A call-by-value λ-calculus with lists and control

Robbert Krebbers

July 08, 2012 @ CL&C
Warwick, United Kingdom

Problem

- A lot of λ-calculi with control have been studied: $\lambda_{\mathcal{C}}$, $\lambda\mu$, $\bar{\lambda}\mu\tilde{\mu}$, $\lambda_{\mathcal{C}^-}$, λ_{Δ}, and many more…

Problem

- A lot of λ-calculi with control have been studied: $\lambda_{\mathcal{C}}$, $\lambda\mu$, $\bar{\lambda}\mu\tilde{\mu}$, $\lambda_{\mathcal{C}^-}$, λ_{Δ}, and many more...
- Nearly all do not support data types in direct style, i.e.
  - inductive types
  - natural numbers
  - lists
  - ...

Problem

- A lot of λ-calculi with control have been studied:
  $\lambda_{\mathcal{C}}$, $\lambda\mu$, $\bar{\lambda}\mu\tilde{\mu}$, $\lambda_{\mathcal{C}^-}$, λ_{Δ}, and many more…
- Nearly all do not support data types in direct style, i.e.
  - inductive types
  - natural numbers
  - lists
  - …
- Actual programming languages do support these

My previous attempt: $\lambda\mu^{\mathbf{T}}$

- $\lambda\mu$ with natural numbers (à la Gödel's **T**)

My previous attempt: $\lambda\mu^{\mathbf{T}}$

- $\lambda\mu$ with natural numbers (à la Gödel's **T**)
- Satisfies [Geuvers/Krebbers/McKinna,2012]
  - subject reduction, confluence, strong normalization
  - unique representation of natural numbers
- Exactly as expressive as Gödel's **T**

My previous attempt: $\lambda\mu^{\mathbf{T}}$

- $\lambda\mu$ with natural numbers (à la Gödel's **T**)
- Satisfies [Geuvers/Krebbers/McKinna,2012]
  - subject reduction, confluence, strong normalization
  - unique representation of natural numbers
- Exactly as expressive as Gödel's **T**
- It was not quite satisfactory:
  - call-by-name reduction with call-by-value data types

My previous attempt: $\lambda\mu^{\mathbf{T}}$

- $\lambda\mu$ with natural numbers (à la Gödel's **T**)
- Satisfies [Geuvers/Krebbers/McKinna,2012]
  - subject reduction, confluence, strong normalization
  - unique representation of natural numbers
- Exactly as expressive as Gödel's **T**
- It was not quite satisfactory:
  - call-by-name reduction with call-by-value data types
  - difficult meta theory

My previous attempt: $\lambda\mu^{\mathbf{T}}$

- $\lambda\mu$ with natural numbers (à la Gödel's **T**)
- Satisfies [Geuvers/Krebbers/McKinna,2012]
  - subject reduction, confluence, strong normalization
  - unique representation of natural numbers
- Exactly as expressive as Gödel's **T**
- It was not quite satisfactory:
  - call-by-name reduction with call-by-value data types
  - difficult meta theory
  - hard to extend

Starting point for an improved system: Herbelin's $\mathrm{IQC}_{\mathrm{MP}}$

- Incorporates the control operators `catch` and `throw`

Starting point for an improved system: Herbelin's $\mathrm{IQC}_{\mathsf{MP}}$

- Incorporates the control operators `catch` and `throw`
- Convenient meta theory

Starting point for an improved system: Herbelin's $\mathsf{IQC}_{\mathsf{MP}}$

- Incorporates the control operators `catch` and `throw`
- Convenient meta theory
- Gives a constructive interpretation to Markov's principle

$$\neg\neg\exists x.P(x) \to \exists x.P(x)$$

Starting point for an improved system: Herbelin's $\mathsf{IQC}_{\mathsf{MP}}$

- Incorporates the control operators `catch` and `throw`
- Convenient meta theory
- Gives a constructive interpretation to Markov's principle

$$\neg\neg\exists x.P(x) \to \exists x.P(x)$$

- However:
  - No types like natural numbers, lists, ...
  - No (direct) proofs of confluence and strong normalization

This talk: the system λ::`catch`

- Based on Herbelin's IQC_{MP}

This talk: the system λ::`catch`

- Based on Herbelin's IQC_{MP}
- Primitive data type of lists and a recursor

This talk: the system λ::`catch`

- Based on Herbelin's $\mathrm{IQC}_{\mathrm{MP}}$
- Primitive data type of lists and a recursor
- Fully-fledged call-by-value system

This talk: the system λ::`catch`

- Based on Herbelin's $\mathrm{IQC}_{\mathrm{MP}}$
- Primitive data type of lists and a recursor
- Fully-fledged call-by-value system
- Satisfies the conventional meta theoretical properties:

This talk: the system λ::`catch`

- Based on Herbelin's $\mathsf{IQC}_{\mathsf{MP}}$
- Primitive data type of lists and a recursor
- Fully-fledged call-by-value system
- Satisfies the conventional meta theoretical properties:

  Subject reduction. $\Gamma; \Delta \vdash t : \rho$ and $t \to t'$, then $\Gamma; \Delta \vdash t' : \rho$

This talk: the system λ::`catch`

- Based on Herbelin's $\mathsf{IQC}_{\mathsf{MP}}$
- Primitive data type of lists and a recursor
- Fully-fledged call-by-value system
- Satisfies the conventional meta theoretical properties:

Subject reduction. $\Gamma;\Delta \vdash t : \rho$ and $t \to t'$, then $\Gamma;\Delta \vdash t' : \rho$

Progress. $;\vdash t : \rho$, then t is a value or $\exists t', t \to t'$

This talk: the system λ::`catch`

- Based on Herbelin's $\mathrm{IQC}_{\mathrm{MP}}$
- Primitive data type of lists and a recursor
- Fully-fledged call-by-value system
- Satisfies the conventional meta theoretical properties:

Subject reduction. $\Gamma;\Delta \vdash t : \rho$ and $t \to t'$, then $\Gamma;\Delta \vdash t' : \rho$

Progress. $; \vdash t : \rho$, then t is a value or $\exists t', t \to t'$

Confluence. $t \twoheadrightarrow r$ and $t \twoheadrightarrow s$, then $\exists q \,.\, r \twoheadrightarrow q$ and $s \twoheadrightarrow q$

This talk: the system λ::`catch`

- Based on Herbelin's $\mathrm{IQC}_{\mathrm{MP}}$
- Primitive data type of lists and a recursor
- Fully-fledged call-by-value system
- Satisfies the conventional meta theoretical properties:

Subject reduction. $\Gamma; \Delta \vdash t : \rho$ and $t \to t'$, then $\Gamma; \Delta \vdash t' : \rho$

Progress. $; \vdash t : \rho$, then t is a value or $\exists t', t \to t'$

Confluence. $t \twoheadrightarrow r$ and $t \twoheadrightarrow s$, then $\exists q \,.\, r \twoheadrightarrow q$ and $s \twoheadrightarrow q$

Strong Normalization. $\Gamma; \Delta \vdash t : \rho$, then no infinite $t \to t_1 \ldots$

This talk: the system λ::`catch`

- Based on Herbelin's $\mathrm{IQC}_{\mathrm{MP}}$
- Primitive data type of lists and a recursor
- Fully-fledged call-by-value system
- Satisfies the conventional meta theoretical properties:

  Subject reduction. $\Gamma; \Delta \vdash t : \rho$ and $t \to t'$, then $\Gamma; \Delta \vdash t' : \rho$

  Progress. $; \vdash t : \rho$, then t is a value or $\exists t', t \to t'$

  Confluence. $t \twoheadrightarrow r$ and $t \twoheadrightarrow s$, then $\exists q \,.\, r \twoheadrightarrow q$ and $s \twoheadrightarrow q$

  Strong Normalization. $\Gamma; \Delta \vdash t : \rho$, then no infinite $t \to t_1 \ldots$
- These properties are relatively easy to prove

The system λ::`catch`

- Typing judgments à la Parigot's $\lambda\mu$

$$\Gamma ; \Delta \vdash t : \rho$$

The system λ::`catch`

- Typing judgments à la Parigot's $\lambda\mu$

$$\Gamma ; \Delta \vdash t : \rho$$

$\alpha : \psi \in \Delta$ are exceptions that may be throw

The system λ::`catch`

- Typing judgments à la Parigot's $\lambda\mu$

  $$\Gamma ; \Delta \vdash t : \rho$$

  $\alpha : \psi \in \Delta$ are exceptions that may be throw
- Another way to think of it: t is a proof of either
  - ρ, or,
  - $\alpha : \psi \in \Delta$

The typing rules of λ::`catch`

The constructs of simple type theory:

$$\frac{x : \rho \in \Gamma}{\Gamma; \Delta \vdash x : \rho} \qquad \frac{\Gamma, x : \sigma; \Delta \vdash t : \tau}{\Gamma; \Delta \vdash \lambda x.t : \sigma \to \tau} \qquad \frac{\Gamma; \Delta \vdash t : \sigma \to \tau \qquad \Gamma; \Delta \vdash s : \sigma}{\Gamma; \Delta \vdash ts : \tau}$$

The typing rules of λ::`catch`

The constructs of simple type theory:

$$\frac{x : \rho \in \Gamma}{\Gamma; \Delta \vdash x : \rho} \qquad \frac{\Gamma, x : \sigma; \Delta \vdash t : \tau}{\Gamma; \Delta \vdash \lambda x.t : \sigma \to \tau} \qquad \frac{\Gamma; \Delta \vdash t : \sigma \to \tau \qquad \Gamma; \Delta \vdash s : \sigma}{\Gamma; \Delta \vdash ts : \tau}$$

