On the expressive power of types

Sam Lindley

The University of Edinburgh

TYPES 2022
What do we mean by expressive power?

Some possible answers:

▶ Computability
▶ Algorithmic complexity
▶ Macro expressiveness

Matthias Felleisen
[Felleisen, 1990, "On the expressive power of programming languages"]
What do we mean by expressive power?

Some possible answers:

- Computability
What do we mean by expressive power?

Some possible answers:

- Computability
- Algorithmic complexity
What do we mean by expressive power?

Some possible answers:

▶ Computability
▶ Algorithmic complexity
▶ Macro expressiveness

Matthias Felleisen

Quiz

Can function types encode product types?

Can positive iso-recursive types encode positive equi-recursive types?

Can existential types encode universal types?

Can simple-typed lambda calculus encode System F?

Can row polymorphism encode row subtyping?
Quiz

Can function types encode product types?
Can function types encode product types?

Can positive iso-recursive types encode positive equi-recursive types?
Quiz

Can function types encode product types?

Can positive iso-recursive types encode positive equi-recursive types?

Can existential types encode universal types?
Quiz

Can function types encode product types?

Can positive iso-recursive types encode positive equi-recursive types?

Can existential types encode universal types?

Can simple-typed lambda calculus encode System F?
Quiz

Can function types encode product types?

Can positive iso-recursive types encode positive equi-recursive types?

Can existential types encode universal types?

Can simple-typed lambda calculus encode System F?

Can row polymorphism encode row subtyping?
Part I

Products
Motivation

The standard Church encoding for a pair can only be ascribed a type in simply-typed lambda calculus if both components of the pair have the same type.
The standard Church encoding for a pair can only be ascribed a type in simply-typed lambda calculus if both components of the pair have the same type.

Do alternative simply-typed encodings exist for heterogeneous pairs?
Call-by-name CPS

\[ \mathcal{N}[A] = (A^* \to R) \to R \]
\[ X^* = X \]
\[ (A \times B)^* = \mathcal{N}[A] \times \mathcal{N}[B] \]
\[ (A \to B)^* = \mathcal{N}[A] \to \mathcal{N}[B] \]
Call-by-name CPS

\[ \mathcal{N}[X] = (X \to R) \to R \]
\[ \mathcal{N}[A \to B] = ((\mathcal{N}[A] \to \mathcal{N}[B]) \to R) \to R \]
\[ \mathcal{N}[A \times B] = ((\mathcal{N}[A] \times \mathcal{N}[B]) \to R) \to R \]
Call-by-name CPS

\[\mathcal{N}[X] = (X \rightarrow R) \rightarrow R\]
\[\mathcal{N}[A \rightarrow B] = ((\mathcal{N}[A] \rightarrow \mathcal{N}[B]) \rightarrow R) \rightarrow R\]
\[\mathcal{N}[A \times B] = ((\mathcal{N}[A] \times \mathcal{N}[B]) \rightarrow R) \rightarrow R\]
\[\mathcal{N}[\lambda x. M] = \lambda k. x \ k\]
\[\mathcal{N}[\lambda x. M] = \lambda k. k \ (\lambda x. \mathcal{N}[M])\]
\[\mathcal{N}[M N] = \lambda k. \mathcal{N}[M] \ (\lambda f. f \ \mathcal{N}[N] \ k)\]
\[\mathcal{N}[\text{pair } M N] = \lambda k. k \ (\text{pair } \mathcal{N}[M] \ \mathcal{N}[N])\]
\[\mathcal{N}[\text{fst } M] = \lambda k. \mathcal{N}[M] \ (\lambda p. (\text{fst } p) \ k)\]
\[\mathcal{N}[\text{snd } M] = \lambda k. \mathcal{N}[M] \ (\lambda p. (\text{snd } p) \ k)\]
Call-by-name CPS

Products are encodable via a curried **global** CPS translation

\[
\begin{align*}
C[X] &= (X \rightarrow R) \rightarrow R \\
C[A \rightarrow B] &= (((C[A] \rightarrow C[B]) \rightarrow R) \rightarrow R \\
C[A \times B] &= (C[A] \rightarrow C[B] \rightarrow R) \rightarrow R \\
C[x] &= \lambda k. x \ k \\
C[\lambda x. M] &= \lambda k. k (\lambda x. C[M]) \\
C[M N] &= \lambda k. C[M] (\lambda f. f C[N] \ k) \\
C[\text{pair } M N] &= \lambda k. k C[M] C[N] \\
C[\text{fst } M] &= \lambda k. C[M] (\lambda x. \lambda y. x \ k) \\
C[\text{snd } M] &= \lambda k. C[M] (\lambda x. \lambda y. y \ k)
\end{align*}
\]

What about a **local** encoding?
Call-by-name CPS

Products are encodable via a curried **global** CPS translation

\[
\begin{align*}
C[X] &= (X \rightarrow R) \rightarrow R \\
C[A \rightarrow B] &= ((C[A] \rightarrow C[B]) \rightarrow R) \rightarrow R \\
C[A \times B] &= (C[A] \rightarrow C[B] \rightarrow R) \rightarrow R \\
C[x] &= \lambda k.x \ k \\
C[\lambda x.M] &= \lambda k.k \ (\lambda x.C[M]) \\
C[M \ N] &= \lambda k.C[M] \ (\lambda f.f \ C[N] \ k) \\
C[\text{pair } M \ N] &= \lambda k.k \ C[M] \ C[N] \\
C[\text{fst } M] &= \lambda k.C[M] \ (\lambda x.\lambda y.x \ k) \\
C[\text{snd } M] &= \lambda k.C[M] \ (\lambda x.\lambda y.y \ k)
\end{align*}
\]

