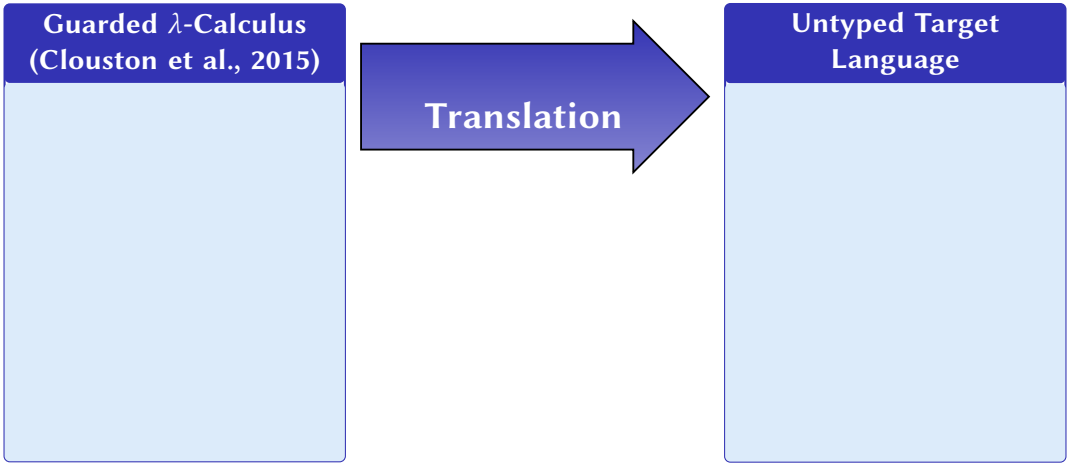
Verified Translation of Guarded Programs

David Läwen, joint work with Robbert Krebbers and Bálint Kocsis

Radboud University

PLNL @ CWI Amsterdam, 28 November 2025

The Main Idea



The Main Idea

Guarded λ-Calculus (Clouston et al., 2015)

- STLC with guarded recursive and coinductive types

Untyped Target Language

The Main Idea

Guarded λ-Calculus (Clouston et al., 2015)

- STLC with guarded recursive and coinductive types
- Call-by-name

Untyped Target Language

The Main Idea

Guarded λ-Calculus (Clouston et al., 2015)

- STLC with guarded recursive and coinductive types
- Call-by-name
- Well-typed programs terminate

Untyped Target Language

The Main Idea

Guarded λ-Calculus (Clouston et al., 2015)

- STLC with guarded recursive and coinductive types
- Call-by-name
- Well-typed programs terminate

Untyped Target Language

- Untyped λ-calculus with extensions

The Main Idea

Guarded λ-Calculus (Clouston et al., 2015)

- STLC with guarded recursive and coinductive types
- Call-by-name
- Well-typed programs terminate

Untyped Target Language

- Untyped λ-calculus with extensions
- Call-by-value

The Main Idea

Guarded λ-Calculus (Clouston et al., 2015)

- STLC with guarded recursive and coinductive types
- Call-by-name
- Well-typed programs terminate

Untyped Target Language

- Untyped λ-calculus with extensions
- Call-by-value
- Resembles e.g. `flambda` intermediate language (OCaml)

The Main Idea

Guarded λ-Calculus (Clouston et al., 2015)

- STLC with guarded recursive and coinductive types
- Call-by-name
- Well-typed programs terminate

- Semantics-preserving

Untyped Target Language

- Untyped λ-calculus with extensions
- Call-by-value
- Resembles e.g. `flambda` intermediate language (OCaml)

The Main Idea

Guarded λ-Calculus (Clouston et al., 2015)

- STLC with guarded recursive and coinductive types
- Call-by-name
- Well-typed programs terminate

- Semantics-preserving
  - Termination

Untyped Target Language

- Untyped λ-calculus with extensions
- Call-by-value
- Resembles e.g. `flambda` intermediate language (OCaml)

The Main Idea

Guarded λ-Calculus (Clouston et al., 2015)