Constructors of the unit and list data type:

$$\frac{}{\Gamma; \Delta \vdash () : \top} \qquad \frac{}{\Gamma; \Delta \vdash \mathtt{nil} : [\sigma]} \qquad \frac{}{\Gamma; \Delta \vdash (::) : \sigma \to [\sigma] \to [\sigma]}$$

The typing rules of λ::`catch`

The constructs of simple type theory:

$$\frac{x : \rho \in \Gamma}{\Gamma; \Delta \vdash x : \rho} \qquad \frac{\Gamma, x : \sigma; \Delta \vdash t : \tau}{\Gamma; \Delta \vdash \lambda x.t : \sigma \to \tau} \qquad \frac{\Gamma; \Delta \vdash t : \sigma \to \tau \qquad \Gamma; \Delta \vdash s : \sigma}{\Gamma; \Delta \vdash ts : \tau}$$

Constructors of the unit and list data type:

$$\frac{}{\Gamma; \Delta \vdash () : \top} \qquad \frac{}{\Gamma; \Delta \vdash \mathtt{nil} : [\sigma]} \qquad \frac{}{\Gamma; \Delta \vdash (::) : \sigma \to [\sigma] \to [\sigma]}$$

Primitive recursion over lists:

$$\frac{}{\Gamma; \Delta \vdash \mathtt{lrec} : \rho \to (\sigma \to [\sigma] \to \rho \to \rho) \to [\sigma] \to \rho}$$

The typing rules of λ::$\texttt{catch}$

The constructs of simple type theory:

$$\frac{x : \rho \in \Gamma}{\Gamma; \Delta \vdash x : \rho} \qquad \frac{\Gamma, x : \sigma; \Delta \vdash t : \tau}{\Gamma; \Delta \vdash \lambda x.t : \sigma \to \tau} \qquad \frac{\Gamma; \Delta \vdash t : \sigma \to \tau \qquad \Gamma; \Delta \vdash s : \sigma}{\Gamma; \Delta \vdash ts : \tau}$$

Constructors of the unit and list data type:

$$\frac{}{\Gamma; \Delta \vdash () : \top} \qquad \frac{}{\Gamma; \Delta \vdash \texttt{nil} : [\sigma]} \qquad \frac{}{\Gamma; \Delta \vdash (::) : \sigma \to [\sigma] \to [\sigma]}$$

Primitive recursion over lists:

$$\frac{}{\Gamma; \Delta \vdash \texttt{lrec} : \rho \to (\sigma \to [\sigma] \to \rho \to \rho) \to [\sigma] \to \rho}$$

The control operators `catch` and `throw`:

$$\frac{\Gamma; \Delta, \alpha : \psi \vdash t : \psi}{\Gamma; \Delta \vdash \texttt{catch}\ \alpha\,.\,t : \psi} \qquad \frac{\Gamma; \Delta \vdash t : \psi \qquad \alpha : \psi \in \Delta}{\Gamma; \Delta \vdash \texttt{throw}\ \alpha\ t : \tau}$$

The typing rules of λ::`catch`

The constructs of simple type theory:

$$\frac{x : \rho \in \Gamma}{\Gamma; \Delta \vdash x : \rho} \qquad \frac{\Gamma, x : \sigma; \Delta \vdash t : \tau}{\Gamma; \Delta \vdash \lambda x.t : \sigma \to \tau} \qquad \frac{\Gamma; \Delta \vdash t : \sigma \to \tau \qquad \Gamma; \Delta \vdash s : \sigma}{\Gamma; \Delta \vdash ts : \tau}$$

Constructors of the unit and list data type:

$$\frac{}{\Gamma; \Delta \vdash () : \top} \qquad \frac{}{\Gamma; \Delta \vdash \mathtt{nil} : [\sigma]} \qquad \frac{}{\Gamma; \Delta \vdash (::) : \sigma \to [\sigma] \to [\sigma]}$$

Primitive recursion over lists:

$$\frac{}{\Gamma; \Delta \vdash \mathtt{lrec} : \rho \to (\sigma \to [\sigma] \to \rho \to \rho) \to [\sigma] \to \rho}$$

The control operators `catch` and `throw`:

$$\frac{\Gamma; \Delta, \alpha : \psi \vdash t : \psi}{\Gamma; \Delta \vdash \mathtt{catch}\, \alpha \,.\, t : \psi} \qquad \frac{\Gamma; \Delta \vdash t : \psi \qquad \alpha : \psi \in \Delta}{\Gamma; \Delta \vdash \mathtt{throw}\, \alpha\ t : \tau}$$

Important: ψ ranges over →-free types [Herbelin, 2010]

Example: typing

$$\frac{}{;\ \vdash \mathtt{catch}\ \alpha\,.\,(\mathtt{throw}\ \alpha\ \mathtt{nil}) :: \mathtt{nil} : [\top]}$$

Example: typing

$$\frac{}{;\vdash \texttt{catch}\,\alpha\,.\,(\texttt{throw}\,\alpha\,\texttt{nil}) :: \texttt{nil} : [\top]}$$

How to think of this derivation:

1. Our goal is $[\top]$

Example: typing

$$\dfrac{\dfrac{}{;\, \alpha : [\top] \vdash (\texttt{throw}\ \alpha\ \texttt{nil}) :: \texttt{nil} : [\top]}}{;\, \vdash \texttt{catch}\ \alpha\, .\, (\texttt{throw}\ \alpha\ \texttt{nil}) :: \texttt{nil} : [\top]}$$

How to think of this derivation:

1. Our goal is $[\top]$
2. We save the current continuation as α

Example: typing

$$\dfrac{\dfrac{}{;\alpha : [\top] \vdash \texttt{throw}\ \alpha\ \texttt{nil} : \top} \qquad \dfrac{}{;\alpha : [\top] \vdash \texttt{nil} : [\top]}}{\dfrac{;\alpha : [\top] \vdash (\texttt{throw}\ \alpha\ \texttt{nil}) :: \texttt{nil} : [\top]}{; \vdash \texttt{catch}\ \alpha\,.\,(\texttt{throw}\ \alpha\ \texttt{nil}) :: \texttt{nil} : [\top]}}$$

How to think of this derivation:

1. Our goal is $[\top]$
2. We save the current continuation as α
3. We construct a singleton list
   . . . leaving us to construct a term of type $\top$

Example: typing

$$\dfrac{\dfrac{\dfrac{\dfrac{}{;\alpha : [\top] \vdash \mathtt{nil} : [\top]}}{;\alpha : [\top] \vdash \mathtt{throw}\ \alpha\ \mathtt{nil} : \top} \qquad \dfrac{}{;\alpha : [\top] \vdash \mathtt{nil} : [\top]}}{;\alpha : [\top] \vdash (\mathtt{throw}\ \alpha\ \mathtt{nil}) :: \mathtt{nil} : [\top]}}{; \vdash \mathtt{catch}\ \alpha\,.\,(\mathtt{throw}\ \alpha\ \mathtt{nil}) :: \mathtt{nil} : [\top]}$$

How to think of this derivation:

1. Our goal is $[\top]$
2. We save the current continuation as α
3. We construct a singleton list
   . . . leaving us to construct a term of type $\top$
4. But instead we jump back to α with `nil`

Reduction

Values:

$$\begin{aligned} v, w ::= {} & x \mid () \mid \texttt{nil} \mid (::) \mid (::)\ v \mid (::)\ v\ w \\ & \mid \texttt{lrec} \mid \texttt{lrec}\ v_r \mid \texttt{lrec}\ v_r\ v_s \mid \lambda x.r \end{aligned}$$

Reduction

Values:

$$v, w ::= x \mid () \mid \texttt{nil} \mid (::) \mid (::)\ v \mid (::)\ v\ w \\ \mid \texttt{lrec} \mid \texttt{lrec}\ v_r \mid \texttt{lrec}\ v_r\ v_s \mid \lambda x.r$$

Reduction:

$$\begin{aligned} (\lambda x.t)\ v &\to t[x := v] \\ \texttt{lrec}\ v_r\ v_s\ \texttt{nil} &\to v_r \\ \texttt{lrec}\ v_r\ v_s\ (v_h :: v_t) &\to v_s\ v_h\ v_t\ (\texttt{lrec}\ v_r\ v_s\ v_t) \end{aligned}$$

Reduction

Values:

$$\begin{aligned} v, w ::= {} & x \mid () \mid \texttt{nil} \mid (::) \mid (::)\ v \mid (::)\ v\ w \\ & \mid \texttt{lrec} \mid \texttt{lrec}\ v_r \mid \texttt{lrec}\ v_r\ v_s \mid \lambda x.r \end{aligned}$$

Contexts:

$$E ::= \Box t \mid v\Box \mid \texttt{throw}\ \beta\ \Box$$

Reduction:

$$\begin{aligned} (\lambda x.t)\ v &\rightarrow t[x := v] \\ \texttt{lrec}\ v_r\ v_s\ \texttt{nil} &\rightarrow v_r \\ \texttt{lrec}\ v_r\ v_s\ (v_h :: v_t) &\rightarrow v_s\ v_h\ v_t\ (\texttt{lrec}\ v_r\ v_s\ v_t) \\ E[\texttt{throw}\ \alpha\ t] &\rightarrow \texttt{throw}\ \alpha\ t \end{aligned}$$

Reduction

Values:

$$v, w ::= x \mid () \mid \texttt{nil} \mid (::) \mid (::)\ v \mid (::)\ v\ w$$
$$\mid \texttt{lrec} \mid \texttt{lrec}\ v_r \mid \texttt{lrec}\ v_r\ v_s \mid \lambda x.r$$

