

- 1 Introduction
- 2 Splitting unpack
- 3 Splitting pack
- 4 Reduction
- 5 Advanced technicalities
- 6 Expressiveness

# Abstract

We present a variant of the explicitly-typed second-order polymorphic  $\lambda$ -calculus with primitive *open existential types*, *i.e.* a collection of more atomic constructs for introduction and elimination of existential types. We equip the language with a call-by-value small-step reduction semantics that enjoys the subject reduction property.

We claim that open existential types model abstract types and type generativity in a modular way. Our proposal can be understood as a logically-motivated variant of Dreyer's RTG where type generativity is no more seen as a side effect.

# Open Existential types for Module systems

## A Logical Account of Type Generativity

Didier Rémy

INRIA-Rocquencourt

IFIP WG 2.8, June 2008

Based on joint work with

**Benoît Montagu**



# Motivations

Modular programming is the key to writing good, maintainable software.  
Will be even more important tomorrow than today.

However, despite 20 years of intensive research on module systems:  
There is a **big gap** between:

- The **intuitive simplicity** of the underlying concepts, and
- The **actual complexity** of existing solutions.

## Our goals

- Explain *or* reduce this gap.
- Design a **core calculus** for the **surface language** of a language with:
  - **first-class** modules
  - that is **conceptually economical**, e.g. avoids duplication of concepts.

# What is needed for module systems?

## Already in the core-calculus

- Structures are records
- Functors are functions
- Signatures are types

## Crucial (and deep) features for expressiveness

- Type abstraction (may already be in the core language)
- Type generativity (the master-key to modules)

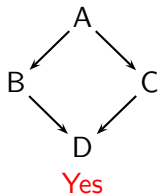
## Important (but not so deep) features for conciseness

- Sharing a posteriori (diamond import problem)
- Flexible naming policy

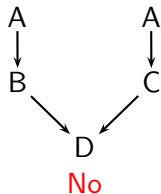
# Type generativity

## The problem

- A defines  $t$  abstractly
- B and C uses A
- Can D assume that B and C have compatible views of  $t$ ?
- Can also two copies/views of A be made incompatible? —this is type generativity.



or



Keep track of identifies of abstract types **in a way or another**

# Previous approaches

~~Existential types~~: model type abstraction but lack modular structure.

## Path-based systems.

- An old idea (Dave MacQueen, Modules for Standard ML, 1984)
- Today, still at the basis of all module systems.

## General idea

- Cannot refer to **how** types have been defined, since they have been forgotten.
- Instead refer to **where** they have been defined.
- An abstract type is referred to as a projection path from a value variable.

# Problem with path-based systems

## General problem

- Types depend on values (at least syntactically)
- Although paths only use a small fragment of dependent types, a much larger fragment is needed to preserve stability under term substitution.

## Dependent types

- An overkill technology.
- They do not carry good intuitions about modules (in our opinion).
- Too complicated to be exposed to the programmer, hence they defined a core calculus in which existing languages are elaborated.

## Elaboration semantics

- Elaboration is a compilation process, may be of arbitrary complexity.
- The user cannot perform it mentally.
- Loses the connection with logic: no small-step reduction semantics.

# Dreyer's RTG: a solution without dependent types!

## Motivations

- Designed and used as an **internal language**
- for a language with **recursive** and **mixin** modules.

## Underlying ideas

- Sees type generativity as a **static** side effect.
- Use of linear types to keep track of such side effects.

## Achievements

- Interesting set of primitives
- which can be used to model recursive and mixin modules.
- Type generativity can be explained without dependent types.

# Problem with RTG

## Based on and carrying wrong intuitions

- **Type generativity is a side effect** (claimed very strongly)
- Their semantics enforces and relies on a strictly deterministic evaluation order.

## Ad hoc meta-theory

- Typechecking in RTG uses an abstract machine that performs side effects into a global store.
- Their dynamic semantics is store based, including the modelling of generativity.

## Consequences

- Unintuitive semantics: programmers can't run the machine mentally.
- Any connection with logic is lost.
- Cannot be exposed to users, *i.e.* used as an external language.