- STLC with guarded recursive and coinductive types
- Call-by-name
- Well-typed programs terminate

- Semantics-preserving
  - Termination
  - Result value

Untyped Target Language

- Untyped λ-calculus with extensions
- Call-by-value
- Resembles e.g. `flambda` intermediate language (OCaml)

The Main Idea

Guarded λ-Calculus (Clouston et al., 2015)

- STLC with guarded recursive and coinductive types
- Call-by-name
- Well-typed programs terminate

Verified Translation

- Semantics-preserving
  - Termination
  - Result value

Untyped Target Language

- Untyped λ-calculus with extensions
- Call-by-value
- Resembles e.g. `flambda` intermediate language (OCaml)

The Main Idea

Guarded λ-Calculus (Clouston et al., 2015)

- STLC with guarded recursive and coinductive types
- Call-by-name
- Well-typed programs terminate

Verified Translation

- Semantics-preserving
  - Termination
  - Result value
- Mechanised in Rocq using Iris framework

Untyped Target Language

- Untyped λ-calculus with extensions
- Call-by-value
- Resembles e.g. `flambda` intermediate language (OCaml)

The Guarded Lambda-Calculus

STLC with Unrestricted Recursive Types

STLC with iso-recursive types $\mu\alpha.\tau$, a.k.a. $\lambda\mu$-calculus

$$\texttt{fold} \;:\; \tau[\mu\alpha.\tau/\alpha] \to \mu\alpha.\tau \qquad\qquad \texttt{unfold} \;:\; \mu\alpha.\tau \to \tau[\mu\alpha.\tau/\tau]$$

STLC with Unrestricted Recursive Types

STLC with iso-recursive types $\mu\alpha.\tau$, a.k.a. $\lambda\mu$-calculus

$$\texttt{fold} \;:\; \tau[\mu\alpha.\tau/\alpha] \to \mu\alpha.\tau \qquad\qquad \texttt{unfold} \;:\; \mu\alpha.\tau \to \tau[\mu\alpha.\tau/\tau]$$

E.g. can express streams over τ

$$\mathrm{Str}\ \tau \;\triangleq\; \mu\alpha.\ \tau \times \alpha \qquad\qquad (::) \;:\; \tau \to \mathrm{Str}\ \tau \to \mathrm{Str}\ \tau$$

STLC with Unrestricted Recursive Types

STLC with iso-recursive types $\mu\alpha.\tau$, a.k.a. $\lambda\mu$-calculus

$$\texttt{fold} \;:\; \tau[\mu\alpha.\tau/\alpha] \to \mu\alpha.\tau \qquad\qquad \texttt{unfold} \;:\; \mu\alpha.\tau \to \tau[\mu\alpha.\tau/\tau]$$

E.g. can express streams over τ

$$\mathrm{Str}\ \tau \;\triangleq\; \mu\alpha.\ \tau \times \alpha \qquad\qquad (::) \;:\; \tau \to \mathrm{Str}\ \tau \to \mathrm{Str}\ \tau$$

Construct streams as fixpoints

$$\mathrm{fix} \;:\; (\tau \to \tau) \to \tau$$

STLC with Unrestricted Recursive Types

STLC with iso-recursive types $\mu\alpha.\tau$, a.k.a. $\lambda\mu$-calculus

$$\texttt{fold} \;:\; \tau[\mu\alpha.\tau/\alpha] \to \mu\alpha.\tau \qquad\qquad \texttt{unfold} \;:\; \mu\alpha.\tau \to \tau[\mu\alpha.\tau/\tau]$$

E.g. can express streams over τ

$$\mathrm{Str}\ \tau \;\triangleq\; \mu\alpha.\ \tau \times \alpha \qquad\qquad (::) \;:\; \tau \to \mathrm{Str}\ \tau \to \mathrm{Str}\ \tau$$