Contexts:

$$E ::= \Box t \mid v\Box \mid \texttt{throw}\ \beta\ \Box$$

Reduction:

$$(\lambda x.t)\ v \to t[x := v]$$
$$\texttt{lrec}\ v_r\ v_s\ \texttt{nil} \to v_r$$
$$\texttt{lrec}\ v_r\ v_s\ (v_h :: v_t) \to v_s\ v_h\ v_t\ (\texttt{lrec}\ v_r\ v_s\ v_t)$$
$$E[\texttt{throw}\ \alpha\ t] \to \texttt{throw}\ \alpha\ t$$
$$\texttt{catch}\ \alpha\,.\,\texttt{throw}\ \alpha\ t \to \texttt{catch}\ \alpha\,.\,t$$

Reduction

Values:

$$\begin{aligned} v, w ::= x \mid () \mid \texttt{nil} \mid (::) \mid (::)\ v \mid (::)\ v\ w \\ \mid \texttt{lrec} \mid \texttt{lrec}\ v_r \mid \texttt{lrec}\ v_r\ v_s \mid \lambda x.r \end{aligned}$$

Contexts:

$$E ::= \Box t \mid v\Box \mid \texttt{throw}\ \beta\ \Box$$

Reduction:

$$\begin{aligned} (\lambda x.t)\ v &\to t[x := v] \\ \texttt{lrec}\ v_r\ v_s\ \texttt{nil} &\to v_r \\ \texttt{lrec}\ v_r\ v_s\ (v_h :: v_t) &\to v_s\ v_h\ v_t\ (\texttt{lrec}\ v_r\ v_s\ v_t) \\ E[\texttt{throw}\ \alpha\ t] &\to \texttt{throw}\ \alpha\ t \\ \texttt{catch}\ \alpha\,.\,\texttt{throw}\ \alpha\ t &\to \texttt{catch}\ \alpha\,.\,t \\ \texttt{catch}\ \alpha\,.\,\texttt{throw}\ \beta\ v &\to \texttt{throw}\ \beta\ v \quad \text{if } \alpha \notin \{\beta\} \cup \mathsf{FCV}(v) \end{aligned}$$

Reduction

Values:

$$\begin{aligned} v, w ::= {} & x \mid () \mid \texttt{nil} \mid (::) \mid (::)\ v \mid (::)\ v\ w \\ & \mid \texttt{lrec} \mid \texttt{lrec}\ v_r \mid \texttt{lrec}\ v_r\ v_s \mid \lambda x.r \end{aligned}$$

Contexts:

$$E ::= \Box t \mid v\Box \mid \texttt{throw}\ \beta\ \Box$$

Reduction:

$$\begin{aligned}
(\lambda x.t)\ v &\to t[x := v] \\
\texttt{lrec}\ v_r\ v_s\ \texttt{nil} &\to v_r \\
\texttt{lrec}\ v_r\ v_s\ (v_h :: v_t) &\to v_s\ v_h\ v_t\ (\texttt{lrec}\ v_r\ v_s\ v_t) \\
E[\texttt{throw}\ \alpha\ t] &\to \texttt{throw}\ \alpha\ t \\
\texttt{catch}\ \alpha\,.\,\texttt{throw}\ \alpha\ t &\to \texttt{catch}\ \alpha\,.\,t \\
\texttt{catch}\ \alpha\,.\,\texttt{throw}\ \beta\ v &\to \texttt{throw}\ \beta\ v \qquad \text{if } \alpha \notin \{\beta\} \cup \mathsf{FCV}(v) \\
\texttt{catch}\ \alpha\,.\,v &\to v \qquad\qquad\qquad\ \ \text{if } \alpha \notin \mathsf{FCV}(v)
\end{aligned}$$

Example: reduction

$$\texttt{catch}\,\alpha\,\texttt{.}\,(\texttt{throw}\,\alpha\,\texttt{nil}) :: \texttt{nil}$$

Example: reduction

$$\texttt{catch}\,\alpha\,\texttt{.}\,(\texttt{throw}\;\alpha\;\texttt{nil}) :: \texttt{nil}$$

Example: reduction

$$\begin{aligned} & \texttt{catch}\,\alpha\,.\,(\texttt{throw}\,\alpha\,\texttt{nil}) :: \texttt{nil} \\ \equiv\ & \texttt{catch}\,\alpha\,.\,(\Box :: \texttt{nil})[\texttt{throw}\,\alpha\,\texttt{nil}] \end{aligned}$$

Example: reduction

$$\begin{aligned} & \texttt{catch}\,\alpha\,.\,(\texttt{throw}\,\alpha\,\texttt{nil}) :: \texttt{nil} \\ \equiv\; & \texttt{catch}\,\alpha\,.\,(\square :: \texttt{nil})[\texttt{throw}\,\alpha\,\texttt{nil}] \end{aligned}$$

Example: reduction

$$\begin{aligned} & \texttt{catch}\,\alpha\,.\,(\texttt{throw}\,\alpha\,\texttt{nil}) :: \texttt{nil} \\ \equiv\ & \texttt{catch}\,\alpha\,.\,(\Box :: \texttt{nil})[\texttt{throw}\,\alpha\,\texttt{nil}] \\ \rightarrow\ & \texttt{catch}\,\alpha\,.\,\texttt{throw}\,\alpha\,\texttt{nil} \end{aligned}$$

Example: reduction

$$\begin{aligned}
&\texttt{catch}\,\alpha\,.\,(\texttt{throw}\,\alpha\,\texttt{nil}) :: \texttt{nil}\\
\equiv\ &\texttt{catch}\,\alpha\,.\,(\Box :: \texttt{nil})[\texttt{throw}\,\alpha\,\texttt{nil}]\\
\to\ &\texttt{catch}\,\alpha\,.\,\texttt{throw}\,\alpha\,\texttt{nil}
\end{aligned}$$

Example: reduction

$$\begin{aligned}
&\texttt{catch}\,\alpha\,\texttt{.}\,(\texttt{throw}\,\alpha\,\texttt{nil}) :: \texttt{nil}\\
\equiv\;&\texttt{catch}\,\alpha\,\texttt{.}\,(\square :: \texttt{nil})[\texttt{throw}\,\alpha\,\texttt{nil}]\\
\rightarrow\;&\texttt{catch}\,\alpha\,\texttt{.}\,\texttt{throw}\,\alpha\,\texttt{nil}\\
\rightarrow\;&\texttt{catch}\,\alpha\,\texttt{.}\,\texttt{nil}
\end{aligned}$$

Example: reduction

$$\begin{aligned} & \texttt{catch}\,\alpha\,.\,(\texttt{throw}\,\alpha\,\texttt{nil}) :: \texttt{nil} \\ \equiv\ & \texttt{catch}\,\alpha\,.\,(\Box :: \texttt{nil})[\texttt{throw}\,\alpha\,\texttt{nil}] \\ \to\ & \texttt{catch}\,\alpha\,.\,\texttt{throw}\,\alpha\,\texttt{nil} \\ \to\ & \texttt{catch}\,\alpha\,.\,\texttt{nil} \end{aligned}$$

Example: reduction

$$\begin{aligned} & \texttt{catch}\,\alpha\,.\,(\texttt{throw}\,\alpha\,\texttt{nil}) :: \texttt{nil} \\ \equiv\ & \texttt{catch}\,\alpha\,.\,(\Box :: \texttt{nil})[\texttt{throw}\,\alpha\,\texttt{nil}] \\ \to\ & \texttt{catch}\,\alpha\,.\,\texttt{throw}\,\alpha\,\texttt{nil} \\ \to\ & \texttt{catch}\,\alpha\,.\,\texttt{nil} \\ \to\ & \texttt{nil} \end{aligned}$$

Why restrict to $\rightarrow$-free types? (1)

Progress: $; \vdash t : \tau \implies t$ is a value or $\exists t'. t \rightarrow t'$

Why restrict to $\rightarrow$-free types? (1)

Progress: $; \vdash t : \tau \implies t$ is a value or $\exists t'.t \rightarrow t'$

- Without the $\rightarrow$-free restriction, the term

  $$\texttt{catch}\,\alpha\,.\,\lambda x.\texttt{throw}\,\alpha\,(\lambda y.y) : \top \rightarrow \top$$

  would not reduce

Why restrict to $\rightarrow$-free types? (1)

Progress: $; \vdash t : \tau \implies t$ is a value or $\exists t'.t \rightarrow t'$

- Without the $\rightarrow$-free restriction, the term

  $$\texttt{catch}\,\alpha\,.\,\lambda x.\texttt{throw}\,\alpha\,(\lambda y.y) : \top \rightarrow \top$$

  would not reduce
- Hence progress would fail

Why restrict to $\rightarrow$-free types? (1)

Progress: $; \vdash t : \tau \implies t$ is a value or $\exists t'.t \to t'$

- Without the $\rightarrow$-free restriction, the term

  $$\mathtt{catch}\,\alpha\,.\,\lambda x.\mathtt{throw}\,\alpha\,(\lambda y.y) : \top \to \top$$

  would not reduce
- Hence progress would fail
- Note: an analogue term in the $\lambda\mu$-calculus

  $$\mu\alpha.[\alpha]\lambda x.\mu\,\_.[\alpha]\lambda y.y$$

  does not reduce either

Why restrict to $\rightarrow$-free types? (2)

Consequences of progress

- In Herbelin's $\mathrm{IQC}_{\mathrm{MP}}$:
  - If $; \vdash t : \rho \vee \sigma$, then $\exists t' \,.\, ; \vdash t' : \rho$ or $; \vdash t' : \sigma$
  - If $; \vdash t : \exists x. P(x)$, then $\exists t' \,.\, ; \vdash t' : P(t')$

Why restrict to $\rightarrow$-free types? (2)

Consequences of progress

- In Herbelin's IQC_{MP}:
  - If $; \vdash t : \rho \vee \sigma$, then $\exists t' .\ ; \vdash t' : \rho$ or $; \vdash t' : \sigma$
  - If $; \vdash t : \exists x.P(x)$, then $\exists t' .\ ; \vdash t' : P(t')$
- In λ::`catch`:
  - Unique representation of data
  - One-to-one correspondence between closed terms of $\mathtt{N}$ and $\mathbb{N}$

