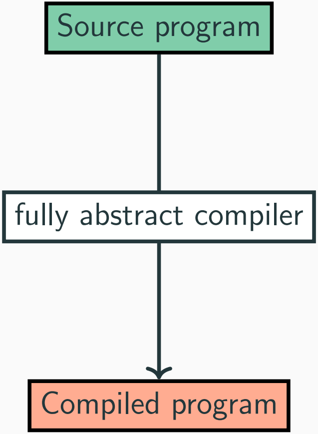
Linking Types: Secure compilation of multi-language programs

Daniel Patterson and Amal Ahmed

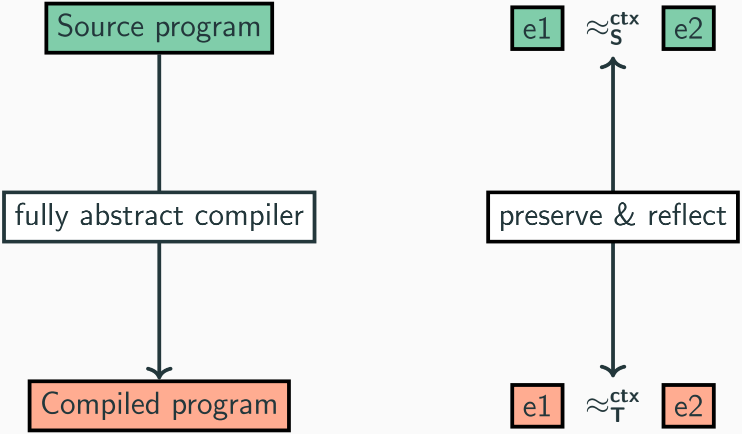
January 15, 2017

Northeastern University

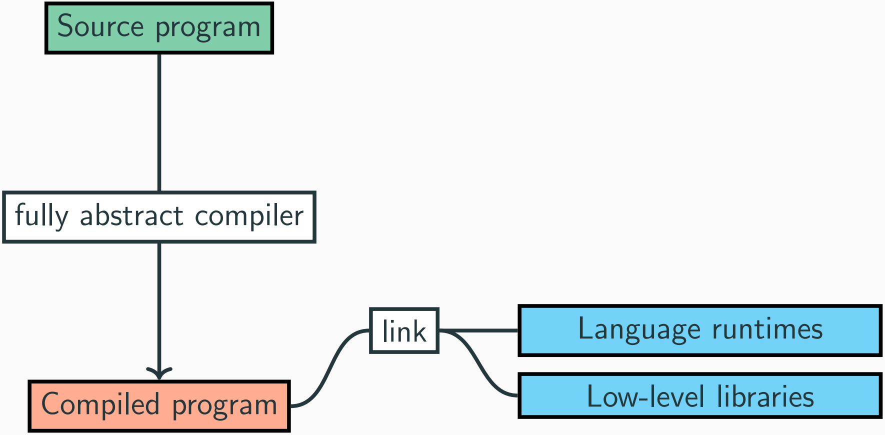
Fully abstract compilers



Fully abstract compilers

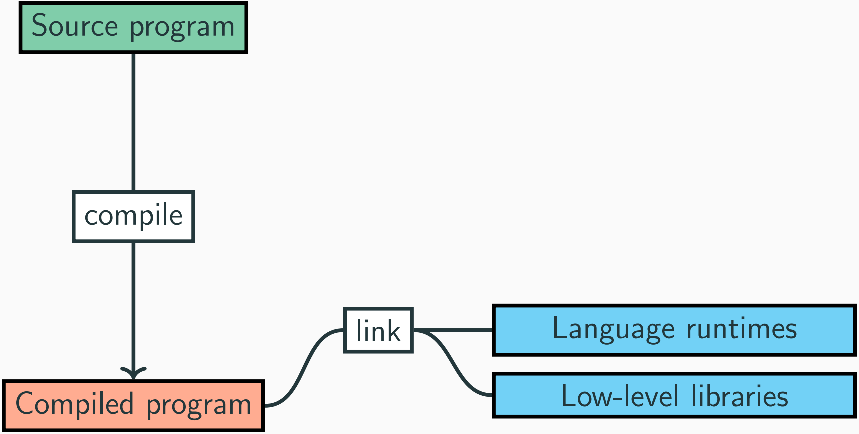


Fully abstract compilers in the real world

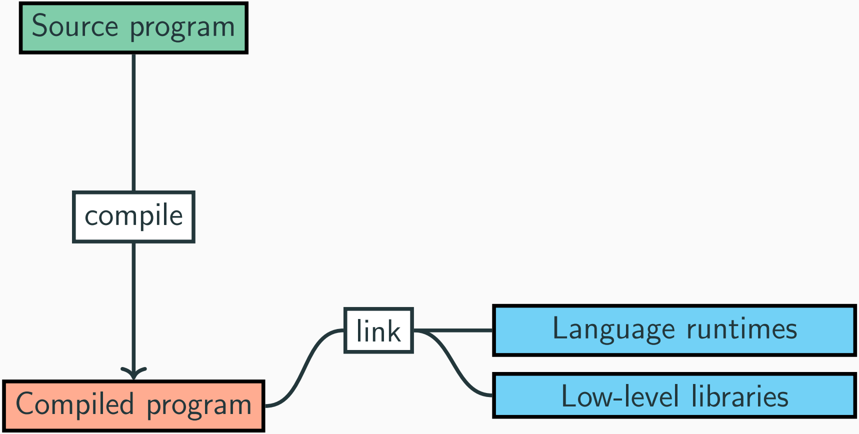


How do we implement fully abstract compilers for multi-language programs?