# $F^{\forall}$ (Fzip): a variant of RTG without the drawbacks

## Standard static and dynamic semantics

- Typing rules are compositional and have a logical flavor.
- Small-step reduction semantics
- The two are related by *subject reduction* and *progress* lemmas.
- No use of recursive types is needed to model type generativity (*but they could be useful with recursive or mixin modules*)

## Curry-Howard isomorphism (for a subset of $F^{\forall}$ )

- Formulae are the same as in System-F with existential types.
- The same formulae are provable.
- There are more proofs—which can be assembled more modularly.
- Reduction is proof normalization, indeed.

# Beyond $F^\forall$

## Modules can be explained as a combination of

- **open abstract types**, to model type generativity
- ***Shape bounded quantification*** to recover conciseness  
*(complementary, not described here)*

# Reminder: pack and unpack

PACK

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

UNPACK

$$\frac{\Gamma \vdash M : \exists \alpha. \tau \quad \Gamma, \alpha, x : \tau \vdash M' : \tau' \quad \alpha \notin \text{ftv}(\tau')}{\Gamma \vdash \text{unpack } M \text{ as } \alpha, x \text{ in } M' : \tau'}$$

# Splitting unpack

unpack  $M$  as  $\alpha, x$  in  $M'$

$\triangleq$

$\nu\alpha.$  let  $x =$  open  $\langle\alpha\rangle M$  in  $M'$

Limits the scope of  $\alpha$

Uses  $\alpha$  for the abstract type of  $M$

Binds  $M$  to  $x$  in  $M'$

# Splitting unpack

$\nu\alpha.$     let  $x =$     open  $\langle\alpha\rangle M$     in  $M'$

## Splitting unpack

advantages

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

$M$  need not be at toplevel.

## Splitting unpack

advantages

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

$\alpha$  need not be hidden immediately.

## Splitting unpack

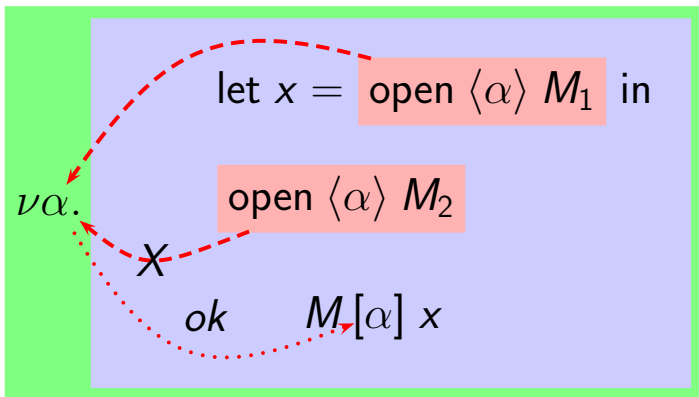
advantages

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

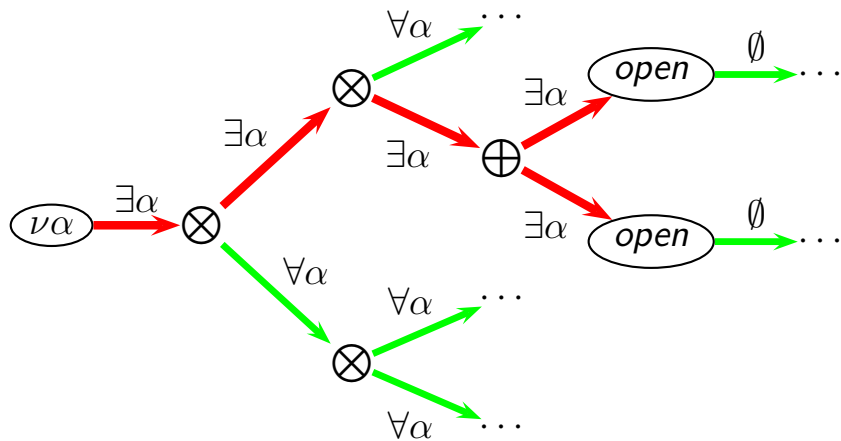
$\alpha$  need not be hidden at all in program *components*

