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 λ-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 langage of a language with:
  - first-class modules
  - that is conceptually economical, *e.g.* avoids dupplication 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.

$$\begin{tikzcd} & A \arrow[dl] \arrow[dr] & \\ B \arrow[dr] & & C \arrow[dl] \\ & D & \end{tikzcd} \quad \text{or} \quad \begin{tikzcd} A \arrow[d] & & A \arrow[d] \\ B \arrow[dr] & & C \arrow[dl] \\ & D & \end{tikzcd}$$

Yes No

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 elabored.

Elaboration semantics

- Elaboration is a compilation process, may be of arbitrary complexity.
- The user cannot perform it mentally.
- Looses 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.

$\mathsf{F}^{\curlyvee}$ (Fzip): a variant of $\textsc{Rtg}$ 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 $\mathsf{F}^{\curlyvee}$)

- 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.

Beyong $\mathsf{F}^{\curlyvee}$

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

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

UNPACK

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

Splitting unpack

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

$$\triangleq$$

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

Limits the scope of α

Uses α for the abstract type of M

Binds M to x in M'

Splitting unpack

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

Splitting unpack advantages

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

M need not be at toplevel.

Splitting unpack advantages

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

α need not be hidden immediately.

Splitting unpack advantages

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

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

Typechecking

Must forbid incorrect programs such as

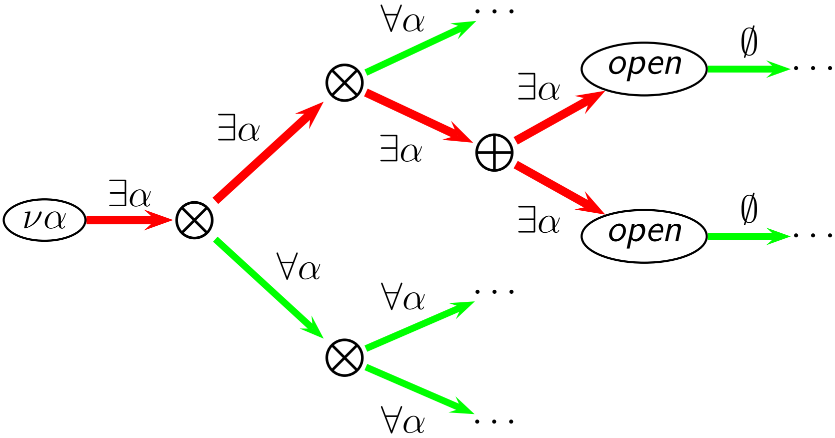
$$\nu\alpha.\quad \text{let } x = \boxed{\text{open } \langle\alpha\rangle\, M_1} \text{ in}$$

$$\boxed{\text{open } \langle\alpha\rangle\, M_2}$$

X

ok $\quad M\,[\alpha]\; x$

Typechecking



Typechecking

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

Typechecking

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

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

Typechecking

Open

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

Let

$$\frac{\Gamma_1 \vdash M_1 \;:\; \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash M_2 \;:\; \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 \;:\; \tau_2}$$

Nu

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

Typechecking

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

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

LET
$$\frac{\Gamma_1 \vdash M_1 : \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \tau_2}$$

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

Typechecking Zipping

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

$$\begin{array}{ccccc} \forall\alpha & \curlyvee & \forall\alpha & = & \forall\alpha \\ \forall\alpha & \curlyvee & \exists\alpha & = & \exists\alpha \\ \exists\alpha & \curlyvee & \forall\alpha & = & \exists\alpha \end{array}$$

$$x : \tau \curlyvee x : \tau = x : \tau$$

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

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

Splitting pack

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

$$\triangleq$$

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

makes α abstract
with witness τ

converts the type of M
using the equation(s)

Splitting pack

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

$$\triangleq$$

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

closes the abstract type β

converts the type of M

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

Splitting pack

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

$$\triangleq$$

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

Splitting pack

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

$$\triangleq$$

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

A module with an open abstract type β.

Splitting pack

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

A sub -module with an open abstract type β.

Typechecking

Exists

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

Typechecking

Exists

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

Open

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

Typechecking

Sigma

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

Exists

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

Open

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

Typechecking

Coerce

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

uses

Sigma

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

Exists

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

Summary

Types are unchanged

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

Exressions are

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

Examples

In ML:

$$\text{module } X = \text{struct} \begin{pmatrix} \text{type } t = \mathit{int} \\ \text{val } z = 0 \\ \text{val } s = \lambda(x : \mathit{int}) x{+}1 \end{pmatrix} : \text{sig} \begin{pmatrix} \text{type } t \\ \text{val } z : t \\ \text{val } s : t \to t \end{pmatrix}$$

In Fzip:

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

Examples

In ML:

$$\text{module } X = \text{struct} \begin{pmatrix} \text{type } t = int \\ \text{val } z = 0 \\ \text{val } s = \lambda(x : int) x{+}1 \end{pmatrix} : \text{sig} \begin{pmatrix} \text{type } t \\ \text{val } z : t \\ \text{val } s : t \to t \end{pmatrix}$$

In Fzip:

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

Examples

In ML:

Making generative views of x

In Fzip:

$$\text{let } x = \exists(\alpha = \mathit{int}) \left(\left\{ \begin{array}{l} z = 0\ ; \\ s = \lambda(x : \mathit{int}) x{+}1 \end{array} \right\} : \left\{ \begin{array}{l} z : \alpha\ ; \\ s : \alpha \rightarrow \alpha \end{array} \right\} \right) \text{ in}$$
$$\text{let } x_1 = \text{open } \langle \beta_1 \rangle\ x \text{ in}$$
$$\text{let } x_2 = \text{open } \langle \beta_2 \rangle\ x \text{ in}$$
$$\ldots$$

Examples

Functors

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

Example

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

Reduction

Problem (well-known)

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

Solution (new)

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

Reduction

Example

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

$$\downarrow$$

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

$$\downarrow$$

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

$$\downarrow$$

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

Reduction Values

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

Definition

Values

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

Results

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

Note

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

Reduction Semantics

Call-by-value small-step reduction semantics

Elimination rules: β-reduction rules plus,

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

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

\+ 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* $M \leadsto M'$, *then* $\Gamma \vdash M' : \tau$.

Theorem (Progress)

If $\Gamma \vdash M : \tau$ *and* Γ *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:

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

reduces to:

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

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

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

External recursion, through open witness definitions:

$$\{\ell_1 = \Sigma\, \langle\beta_1\rangle\, (\alpha_1 = \beta_2 \to \beta_2)\, M_1\ ;$$
$$\ell_2 = \Sigma\, \langle\beta_2\rangle\, (\alpha_2 = \beta_1 \to \beta_1)\, M_2\ \}$$

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

Cannot occur in System F.

The appearance of recursive types

Origin of the problem

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

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

Solutions

1. Remove $\boxed{\forall\beta}$ from the premisse:
   - 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.

Tracking dependencies

▶ more

Traditional view

- Γ 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 $\mathsf{F}^{\curlyvee -}$ with more restrictive dependencies

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

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.

Conclusions

Type generativity can be explained by open existential types

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

Modelling of double-vision is already in $F^{\curlyvee}$ (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

Tracking dependencies

◀ back

Traditional view

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

Generalization

- Organize the context as a strict partial order.
- Γ is a pair $(\mathcal{E}, \prec)$ where $\mathcal{E}$ is a set of bindings ordered by $\prec$.
- We write $\Gamma, (b \prec \mathcal{D}), \Gamma'$ when
  - $dom\,\Gamma \not\prec b$ and $b \not\prec dom\,\Gamma'$ and $\mathcal{D}$ is the set b depends on.

Zipping of contexts is redefined

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

Tracking dependencies

◀ back

Sigma

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

In particular,

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

Tracking dependencies

◀ back

Sigma

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

Prevents typechecking:

$$\begin{array}{ll} \{\ell_1 = \Sigma \langle \beta_1 \rangle \, (\alpha_1 = \beta_2 \to \beta_2)\, M_1 \; ; & \textit{implies } \beta_1 \prec \beta_2 \\ \;\ell_2 = \Sigma \langle \beta_2 \rangle \, (\alpha_2 = \beta_1 \to \beta_1)\, M_2 \; \} & \textit{implies } \beta_2 \prec \beta_1 \end{array}$$

But allows typechecking:

$$\begin{array}{l} \{\ell_1 = \Sigma \langle \beta_1 \rangle \, (\alpha_1 = \mathit{int})\, M_1 \; ; \\ \;\ell_2 = \Sigma \langle \beta_2 \rangle \, (\alpha_2 = \beta_1 \to \beta_1)\, M_2 \; \} \end{array}$$

Tracking dependencies

◀ back

$$\textsc{Open}\ \dfrac{\Gamma \vdash M : \exists \beta.\, \tau \qquad \boxed{\mathcal{D} = dom\, \Gamma}}{\Gamma, (\exists \beta \prec \mathcal{D}) \vdash \text{open}\ \langle \beta \rangle\ M : \tau}$$

$$\textsc{Let}\ \dfrac{\begin{array}{c}\boxed{\{\alpha \mid (\exists \alpha) \in \Gamma_2 \text{ and } (\forall \alpha) \in \Gamma_1\} \subseteq \mathcal{D}} \\ \Gamma_1 \vdash M_1 : \tau_1 \qquad \Gamma_2, (x : \tau_1 \prec \mathcal{D}) \vdash M_2 : \tau_2\end{array}}{\Gamma_1 \curlyvee \Gamma_2 \vdash \text{let}\ x = M_1\ \text{in}\ M_2 : \tau_2}$$

Open: α 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

◀ back

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

$$\textsc{Let}\quad \frac{\boxed{\{\alpha \mid (\exists\alpha) \in \Gamma_2 \text{ and } (\forall\alpha) \in \Gamma_1\} \subseteq \mathcal{D}} \quad \Gamma_1 \vdash M_1 : \tau_1 \qquad \Gamma_2, (x : \tau_1 \prec \mathcal{D}) \vdash M_2 : \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \tau_2}$$

Prevents typechecking:

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

Double vision

This example is rejected

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

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

Keep this information in the context

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

and use it whenever needed

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

Comparisson with Derek's RTG

The primitives are similar, with small differences

Fzip	RTG
$\nu\alpha.\, M$	new α 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^ω 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 α 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^ω 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 λ_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 Σ binder.
- But the motivations and technical details are quire different.
  In particular, parametricity is purposedly violates in λ_N.

Russo (2003)

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