Construct streams as fixpoints

$$\mathrm{fix} \;:\; (\tau \to \tau) \to \tau$$

$$\mathrm{zeroes} \;\triangleq\; \mathrm{fix}\ (\lambda s.\ 0 :: s) \;=_\beta\; 0 :: 0 :: 0 :: \ldots$$

STLC with Unrestricted Recursive Types

STLC with iso-recursive types $\mu\alpha.\tau$, a.k.a. $\lambda\mu$-calculus

$$\texttt{fold} \;:\; \tau[\mu\alpha.\tau/\alpha] \to \mu\alpha.\tau \qquad\qquad \texttt{unfold} \;:\; \mu\alpha.\tau \to \tau[\mu\alpha.\tau/\tau]$$

E.g. can express streams over τ

$$\text{Str}\ \tau \;\triangleq\; \mu\alpha.\ \tau \times \alpha \qquad\qquad (::) \;:\; \tau \to \text{Str}\ \tau \to \text{Str}\ \tau$$

Construct streams as fixpoints

$$\text{fix} \;:\; (\tau \to \tau) \to \tau$$

$$\text{zeroes} \;\triangleq\; \text{fix}\ (\lambda s.\ 0 :: s) \;=_\beta\; 0 :: 0 :: 0 :: \ldots$$

STLC with Unrestricted Recursive Types

STLC with iso-recursive types $\mu\alpha.\tau$, a.k.a. $\lambda\mu$-calculus

$$\texttt{fold} \;:\; \tau[\mu\alpha.\tau/\alpha] \to \mu\alpha.\tau \qquad\qquad \texttt{unfold} \;:\; \mu\alpha.\tau \to \tau[\mu\alpha.\tau/\tau]$$

E.g. can express streams over τ

$$\text{Str}\ \tau \;\triangleq\; \mu\alpha.\ \tau \times \alpha \qquad\qquad (::) \;:\; \tau \to \text{Str}\ \tau \to \text{Str}\ \tau$$

Construct streams as fixpoints

$$\text{fix} \;:\; (\tau \to \tau) \to \tau$$

$$\text{zeroes} \;\triangleq\; \text{fix}\ (\lambda s.\ 0 :: s) \;=_\beta\; 0 :: 0 :: 0 :: \ldots$$

$$\text{loop} \;\triangleq\; \text{fix}\ (\lambda s.\ s) \;\approx_\beta\; \Omega$$

STLC with Unrestricted Recursive Types

STLC with iso-recursive types $\mu\alpha.\tau$, a.k.a. $\lambda\mu$-calculus

$$\texttt{fold} \;:\; \tau[\mu\alpha.\tau/\alpha] \to \mu\alpha.\tau \qquad\qquad \texttt{unfold} \;:\; \mu\alpha.\tau \to \tau[\mu\alpha.\tau/\tau]$$

E.g. can express streams over τ

$$\text{Str}\ \tau \;\triangleq\; \mu\alpha.\ \tau \times \alpha \qquad\qquad (::) \;:\; \tau \to \text{Str}\ \tau \to \text{Str}\ \tau$$

Construct streams as fixpoints

$$\text{fix} \;:\; (\tau \to \tau) \to \tau$$

$$\text{zeroes} \;\triangleq\; \text{fix}\ (\lambda s.\ 0 :: s) \;=_\beta\; 0 :: 0 :: 0 :: \ldots$$

$$\text{loop} \;\triangleq\; \text{fix}\ (\lambda s.\ s) \;\approx_\beta\; \Omega$$

STLC with Unrestricted Recursive Types

STLC with iso-recursive types $\mu\alpha.\tau$, a.k.a. $\lambda\mu$-calculus

$$\texttt{fold} \;:\; \tau[\mu\alpha.\tau/\alpha] \to \mu\alpha.\tau \qquad\qquad \texttt{unfold} \;:\; \mu\alpha.\tau \to \tau[\mu\alpha.\tau/\tau]$$