# Typechecking

Must forbid incorrect programs such as



# Typechecking



# Typechecking

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

# Typechecking

OPEN

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

NU

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

# Typechecking

OPEN

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

LET

$$\frac{\Gamma_1 \vdash M_1 : \tau_1 \quad \Gamma_2, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma_1 \Downarrow \Gamma_2 \vdash \text{let } x = M_1 \text{ in } M_2 : \tau_2}$$

NU

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

# Typechecking

OPEN

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

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

LET

$$\frac{\Gamma_1 \vdash M_1 : \tau_1 \quad \Gamma_2, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma_1 \Downarrow \Gamma_2 \vdash \text{let } x = M_1 \text{ in } M_2 : \tau_2}$$

NU

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

## Typechecking

## Zipping

Zipping of two type environments ensures that every existential type appears in at most one of the environments.

$$\begin{array}{l} \forall\alpha \ \forall\alpha = \forall\alpha \\ \forall\alpha \ \exists\alpha = \exists\alpha \\ \exists\alpha \ \forall\alpha = \exists\alpha \end{array}$$

$$x : \tau \ \forall\ x : \tau = x : \tau$$

$$\emptyset \ \forall\ \emptyset = \emptyset$$

$$(\Gamma_1, b_1) \ \forall\ (\Gamma_2, b_2) = (\Gamma_1 \ \forall\ \Gamma_2), (b_1 \ \forall\ b_2)$$

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

# Splitting pack

pack  $\langle \tau, M \rangle$  as  $\exists \alpha. \tau'$

$\triangleq$

$\exists(\alpha = \tau) (M : \tau')$

makes  $\alpha$  abstract  
with witness  $\tau$

converts the type of  $M$   
using the equation(s)

# Splitting pack

pack  $\langle \tau, M \rangle$  as  $\exists \alpha. \tau'$

$\triangleq$

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

closes the abstract type  $\beta$

converts the type of  $M$

defines the **open** abstract type  $\beta$   
with internal name  $\alpha$  and witness  $\tau$

# Splitting pack

pack  $\langle \tau, M \rangle$  as  $\exists \alpha. \tau'$

$\triangleq$

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

# Splitting pack

pack  $\langle \tau, M \rangle$  as  $\exists \alpha. \tau'$

$\triangleq$

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

A module with an open abstract type  $\beta$ .

# Splitting pack

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

A **sub**-module with an open abstract type  $\beta$ .

# Typechecking

EXISTS

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

# Typechecking

EXISTS

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

OPEN

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

# Typechecking

SIGMA

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

EXISTS

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

OPEN

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

# Typechecking

COERCE

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

uses

SIGMA

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

EXISTS

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

# Summary

## Types are unchanged

$$\tau ::= \alpha \quad | \quad \tau \rightarrow \tau \quad | \quad \forall \alpha. \tau \quad | \quad \exists \alpha. \tau$$

## Expressions are

$$\begin{array}{l}
 M ::= \dots \\
 \quad | \quad \exists \alpha. M \quad | \quad \Sigma \langle \beta \rangle (\alpha = \tau) M \quad | \quad (M : \tau) \\
 \quad | \quad \nu \alpha. M \quad | \quad \text{open } \langle \alpha \rangle M
 \end{array}$$

# Examples

In ML:

$$\text{module } X = \text{struct} \left( \begin{array}{l} \text{type } t = \text{int} \\ \text{val } z = 0 \\ \text{val } s = \lambda(x : \text{int})x+1 \end{array} \right) : \text{sig} \left( \begin{array}{l} \text{type } t \\ \text{val } z : t \\ \text{val } s : t \rightarrow t \end{array} \right)$$

In Fzip:

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

# Examples

In ML:

$$\text{module } X = \text{struct} \left( \begin{array}{l} \text{type } t = \text{int} \\ \text{val } z = 0 \\ \text{val } s = \lambda(x : \text{int})x+1 \end{array} \right) : \text{sig} \left( \begin{array}{l} \text{type } t \\ \text{val } z : t \\ \text{val } s : t \rightarrow t \end{array} \right)$$

In Fzip:

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

open  $\langle \beta \rangle x$

# Examples

In ML:

Making generative views of  $x$

In Fzip:

```
let x =  $\exists(\alpha = \text{int})$   $\left( \left\{ \begin{array}{l} z = 0 ; \\ s = \lambda(x : \text{int})x+1 \end{array} \right\} : \left\{ \begin{array}{l} z : \alpha ; \\ s : \alpha \rightarrow \alpha \end{array} \right\} \right)$  in
let x1 = open  $\langle \beta_1 \rangle$  x in
let x2 = open  $\langle \beta_2 \rangle$  x in
...
```

# Examples

## Functors

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

## Example

let *MakeSet* =

$\Lambda\alpha. \lambda(cmp : \alpha \rightarrow \alpha \rightarrow bool) \exists(\beta = set(\alpha)) (\dots : set(\beta))$  in

let  $s_1 = \text{open } \langle \beta_1 \rangle$  *MakeSet* [*int*] (<) in

let  $s_2 = \text{open } \langle \beta_2 \rangle$  *MakeSet* [ $\beta_1$ ] ( $s_1$ .*cmp*) in

...

# 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 (new)

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

## Reduction

## Example

$$\text{let } x = \Sigma \langle \beta \rangle (\alpha = \text{int}) (1 : \alpha) \text{ in } \{l_1 = x ; l_2 = (\lambda(y : \beta)y) x\}$$

$$\downarrow$$

$$\Sigma \langle \beta \rangle (\alpha = \text{int}) \text{ let } x = (1 : \alpha) \text{ in } \{l_1 = x ; l_2 = (\lambda(y : \beta)y) x\}$$

$$\downarrow$$

$$\Sigma \langle \beta \rangle (\alpha = \text{int}) \{l_1 = (1 : \alpha) ; l_2 = (\lambda(y : \beta)y) (1 : \alpha)\}$$

$$\downarrow$$

$$\Sigma \langle \beta \rangle (\alpha = \text{int}) \{l_1 = (1 : \alpha) ; l_2 = (1 : \alpha)\}$$

## Reduction

## Values

- 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.

## Definition

*Values*

$$v ::= u \quad | \quad (u : \tau)$$

$$u ::= x \quad | \quad \lambda(x : \tau)M \quad | \quad \Lambda\alpha. M \quad | \quad \exists\beta. \Sigma \langle\beta\rangle (\alpha = \tau) v$$

*Results*

$$w ::= v \quad | \quad \Sigma \langle\beta\rangle (\alpha = \tau) w$$

## Note

- Abstractions  $\lambda$ 's and  $\Lambda$ 's are always values because they are pure, *i.e.* typechecked in  $\Gamma$  without  $\exists\alpha$ 's.
- Otherwise, unpure abstractions should be treated linearly.

## Reduction

## Semantics

## Call-by-value small-step reduction semantics

Elimination rules:  $\beta$ -reduction rules plus,

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

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

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

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

+ Propagation of coercions (uninteresting reduction rules)

## Reduction

## Type soundness

## Theorem (Subject reduction)

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

## Theorem (Progress)

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

# The appearance of recursive types

## Internal recursion, through openings:

let  $x = \exists(\alpha = \beta \rightarrow \beta) M$  in open  $\langle \beta \rangle x$

reduces to:

open  $\langle \beta \rangle \exists(\alpha = \beta \rightarrow \beta) M$

$\exists(\alpha = \tau) M$  stands for  
 $\exists \gamma. \Sigma \langle \gamma \rangle (\alpha = \beta \rightarrow \beta) M$

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

## External recursion, through open witness definitions:

$$\{ \begin{array}{l} l_1 = \Sigma \langle \beta_1 \rangle (\alpha_1 = \beta_2 \rightarrow \beta_2) M_1 ; \\ l_2 = \Sigma \langle \beta_2 \rangle (\alpha_2 = \beta_1 \rightarrow \beta_1) M_2 \end{array} \}$$

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

Cannot occur in System F.

# The appearance of recursive types

## Origin of the problem

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

$\beta$  may appear in  $\tau$  which is later meant to be equated with  $\beta$ .

## Solutions

- 1 Remove  $\forall\beta$  from the premisses:
  - requires that  $\Gamma'$  does not depend on  $\beta$  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.

# Tracking dependencies



## Traditional view

- $\Gamma$  is a mapping together with a total ordering on its domain.