Natural numbers

We define a type $\mathtt{N} := [\top]$, with:

$$\begin{aligned} \mathtt{0} &:= \mathtt{nil} \\ \mathtt{S} &:= (::)\,() \\ \mathtt{nrec} &:= \lambda x_r y_s \,.\, \mathtt{lrec}\; x_r\; (\lambda\,\_\,.\, x_s) \end{aligned}$$

Notation: $\underline{n} := \mathtt{S}^n\mathtt{0}$

Inefficient predecessor

We could define $\texttt{pred} : \texttt{N} \to \texttt{N}$ as

$$\texttt{pred} := \texttt{nrec}\ \texttt{0}\ (\lambda x h \,.\, x)$$

Inefficient predecessor

We could define $\texttt{pred} : \texttt{N} \to \texttt{N}$ as

$$\texttt{pred} := \texttt{nrec}\ \texttt{0}\ (\lambda x h\,.\,x)$$

Inefficient with call-by-value reduction

$$\texttt{pred}\ \underline{n} \twoheadrightarrow (\lambda x h\,.\,x)\ \underline{n-1}\ (\texttt{pred}\ \underline{n-1})$$

Inefficient predecessor

We could define $\texttt{pred} : \texttt{N} \to \texttt{N}$ as

$$\texttt{pred} := \texttt{nrec}\ \texttt{0}\ (\lambda x h \,.\, x)$$

Inefficient with call-by-value reduction

$$\begin{aligned} \texttt{pred}\ \underline{n} &\twoheadrightarrow (\lambda x h \,.\, x)\ \underline{n-1}\ (\texttt{pred}\ \underline{n-1}) \\ &\twoheadrightarrow (\lambda h \,.\, \underline{n-1})\ \underbrace{(\texttt{pred}\ \underline{n-2})}_{\text{not a value}} \end{aligned}$$

Inefficient predecessor

We could define $\mathtt{pred} : \mathtt{N} \to \mathtt{N}$ as

$$\mathtt{pred} := \mathtt{nrec}\ \mathtt{0}\ (\lambda x h \,.\, x)$$

Inefficient with call-by-value reduction

$$\begin{aligned}
\mathtt{pred}\ \underline{n} &\twoheadrightarrow (\lambda x h \,.\, x)\ \underline{n-1}\ (\mathtt{pred}\ \underline{n-1}) \\
&\twoheadrightarrow (\lambda h \,.\, \underline{n-1})\ \underbrace{(\mathtt{pred}\ \underline{n-2})}_{\text{not a value}} \\
&\twoheadrightarrow (\lambda h \,.\, \underline{n-1})\ ((\lambda h \,.\, \underline{n-2})\ \underbrace{(\mathtt{pred}\ \underline{n-2})}_{\text{not a value}})
\end{aligned}$$

Inefficient predecessor

We could define $\mathtt{pred} : \mathtt{N} \to \mathtt{N}$ as

$$\mathtt{pred} := \mathtt{nrec}\ 0\ (\lambda x h \,.\, x)$$

Inefficient with call-by-value reduction

$$\begin{aligned}
\mathtt{pred}\ \underline{n} &\twoheadrightarrow (\lambda x h \,.\, x)\ \underline{n-1}\ (\mathtt{pred}\ \underline{n-1}) \\
&\twoheadrightarrow (\lambda h \,.\, \underline{n-1})\ \underbrace{(\mathtt{pred}\ \underline{n-2})}_{\text{not a value}} \\
&\twoheadrightarrow (\lambda h \,.\, \underline{n-1})\ ((\lambda h \,.\, \underline{n-2})\ \underbrace{(\mathtt{pred}\ \underline{n-2})}_{\text{not a value}}) \\
&\twoheadrightarrow \ldots
\end{aligned}$$

A more efficient predecessor in λ::`catch`

We redefine $\texttt{pred} : \texttt{N} \to \texttt{N}$ as

$$\texttt{pred} := \lambda n\,.\,\texttt{catch}\,\alpha\,.\,\texttt{nrec}\;0\;(\lambda x\,.\,\texttt{throw}\;\alpha\;x)\;n$$

A more efficient predecessor in λ::`catch`

We redefine `pred` : $\mathtt{N} \to \mathtt{N}$ as

$$\mathtt{pred} := \lambda n\,.\,\mathtt{catch}\,\alpha\,.\,\mathtt{nrec}\;0\;(\lambda x\,.\,\mathtt{throw}\;\alpha\;x)\;n$$

Using `catch` and `throw` it becomes more efficient

$$\mathtt{pred}\;\underline{n+1}$$

A more efficient predecessor in λ::`catch`

We redefine `pred` : $\mathtt{N} \to \mathtt{N}$ as

$$\mathtt{pred} := \lambda n\,.\,\mathtt{catch}\,\alpha\,.\,\mathtt{nrec}\;0\;(\lambda x\,.\,\mathtt{throw}\;\alpha\;x)\;n$$

Using `catch` and `throw` it becomes more efficient

$$\mathtt{pred}\ \underline{n+1}$$

A more efficient predecessor in λ::`catch`

We redefine `pred` : `N` → `N` as

$$\mathtt{pred} := \lambda n\,.\,\mathtt{catch}\,\alpha\,.\,\mathtt{nrec}\;0\;(\lambda x\,.\,\mathtt{throw}\;\alpha\;x)\;n$$

Using `catch` and `throw` it becomes more efficient

$$\begin{aligned} & \mathtt{pred}\;\underline{n+1} \\ \twoheadrightarrow\; & \mathtt{catch}\,\alpha\,.\,\mathtt{nrec}\;0\;(\lambda x\,.\,\mathtt{throw}\;\alpha\;x)\;(\mathtt{S}\;\underline{n}) \end{aligned}$$

A more efficient predecessor in λ::`catch`

We redefine $\texttt{pred} : \texttt{N} \to \texttt{N}$ as

$$\texttt{pred} := \lambda n \,.\, \texttt{catch}\, \alpha \,.\, \texttt{nrec}\; 0\; (\lambda x \,.\, \texttt{throw}\; \alpha\; x)\; n$$

Using `catch` and `throw` it becomes more efficient

$$\begin{aligned} & \texttt{pred}\; \underline{n+1} \\ \twoheadrightarrow\; & \texttt{catch}\, \alpha \,.\, \color{red}{\texttt{nrec}\; 0\; (\lambda x \,.\, \texttt{throw}\; \alpha\; x)\; (\texttt{S}\; \underline{n})} \end{aligned}$$

A more efficient predecessor in λ::`catch`

We redefine `pred` $: \mathtt{N} \to \mathtt{N}$ as

$$\mathtt{pred} := \lambda n\,.\,\mathtt{catch}\,\alpha\,.\,\mathtt{nrec}\;0\;(\lambda x\,.\,\mathtt{throw}\;\alpha\;x)\;n$$

Using `catch` and `throw` it becomes more efficient

$$\begin{aligned}
&\mathtt{pred}\;\underline{n+1}\\
\twoheadrightarrow\;&\mathtt{catch}\,\alpha\,.\,\mathtt{nrec}\;0\;(\lambda x\,.\,\mathtt{throw}\;\alpha\;x)\;(\mathtt{S}\;\underline{n})\\
\twoheadrightarrow\;&\mathtt{catch}\,\alpha\,.\,(\lambda x\,.\,\mathtt{throw}\;\alpha\;x)\;\underline{n}\;(\mathtt{lrec}\;0\;(\lambda\,\_\;x\,.\,\mathtt{throw}\;\alpha\;x)\;\underline{n})
\end{aligned}$$

A more efficient predecessor in λ::`catch`

We redefine `pred` : $\mathtt{N} \to \mathtt{N}$ as

$$\mathtt{pred} := \lambda n \,.\, \mathtt{catch}\ \alpha \,.\, \mathtt{nrec}\ 0\ (\lambda x \,.\, \mathtt{throw}\ \alpha\ x)\ n$$

Using `catch` and `throw` it becomes more efficient

$$\begin{aligned} & \mathtt{pred}\ \underline{n+1} \\ \twoheadrightarrow\ & \mathtt{catch}\ \alpha \,.\, \mathtt{nrec}\ 0\ (\lambda x \,.\, \mathtt{throw}\ \alpha\ x)\ (\mathtt{S}\ \underline{n}) \\ \twoheadrightarrow\ & \mathtt{catch}\ \alpha \,.\, {\color{red}(\lambda x \,.\, \mathtt{throw}\ \alpha\ x)\ \underline{n}}\ (\mathtt{lrec}\ 0\ (\lambda\, \_\, x \,.\, \mathtt{throw}\ \alpha\ x)\ \underline{n}) \end{aligned}$$

A more efficient predecessor in λ::`catch`

We redefine $\texttt{pred} : \texttt{N} \to \texttt{N}$ as

$$\texttt{pred} := \lambda n \,.\, \texttt{catch}\; \alpha \,.\, \texttt{nrec}\; 0\; (\lambda x \,.\, \texttt{throw}\; \alpha\; x)\; n$$

Using `catch` and `throw` it becomes more efficient

$$\begin{aligned}
&\texttt{pred}\; \underline{n+1} \\
\twoheadrightarrow\; &\texttt{catch}\; \alpha \,.\, \texttt{nrec}\; 0\; (\lambda x \,.\, \texttt{throw}\; \alpha\; x)\; (\texttt{S}\; \underline{n}) \\
\twoheadrightarrow\; &\texttt{catch}\; \alpha \,.\, (\lambda x \,.\, \texttt{throw}\; \alpha\; x)\; \underline{n}\; (\texttt{lrec}\; 0\; (\lambda \_\, x \,.\, \texttt{throw}\; \alpha\; x)\; \underline{n}) \\
\twoheadrightarrow\; &\texttt{catch}\; \alpha \,.\, (\texttt{throw}\; \alpha\; \underline{n})\; (\texttt{lrec}\; 0\; (\lambda \_\, x \,.\, \texttt{throw}\; \alpha\; x)\; \underline{n})
\end{aligned}$$