E.g. can express streams over τ

$$\mathrm{Str}\ \tau \triangleq \mu\alpha.\ \tau \times \alpha \qquad\qquad (::) \;:\; \tau \to \mathrm{Str}\ \tau \to \mathrm{Str}\ \tau$$

Construct streams as fixpoints

$$\mathrm{fix} \;:\; (\tau \to \tau) \to \tau$$

$$\mathrm{zeroes} \triangleq \mathrm{fix}\ (\lambda s.\ 0 :: s) \;=_\beta\; 0 :: 0 :: 0 :: \ldots$$

$$\mathrm{loop} \triangleq \mathrm{fix}\ (\lambda s.\ s) \;\approx_\beta\; \Omega$$

Problem: Type system accepts diverging programs!

Guarded Recursive Types

Idea: Restrict recursive types with new type modality ▶ (‘later’)

Guarded Recursive Types

Idea: Restrict recursive types with new type modality ▶ (‘later’)

- In $\mu\alpha.\tau$, the α can only appear under a ▶

$$\mathrm{Str}^{\mathrm{g}}\ \tau \triangleq \mu\alpha.\ \tau \times \blacktriangleright\alpha \qquad\qquad (::)\ :\ \tau \to \blacktriangleright(\mathrm{Str}^{\mathrm{g}}\ \tau) \to \mathrm{Str}^{\mathrm{g}}\ \tau$$

Guarded Recursive Types

Idea: Restrict recursive types with new type modality $\blacktriangleright$ ('later')

- In $\mu\alpha.\tau$, the α can only appear under a $\blacktriangleright$

$$\mathrm{Str}^{\mathrm{g}}\ \tau \triangleq \mu\alpha.\ \tau \times \blacktriangleright\alpha \qquad\qquad (::)\ :\ \tau \to \blacktriangleright(\mathrm{Str}^{\mathrm{g}}\ \tau) \to \mathrm{Str}^{\mathrm{g}}\ \tau$$

Term can refer to itself only 'later'

$$\mathrm{fix}^{\mathrm{g}}\ :\ (\blacktriangleright\tau \to \tau) \to \tau$$

Guarded Recursive Types

Idea: Restrict recursive types with new type modality ▶ (‘later’)

- In $\mu\alpha.\tau$, the α can only appear under a ▶

$$\mathrm{Str}^{\mathrm{g}}\ \tau \triangleq \mu\alpha.\ \tau \times \blacktriangleright\alpha \qquad\qquad (::)\ :\ \tau \to \blacktriangleright(\mathrm{Str}^{\mathrm{g}}\ \tau) \to \mathrm{Str}^{\mathrm{g}}\ \tau$$

Term can refer to itself only ‘later’

$$\mathrm{fix}^{\mathrm{g}}\ :\ (\blacktriangleright\tau \to \tau) \to \tau$$

Examples:

$$\vdash \mathrm{fix}^{\mathrm{g}}\ (\lambda s : \blacktriangleright\ \mathrm{Str}^{\mathrm{g}}\ \mathbf{N}.\ 0 :: s) : \mathrm{Str}^{\mathrm{g}}\ \mathbf{N}$$

Guarded Recursive Types

Idea: Restrict recursive types with new type modality ▶ (‘later’)

- In $\mu\alpha.\tau$, the α can only appear under a ▶

$$\mathrm{Str}^{\mathrm{g}}\ \tau \triangleq \mu\alpha.\ \tau \times \blacktriangleright\alpha \qquad\qquad (::)\ :\ \tau \to \blacktriangleright(\mathrm{Str}^{\mathrm{g}}\ \tau) \to \mathrm{Str}^{\mathrm{g}}\ \tau$$

Term can refer to itself only ‘later’

$$\mathrm{fix}^{\mathrm{g}}\ :\ (\blacktriangleright\tau \to \tau) \to \tau$$

Examples:

$\vdash \mathrm{fix}^{\mathrm{g}}\ (\lambda s : \blacktriangleright\ \mathrm{Str}^{\mathrm{g}}\ \mathbf{N}.\ 0 :: s) : \mathrm{Str}^{\mathrm{g}}\ \mathbf{N}$ 