What about a **local** encoding?
Localising CPS

Untyped

\[ U[pair \ M \ N] = \lambda s. \ U[M] \ U[N] \]
\[ U[fst \ M] = U[M] (\lambda x. \lambda y. x) \]
\[ U[snd \ M] = U[M] (\lambda x. \lambda y. y) \]
Localising CPS

Untyped

\[\mathcal{U}[\text{pair } M N] = \lambda s. s \mathcal{U}[M] \mathcal{U}[N]\]
\[\mathcal{U}[\text{fst } M] = \mathcal{U}[M] (\lambda x. \lambda y. x)\]
\[\mathcal{U}[\text{snd } M] = \mathcal{U}[M] (\lambda x. \lambda y. y)\]

Simply typed — homogeneous products

\[\mathcal{H}[A \times A] = (\mathcal{H}[A] \rightarrow \mathcal{H}[A] \rightarrow \mathcal{H}[A]) \rightarrow \mathcal{H}[A]\]
\[\mathcal{H}[\text{pair } M^A N^A] = \lambda s. \mathcal{H}[A] \rightarrow \mathcal{H}[A] \rightarrow \mathcal{H}[A]. s \mathcal{H}[M] \mathcal{H}[N]\]
\[\mathcal{H}[\text{fst } M^{A \times A}] = \mathcal{H}[M] (\lambda x. \mathcal{H}[A] \rightarrow \mathcal{H}[A]. x)\]
\[\mathcal{H}[\text{snd } M^{A \times A}] = \mathcal{H}[M] (\lambda x. \mathcal{H}[A] \rightarrow \mathcal{H}[A]. y)\]
Localising CPS

Untyped

\[
\begin{align*}
U[\text{pair } M \ N] &= \lambda s . U[M] \ U[N] \\
U[\text{fst } M] &= U[M] \ (\lambda x . \lambda y . x) \\
U[\text{snd } M] &= U[M] \ (\lambda x . \lambda y . y)
\end{align*}
\]

Simply typed — homogeneous products

\[
\begin{align*}
H[A \times A] &= (H[A] \rightarrow H[A] \rightarrow H[A]) \rightarrow H[A] \\
H[\text{pair } M^A \ N^A] &= \lambda s . H[A] \rightarrow H[A] \rightarrow H[A] . s \ H[M] \ H[N] \\
H[\text{fst } M^{A \times A}] &= H[M] \ (\lambda x . H[A] . \lambda y . H[A] . x) \\
H[\text{snd } M^{A \times A}] &= H[M] \ (\lambda x . H[A] . \lambda y . H[A] . y)
\end{align*}
\]

Polymorphic

\[
\begin{align*}
F[A \times B] &= \forall Z. (F[A] \rightarrow F[B] \rightarrow Z) \rightarrow Z \\
F[\text{pair}_{A,B} \ M \ N] &= \exists Z . \lambda s . F[A] \rightarrow F[B] \rightarrow Z . s \ F[M] \ F[N] \\
F[\text{fst}_{A,B} \ M] &= F[M] \ F[A] \ (\lambda x . F[A] . \lambda y . F[B] . x) \\
\end{align*}
\]
No local encoding of $X \times Y$

We seek $\beta$-normal forms $\text{fst}_{X,Y}$ and $\text{snd}_{X,Y}$ such that:

\[
\begin{align*}
[p : X \times Y \vdash \text{fst } p : X] & = p : [X \times Y] \vdash \text{fst}_{X,Y} : X \\
[p : X \times Y \vdash \text{snd } p : Y] & = p : [X \times Y] \vdash \text{snd}_{X,Y} : Y
\end{align*}
\]
No local encoding of \( X \times Y \)

We seek \( \beta \)-normal forms \( \text{fst}_{X,Y} \) and \( \text{snd}_{X,Y} \) such that:

\[
\begin{align*}
[p : X \times Y \vdash \text{fst} \ x : X] & = p : [X \times Y] \vdash \text{fst}_{X,Y} : X \\
[p : X \times Y \vdash \text{snd} \ x : Y] & = p : [X \times Y] \vdash \text{snd}_{X,Y} : Y
\end{align*}
\]

So we must have \( m, n, M_1, \ldots, M_m, N_1, \ldots, N_n \) such that:

\[
\begin{align*}
\text{fst}_{X,Y} & = p \ M_1 \ \ldots \ \ M_m \\
\text{snd}_{X,Y} & = p \ N_1 \ \ldots \ \ N_n
\end{align*}
\]
No local encoding of $X \times Y$

We seek $\beta$-normal forms $\text{fst}_{X,Y}$ and $\text{snd}_{X,Y}$ such that:

$$
\begin{align*}
[p : X \times Y \vdash \text{fst} \ p : X] & = p : [X \times Y] \vdash \text{fst}_{X,Y} : X \\
[p : X \times Y \vdash \text{snd} \ p : Y] & = p : [X \times Y] \vdash \text{snd}_{X,Y} : Y
\end{align*}
$$