Fully abstract compilers in the real world

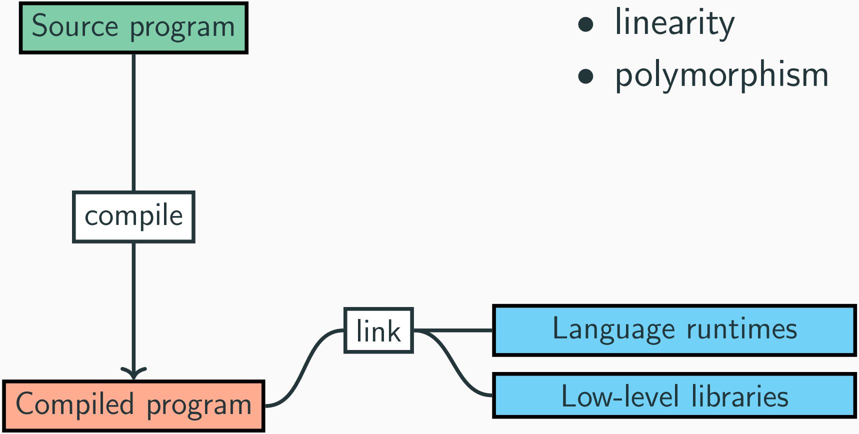


Fully abstract compilers in the real world



- references
- first-class control

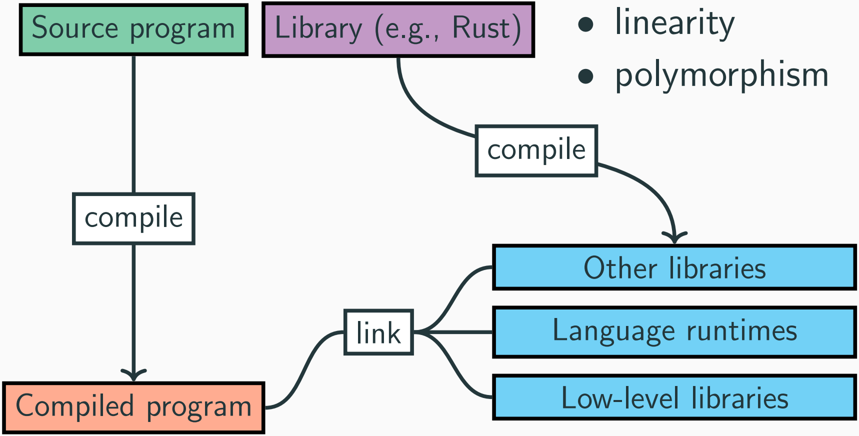
Fully abstract compilers in the real world



- linearity
- polymorphism

- references
- first-class control

Fully abstract compilers in the real world



- linearity
- polymorphism

- references
- first-class control

Without a full accounting for linking, full abstraction—and secure compilation—is doomed.