A more efficient predecessor in λ::$\texttt{catch}$

We redefine $\texttt{pred} : \texttt{N} \to \texttt{N}$ as

$$\texttt{pred} := \lambda n \,.\, \texttt{catch}\, \alpha \,.\, \texttt{nrec}\; 0\; (\lambda x \,.\, \texttt{throw}\; \alpha\; x)\; n$$

Using $\texttt{catch}$ and $\texttt{throw}$ it becomes more efficient

$$\begin{aligned}
& \texttt{pred}\; \underline{n+1} \\
\twoheadrightarrow\; & \texttt{catch}\, \alpha \,.\, \texttt{nrec}\; 0\; (\lambda x \,.\, \texttt{throw}\; \alpha\; x)\; (\texttt{S}\; \underline{n}) \\
\twoheadrightarrow\; & \texttt{catch}\, \alpha \,.\, (\lambda x \,.\, \texttt{throw}\; \alpha\; x)\; \underline{n}\; (\texttt{lrec}\; 0\; (\lambda \_\, x \,.\, \texttt{throw}\; \alpha\; x)\; \underline{n}) \\
\twoheadrightarrow\; & \texttt{catch}\, \alpha \,.\, (\texttt{throw}\; \alpha\; \underline{n})\; (\texttt{lrec}\; 0\; (\lambda \_\, x \,.\, \texttt{throw}\; \alpha\; x)\; \underline{n})
\end{aligned}$$

We use the rule $(\texttt{throw}\; \alpha\; t)\; r \to \texttt{throw}\; \alpha\; t$ to discard the recursive call

A more efficient predecessor in λ::catch

We redefine $\texttt{pred} : \texttt{N} \to \texttt{N}$ as

$$\texttt{pred} := \lambda n\,.\,\texttt{catch}\,\alpha\,.\,\texttt{nrec}\;0\;(\lambda x\,.\,\texttt{throw}\;\alpha\;x)\;n$$

Using catch and throw it becomes more efficient

$$\begin{aligned}
&\texttt{pred}\;\underline{n+1}\\
\twoheadrightarrow\;&\texttt{catch}\,\alpha\,.\,\texttt{nrec}\;0\;(\lambda x\,.\,\texttt{throw}\;\alpha\;x)\;(\texttt{S}\;\underline{n})\\
\twoheadrightarrow\;&\texttt{catch}\,\alpha\,.\,(\lambda x\,.\,\texttt{throw}\;\alpha\;x)\;\underline{n}\;(\texttt{lrec}\;0\;(\lambda\,\_\,x\,.\,\texttt{throw}\;\alpha\;x)\;\underline{n})\\
\twoheadrightarrow\;&\texttt{catch}\,\alpha\,.\,(\texttt{throw}\;\alpha\;\underline{n})\;(\texttt{lrec}\;0\;(\lambda\,\_\,x\,.\,\texttt{throw}\;\alpha\;x)\;\underline{n})\\
\twoheadrightarrow\;&\texttt{catch}\,\alpha\,.\,\texttt{throw}\;\alpha\;\underline{n}
\end{aligned}$$

We use the rule $(\texttt{throw}\;\alpha\;t)\;r \to \texttt{throw}\;\alpha\;t$ to discard the recursive call

A more efficient predecessor in λ::`catch`

We redefine $\texttt{pred} : \texttt{N} \to \texttt{N}$ as

$$\texttt{pred} := \lambda n \,.\, \texttt{catch}\, \alpha \,.\, \texttt{nrec}\ 0\ (\lambda x \,.\, \texttt{throw}\ \alpha\ x)\ n$$

Using `catch` and `throw` it becomes more efficient

$$\begin{aligned}
& \texttt{pred}\ \underline{n+1} \\
\twoheadrightarrow\ & \texttt{catch}\, \alpha \,.\, \texttt{nrec}\ 0\ (\lambda x \,.\, \texttt{throw}\ \alpha\ x)\ (\texttt{S}\ \underline{n}) \\
\twoheadrightarrow\ & \texttt{catch}\, \alpha \,.\, (\lambda x \,.\, \texttt{throw}\ \alpha\ x)\ \underline{n}\ (\texttt{lrec}\ 0\ (\lambda\, \_\, x \,.\, \texttt{throw}\ \alpha\ x)\ \underline{n}) \\
\twoheadrightarrow\ & \texttt{catch}\, \alpha \,.\, (\texttt{throw}\ \alpha\ \underline{n})\ (\texttt{lrec}\ 0\ (\lambda\, \_\, x \,.\, \texttt{throw}\ \alpha\ x)\ \underline{n}) \\
\twoheadrightarrow\ & \texttt{catch}\, \alpha \,.\, \texttt{throw}\ \alpha\ \underline{n}
\end{aligned}$$

A more efficient predecessor in λ::`catch`

We redefine $\mathtt{pred} : \mathtt{N} \to \mathtt{N}$ as

$$\mathtt{pred} := \lambda n\,.\,\mathtt{catch}\,\alpha\,.\,\mathtt{nrec}\;0\;(\lambda x\,.\,\mathtt{throw}\;\alpha\;x)\;n$$

Using `catch` and `throw` it becomes more efficient

$$\begin{aligned}
&\mathtt{pred}\;\underline{n+1}\\
\twoheadrightarrow\;&\mathtt{catch}\,\alpha\,.\,\mathtt{nrec}\;0\;(\lambda x\,.\,\mathtt{throw}\;\alpha\;x)\;(\mathtt{S}\;\underline{n})\\
\twoheadrightarrow\;&\mathtt{catch}\,\alpha\,.\,(\lambda x\,.\,\mathtt{throw}\;\alpha\;x)\;\underline{n}\;(\mathtt{lrec}\;0\;(\lambda\,\_\,x\,.\,\mathtt{throw}\;\alpha\;x)\;\underline{n})\\
\twoheadrightarrow\;&\mathtt{catch}\,\alpha\,.\,(\mathtt{throw}\;\alpha\;\underline{n})\;(\mathtt{lrec}\;0\;(\lambda\,\_\,x\,.\,\mathtt{throw}\;\alpha\;x)\;\underline{n})\\
\twoheadrightarrow\;&\mathtt{catch}\,\alpha\,.\,\mathtt{throw}\;\alpha\;\underline{n}\\
\twoheadrightarrow\;&\underline{n}
\end{aligned}$$

Example: list multiplication

- We want to define $F : [\mathtt{N}] \to \mathtt{N}$ such that

$$F[t_1, \ldots, t_n] = t_1 * \ldots * t_n$$

Example: list multiplication

- We want to define $F : [\mathtt{N}] \to \mathtt{N}$ such that

$$F[t_1, \ldots, t_n] = t_1 * \ldots * t_n$$

- The straightforward definition

$$F := \mathtt{lrec}\; \underline{1}\; (\lambda x \mathbin{\_} h \,.\, x * h)$$

  continues to multiply once a zero has been encountered

Example: list multiplication

- We want to define $F : [\mathtt{N}] \to \mathtt{N}$ such that

$$F[t_1, \ldots, t_n] = t_1 * \ldots * t_n$$

- The straightforward definition

$$F := \mathtt{lrec}\, \underline{1}\, (\lambda x\, \_\, h\, .\, x * h)$$

  continues to multiply once a zero has been encountered

- We use control to jump out when we encounter a zero

$$\begin{aligned} F &:= \lambda l\, .\, \mathtt{catch}\, \alpha\, .\, \mathtt{lrec}\, \underline{1}\, H\, l \\ H &:= \lambda x\, \_\, .\, \mathtt{nrec}\, (\mathtt{throw}\, \alpha\, 0)\, (\lambda y\, \_\, h\, .\, \mathtt{S}\, y * h)\, x \end{aligned}$$

Example: list multiplication (continued)

The definition of list multiplication:

$$F := \lambda l\,.\,\texttt{catch}\,\alpha\,.\,\texttt{lrec}\,\underline{1}\;H\;l$$

$$H := \lambda x\,\_\,.\,\texttt{nrec}\,(\texttt{throw}\,\alpha\,\texttt{0})\,(\lambda y\,\_\,h\,.\,\texttt{S}\,y * h)\;x$$

A computation of $F\,[\underline{4}, \underline{0}, \underline{9}]$:

Example: list multiplication (continued)

The definition of list multiplication:

$$F := \lambda l\,.\,\texttt{catch}\,\alpha\,.\,\texttt{lrec}\;\underline{1}\;H\;l$$
$$H := \lambda x\,\_\,.\,\texttt{nrec}\,(\texttt{throw}\,\alpha\,\texttt{0})\,(\lambda y\,\_\,h\,.\,\texttt{S}\,y * h)\;x$$

A computation of $F\,[\underline{4},\underline{0},\underline{9}]$:

$$\twoheadrightarrow \texttt{catch}\,\alpha\,.\,\texttt{lrec}\;\underline{1}\;H\;[\underline{4},\underline{0},\underline{9}]$$

Example: list multiplication (continued)

The definition of list multiplication:

$$F := \lambda l\,.\,\texttt{catch}\,\alpha\,.\,\texttt{lrec}\,\underline{1}\;H\;l$$

$$H := \lambda x\,\_\,.\,\texttt{nrec}\,(\texttt{throw}\,\alpha\;0)\,(\lambda y\,\_\,h\,.\,\texttt{S}\,y * h)\;x$$