So we must have $m, n, M_1, \ldots, M_m, N_1, \ldots, N_n$ such that:

$$
\begin{align*}
\text{fst}_{X,Y} & = p \ M_1 \ldots \ M_m \\
\text{snd}_{X,Y} & = p \ N_1 \ldots \ N_n
\end{align*}
$$

The typing rule for application means that we also have

$$
A_1 \to \cdots \to A_m \to X = [X \times Y] = B_1 \to \cdots \to B_n \to Y
$$

where

$$
\begin{align*}
(p : [X \times Y] \vdash M_i : A_i)_{1 \leq i \leq m} \\
(p : [X \times Y] \vdash N_j : B_i)_{1 \leq j \leq n}
\end{align*}
$$
No local encoding of $X \times Y$

We seek $\beta$-normal forms $\text{fst}_{X,Y}$ and $\text{snd}_{X,Y}$ such that:

$$
\begin{align*}
[p : X \times Y \vdash \text{fst} \; p : X] &= p : [X \times Y] \vdash \text{fst}_{X,Y} : X \\
[p : X \times Y \vdash \text{snd} \; p : Y] &= p : [X \times Y] \vdash \text{snd}_{X,Y} : Y
\end{align*}
$$

So we must have $m, n, M_1, \ldots, M_m, N_1, \ldots, N_n$ such that:

$$
\begin{align*}
\text{fst}_{X,Y} &= p \; M_1 \ldots M_m \\
\text{snd}_{X,Y} &= p \; N_1 \ldots N_n
\end{align*}
$$

The typing rule for application means that we also have

$$
A_1 \to \cdots \to A_m \to X = [X \times Y] = B_1 \to \cdots \to B_n \to Y
$$

where

$$(p : [X \times Y] \vdash M_i : A_i)_{1 \leq i \leq m}$$
$$(p : [X \times Y] \vdash N_j : B_i)_{1 \leq j \leq n}$$

But these equations could only hold if $X$ and $Y$ were the same type!
Hang on a minute!

Type-indexed local encodings of products are well-known in PCF and System T. Examples:

- [Longley and Normann, 2015]
- [Kiselyov, 2021]

http://okmij.org/ftp/Computation/simple-encodings.html#product

How do we reconcile the existence of such encodings with the non-existence result?
No local encoding of $X \times (X \to X)$

Consider $X \times (X \to X)$. We seek $\beta$-normal forms $\text{fst}_{X, X\to X}$ and $\text{snd}_{X, X\to X}$ such that:

$$\llbracket p : X \times (X \to X) \vdash \text{fst } p : X \rrbracket = p : \llbracket X \times (X \to X) \rrbracket \vdash \text{fst}_{X, X\to X} : X$$

$$\llbracket p : X \times (X \to X) \vdash \text{snd } p : X \to X \rrbracket = p : \llbracket X \times (X \to X) \rrbracket \vdash \text{snd}_{X, X\to X} : X \to X$$
No local encoding of $X \times (X \to X)$

Consider $X \times (X \to X)$. We seek $\beta$-normal forms $\text{fst}_{X, X \to X}$ and $\text{snd}_{X, X \to X}$ such that:

$$\begin{align*}
[p : X \times (X \to X) \vdash \text{fst} \ p : X] &= p : [X \times (X \to X)] \vdash \text{fst}_{X, X \to X} : X \\
[p : X \times (X \to X) \vdash \text{snd} \ p : X \to X] &= p : [X \times (X \to X)] \vdash \text{snd}_{X, X \to X} : X \to X
\end{align*}$$

As before, $\text{fst}_{X, X \to X}$ must be of the form

$$p \ M_1 \ \ldots \ M_m$$

and hence:

$$[X \times (X \to X)] = A_1 \to \cdots \to A_m \to X$$
Two choices for $\text{snd}_{X,X \rightarrow X}$:
Two choices for \( \text{snd}_{X,X \to X} \):

1. \( \text{snd}_{X,X \to X} : p \ N_1 \ldots N_{m-1} \)
Two choices for $\text{snd}_{X, X \to X}$:

1. $p \ N_1 \ldots \ N_{m-1}$
   \[ \implies A_m = X \text{ and } M_m = p \ M'_1 \ldots \ M'_m \]
   \[ M'_m = p \ M''_1 \ldots \ M''_m \]
   ...

No finite such $\text{snd}$ can exist.

No lambda abstraction can be $\beta$-converted to $y$. 
No local encoding of $X \times (X \to X)$ (continued)

Two choices for $\text{snd}_{X, X \to X}$:

1. $p \ N_1 \ \ldots \ N_{m-1}$
   
   $\implies A_m = X$ and $M_m = p \ M'_1 \ \ldots \ M'_m$
   
   $M'_m = p \ M''_1 \ \ldots \ M''_m$

   ... 

   No finite such $\text{snd}$ can exist.
No local encoding of $X \times (X \to X)$ (continued)

Two choices for $\text{snd}_{X,X\to X}$:

1. $p\ N_1\ \ldots\ N_{m-1}$
   \[\implies \ A_m = X\text{ and } M_m = p\ M'_1\ \ldots\ M'_m\]
   \[M'_m = p\ M''_1\ \ldots\ M''_m\]
   
   No finite such $\text{snd}$ can exist.