~~$\vdash$~~ $\mathrm{fix}^{\mathrm{g}}\ (\lambda s : \blacktriangleright\mathrm{Str}^{\mathrm{g}}\ \tau.\ s)$ 

Guarded Recursive Types

Idea: Restrict recursive types with new type modality ▶ (‘later’)

- In $\mu\alpha.\tau$, the α can only appear under a ▶

$$\mathrm{Str}^{\mathrm{g}}\ \tau \triangleq \mu\alpha.\ \tau \times \blacktriangleright\alpha \qquad\qquad (::)\ :\ \tau \to \blacktriangleright(\mathrm{Str}^{\mathrm{g}}\ \tau) \to \mathrm{Str}^{\mathrm{g}}\ \tau$$

Term can refer to itself only ‘later’

$$\mathrm{fix}^{\mathrm{g}}\ :\ (\blacktriangleright\tau \to \tau) \to \tau$$

Examples:

$\vdash \mathrm{fix}^{\mathrm{g}}\ (\lambda s : \blacktriangleright\ \mathrm{Str}^{\mathrm{g}}\ \mathbf{N}.\ 0 :: s) : \mathrm{Str}^{\mathrm{g}}\ \mathbf{N}$ 

~~$\vdash$~~ $\mathrm{fix}^{\mathrm{g}}\ (\lambda s : \blacktriangleright\mathrm{Str}^{\mathrm{g}}\ \tau.\ s)$ 

Problem solved: Diverging programs are rejected.

Guarded Term Formers

New term syntax:

$$\texttt{next} \;:\; \tau \to \blacktriangleright\tau \qquad\qquad \circledast \;:\; \blacktriangleright(\tau_1 \to \tau_2) \to \blacktriangleright\tau_1 \to \blacktriangleright\tau_2$$

Guarded Term Formers

New term syntax:

$$\texttt{next} \;:\; \tau \to \blacktriangleright\tau \qquad\qquad \circledast \;:\; \blacktriangleright(\tau_1 \to \tau_2) \to \blacktriangleright\tau_1 \to \blacktriangleright\tau_2$$

Needed for more interesting examples:

$$\begin{array}{rcll} \text{toggle} & \triangleq & \text{fix}^{\text{g}}\ (\lambda s.\ 1 :: \texttt{next}\ (0 :: s)) & : \quad \text{Str}^{\text{g}}\ \mathbf{N} \\ \text{interleave} & \triangleq & \text{fix}^{\text{g}}\ (\lambda f\ s\ t.\ \text{hd}\ s :: (f \circledast t \circledast \texttt{next}\ (\text{tl}\ s))) & : \quad \text{Str}^{\text{g}}\ \tau \to \blacktriangleright\text{Str}^{\text{g}}\ \tau \to \text{Str}^{\text{g}}\ \tau \end{array}$$

Guarded Term Formers

New term syntax:

$$\texttt{next} \;:\; \tau \to \blacktriangleright\tau \qquad\qquad \circledast \;:\; \blacktriangleright(\tau_1 \to \tau_2) \to \blacktriangleright\tau_1 \to \blacktriangleright\tau_2$$

Needed for more interesting examples:

$$x :: s \triangleq \texttt{fold}\ \langle x, s\rangle$$

$$\begin{aligned}
\text{toggle} &\triangleq \text{fix}^{\text{g}}\ (\lambda s.\ 1 :: \texttt{next}\ (0 :: s)) &&: \text{Str}^{\text{g}}\ \mathbf{N}\\
\text{interleave} &\triangleq \text{fix}^{\text{g}}\ (\lambda f\ s\ t.\ \text{hd}\ s :: (f \circledast t \circledast \texttt{next}\ (\text{tl}\ s))) &&: \text{Str}^{\text{g}}\ \tau \to \blacktriangleright\text{Str}^{\text{g}}\ \tau \to \text{Str}^{\text{g}}\ \tau
\end{aligned}$$