A computation of $F\,[\underline{4},\underline{0},\underline{9}]$:

$$\twoheadrightarrow \texttt{catch}\,\alpha\,.\,\texttt{lrec}\,\underline{1}\;H\;[\underline{4},\underline{0},\underline{9}]$$

Example: list multiplication (continued)

The definition of list multiplication:

$$\begin{aligned} F &:= \lambda l\,.\,\texttt{catch}\;\alpha\,.\,\texttt{lrec}\;\underline{1}\;H\;l \\ H &:= \lambda x\;\_\,.\,\texttt{nrec}\;(\texttt{throw}\;\alpha\;\texttt{0})\;(\lambda y\;\_\;h\,.\,\texttt{S}\;y * h)\;x \end{aligned}$$

A computation of $F\,[\underline{4},\underline{0},\underline{9}]$:

$$\begin{aligned} &\twoheadrightarrow \texttt{catch}\;\alpha\,.\,\texttt{lrec}\;\underline{1}\;H\;[\underline{4},\underline{0},\underline{9}] \\ &\twoheadrightarrow \texttt{catch}\;\alpha\,.\,\texttt{nrec}\;(\texttt{throw}\;\alpha\;\texttt{0})\;(\lambda y\;\_\;h\,.\,\texttt{S}\;y * h)\;\underline{4}\;(\texttt{lrec}\;\underline{1}\;H\;[\underline{0},\underline{9}]) \end{aligned}$$

Example: list multiplication (continued)

The definition of list multiplication:

$$\begin{aligned} F &:= \lambda l\,.\,\texttt{catch}\,\alpha\,.\,\texttt{lrec}\;\underline{1}\;H\;l \\ H &:= \lambda x\,\_\,.\,\texttt{nrec}\,(\texttt{throw}\,\alpha\;\texttt{0})\,(\lambda y\,\_\,h\,.\,\texttt{S}\,y * h)\;x \end{aligned}$$

A computation of $F\,[\underline{4}, \underline{0}, \underline{9}]$:

$\twoheadrightarrow \texttt{catch}\,\alpha\,.\,\texttt{lrec}\;\underline{1}\;H\;[\underline{4}, \underline{0}, \underline{9}]$

$\twoheadrightarrow \texttt{catch}\,\alpha\,.\,\texttt{nrec}\,(\texttt{throw}\,\alpha\;\texttt{0})\,(\lambda y\,\_\,h\,.\,\texttt{S}\,y * h)\;\underline{4}\;(\texttt{lrec}\;\underline{1}\;H\;[\underline{0}, \underline{9}])$

Example: list multiplication (continued)

The definition of list multiplication:

$$\begin{aligned} F &:= \lambda l \,.\, \texttt{catch}\,\alpha\,.\,\texttt{lrec}\,\underline{1}\;H\;l \\ H &:= \lambda x\,\_\,.\,\texttt{nrec}\,(\texttt{throw}\,\alpha\,\texttt{0})\,(\lambda y\,\_\,h\,.\,\texttt{S}\,y * h)\;x \end{aligned}$$

A computation of $F\,[\underline{4},\underline{0},\underline{9}]$:

$$\begin{aligned} &\twoheadrightarrow \texttt{catch}\,\alpha\,.\,\texttt{lrec}\,\underline{1}\;H\;[\underline{4},\underline{0},\underline{9}] \\ &\twoheadrightarrow \texttt{catch}\,\alpha\,.\,\texttt{nrec}\,(\texttt{throw}\,\alpha\,\texttt{0})\,(\lambda y\,\_\,h\,.\,\texttt{S}\,y * h)\;\underline{4}\;(\texttt{lrec}\,\underline{1}\;H\;[\underline{0},\underline{9}]) \\ &\twoheadrightarrow \texttt{catch}\,\alpha\,.\,(\lambda h\,.\,\underline{4} * h)\;(\texttt{lrec}\,\underline{1}\;H\;[\underline{0},\underline{9}]) \end{aligned}$$

Example: list multiplication (continued)

The definition of list multiplication:

$$\begin{aligned} F &:= \lambda l \,.\, \texttt{catch}\, \alpha \,.\, \texttt{lrec}\, \underline{1}\ H\ l \\ H &:= \lambda x\, \_ \,.\, \texttt{nrec}\, (\texttt{throw}\, \alpha\, \texttt{0})\, (\lambda y\, \_\, h \,.\, \texttt{S}\, y * h)\, x \end{aligned}$$

A computation of $F\,[\underline{4}, \underline{0}, \underline{9}]$:

$$\begin{aligned} &\twoheadrightarrow \texttt{catch}\, \alpha \,.\, \texttt{lrec}\, \underline{1}\ H\ [\underline{4}, \underline{0}, \underline{9}] \\ &\twoheadrightarrow \texttt{catch}\, \alpha \,.\, \texttt{nrec}\, (\texttt{throw}\, \alpha\, \texttt{0})\, (\lambda y\, \_\, h \,.\, \texttt{S}\, y * h)\ \underline{4}\ (\texttt{lrec}\, \underline{1}\ H\ [\underline{0}, \underline{9}]) \\ &\twoheadrightarrow \texttt{catch}\, \alpha \,.\, (\lambda h \,.\, \underline{4} * h)\, (\texttt{lrec}\, \underline{1}\ H\ [\underline{0}, \underline{9}]) \end{aligned}$$

Example: list multiplication (continued)

The definition of list multiplication:

$$F := \lambda l \,.\, \texttt{catch}\, \alpha \,.\, \texttt{lrec}\, \underline{1}\ H\ l$$
$$H := \lambda x \,\_\,.\, \texttt{nrec}\, (\texttt{throw}\, \alpha\ \texttt{0})\ (\lambda y \,\_\, h \,.\, \texttt{S}\, y * h)\ x$$

A computation of $F\,[\underline{4}, \underline{0}, \underline{9}]$:

$$\twoheadrightarrow \texttt{catch}\, \alpha \,.\, \texttt{lrec}\, \underline{1}\ H\ [\underline{4}, \underline{0}, \underline{9}]$$
$$\twoheadrightarrow \texttt{catch}\, \alpha \,.\, \texttt{nrec}\, (\texttt{throw}\, \alpha\ \texttt{0})\ (\lambda y \,\_\, h \,.\, \texttt{S}\, y * h)\ \underline{4}\ (\texttt{lrec}\, \underline{1}\ H\ [\underline{0}, \underline{9}])$$
$$\twoheadrightarrow \texttt{catch}\, \alpha \,.\, (\lambda h \,.\, \underline{4} * h)\ (\texttt{lrec}\, \underline{1}\ H\ [\underline{0}, \underline{9}])$$
$$\twoheadrightarrow \texttt{catch}\, \alpha \,.\, (\lambda h \,.\, \underline{4} * h)\ (\texttt{throw}\, \alpha\ \texttt{0})$$

Example: list multiplication (continued)

The definition of list multiplication:

$$F := \lambda l \,.\, \mathtt{catch}\ \alpha \,.\, \mathtt{lrec}\ \underline{1}\ H\ l$$
$$H := \lambda x \mathbin{\_} .\, \mathtt{nrec}\ (\mathtt{throw}\ \alpha\ \mathtt{0})\ (\lambda y \mathbin{\_} h \,.\, \mathtt{S}\, y * h)\ x$$

A computation of $F\,[\underline{4}, \underline{0}, \underline{9}]$:

$$\twoheadrightarrow \mathtt{catch}\ \alpha \,.\, \mathtt{lrec}\ \underline{1}\ H\ [\underline{4}, \underline{0}, \underline{9}]$$
$$\twoheadrightarrow \mathtt{catch}\ \alpha \,.\, \mathtt{nrec}\ (\mathtt{throw}\ \alpha\ \mathtt{0})\ (\lambda y \mathbin{\_} h \,.\, \mathtt{S}\, y * h)\ \underline{4}\ (\mathtt{lrec}\ \underline{1}\ H\ [\underline{0}, \underline{9}])$$
$$\twoheadrightarrow \mathtt{catch}\ \alpha \,.\, (\lambda h \,.\, \underline{4} * h)\ (\mathtt{lrec}\ \underline{1}\ H\ [\underline{0}, \underline{9}])$$
$$\twoheadrightarrow \mathtt{catch}\ \alpha \,.\, (\lambda h \,.\, \underline{4} * h)\ (\mathtt{throw}\ \alpha\ \mathtt{0})$$

Example: list multiplication (continued)

The definition of list multiplication:

$$\begin{aligned} F &:= \lambda l \,.\, \texttt{catch}\, \alpha \,.\, \texttt{lrec}\; \underline{1}\; H\; l \\ H &:= \lambda x \,\_\,.\, \texttt{nrec}\; (\texttt{throw}\; \alpha\; \texttt{0})\; (\lambda y \,\_\, h \,.\, \texttt{S}\; y * h)\; x \end{aligned}$$

A computation of $F\,[\underline{4}, \underline{0}, \underline{9}]$:

$$\begin{aligned} &\twoheadrightarrow \texttt{catch}\, \alpha \,.\, \texttt{lrec}\; \underline{1}\; H\; [\underline{4}, \underline{0}, \underline{9}] \\ &\twoheadrightarrow \texttt{catch}\, \alpha \,.\, \texttt{nrec}\; (\texttt{throw}\; \alpha\; \texttt{0})\; (\lambda y \,\_\, h \,.\, \texttt{S}\; y * h)\; \underline{4}\; (\texttt{lrec}\; \underline{1}\; H\; [\underline{0}, \underline{9}]) \\ &\twoheadrightarrow \texttt{catch}\, \alpha \,.\, (\lambda h \,.\, \underline{4} * h)\; (\texttt{lrec}\; \underline{1}\; H\; [\underline{0}, \underline{9}]) \\ &\twoheadrightarrow \texttt{catch}\, \alpha \,.\, (\lambda h \,.\, \underline{4} * h)\; (\texttt{throw}\; \alpha\; \texttt{0}) \\ &\twoheadrightarrow \texttt{catch}\, \alpha \,.\, \texttt{throw}\; \alpha\; \texttt{0} \end{aligned}$$