2. $\lambda z. N'$
No local encoding of $X \times (X \to X)$ (continued)

Two choices for $\text{snd}_{X, X \to X}$:

1. $p \ N_1 \ldots \ N_{m-1}$
   \[ \implies A_m = X \text{ and } M_m = p \ M'_1 \ldots \ M'_m \]
   \[ M'_m = p \ M''_1 \ldots \ M''_m \]
   ... 
   No finite such $\text{snd}$ can exist.

2. $\lambda z. N'$
   \[ \implies [\text{snd (pair x y)}] = \lambda z. N'[\text{[pair x y]}/p] \]
Two choices for $\text{snd}_{X, X \to X}$:

1. $p \, N_1 \ldots \, N_{m-1}$
   $\implies A_m = X$ and $M_m = p \, M'_1 \ldots \, M'_m$
   $M'_m = p \, M''_1 \ldots \, M''_m$
   
   No finite such $\text{snd}$ can exist.

2. $\lambda z. N'$

   $\implies [\text{snd}(\text{pair} \, x \, y)] = \lambda z. N'[\text{pair} \, x \, y / p]$

   No lambda abstraction can be $\beta$-converted to $y$. 
Local encoding of $X \times (X \to X)$ with $\eta$

\[
\begin{align*}
E[X \times (X \to X)] &= (X \to (X \to X) \to X) \to X \\
E[\text{pair } M^X N^{X \to X}] &= \lambda f. f \ E[M] \ E[N] \\
E[\text{fst } M^{X \times (X \to X)}] &= E[M] (\lambda x. y.x) \\
E[\text{snd } M^{X \times (X \to X)}] &= \lambda z. E[M] (\lambda x. y. y z)
\end{align*}
\]
Local encoding of $X \times (X \to X)$ with $\eta$

\[
\begin{align*}
\mathcal{E}[X \times (X \to X)] &= (X \to (X \to X) \to X) \to X \\
\mathcal{E}[\text{pair } M^X N^{X \to X}] &= \lambda f. f \mathcal{E}[M] \mathcal{E}[N] \\
\mathcal{E}[\text{fst } M^{X \times (X \to X)}] &= \mathcal{E}[M] (\lambda x y. x) \\
\mathcal{E}[\text{snd } M^{X \times (X \to X)}] &= \lambda z. \mathcal{E}[M] (\lambda x y. y z)
\end{align*}
\]

Now we have

\[
\begin{align*}
\mathcal{E}[\text{fst } (\mathcal{E}[\text{pair } x y])] &\sim_\beta x \\
\mathcal{E}[\text{snd } (\mathcal{E}[\text{pair } x y])] &\sim_\beta \lambda z. y z \sim_\eta y
\end{align*}
\]
Local encoding of $A \times B$ with $\eta$ and a single base type $X$

$$\mathcal{E}[A \times B] = (\mathcal{E}[A] \rightarrow \mathcal{E}[B] \rightarrow X) \rightarrow X$$

$$\mathcal{E}[^{\text{pair }} MN] = \lambda f.f \mathcal{E}[M] \mathcal{E}[N]$$

$$\mathcal{E}[^{\text{fst}} M^{A_1 \rightarrow \cdots A_n \rightarrow X,B}] = \lambda z_1 \ldots z_n.\mathcal{E}[M] \ (\lambda x \ y. x \ z_1 \ldots z_n)$$

$$\mathcal{E}[^{\text{snd}} M^{A,B_1 \rightarrow \cdots B_n \rightarrow X}] = \lambda z_1 \ldots z_n.\mathcal{E}[M] \ (\lambda x \ y. y \ z_1 \ldots z_n)$$
Local encoding of $A \times B$ with $\eta$ and a single base type $X$

$$\mathcal{E}[A \times B] = (\mathcal{E}[A] \to \mathcal{E}[B] \to X) \to X$$

$$\mathcal{E}[^{\text{pair}} M N] = \lambda f. f \mathcal{E}[M] \mathcal{E}[N]$$

$$\mathcal{E}[^{\text{fst}} M^{A_1 \to \ldots A_n \to X, B}] = \lambda z_1 \ldots z_n. \mathcal{E}[M] (\lambda x y.x z_1 \ldots z_n)$$

$$\mathcal{E}[^{\text{snd}} M^{A, B_1 \to \ldots B_n \to X}] = \lambda z_1 \ldots z_n. \mathcal{E}[M] (\lambda x y.y z_1 \ldots z_n)$$

This is a **type-indexed** local encoding.
Can function types encode product types?

It depends...

- Parametric global CPS encoding
- Parametric local Church encodings
- Untyped
- Simple types, but only homogeneous products
- Polymorphic
- Multiple base types — no local encoding
- Single base type without $\eta$ — no local encoding
- Single base type with $\eta$ — type-indexed local encoding

Incidentally, none of these encodings preserves the $\eta$-rule for products.
Can function types encode product types?

It depends...
Can function types encode product types?

It depends...

- Parametric global CPS encoding
- Parametric local Church encodings
  - untyped
  - simple types, but only homogeneous products
  - polymorphic
- Multiple base types — no local encoding
- Single base type without $\eta$ — no local encoding
- Single base type with $\eta$ — type-indexed local encoding

Incidentally, none of these encodings preserves the $\eta$-rule for products.
Can function types encode product types?

It depends...