Full abstraction: by example

e.g. a pure language λ linking with a counter library

$$\lambda \mathsf{c}.\, \mathsf{c}()$$

Full abstraction: by example

e.g. a pure language λ linking with a counter library

$$\lambda \mathtt{c}.\, \mathtt{c}() \approx^{ctx}_{\lambda} \lambda \mathtt{c}.\, \mathtt{c}(); \mathtt{c}() : (\mathtt{unit} \to \mathtt{int}) \to \mathtt{int}$$

Full abstraction: by example

e.g. a pure language λ linking with a counter library

$$\lambda \mathsf{c}.\, \mathsf{c}() \approx^{ctx}_{\lambda} \lambda \mathsf{c}.\, \mathsf{c}(); \mathsf{c}() : (\mathtt{unit} \to \mathtt{int}) \to \mathtt{int}$$

$$\begin{aligned} \mathtt{C}^{\mathtt{counter}} = \ & \mathtt{let}\, \mathtt{v} = \mathtt{ref}\, 0\, \mathtt{in} \\ & \mathtt{let}\, \mathsf{c}'\, () = \mathtt{v} := !\mathtt{v} + 1;\ !\mathtt{v}\ \mathtt{in} \\ & [\cdot]\, \mathsf{c}' \end{aligned}$$

Full abstraction: by example

e.g. a pure language λ linking with a counter library

$$\lambda \mathtt{c}.\, \mathtt{c}() \approx^{ctx}_{\lambda} \lambda \mathtt{c}.\, \mathtt{c}(); \mathtt{c}() : (\mathtt{unit} \to \mathtt{int}) \to \mathtt{int}$$

$$\begin{aligned} \mathtt{C}^{\mathtt{counter}} = \;& \mathtt{let}\, \mathtt{v} = \mathtt{ref}\, 0\, \mathtt{in} \\ & \mathtt{let}\, \mathtt{c}'\, () = \mathtt{v} := !\mathtt{v} + 1;\ !\mathtt{v}\, \mathtt{in} \\ & [\cdot]\, \mathtt{c}' \end{aligned}$$

$$\mathtt{C}^{\mathtt{counter}}[\lambda \mathtt{c}.\, \mathtt{c}()] \Downarrow 1 \qquad \mathtt{C}^{\mathtt{counter}}[\lambda \mathtt{c}.\, \mathtt{c}(); \mathtt{c}()] \Downarrow 2$$

Full abstraction: by example

e.g. a pure language λ linking with a counter library

$$\lambda \mathsf{c}.\, \mathsf{c}() \approx_{\lambda}^{ctx} \lambda \mathsf{c}.\, \mathsf{c}(); \mathsf{c}() : (\mathtt{unit} \to \mathtt{int}) \to \mathtt{int}$$

$$\begin{aligned} \mathtt{C}^{\mathtt{counter}} = \;& \mathtt{let}\, \mathtt{v} = \mathtt{ref}\, 0\, \mathtt{in} \\ & \mathtt{let}\, \mathsf{c}'\, () = \mathtt{v} := !\mathtt{v} + 1;\; !\mathtt{v}\, \mathtt{in} \\ & [\cdot]\, \mathsf{c}' \end{aligned}$$

$$\mathtt{C}^{\mathtt{counter}}[\lambda \mathsf{c}.\, \mathsf{c}()] \Downarrow 1 \qquad \mathtt{C}^{\mathtt{counter}}[\lambda \mathsf{c}.\, \mathsf{c}(); \mathsf{c}()] \Downarrow 2$$

$$\mathsf{c}' : \mathtt{unit} \to \mathtt{int} \quad \neq \quad \mathsf{c} : \mathtt{unit} \to \mathtt{int}$$

Full abstraction: by example

e.g. a pure language λ linking with a counter library

$$\lambda c.\, c() \not\approx^{ctx}_{\lambda^?} \lambda c.\, c(); c() : (\mathtt{unit} \rightarrow \mathtt{int}) \rightarrow \mathtt{int}$$

$$\begin{aligned} C^{\mathtt{counter}} = \; & \mathtt{let}\, v = \mathtt{ref}\, 0\, \mathtt{in} \\ & \mathtt{let}\, c'\, () = v := !v + 1;\ !v\ \mathtt{in} \\ & [\cdot]\, c' \end{aligned}$$

$$C^{\mathtt{counter}}[\lambda c.\, c()] \Downarrow 1 \qquad C^{\mathtt{counter}}[\lambda c.\, c(); c()] \Downarrow 2$$

$$c' : \mathtt{unit} \rightarrow \mathtt{int} \quad \neq \quad c : \mathtt{unit} \rightarrow \mathtt{int}$$

Writing a type which corresponds to behavior **inexpressible** in your language is the essence of **linking types**.

