every $\mathcal{G}' \sqsupseteq \mathcal{G}$, if $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1$, then $\mathcal{G}' \models \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$;
 - ▶ or, $\mathcal{K} = \Sigma(\alpha : \mathcal{K}_1) \mathcal{K}_2$ and $\mathcal{G} \models \mathcal{K}_1$ and for every $\mathcal{G}' \sqsupseteq \mathcal{G}$, if $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1$, then $\mathcal{G}' \models \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$;
- ▶ $\mathcal{G} \models \mathcal{T} :: \mathcal{K}$ holds if $\mathcal{G} \models \mathcal{K}$ holds and:
 - ▶ $\mathcal{K} = \{\star\}$ and for every $\Gamma \in \mathcal{G}$ and $\tau_1, \tau_2 \in \mathcal{T}$, $[\tau_1]^{[\Gamma]} \stackrel{\beta\pi\eta}{\longleftrightarrow} [\tau_2]^{[\Gamma]}$;
 - ▶ or, $\mathcal{K} = \mathcal{S}(\mathcal{U})$ and $\mathcal{G} \models \mathcal{T} \cup \mathcal{U} :: \{\star\}$
 - ▶ or, $\mathcal{K} = \Pi(\alpha : \mathcal{K}_1) \mathcal{K}_2$ and for every $\mathcal{G}' \sqsupseteq \mathcal{G}$, if $\mathcal{G}' \models \mathcal{T}_1 :: \mathcal{K}_1$, then $\mathcal{G}' \models \mathcal{T} \mathcal{T}_1 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}_1]$;
 - ▶ or, $\mathcal{K} = \Sigma(\alpha : \mathcal{K}_1) \mathcal{K}_2$ and $\mathcal{G} \models \mathcal{T}.1 :: \mathcal{K}_1$ and $\mathcal{G} \models \mathcal{T}.2 :: \mathcal{K}_2[\alpha \leftarrow \mathcal{T}.1]$.

Outline

5 Decreasing diagrams

6 The logical relation for completeness of convertibility

7 References

References I



Roberto Di Cosmo.

On the power of simple diagrams.

In *RTA '96: Proceedings of the 7th International Conference on Rewriting Techniques and Applications*, pages 200–214, London, UK, 1996.
Springer-Verlag.



Judicaël Courant.

Strong normalization with singleton types.

Electronic Notes in Theoretical Computer Science, 70(1):53 – 71, 2003.
ITRS '02, Intersection Types and Related Systems (FLoC Satellite Event).



Karl Crary.

Sound and complete elimination of singleton kinds.

ACM Trans. Comput. Logic, 8(2):8, 2007.

References II



Derek Dreyer.

Recursive type generativity.

Journal of Functional Programming, pages 433–471, 2007.



Daniel K. Lee, Karl Cray, and Robert Harper.

Towards a mechanized metatheory of Standard ML.

SIGPLAN Not., 42(1):173–184, 2007.



John C. Mitchell and Gordon D. Plotkin.

Abstract types have existential type.

ACM Trans. Program. Lang. Syst., 10(3):470–502, 1988.



Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone.

Implementing the TILT internal language.

Technical Report CMU-CS-00-180, Carnegie Mellon School of Computer Science, 2001.

References III



Andreas Rossberg, Claudio V. Russo, and Derek Dreyer.

F-ing modules.

In *2010 ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI2010)*, January 2010.



Claudio V. Russo.

Types for modules.

Electronic Notes in Theoretical Computer Science, 60, January 2003.



Christopher A. Stone and Robert Harper.

Extensional equivalence and singleton types.

ACM Trans. Comput. Logic, 7(4):676–722, 2006.