- Parametric global CPS encoding
- Parametric local Church encodings
  - untyped
  - simple types, but only homogeneous products
  - polymorphic
- Multiple base types — no local encoding
- Single base type without $\eta$ — no local encoding
- Single base type with $\eta$ — type-indexed local encoding

Incidentally, none of these encodings preserves the $\eta$-rule for products.
Part II

Recursive types
Motivation

[Dolan and Mycroft, 2017, “Polymorphism, subtyping, and type inference in MLsub”]
Motivation

[Dolan and Mycroft, 2017, “Polymorphism, subtyping, and type inference in MLsub”]

MLsub relies on simulating general equi-recursive types by positive equi-recursive types.
Motivation

[Dolan and Mycroft, 2017, “Polymorphism, subtyping, and type inference in MLsub”]

MLsub relies on simulating general equi-recursive types by positive equi-recursive types.

General equi-recursive types can type non-terminating programs.
Motivation

[Dolan and Mycroft, 2017, “Polymorphism, subtyping, and type inference in MLsub”]

MLsub relies on simulating general equi-recursive types by positive equi-recursive types.

General equi-recursive types can type non-terminating programs.

Positive iso-recursive types can be encoded in System F.
Motivation

[Dolan and Mycroft, 2017, “Polymorphism, subtyping, and type inference in MLsub”]

MLsub relies on simulating general equi-recursive types by positive equi-recursive types.

General equi-recursive types can type non-terminating programs.

Positive iso-recursive types can be encoded in System F.

Why the discrepancy between equi and iso?
Recursive types

Algebraic datatypes

\[
\begin{align*}
\text{data Nat} & = \ Z \mid \text{S Nat} \\
\text{data List} & = \ \text{Nil} \mid \text{Cons Int List} \\
\text{data Tree} & = \ \text{Leaf} \mid \text{Node Tree Int Tree}
\end{align*}
\]

Inline recursive types

\[
\begin{align*}
\text{Nat} & = \mu X.1 + X \\
\text{List} & = \mu X.1 + \text{Int} \times X \\
\text{Tree} & = \mu X.1 + X \times \text{Int} \times X
\end{align*}
\]

Recursive type equations

\[
\begin{align*}
\text{Nat} & = 1 + \text{Nat} \\
\text{List} & = 1 + \text{Int} \times \text{List} \\
\text{Tree} & = 1 + \text{Tree} \times \text{Int} \times \text{Tree}
\end{align*}
\]
Recursive types as regular trees

Nat \quad \mu X.1 + X

1

\begin{center}
\begin{tikzpicture}
\node (1) at (0,0) {1};
\node (x) at (1,0) {\mu X.1 + X};
\node (nat) at (-2,0) {Nat};
\draw[->] (1) to (x);
\draw[->] (1) to (x);
\end{tikzpicture}
\end{center}
Recursive types as regular trees

\[
\text{Nat} \quad \mu X. 1 + X
\]

\[
\text{List} \quad \mu X. 1 + \text{Int} \times X
\]
Recursive types as regular trees

Nat \[ \mu X. 1 + X \]

List \[ \mu X. 1 + \text{Int} \times X \]

Tree \[ \mu X. 1 + X \times \text{Int} \times X \]
Equi-recursive types

\[ \Gamma \vdash M : A \quad \vdash A \approx B \]

\[ \vdash A \approx B \text{ means} \]

A and B are equivalent up to infinite unrolling
Equi-recursive types

\[
\Gamma \vdash M : A \quad \vdash A \approx B
\]

\[\vdash A \approx B \text{ means}\]

\[A \text{ and } B \text{ are equivalent up to infinite unrolling}\]

Example

\[\mu X.1 + X \quad \approx \quad 1 + \mu X.1 + X \quad \approx \quad \mu X.1 + (1 + X)\]
Deciding equality by unrolling

\[ \Phi \vdash a \approx b \]

Repeat

\[ \Phi, a \approx b \vdash a \approx b \]

Rec-Cons

\[ \text{label}(a) = \text{label}(b) \]

\[ \Phi, a \approx b \vdash \text{children}(a) \approx \text{children}(b) \]

\[ \Phi \vdash a \approx b \]

\Phi a \text{ set – comma is disjoint extension}
Deciding equality by unrolling

\[ a \approx b \]

\[ a \approx b \vdash a \approx b \]

Rec-Cons

\[ \text{label}(a) = \text{label}(b) \]

\[ \Phi, a \approx b \vdash \text{children}(a) \approx \text{children}(b) \]

\[ \Phi \vdash a \approx b \]

\[ \Phi \text{ a set — comma is disjoint extension} \]
Deciding equality by unrolling

\[ a \approx c \]
Deciding equality by unrolling

\[ a \overset{+}{\to} b \]

\[ 1 \overset{a}{\to} c \]

\[ \Phi \vdash a \approx b \]

\[ \text{Repeat } \Phi, \ a \approx b \vdash a \approx b \]

\[ \text{Rec-Cons} \]

\[ \text{label}(a) = \text{label}(b) \]

\[ \Phi, a \approx b \vdash \text{children}(a) \approx \text{children}(b) \]

\[ \Phi \vdash a \approx b \]

\[ \Phi \text{ a set — comma is disjoint extension} \]

\[ a \approx c \]
Deciding equality by unrolling

\[ a \approx_1 b \] \quad \Phi \vdash a \approx b \\
Repeat \ \Phi, \ a \approx b \vdash a \approx b \\
Rec-Cons \ label(a) = label(b) \\
\Phi \vdash \text{children}(a) \approx \text{children}(b) \\
\Phi \vdash a \approx b \\
\Phi \text{ a set — comma is disjoint extension}
Deciding equality by unrolling

\[ a \approx c, \ a \approx e \]
Deciding equality by unrolling

\[
a \approx c, \ a \approx e
\]

\[\Phi \vdash a \approx b\]