Guarded Term Formers

New term syntax:

$$\texttt{next} \;:\; \tau \to \blacktriangleright\tau \qquad\qquad \circledast \;:\; \blacktriangleright(\tau_1 \to \tau_2) \to \blacktriangleright\tau_1 \to \blacktriangleright\tau_2$$

Needed for more interesting examples:

$$x :: s \triangleq \texttt{fold}\ \langle x, s\rangle$$

$$\begin{aligned}
\text{toggle} &\triangleq \text{fix}^{\text{g}}\ (\lambda s.\ 1 :: \texttt{next}\ (0 :: s)) &&: \text{Str}^{\text{g}}\ \mathbf{N}\\
\text{interleave} &\triangleq \text{fix}^{\text{g}}\ (\lambda f\ s\ t.\ \text{hd}\ s :: (f \circledast t \circledast \texttt{next}\ (\text{tl}\ s))) &&: \text{Str}^{\text{g}}\ \tau \to \blacktriangleright\text{Str}^{\text{g}}\ \tau \to \text{Str}^{\text{g}}\ \tau
\end{aligned}$$

$$\text{hd}\ s \triangleq \texttt{proj}_1\ (\texttt{unfold}\ s)$$

Guarded Term Formers

New term syntax:

$$\texttt{next} \;:\; \tau \to \blacktriangleright\tau \qquad\qquad \circledast \;:\; \blacktriangleright(\tau_1 \to \tau_2) \to \blacktriangleright\tau_1 \to \blacktriangleright\tau_2$$

Needed for more interesting examples:

$$x :: s \triangleq \texttt{fold}\ \langle x, s\rangle$$

$$\begin{aligned}
\text{toggle} &\triangleq \text{fix}^{\text{g}}\ (\lambda s.\ 1 :: \texttt{next}\ (0 :: s)) &&: \text{Str}^{\text{g}}\ \mathbf{N}\\
\text{interleave} &\triangleq \text{fix}^{\text{g}}\ (\lambda f\ s\ t.\ \text{hd}\ s :: (f \circledast t \circledast \texttt{next}\ (\text{tl}\ s))) &&: \text{Str}^{\text{g}}\ \tau \to \blacktriangleright\text{Str}^{\text{g}}\ \tau \to \text{Str}^{\text{g}}\ \tau
\end{aligned}$$

$$\text{hd}\ s \triangleq \texttt{proj}_1\ (\texttt{unfold}\ s)$$

Can we erase `fold`, `unfold`, `next` and $\circledast$ while preserving program behaviour?

Translation & Correctness

Source and Target Language

Goal: Translate $ge \in GExp$ to some $ue \in UExp$

$$
\begin{array}{rcl}
ge & ::= & x \mid \lambda x.ge \mid ge\ ge \\
& \mid & \texttt{lit}\ n \\
& \mid & \langle ge, ge \rangle \mid \langle\rangle \\
& \mid & \texttt{proj}_1\ ge \mid \texttt{proj}_2\ ge \\
& \mid & \texttt{inj}_1\ ge \mid \texttt{inj}_2\ ge \\
& \mid & \texttt{case}\ ge\ \texttt{of}\ x.ge; x.ge \\
& \mid & \texttt{fold}\ ge \mid \texttt{unfold}\ ge \\
& \mid & \texttt{next}\ ge \mid ge \circledast ge
\end{array}
\qquad
\begin{array}{rcl}
ue & ::= & x \mid \lambda x.ue \mid ue\ ue \\
& \mid & \texttt{lit}\ n \\
& \mid & \langle ue, ue \rangle \mid \langle\rangle \\
& \mid & \texttt{proj}_1\ ue \mid \texttt{proj}_2\ ue \\
& \mid & \texttt{inj}_1\ ue \mid \texttt{inj}_2\ ue \\
& \mid & \texttt{case}\ ue\ \texttt{of}\ x.ue; x.ue
\end{array}
$$