Linking types: in more detail

Two source languages: λ and $\lambda^{\texttt{ref}}$.

$$
\begin{array}{lcl}
\tau & ::= & \texttt{unit} \mid \texttt{int} \mid \tau \rightarrow \tau \\
\texttt{e} & ::= & () \mid \texttt{n} \mid \texttt{x} \mid \lambda \texttt{x} : \tau . \texttt{e} \\
 & & \mid \texttt{e}\,\texttt{e} \mid \texttt{e} + \texttt{e} \mid \texttt{e} * \texttt{e} \\
\texttt{v} & ::= & () \mid \texttt{n} \mid \lambda \texttt{x} : \tau . \texttt{e}
\end{array}
\qquad
\begin{array}{lcl}
\tau & ::= & \ldots \mid \texttt{ref}\,\tau \\
\texttt{e} & ::= & \ldots \mid \texttt{ref}\,\texttt{e} \mid \texttt{e} := \texttt{e} \mid !\texttt{e} \\
\texttt{v} & ::= & \ldots \mid \ell
\end{array}
$$

Compiled to a shared, typed, target (not shown).

Linking types: extended language λ^κ

Extend λ to λ^κ:

$$
\begin{aligned}
\tau &::= \texttt{unit} \mid \texttt{int} \mid \texttt{ref}\,\tau \mid \tau \to \mathrm{R}^{\epsilon}\,\tau \\
\texttt{e} &::= () \mid \texttt{n} \mid \texttt{x} \mid \lambda\texttt{x} : \tau.\,\texttt{e} \mid \texttt{e}\,\texttt{e} \mid \texttt{e} + \texttt{e} \\
&\quad\;\mid \texttt{e} * \texttt{e} \\
\texttt{v} &::= () \mid \texttt{n} \mid \lambda\texttt{x} : \tau.\,\texttt{e} \\
\epsilon &::= \bullet \mid \circ
\end{aligned}
$$

Linking types: extended language λ^κ

Extend λ **types** to λ^κ:

$$\begin{aligned}
\tau &::= \texttt{unit} \mid \texttt{int} \mid \textbf{ref}\,\tau \mid \tau \to \mathbf{R}^{\epsilon}\,\tau \\
\texttt{e} &::= () \mid \texttt{n} \mid \texttt{x} \mid \lambda\texttt{x}:\tau.\,\texttt{e} \mid \texttt{e}\;\texttt{e} \mid \texttt{e} + \texttt{e} \\
&\quad\mid \texttt{e} * \texttt{e} \\
\texttt{v} &::= () \mid \texttt{n} \mid \lambda\texttt{x}:\tau.\,\texttt{e} \\
\epsilon &::= \bullet \mid \circ
\end{aligned}$$

- $\mathtt{R}^{\circ}\,\tau$ is a pure computation.
- $\mathtt{R}^{\bullet}\,\tau$ may allocate, read, or write references.

Linking types: extended language λ^κ

Extend λ **terms** to λ^κ:

$$
\begin{aligned}
\tau &::= \texttt{unit} \mid \texttt{int} \mid \texttt{ref}\,\tau \mid \tau \to \texttt{R}^{\epsilon}\,\tau \\
\texttt{e} &::= () \mid \texttt{n} \mid \texttt{x} \mid \lambda \texttt{x} : \tau.\,\texttt{e} \mid \texttt{e}\,\texttt{e} \mid \texttt{e} + \texttt{e} \\
&\quad \mid \texttt{e} * \texttt{e} \mid \textbf{ref e} \mid \textbf{e} := \textbf{e} \mid !\textbf{e} \\
\texttt{v} &::= () \mid \texttt{n} \mid \lambda \texttt{x} : \tau.\,\texttt{e} \mid \ell \\
\epsilon &::= \bullet \mid \circ
\end{aligned}
$$

- Add *representative* terms for new behavior.
- Programmers use these to *reason* about inhabitants of type $\texttt{R}^{\bullet}\,\tau$.