**Repeat**

\[
\frac{\Phi, a \approx b}{\Phi, a \approx b \vdash a \approx b}
\]

**Rec-Cons**

\[
\text{label}(a) = \text{label}(b) \\
\Phi, a \approx b \vdash \text{children}(a) \approx \text{children}(b) \\
\Phi \vdash a \approx b
\]

\(\Phi\) a set — comma is disjoint extension
Iso-recursive types

**Roll**
\[
\Gamma \vdash M : A[\mu X.A/X]
\]
\[
\Gamma \vdash \text{roll } M : \mu X.A
\]

**Unroll**
\[
\Gamma \vdash M : \mu X.A
\]
\[
\Gamma \vdash \text{unroll } M : A[\mu X.A/X]
\]
Iso-recursive types

**Roll**

\[ \Gamma \vdash M : A[\mu X.A/X] \]

\[ \Gamma \vdash \text{roll } M : \mu X.A \]

**Unroll**

\[ \Gamma \vdash M : \mu X.A \]

\[ \Gamma \vdash \text{unroll } M : A[\mu X.A/X] \]

Example

\[ \mu X.1 + X \neq 1 + \mu X.1 + X \neq \mu X.1 + (1 + X) \]
Equi-recursive versus iso-recursive types

\[
\begin{align*}
\text{STLC} & \quad \text{Simply-Typed Lambda Calculus} \\
\text{FPC} & \quad \text{Fixed Point Calculus} \\
\text{FPC} & \quad \text{STLC + iso-recursive types} \\
\text{FPC} & \quad \text{STLC + equi-recursive types}
\end{align*}
\]

[Abadi and Fiore, 1996; Brandt and Henglein, 1998]
Equi-recursive versus iso-recursive types

- **STLC**: Simply-Typed Lambda Calculus
- **FPC**: Fixed Point Calculus
- **FPC**: STLC + iso-recursive types
- **FPC**: STLC + equi-recursive types

**Interdefinability**

- **FPC** can simulate **FPC**
  - just erase roll and unroll
- **FPC** can simulate **FPC** up to observational equivalence
  - coercion functions witness type equivalence

[Abadi and Fiore, 1996; Brandt and Henglein, 1998]
Positive recursive types

**Positive** type variable: occurs on left of even number of arrows

**Negative** type variable: occurs on left of odd number of arrows

**Strictly positive** type variable: occurs on right of arrows

Examples — $X$ occurs:

- strictly positively in $X$, $1 + \text{Int} \times X$, and $\text{Int} \to X$
- positively in $(X \to \text{Int}) \to \text{Int}$
- negatively in $X \to \text{Int}$
- both positively and negatively in $X \to X$

A recursive type is positive if all occurrences of the bound variable are positive, e.g:

$$\mu X.1 + \text{Int} \times X \quad \quad \mu X.(X \to \text{Int}) \to \text{Int}$$
Positive equi- versus positive iso-

- FPC\(^+\) \(\rightarrow\) STLC + positive iso-recursive types
- FPC\(^\pm\) \(\rightarrow\) STLC + positive equi-recursive types
- FPC\(^++\) \(\rightarrow\) STLC + strictly positive iso-recursive types
- FPC\(^{++}\) \(\rightarrow\) STLC + strictly positive equi-recursive types
Positive equi- versus positive iso-

<table>
<thead>
<tr>
<th>FPC⁺</th>
<th>STLC + positive iso-recursive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPCₑ⁺</td>
<td>STLC + positive equi-recursive types</td>
</tr>
<tr>
<td>FPCₑ⁺⁺</td>
<td>STLC + strictly positive iso-recursive types</td>
</tr>
<tr>
<td>FPCₑ⁺⁺⁺</td>
<td>STLC + strictly positive equi-recursive types</td>
</tr>
</tbody>
</table>

Interdefinability

- FPCₑ⁺ can simulate FPC⁺ — just erase roll and unroll
- FPCₑ⁺⁺ can simulate FPCₑ⁺⁺ — just erase roll and unroll
- System F can simulate FPC⁺ (strong normalisation)
- FPCₑ⁺ can simulate FPCₑ⁺ (non-termination)
- FPCₑ⁺ is strictly more expressive than FPC⁺
- FPC⁺ + general recursion can simulate FPC up to observational equivalence
- FPCₑ⁺⁺ + fold can simulate FPCₑ⁺⁺⁺ up to observational equivalence
Universal type with a **negative** occurrence

\[ U = U \rightarrow U = \mu X.X \rightarrow X \]

All untyped lambda terms can be typed with \( U \)

\[ \omega = \lambda x. x \ x \quad \Omega = \omega \ \omega \quad \Omega \xrightarrow{\beta} \Omega \]
Universal type with a **negative** occurrence

\[ U = U \to U = \mu X. X \to X \]