## Generalization

- Organize the context as a strict partial order.

## Relation to System F (with pack and unpack)

There is a subset  $F^{\forall-}$  with more restrictive dependencies

- System F is a subset of  $F^{\forall-}$
- There is a translation of pure expressions of  $F^{\forall-}$  to System F that
  - preserves the semantics, abstraction, and typings.
  - preserves  $\beta$ -reduction steps, but increases *let*-reduction steps.

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

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

# Conclusions

Type generativity can be explained by open existential types

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

Modelling of double-vision is already in  $F^\forall$  (omitted)

Extension to recursive values and types (with no expected difficulties)

Shapes bounded polymorphism and projections (complementary)

Good basis for a core calculus for a rich surface language with

- first-class, recursive and mixin modules and **no redundancies**.



# Appendix

7 Dependencies

8 Double vision

9 Related works

# Tracking dependencies



## Traditional view

- $\Gamma$  is a mapping together with a total ordering on its domain.

## Generalization

- Organize the context as a strict partial order.
- $\Gamma$  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.

## Zippping of contexts is redefined

- $(\mathcal{E}_1, \prec_1) \Downarrow (\mathcal{E}_2, \prec_2) = ((\mathcal{E}_1 \Downarrow \mathcal{E}_2), (\prec_1 \cup \prec_2)^+)$
- $\mathcal{E}_1 \Downarrow \mathcal{E}_2 = \{b_1 \Downarrow b_2 \mid b_1 \in \mathcal{E}_1, b_2 \in \mathcal{E}_2, dom b_1 = dom b_2\}$   
 $\cup \{\exists \beta \mid \beta \in dom \mathcal{E}_1 \Delta dom \mathcal{E}_2\}$   
 (weakening to remove unnecessary dependencies)

## Tracking dependencies



SIGMA

$$\mathcal{D}' \setminus (\{\beta\} \cup \text{dom } \Gamma') \subseteq \mathcal{D}$$

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

In particular,

- Free variables of the witness type  $\tau'$  are in  $\mathcal{D}'$  (by well-formedness).
- Those that are in  $\text{dom } \Gamma$  are not in  $\text{dom } \Gamma'$  and thus must be in  $\mathcal{D}$ .

## Tracking dependencies



SIGMA

$$\mathcal{D}' \setminus (\{\beta\} \cup \text{dom } \Gamma') \subseteq \mathcal{D}$$

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

Prevents typechecking:

$$\begin{aligned} \{ \ell_1 = \Sigma \langle \beta_1 \rangle (\alpha_1 = \beta_2 \rightarrow \beta_2) M_1 ; & \quad \textit{implies } \beta_1 \prec \beta_2 \\ \ell_2 = \Sigma \langle \beta_2 \rangle (\alpha_2 = \beta_1 \rightarrow \beta_1) M_2 \} & \quad \textit{implies } \beta_2 \prec \beta_1 \end{aligned}$$

But allows typechecking:

$$\begin{aligned} \{ \ell_1 = \Sigma \langle \beta_1 \rangle (\alpha_1 = \textit{int}) M_1 ; \\ \ell_2 = \Sigma \langle \beta_2 \rangle (\alpha_2 = \beta_1 \rightarrow \beta_1) M_2 \} \end{aligned}$$

## Tracking dependencies



OPEN

$$\frac{\Gamma \vdash M : \exists\beta.\tau \quad \mathcal{D} = \text{dom } \Gamma}{\Gamma, (\exists\beta \prec \mathcal{D}) \vdash \text{open } \langle\beta\rangle M : \tau}$$

LET

$$\frac{\begin{array}{c} \{\alpha \mid (\exists\alpha) \in \Gamma_2 \text{ and } (\forall\alpha) \in \Gamma_1\} \subseteq \mathcal{D} \\ \Gamma_1 \vdash M_1 : \tau_1 \quad \Gamma_2, (x : \tau_1 \prec \mathcal{D}) \vdash M_2 : \tau_2 \end{array}}{\Gamma_1 \curlywedge \Gamma_2 \vdash \text{let } x = M_1 \text{ in } M_2 : \tau_2}$$