Linking types: using them

Consider the following λ programs

$$\texttt{program 1} \quad \lambda \texttt{f} : \texttt{int} \rightarrow \texttt{int}.\, \texttt{f}\ 0$$
$$\texttt{program 2} \quad \lambda \texttt{f} : \texttt{int} \rightarrow \texttt{int}.\, \texttt{f}\ 0;\ \texttt{f}\ 0$$

In λ^{κ}, by "default" $\texttt{f}$ has type: $\texttt{int} \rightarrow \texttt{R}^{\circ}\, \texttt{int}$

Linking types: using them

Consider the following λ programs

$$\begin{array}{ll} \texttt{program 1} & \lambda \texttt{f} : \texttt{int} \rightarrow \texttt{int}.\, \texttt{f}\, 0 \\ \texttt{program 2} & \lambda \texttt{f} : \texttt{int} \rightarrow \texttt{int}.\, \texttt{f}\, 0;\ \texttt{f}\, 0 \end{array}$$

In λ^{κ}, by "default" $\texttt{f}$ has type: $\texttt{int} \rightarrow \texttt{R}^{\circ}\, \texttt{int}$

But a programmer can annotate:

$$\begin{array}{ll} \texttt{program 1} & \lambda \texttt{f} : \texttt{int} \rightarrow \texttt{R}^{\bullet}\, \texttt{int}.\, \texttt{f}\, 0 \\ \texttt{program 2} & \lambda \texttt{f} : \texttt{int} \rightarrow \texttt{R}^{\bullet}\, \texttt{int}.\, \texttt{f}\, 0;\ \texttt{f}\, 0 \end{array}$$

Linking types: default embedding κ^+

Property 1: κ^+ is Equivalence Preserving & Reflecting

Linking types: default embedding κ^+

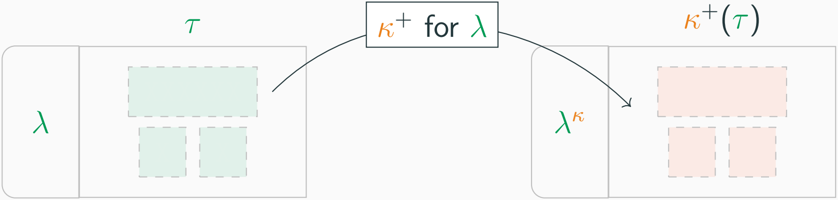
Property 1: κ^+ is Equivalence Preserving & Reflecting

$$\forall \mathtt{e}_1, \mathtt{e}_2.\ \mathtt{e}_1 \approx^{ctx}_{\lambda} \mathtt{e}_2 : \tau \implies \mathtt{e}_1 \approx^{ctx}_{\lambda^\kappa} \mathtt{e}_2 : \kappa^+(\tau).$$

Linking types: default embedding κ^+

Property 1: κ^+ is Equivalence Preserving & Reflecting

$$\forall \mathtt{e}_1, \mathtt{e}_2.\ \mathtt{e}_1 \approx^{ctx}_{\lambda} \mathtt{e}_2 : \tau \implies \mathtt{e}_1 \approx^{ctx}_{\lambda^\kappa} \mathtt{e}_2 : \kappa^+(\tau).$$

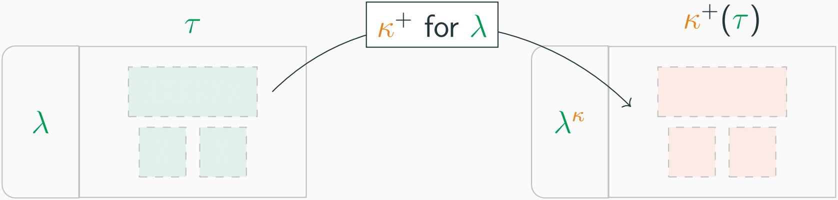


Linking types: default embedding κ^+

Property 1: κ^+ is Equivalence Preserving & Reflecting

$$\forall \mathtt{e}_1, \mathtt{e}_2.\ \mathtt{e}_1 \approx^{ctx}_{\lambda} \mathtt{e}_2 : \tau \implies \mathtt{e}_1 \approx^{ctx}_{\lambda^\kappa} \mathtt{e}_2 : \kappa^+(\tau).$$

$$\begin{aligned} \kappa^+(\mathtt{unit}) &= \mathtt{unit} \\ \kappa^+(\mathtt{int}) &= \mathtt{int} \\ \kappa^+(\tau_1 \to \tau_2) &= \kappa^+(\tau_1) \to \mathtt{R}^\circ\, \kappa^+(\tau_2) \end{aligned}$$



Linking types: project κ^-

Property 2: No New Programs.

Linking types: project κ^-

Property 2: No New Programs.

For e made of λ terms: $\forall \tau .\, \mathsf{e} : \tau \implies \mathsf{e} : \kappa^-(\tau)$

Linking types: project κ^-

Property 2: No New Programs.