All untyped lambda terms can be typed with \( U \)

\[ \omega = \lambda x.x \ x \quad \Omega = \omega \ \omega \quad \Omega \leadsto_{\beta} \Omega \]

Universal type as mutually recursive **positive** type

\[ P = Q \to P = \mu X. (\mu Y. X \to Y) \to X \]
\[ Q = P \to Q = \mu X. (\mu Y. X \to Y) \to X \]

\( U, P, Q \) all represent infinite binary tree of \( \to \) nodes

\[ \vdash U \approx P \approx Q \]
Universal type with a **negative** occurrence

\[ U = U \to U = \mu X.X \to X \]

All untyped lambda terms can be typed with \( U \)

\[ \omega = \lambda x.x \ x \quad \Omega = \omega \ \omega \quad \Omega \rightsquigarrow_\beta \Omega \]

Universal type as mutually recursive **positive** type

\[ P = Q \to P = \mu X.(\mu Y.X \to Y) \to X \]
\[ Q = P \to Q = \mu X.(\mu Y.X \to Y) \to X \]

\( U, P, Q \) all represent infinite binary tree of \( \to \) nodes

\[ \vdash U \approx P \approx Q \]

All untyped lambda terms can be typed with \( P \)!
**FPC\_≠ in FPC\_±**

Idea: split the positive and negative occurrences

\[ \mu X. F[X, X] \approx \mu X. F[\mu Y. F[X, Y], X] \]

[Bekić, 1984]

Example: universal data type

\[ F[X^-, X^+] = X^- \rightarrow X^+ \]

\[ U = \mu X. F[X, X] = \mu X. X \rightarrow X \]

\[ P = \mu X. F[\mu Y. F[X, Y], X] = \mu X.(\mu Y. X \rightarrow Y) \rightarrow X \]
FPC in $\text{FPC}^+ + \text{general recursion}$

Coercions for simulating FPC in $\text{FPC}_{\perp}$ use general recursion

$\text{FPC}^+ + \text{general recursion} \rightarrow \text{FPC}^+ \rightarrow \text{FPC}_{\perp} \rightarrow \text{FPC}$

Example:

$$\omega^{U \rightarrow U} = \lambda x^U . (\text{unroll } x) \times$$
$$\Omega^U = \omega^{U \rightarrow U} \left( \text{roll } \omega^{U \rightarrow U} \right)$$

$$\omega^{Q \rightarrow P} = \lambda x^Q . (\text{unroll } (V_{QP} \times)) \ (V_{QP} \times)$$
$$\Omega^P = \omega^{Q \rightarrow P} \left( V_{PQ} \left( \text{roll } \omega^{Q \rightarrow P} \right) \right)$$

where

$$V_{QP} = \text{rec } f^{Q \rightarrow P} \times^Q . \text{roll } (f \circ (\text{unroll } x) \circ f)$$
$$V_{PQ} = \text{rec } f^{P \rightarrow Q} \times^P . \text{roll } (f \circ (\text{unroll } x) \circ f)$$
Summary

\[ FPC \sim FPC_{=} \sim FPC_{=}^{+} \succ FPC^{+} \preceq \text{System F} \]

\[ FPC^{++} \sim FPC_{=}^{++} \]
Can positive iso-recursive types encode positive equi-recursive types?
Can positive iso-recursive types encode positive equi-recursive types?

Not without general recursion.
Can positive iso-recursive types encode positive equi-recursive types?

Not without general recursion

Intuitively the encoding requires the insertion of an infinite number of rolls and unrolls
Can positive iso-recursive types encode positive equi-recursive types?

Not without general recursion

Intuitively the encoding requires the insertion of an infinite number of rolls and unrolls

Morally the notion of a “positive” equi-recursive type is rather misleading
Part III

Existential types
Motivation

Well-known how universal types can encode existential types
Motivation

Well-known how universal types can encode existential types

What if we already have existential types, and want to encode universal types?

(I ran into this situation when trying to define a minimal effect handler calculus where parametric algebraic operations provide existentials, but there are no universals.)
Existentials as universals

De Morgan dual

$$\exists X. A \equiv \neg \forall X. \neg A = (\forall X. (A \rightarrow \bot)) \rightarrow \bot$$
Existentials as universals

De Morgan dual

$$\exists X. A \equiv \neg \forall X. \neg A = (\forall X. (A \rightarrow \bot)) \rightarrow \bot$$

System F encoding generalises the de Morgan dual

$$\exists X. A \equiv \forall Z. (\forall X. (A \rightarrow Z)) \rightarrow Z$$
Existentials as universals

De Morgan dual
\[ \exists X. A \equiv \neg \forall X. \neg A = (\forall X. (A \rightarrow \bot)) \rightarrow \bot \]

System F encoding generalises the de Morgan dual
\[ \exists X. A \equiv \forall Z. (\forall X. (A \rightarrow Z)) \rightarrow Z \]

What about the other way round?
\[ \forall X. A \equiv \neg \exists X. \neg A = (\exists X. (A \rightarrow \bot)) \rightarrow \bot \]
Existentials as universals

De Morgan dual

\[ \exists X. A \equiv \neg \forall X. \neg A = (\forall X.(A \rightarrow \bot)) \rightarrow \bot \]