Example: list multiplication (continued)

The definition of list multiplication:

$$F := \lambda l \,.\, \texttt{catch}\, \alpha \,.\, \texttt{lrec}\, \underline{1}\; H\; l$$

$$H := \lambda x\, \_ \,.\, \texttt{nrec}\, (\texttt{throw}\, \alpha\, \mathsf{0})\, (\lambda y\, \_\, h \,.\, \texttt{S}\, y * h)\; x$$

A computation of $F\,[\underline{4}, \underline{0}, \underline{9}]$:

$$\twoheadrightarrow \texttt{catch}\, \alpha \,.\, \texttt{lrec}\, \underline{1}\; H\; [\underline{4}, \underline{0}, \underline{9}]$$

$$\twoheadrightarrow \texttt{catch}\, \alpha \,.\, \texttt{nrec}\, (\texttt{throw}\, \alpha\, \mathsf{0})\, (\lambda y\, \_\, h \,.\, \texttt{S}\, y * h)\; \underline{4}\; (\texttt{lrec}\, \underline{1}\; H\; [\underline{0}, \underline{9}])$$

$$\twoheadrightarrow \texttt{catch}\, \alpha \,.\, (\lambda h \,.\, \underline{4} * h)\, (\texttt{lrec}\, \underline{1}\; H\; [\underline{0}, \underline{9}])$$

$$\twoheadrightarrow \texttt{catch}\, \alpha \,.\, (\lambda h \,.\, \underline{4} * h)\, (\texttt{throw}\, \alpha\, \mathsf{0})$$

$$\twoheadrightarrow \texttt{catch}\, \alpha \,.\, \texttt{throw}\, \alpha\, \mathsf{0}$$

Example: list multiplication (continued)

The definition of list multiplication:

$$\begin{aligned} F &:= \lambda l \,.\, \mathtt{catch}\, \alpha \,.\, \mathtt{lrec}\ \underline{1}\ H\ l \\ H &:= \lambda x\, \_ \,.\, \mathtt{nrec}\ (\mathtt{throw}\ \alpha\ \mathtt{0})\ (\lambda y\, \_\, h \,.\, \mathtt{S}\, y * h)\ x \end{aligned}$$

A computation of $F\,[\underline{4}, \underline{0}, \underline{9}]$:

$$\begin{aligned} &\twoheadrightarrow \mathtt{catch}\, \alpha \,.\, \mathtt{lrec}\ \underline{1}\ H\ [\underline{4}, \underline{0}, \underline{9}] \\ &\twoheadrightarrow \mathtt{catch}\, \alpha \,.\, \mathtt{nrec}\ (\mathtt{throw}\ \alpha\ \mathtt{0})\ (\lambda y\, \_\, h \,.\, \mathtt{S}\, y * h)\ \underline{4}\ (\mathtt{lrec}\ \underline{1}\ H\ [\underline{0}, \underline{9}]) \\ &\twoheadrightarrow \mathtt{catch}\, \alpha \,.\, (\lambda h \,.\, \underline{4} * h)\ (\mathtt{lrec}\ \underline{1}\ H\ [\underline{0}, \underline{9}]) \\ &\twoheadrightarrow \mathtt{catch}\, \alpha \,.\, (\lambda h \,.\, \underline{4} * h)\ (\mathtt{throw}\ \alpha\ \mathtt{0}) \\ &\twoheadrightarrow \mathtt{catch}\, \alpha \,.\, \mathtt{throw}\ \alpha\ \mathtt{0} \\ &\twoheadrightarrow \mathtt{0} \end{aligned}$$

Properties of λ::`catch`

- **Subject reduction.**
  $\Gamma; \Delta \vdash t : \rho$ and $t \to t'$, then $\Gamma; \Delta \vdash t' : \rho$
  - A standard substitution lemma is needed
  - Induction on the structure of $t \to t'$

Properties of λ::`catch`

- **Subject reduction.**
  $\Gamma; \Delta \vdash t : \rho$ and $t \rightarrow t'$, then $\Gamma; \Delta \vdash t' : \rho$
  - A standard substitution lemma is needed
  - Induction on the structure of $t \rightarrow t'$
- **Progress.**
  $; \vdash t : \rho$, then t is a value or $\exists t', t \rightarrow t'$
  - A simple generalization is needed
  - Induction on the typing judgment.

Properties of λ::`catch`

- **Subject reduction.**
  $\Gamma; \Delta \vdash t : \rho$ and $t \rightarrow t'$, then $\Gamma; \Delta \vdash t' : \rho$
  - A standard substitution lemma is needed
  - Induction on the structure of $t \rightarrow t'$
- **Progress.**
  $; \vdash t : \rho$, then t is a value or $\exists t', t \rightarrow t'$
  - A simple generalization is needed
  - Induction on the typing judgment.
- **Confluence.**
  $t \twoheadrightarrow r$ and $t \twoheadrightarrow s$, then $\exists q \,.\, r \twoheadrightarrow q$ and $s \twoheadrightarrow q$

Properties of λ::`catch`

- **Subject reduction.**
  $\Gamma; \Delta \vdash t : \rho$ and $t \to t'$, then $\Gamma; \Delta \vdash t' : \rho$
  - A standard substitution lemma is needed
  - Induction on the structure of $t \to t'$
- **Progress.**
  $; \vdash t : \rho$, then t is a value or $\exists t', t \to t'$
  - A simple generalization is needed
  - Induction on the typing judgment.
- **Confluence.**
  $t \twoheadrightarrow r$ and $t \twoheadrightarrow s$, then $\exists q \,.\, r \twoheadrightarrow q$ and $s \twoheadrightarrow q$
- **Strong Normalization.**
  $\Gamma; \Delta \vdash t : \rho$, then no infinite $t \to t_1 \ldots$

Parallel reduction

Usual approach [Tait/Martin-Löf]

1. Define a parallel reduction $\Rightarrow$
2. Prove that $\Rightarrow$ is confluent
3. Prove that $t_1 \to t_2$ implies $t_1 \Rightarrow t_2$
4. Prove that $t_1 \Rightarrow t_2$ implies $t_1 \twoheadrightarrow t_2$

Parallel reduction

Usual approach [Tait/Martin-Löf]

1. Define a parallel reduction $\Rightarrow$
2. Prove that $\Rightarrow$ is confluent
3. Prove that $t_1 \rightarrow t_2$ implies $t_1 \Rightarrow t_2$
4. Prove that $t_1 \Rightarrow t_2$ implies $t_1 \twoheadrightarrow t_2$

For the ordinary λ-calculus

$$\frac{}{x \Rightarrow x} \qquad \frac{t \Rightarrow t'}{\lambda x.t \Rightarrow \lambda x.t'} \qquad \frac{t \Rightarrow t' \qquad r \Rightarrow r'}{tr \Rightarrow t'r'}$$

$$\frac{t \Rightarrow t' \qquad r \Rightarrow r'}{(\lambda x.t)r \Rightarrow t'[x := r']}$$

Parallel reduction for λ::`catch`

- Consider the naive rule for `throw`

$$\frac{t \Rightarrow t'}{E[\texttt{throw}\ \alpha\ t] \Rightarrow \texttt{throw}\ \alpha\ t'}$$

Parallel reduction for λ::`catch`

- Consider the naive rule for `throw`

$$\frac{t \Rightarrow t'}{E[\texttt{throw}\ \alpha\ t] \Rightarrow \texttt{throw}\ \alpha\ t'}$$

- **Problem:** not confluent

$$\begin{tikzcd}
& \texttt{throw}\ \alpha_1\ (\texttt{throw}\ \alpha_2\ (\texttt{throw}\ \alpha_3\ (\texttt{throw}\ \alpha_4\ (\texttt{throw}\ \alpha_5\ x)))) \arrow[dl, Rightarrow] \arrow[dd, Rightarrow] \\
\texttt{throw}\ \alpha_1\ (\texttt{throw}\ \alpha_3\ (\texttt{throw}\ \alpha_5\ x)) & \\
& \texttt{throw}\ \alpha_2\ (\texttt{throw}\ \alpha_4\ (\texttt{throw}\ \alpha_5\ x))
\end{tikzcd}$$

Parallel reduction for λ::`catch`

- Consider the naive rule for `throw`

$$\frac{t \Rightarrow t'}{E[\texttt{throw}\ \alpha\ t] \Rightarrow \texttt{throw}\ \alpha\ t'}$$

- **Problem:** not confluent

$$\begin{tikzcd}
 & \texttt{throw}\ \alpha_1\ (\texttt{throw}\ \alpha_2\ (\texttt{throw}\ \alpha_3\ (\texttt{throw}\ \alpha_4\ (\texttt{throw}\ \alpha_5\ x)))) \arrow[dl, Rightarrow] \arrow[dd, Rightarrow] \\
\texttt{throw}\ \alpha_1\ (\texttt{throw}\ \alpha_3\ (\texttt{throw}\ \alpha_5\ x)) & \\
 & \texttt{throw}\ \alpha_2\ (\texttt{throw}\ \alpha_4\ (\texttt{throw}\ \alpha_5\ x))
\end{tikzcd}$$

- **Solution:** jump over a compound context

$$\frac{t \Rightarrow t'}{\vec{E}[\texttt{throw}\ \alpha\ t] \Rightarrow \texttt{throw}\ \alpha\ t'}$$

Complete development

Define $t^{\diamond}$ such that if $t_1 \Rightarrow t_2$, then $t_2 \Rightarrow t_1^{\diamond}$ [Takahashi, 1995]