Source and Target Language

Goal: Translate $ge \in GExp$ to some $ue \in UExp$

$$
\begin{array}{rcl}
ge & ::= & x \mid \lambda x.ge \mid ge\ ge \\
& \mid & \mathtt{lit}\ n \\
& \mid & \langle ge, ge \rangle \mid \langle\rangle \\
& \mid & \mathtt{proj}_1\ ge \mid \mathtt{proj}_2\ ge \\
& \mid & \mathtt{inj}_1\ ge \mid \mathtt{inj}_2\ ge \\
& \mid & \mathtt{case}\ ge\ \mathtt{of}\ x.ge; x.ge \\
& \mid & \mathtt{fold}\ ge \mid \mathtt{unfold}\ ge \\
& \mid & \mathtt{next}\ ge \mid ge \circledast ge
\end{array}
\qquad
\begin{array}{rcl}
ue & ::= & x \mid \lambda x.ue \mid ue\ ue \\
& \mid & \mathtt{lit}\ n \\
& \mid & \langle ue, ue \rangle \mid \langle\rangle \\
& \mid & \mathtt{proj}_1\ ue \mid \mathtt{proj}_2\ ue \\
& \mid & \mathtt{inj}_1\ ue \mid \mathtt{inj}_2\ ue \\
& \mid & \mathtt{case}\ ue\ \mathtt{of}\ x.ue; x.ue
\end{array}
$$

Translation to Untyped Target

$$(\!|-|\!) \;:\; GExp \to UExp$$

$$
\begin{aligned}
(\!|\lambda x.ge|\!) &\triangleq \lambda x.(\!|ge|\!) \\
(\!|\texttt{lit}\ n|\!) &\triangleq \texttt{lit}\ n \\
(\!|\langle ge_1, ge_2\rangle|\!) &\triangleq \langle (\!|ge_1|\!), (\!|ge_2|\!)\rangle \\
(\!|\texttt{proj}_i\ ge|\!) &\triangleq \texttt{proj}_i\ (\!|ge|\!) \\
&\vdots
\end{aligned}
$$

Translation to Untyped Target

$$(\!|-|\!) \;:\; GExp \to UExp$$

$$
\begin{array}{rcl}
(\!|\lambda x.ge|\!) & \triangleq & \lambda x.(\!|ge|\!) \\
(\!|\texttt{lit}\ n|\!) & \triangleq & \texttt{lit}\ n \\
(\!|\langle ge_1, ge_2 \rangle|\!) & \triangleq & \langle (\!|ge_1|\!), (\!|ge_2|\!) \rangle \\
(\!|\texttt{proj}_i\ ge|\!) & \triangleq & \texttt{proj}_i\ (\!|ge|\!) \\
\vdots & &
\end{array}
$$

$$
\begin{array}{rcl}
(\!|\texttt{fold}\ ge|\!) & \triangleq & (\!|ge|\!) \\
(\!|\texttt{unfold}\ ge|\!) & \triangleq & (\!|ge|\!) \\
(\!|\texttt{next}\ ge|\!) & \triangleq & \mathit{delay}\ (\!|ge|\!) \\
(\!|ge_1 \circledast ge_2|\!) & \triangleq & \mathit{delay}\ (\mathit{force}\ (\!|ge_1|\!))\ (\mathit{force}\ (\!|ge_2|\!)) \\
\mathit{delay}\ ue & \triangleq & \lambda x.ue,\ x \notin FV(ue) \\
\mathit{force}\ ue & \triangleq & ue\ \langle\rangle
\end{array}
$$

Correctness of Translation

Theorem (Correctness of $(\!|-|\!)$)

Let $ge \in GExp$. If $\emptyset \vdash ge : \mathbf{N}$, we have that

$$ge \mapsto^* \texttt{lit}\ n \iff (\!|ge|\!) \mapsto^* \texttt{lit}\ n.$$

Correctness of Translation

Theorem (Correctness of $(\!|-|\!)$)