System F encoding generalises the de Morgan dual

\[ \exists X. A \equiv \forall Z.(\forall X.(A \rightarrow Z)) \rightarrow Z \]

What about the other way round?

\[ \forall X. A \equiv \neg \exists X. \neg A = (\exists X.(A \rightarrow \bot)) \rightarrow \bot \]

The generalisation trick is no good as it depends on another universal quantifier

\[ \forall X. A \equiv \forall Z.(\exists X. A \rightarrow Z) \rightarrow Z \]
Minimal existential logic

Types

\[ A, B ::= \bot \mid \neg A \mid A \times B \mid \exists X. A \mid X \]

Terms

\[ M, N ::= x \]
\[ \quad \mid \lambda x^A. M \mid M N \]
\[ \quad \mid (M, N) \mid \text{let } (x, y) = M \text{ in } N \]
\[ \quad \mid (A, M) \mid \text{let } (X, y) = M \text{ in } N \]
Universals as existentials [Fujita, 2010]

Judgements

\[(\Gamma \vdash M : A)^* = \neg \Gamma^* \vdash M^* : \neg A^*\]

Types

\[X^* = X\]
\[(A \to B)^* = \neg A^* \times B^*\]
\[(\forall X.A)^* = \exists X.A^*\]

Terms

\[(x : A)^* = \lambda k^A^* . x \ k\]
\[(\lambda x . M : A)^* = \lambda k^A^* . \text{let } (x, k) = k \text{ in } M^* \ k\]
\[(M \ N : A)^* = \lambda k^A^* . M^* (N^*, k)\]
\[(\Lambda X . M : A)^* = \lambda k^A^* . \text{let } (X, k) = k \text{ in } M^* \ k\]
\[(M \ B : A)^* = \lambda k^A^* . M^* (B^*, k)\]
Can existentials encode universals?
Can existentials encode universals?

Yes, with a global CPS translation [Fujita, 2010]
Can existentials encode universals?

Yes, with a global CPS translation [Fujita, 2010]

Open question: can we prove that there is no local encoding?
Part IV

Effectful anecdotes
Idioms are oblivious, arrows are meticulous, monads are meticulous

[Philip Wadler, Jeremy Yallop]

[Lindley, Wadler, and Yallop, 2008]

Semantically: (monads $<$ arrows) and (idioms $<$ arrows)

As programs: idioms $<$ arrows $<$ monads
On the expressive power of user-defined effects

Yannick Forster  Ohad Kammar  Matija Pretnar

[Forster, Kammar, Lindley, and Pretnar, 2019]

Eff = effect handlers  Mon = monadic reflection  Del = delimited continuations

Untyped Eff $\iff$ Mon $\iff$ Del

Translations Mon $\rightarrow$ Eff and Del $\rightarrow$ Eff simulate reduction on the nose

Others translations don't

Simply-typed Eff $\not\iff$ Del

Polymorphic Eff $\iff$ Del

[Pirog, Polesiuk, Sieczkowski, 2019] Novel form of answer-type polymorphism
On the expressive power of user-defined effects

Yannick Forster  Ohad Kammar  Matija Pretnar

[Forster, Kammar, Lindley, and Pretnar, 2019]

Eff = effect handlers    Mon = monadic reflection    Del = delimited continuations

Untyped    Eff ↔ Mon ↔ Del ↔ Eff

- Translations Mon → Eff and Del → Eff simulate reduction on the nose
- Others translations don’t
On the expressive power of user-defined effects

Eff = effect handlers  Mon = monadic reflection  Del = delimited continuations

Untyped  \( \text{Eff} \leftrightarrow \text{Mon} \leftrightarrow \text{Del} \leftrightarrow \text{Eff} \)

- Translations \( \text{Mon} \rightarrow \text{Eff} \) and \( \text{Del} \rightarrow \text{Eff} \) simulate reduction on the nose.
- Others translations don’t.

Simply-typed  \( \text{Eff} \leftrightarrow \neg \leftrightarrow \text{Del} \)
On the expressive power of user-defined effects

[Yannick Forster, Ohad Kammar, Matija Pretnar, 2019]

\[\text{Eff} = \text{effect handlers} \quad \text{Mon} = \text{monadic reflection} \quad \text{Del} = \text{delimited continuations}\]

Untyped \hspace{1cm} \text{Eff} \leftrightarrow \text{Mon} \leftrightarrow \text{Del} \leftrightarrow \text{Eff}

\begin{itemize}
  \item Translations Mon \rightarrow \text{Eff} and Del \rightarrow \text{Eff} simulate reduction on the nose
  \item Others translations don’t
\end{itemize}

Simply-typed \hspace{1cm} \text{Eff} \leftrightarrow \not\leftrightarrow \text{Del}

Polymorphic \hspace{1cm} \text{Eff} \leftrightarrow \text{Del}

[Piróg, Polesiuk, Sieczkowski, 2019] Novel form of answer-type polymorphism
Asymptotic improvement with control operators

[Hillerström, Lindley, and Longley, 2020]

Generic search algorithm is:

- $\Omega(n2^n)$ in PCF
- $O(2^n)$ in PCF + effect handlers

Key constraint: no change of types

Higher-order computability [Longley and Normann, 2015]
Part V

Wrapping up
Closing remarks

The range of notions of expressiveness is broad

Expressiveness results are fragile

Types enable richer notions of expressiveness