**Open:**  $\alpha$  depends on all that precedes him, since the witness is unknown.

**Let:**  $x$  depends on all abstract types that are used in  $M_2$  and could be seen in  $M_1$ .

## Tracking dependencies



OPEN

$$\frac{\Gamma \vdash M : \exists\beta.\tau \quad \mathcal{D} = \text{dom } \Gamma}{\Gamma, (\exists\beta \prec \mathcal{D}) \vdash \text{open } \langle\beta\rangle M : \tau}$$

LET

$$\frac{\begin{array}{c} \{\alpha \mid (\exists\alpha) \in \Gamma_2 \text{ and } (\forall\alpha) \in \Gamma_1\} \subseteq \mathcal{D} \\ \Gamma_1 \vdash M_1 : \tau_1 \quad \Gamma_2, (x : \tau_1 \prec \mathcal{D}) \vdash M_2 : \tau_2 \end{array}}{\Gamma_1 \Downarrow \Gamma_2 \vdash \text{let } x = M_1 \text{ in } M_2 : \tau_2}$$

Prevents typechecking:

let  $x = \exists(\alpha = \beta \rightarrow \beta) M$  in *implies*  $x \prec \beta$ , since  $\beta \in \text{dom } \Gamma_2$   
 open  $\langle\beta\rangle x$  *implies*  $\beta \prec x$

# Double vision

This example is rejected

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

*We do not know that the external type  $\beta$  in the type of  $f$  is equal to the internal view  $\alpha$  also equal to  $\text{int}$ .*

Keep this information in the context

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

and use it whenever needed

$$\frac{\text{SIM} \quad \Gamma \vdash M : \tau' \quad \Gamma \vdash \tau \triangleleft \tau'}{\Gamma \vdash M : \tau}$$

# Comparisson with Derek's RTG

The primitives are similar, with small differences

Fzip	RTG
$\nu\alpha. M$	new $\alpha$ in $M$
$\Sigma \langle\alpha\rangle (\alpha = \tau) M$	set $\alpha := \tau$ in $M$
$\exists\alpha. M$	$\Lambda\alpha \uparrow K. \lambda(: ())\mathbb{1} M$
open $\langle\alpha\rangle M$	$M[\alpha] () M$

- We evaluate under existentials while RTG does not.
- RTG uses  $F^\omega$  while we restrict to System F.
- RTG allows recursive values and types, while we do not.

# Comparisson with Derek's RTG

The primitives are similar, with small differences

Fzip	RTG
$\nu\alpha. M$	new $\alpha$ in $M$
$\Sigma \langle\alpha\rangle (\alpha = \tau) M$	set $\alpha := \tau$ in $M$
$\exists\alpha. M$	$\Lambda\alpha \uparrow K. \lambda(: ()) \perp M$
open $\langle\alpha\rangle M$	$M[\alpha] () M$

- We evaluate under existentials while RTG does not.
- RTG uses  $F^\omega$  while we restrict to System F.
- RTG allows recursive values and types, while we do not.

## Shared ideas with RTG

- Use of linear types  
(only in typing contexts in Fzip, exposed in RTG.)
- Similar decomposition of constructs  
(by design in Fzip, observed a posteriori in RTG.)

# Comparisson with Derek's RTG

The primitives are similar, with small differences

Shared ideas with RTG

The “inside” differs significantly

- Typechecking in RTG uses an abstract machine that performs side effects into a global store.
- Unintuitive for programmers (who can't run the machine mentally).
- Looses the connection with logic.
- Does not isolate type abstraction from the use of recursive types.

The motivations and uses also differs

- Designed and used as an internal language (opposite to our goals)
- Used to model recursive and mixin modules (complementary)

# Other related works

## Rossberg (2003)

Introduces  $\lambda_N$ , a version of System-F to define abstract types, that can automatically be extruded to allow sharper type analysis.

- Many similarities in spirit with our  $\Sigma$  binder.
- But the motivations and technical details are quite different.  
In particular, parametricity is purposely violated in  $\lambda_N$ .

## Russo (2003)

- He first explained that paths are meaningless for module types.
- He interpretes modules and signatures into semantic objects within  $F^\omega$ .
- However
  - his existential types are implicitly opened.
  - no dynamic semantics for objects.