Complete development

Define $t^\diamond$ such that if $t_1 \Rightarrow t_2$, then $t_2 \Rightarrow t_1^\diamond$ [Takahashi, 1995]

Confluence of $\Rightarrow$ is a direct consequence

$$\begin{tikzcd}
 & t_1 \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
t_2 & & t_3
\end{tikzcd}$$

Complete development

Define $t^\diamond$ such that if $t_1 \Rightarrow t_2$, then $t_2 \Rightarrow t_1^\diamond$ [Takahashi, 1995]

Confluence of $\Rightarrow$ is a direct consequence

$$\begin{tikzcd}
 & t_1 \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
t_2 \arrow[dr, Rightarrow, dashed] & & t_3 \arrow[dl, Rightarrow, dashed] \\
 & t_1^\diamond &
\end{tikzcd}$$

Complete development for λ::`catch`

$$\begin{aligned}
((\lambda x.t)\,v)^\diamond &:= t^\diamond[x := v^\diamond] \\
(\vec{E}[\mathtt{throw}\;\alpha\;t])^\diamond &:= \mathtt{throw}\;\alpha\;t^\diamond \text{ if } t \not\equiv \mathtt{throw}\;\gamma\;s \\
(\mathtt{catch}\;\alpha\,.\,\mathtt{throw}\;\alpha\;t)^\diamond &:= \mathtt{catch}\;\alpha\,.\,t^\diamond \\
(\mathtt{catch}\;\alpha\,.\,\mathtt{throw}\;\beta\;v)^\diamond &:= \mathtt{throw}\;\beta\;v^\diamond \text{ if } \alpha \notin \{\beta\} \cup \mathsf{FCV}(v) \\
(\mathtt{catch}\;\alpha\,.\,v)^\diamond &:= v^\diamond \qquad\qquad \text{if } \alpha \notin \mathsf{FCV}(v) \\
(\mathtt{lrec}\;v_r\;v_s\;\mathtt{nil})^\diamond &:= v_r^\diamond \\
(\mathtt{lrec}\;v_r\;v_s\;(v_h :: v_t))^\diamond &:= v_s^\diamond\;v_h^\diamond\;v_t^\diamond\;(\mathtt{lrec}\;v_r^\diamond\;v_s^\diamond\;v_t^\diamond) \\
&\dots
\end{aligned}$$

Strong Normalization

The interpretation $\llbracket \rho \rrbracket$ of a type ρ is defined as:

$$\begin{aligned} \llbracket \top \rrbracket &:= \mathtt{SN} \\ \llbracket \sigma \to \tau \rrbracket &:= \{ t \mid \forall s \in \llbracket \sigma \rrbracket \, . \; ts \in \llbracket \tau \rrbracket \} \end{aligned}$$

Strong Normalization

The interpretation $[\![\rho]\!]$ of a type ρ is defined as:

$$
\begin{aligned}
[\![\top]\!] &:= \mathtt{SN} \\
[\![\sigma \to \tau]\!] &:= \{t \mid \forall s \in [\![\sigma]\!] \,.\, ts \in [\![\tau]\!]\} \\
[\![[\sigma]]\!] &:=
\end{aligned}
$$

Strong Normalization

The interpretation $[\![\rho]\!]$ of a type ρ is defined as:

$$\begin{aligned}
[\![\top]\!] &:= \mathtt{SN} \\
[\![\sigma \to \tau]\!] &:= \{ t \mid \forall s \in [\![\sigma]\!] \ . \ ts \in [\![\tau]\!] \} \\
[\![[\sigma]]\!] &:= \mathtt{SN}
\end{aligned}$$

Strong Normalization

The interpretation $[\![\rho]\!]$ of a type ρ is defined as:

$$\begin{aligned} [\![\top]\!] &:= \mathtt{SN} \\ [\![\sigma \to \tau]\!] &:= \{t \mid \forall s \in [\![\sigma]\!] \ . \ ts \in [\![\tau]\!]\} \\ [\![[\sigma]]\!] &:= \mathtt{SN} \cap \mathcal{L}_{[\![\sigma]\!]} \end{aligned}$$

where for a set of terms S, the set of terms $\mathcal{L}_S$ is defined as

$$\frac{\forall v\ w\ .\ \text{if } t \twoheadrightarrow v :: w \text{ then } v \in S \text{ and } w \in \mathcal{L}_S}{t \in \mathcal{L}_S}$$

Strong Normalization

The interpretation $\llbracket \rho \rrbracket$ of a type ρ is defined as:

$$\begin{aligned} \llbracket \top \rrbracket &:= \mathtt{SN} \\ \llbracket \sigma \to \tau \rrbracket &:= \{ t \mid \forall s \in \llbracket \sigma \rrbracket \, . \, ts \in \llbracket \tau \rrbracket \} \\ \llbracket [\sigma] \rrbracket &:= \mathtt{SN} \cap \mathcal{L}_{\llbracket \sigma \rrbracket} \end{aligned}$$

where for a set of terms S, the set of terms $\mathcal{L}_S$ is defined as

$$\frac{\forall v \; w \; . \text{ if } t \twoheadrightarrow v :: w \text{ then } v \in S \text{ and } w \in \mathcal{L}_S}{t \in \mathcal{L}_S}$$

Key lemmas

- $\llbracket \psi \rrbracket = \mathtt{SN}$ for ψ $\to$-free

Strong Normalization

The interpretation $[\![\rho]\!]$ of a type ρ is defined as:

$$\begin{aligned} [\![\top]\!] &:= \mathtt{SN} \\ [\![\sigma \to \tau]\!] &:= \{t \mid \forall s \in [\![\sigma]\!] \;.\; ts \in [\![\tau]\!]\} \\ [\![[\sigma]]\!] &:= \mathtt{SN} \cap \mathcal{L}_{[\![\sigma]\!]} \end{aligned}$$

where for a set of terms S, the set of terms $\mathcal{L}_S$ is defined as

$$\frac{\forall v\ w \;.\; \text{if } t \twoheadrightarrow v :: w \text{ then } v \in S \text{ and } w \in \mathcal{L}_S}{t \in \mathcal{L}_S}$$

Key lemmas

- $[\![\psi]\!] = \mathtt{SN}$ for ψ $\to$-free
- If $r \in [\![\psi]\!]$, then $\mathtt{catch}\,\alpha\,.\,r \in [\![\psi]\!]$

Strong Normalization

The interpretation $[\![\rho]\!]$ of a type ρ is defined as:

$$\begin{aligned} [\![\top]\!] &:= \mathtt{SN} \\ [\![\sigma \to \tau]\!] &:= \{t \mid \forall s \in [\![\sigma]\!] \,.\, ts \in [\![\tau]\!]\} \\ [\![[\sigma]]\!] &:= \mathtt{SN} \cap \mathcal{L}_{[\![\sigma]\!]} \end{aligned}$$

where for a set of terms S, the set of terms $\mathcal{L}_S$ is defined as

$$\frac{\forall v\ w \,.\, \text{if } t \twoheadrightarrow v :: w \text{ then } v \in S \text{ and } w \in \mathcal{L}_S}{t \in \mathcal{L}_S}$$

Key lemmas

- $[\![\psi]\!] = \mathtt{SN}$ for ψ $\to$-free
- If $r \in [\![\psi]\!]$, then $\mathtt{catch}\,\alpha\,.\,r \in [\![\psi]\!]$
- If $r \in \mathtt{SN}$ and $t[x := r] \in [\![\sigma]\!]$, then $(\lambda x.t)\,r \in [\![\sigma]\!]$

Strong Normalization

The interpretation $[\![\rho]\!]$ of a type ρ is defined as:

$$\begin{aligned} [\![\top]\!] &:= \mathtt{SN} \\ [\![\sigma \to \tau]\!] &:= \{t \mid \forall s \in [\![\sigma]\!] \,.\, ts \in [\![\tau]\!]\} \\ [\![[\sigma]]\!] &:= \mathtt{SN} \cap \mathcal{L}_{[\![\sigma]\!]} \end{aligned}$$

where for a set of terms S, the set of terms $\mathcal{L}_S$ is defined as

$$\frac{\forall v\ w \,.\, \text{if } t \twoheadrightarrow v :: w \text{ then } v \in S \text{ and } w \in \mathcal{L}_S}{t \in \mathcal{L}_S}$$

Key lemmas

- $[\![\psi]\!] = \mathtt{SN}$ for ψ $\to$-free
- If $r \in [\![\psi]\!]$, then $\mathtt{catch}\, \alpha \,.\, r \in [\![\psi]\!]$
- If $r \in \mathtt{SN}$ and $t[x := r] \in [\![\sigma]\!]$, then $(\lambda x.t)\, r \in [\![\sigma]\!]$
- If $x_1 : \rho_1, \ldots, x_n : \rho_n; \Delta \vdash t : \tau$ and $r_i \in [\![\rho_i]\!]$ for all $1 \leq i \leq n$, then $t[x_1 := r_1, \ldots, x_n := r_n] \in [\![\tau]\!]$

Future work

- More interesting data types

Future work

- More interesting data types
- Pattern match and `fix` construct

Future work

- More interesting data types
- Pattern match and `fix` construct
- Dynamically bound exceptions

Future work

- More interesting data types
- Pattern match and `fix` construct
- Dynamically bound exceptions
- Corresponding abstract machine

Future work

- More interesting data types
- Pattern match and `fix` construct
- Dynamically bound exceptions
- Corresponding abstract machine
- Polymorphism

Future work

- More interesting data types
- Pattern match and `fix` construct
- Dynamically bound exceptions
- Corresponding abstract machine
- Polymorphism
- Dependent types

Future work

- More interesting data types
- Pattern match and `fix` construct
- Dynamically bound exceptions
- Corresponding abstract machine
- Polymorphism
- Dependent types
- Program extraction à la Paulin/Letouzey