Let $ge \in GExp$. If $\emptyset \vdash ge : \mathbf{N}$, we have that

$$ge \mapsto^* \texttt{lit}\ n \iff (\!|ge|\!) \mapsto^* \texttt{lit}\ n.$$

Use binary *logical relation* between source and target, indexed by source types

$$\Gamma \vDash ge \approx ue : \tau$$

Correctness of Translation

Theorem (Correctness of $(\!|-|\!)$)

Let $ge \in GExp$. If $\emptyset \vdash ge : \mathbf{N}$, we have that

$$ge \mapsto^* \texttt{lit}\ n \iff (\!|ge|\!) \mapsto^* \texttt{lit}\ n.$$

Use binary *logical relation* between source and target, indexed by source types

$$\Gamma \vDash ge \approx ue : \tau$$

- Interpret types as binary predicates in *step-indexed* higher-order logic *siProp*

Correctness of Translation

Theorem (Correctness of $(\!|-|\!)$)

Let $ge \in GExp$. If $\emptyset \vdash ge : \mathbf{N}$, we have that

$$ge \mapsto^* \texttt{lit}\ n \iff (\!|ge|\!) \mapsto^* \texttt{lit}\ n.$$

Use binary *logical relation* between source and target, indexed by source types

$$\Gamma \vDash ge \approx ue : \tau$$

- Interpret types as binary predicates in *step-indexed* higher-order logic *siProp*

Novel/interesting aspects

- Reasoning about $\blacktriangleright$ with *abstract* step-indexing

Correctness of Translation

Theorem (Correctness of $(\!|-|\!)$)

Let $ge \in GExp$. If $\emptyset \vdash ge : \mathbf{N}$, we have that

$$ge \mapsto^* \texttt{lit}\ n \iff (\!|ge|\!) \mapsto^* \texttt{lit}\ n.$$

Use binary *logical relation* between source and target, indexed by source types

$$\Gamma \vDash ge \approx ue : \tau$$

- Interpret types as binary predicates in *step-indexed* higher-order logic *siProp*

Novel/interesting aspects

- Reasoning about $\blacktriangleright$ with *abstract* step-indexing
- Relating call-by-name to call-by-value

Outlook & Ongoing Work

- Limitation: Source language cannot express *non-causal* functions, e.g.

$$\text{every2nd}\ (x :: x' :: xs) \quad \triangleq \quad x :: \text{every2nd}\ xs$$

Outlook & Ongoing Work

- Limitation: Source language cannot express *non-causal* functions, e.g.

$$\text{every2nd}\ (x :: x' :: xs) \quad \triangleq \quad x :: \text{every2nd}\ xs$$

Ongoing work:

- Lift causality restriction using *constant* type modality ■

Outlook & Ongoing Work

- Limitation: Source language cannot express *non-causal* functions, e.g.

$$\text{every2nd}\ (x :: x' :: xs) \quad \triangleq \quad x :: \text{every2nd}\ xs$$

Ongoing work:

- Lift causality restriction using *constant* type modality ■
  - Allows controlled elimination of ▶

Outlook & Ongoing Work

- Limitation: Source language cannot express *non-causal* functions, e.g.

$$\text{every2nd}\ (x :: x' :: xs) \quad \triangleq \quad x :: \text{every2nd}\ xs$$

Ongoing work:

- Lift causality restriction using *constant* type modality ■
  - Allows controlled elimination of ▶

Future work:

- Extend source with polymorphism and existential types
- Support inductive types alongside guarded recursive types

Bibliography


Clouston, Ranald et al. (2015). ‘Programming and Reasoning with Guarded Recursion for Coinductive Types’. In: *FoSSaCS*. Vol. 9034. LNCS, pp. 407–421. DOI: 10.1007/978-3-662-46678-0_26.


Nakano, Hiroshi (2000). ‘A Modality for Recursion’. In: *15th Annual IEEE Symposium on Logic in Computer Science*, pp. 255–266. DOI: 10.1109/LICS.2000.855774.