For $\mathtt{e}$ made of λ terms: $\forall \tau.\, \mathtt{e} : \tau \implies \mathtt{e} : \kappa^-(\tau)$

$$
\begin{aligned}
\kappa^-(\mathtt{unit}) &= \mathtt{unit} \\
\kappa^-(\mathtt{int}) &= \mathtt{int} \\
\kappa^-(\mathtt{ref}\ \tau) &= \kappa^-(\tau) \\
\kappa^-(\tau_1 \to \mathtt{R}^{\epsilon}\, \tau_2) &= \kappa^-(\tau_1) \to \kappa^-(\tau_2)
\end{aligned}
$$

Linking types: project κ^-

Property 2: No New Programs.

For $\mathtt{e}$ made of λ terms: $\forall \tau.\, \mathtt{e} : \tau \implies \mathtt{e} : \kappa^-(\tau)$

$$
\begin{aligned}
\kappa^-(\mathtt{unit}) &= \mathtt{unit} \\
\kappa^-(\mathtt{int}) &= \mathtt{int} \\
\kappa^-(\mathtt{ref}\, \tau) &= \kappa^-(\tau) \\
\kappa^-(\tau_1 \to \mathtt{R}^{\epsilon}\, \tau_2) &= \kappa^-(\tau_1) \to \kappa^-(\tau_2)
\end{aligned}
$$

e.g.,

$$\lambda(\mathtt{r} : \mathtt{ref}\, \mathtt{int})(\mathtt{f} : \mathtt{ref}\, \mathtt{int} \to \mathtt{R}^{\bullet}\, \mathtt{int}).\, \mathtt{f}\, \mathtt{r}$$

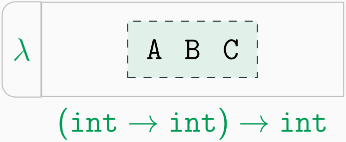
$$\stackrel{\kappa^-}{\Longrightarrow} \lambda(\mathtt{r} : \mathtt{int})(\mathtt{f} : \mathtt{int} \to \mathtt{int}).\, \mathtt{f}\, \mathtt{r}$$

More about equivalences

$\texttt{program}\,\texttt{A} - \lambda\texttt{f} : \texttt{int} \to \texttt{int}.\, \texttt{1}$

$\texttt{program}\,\texttt{B} - \lambda\texttt{f} : \texttt{int} \to \texttt{int}.\, \texttt{f}\,\texttt{0};\ \texttt{1}$

$\texttt{program}\,\texttt{C} - \lambda\texttt{f} : \texttt{int} \to \texttt{int}.\, \texttt{f}\,\texttt{0};\ \texttt{f}\,\texttt{0};\ \texttt{1}$

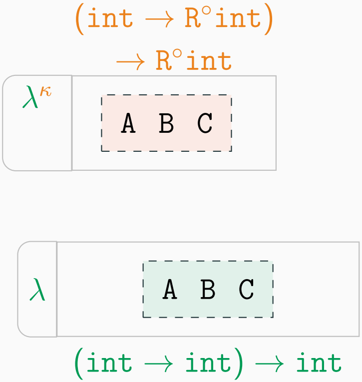


More about equivalences

$\texttt{program}\,\texttt{A} - \lambda\texttt{f} : \texttt{int} \to \texttt{int}.\,\texttt{1}$

$\texttt{program}\,\texttt{B} - \lambda\texttt{f} : \texttt{int} \to \texttt{int}.\,\texttt{f}\;\texttt{0};\;\texttt{1}$

$\texttt{program}\,\texttt{C} - \lambda\texttt{f} : \texttt{int} \to \texttt{int}.\,\texttt{f}\;\texttt{0};\;\texttt{f}\;\texttt{0};\;\texttt{1}$

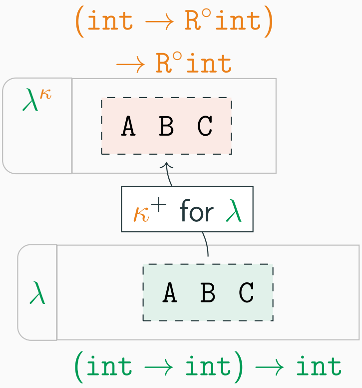


More about equivalences

$\texttt{program}\,\mathtt{A} - \lambda \mathtt{f} : \mathtt{int} \to \mathtt{int}.\, 1$

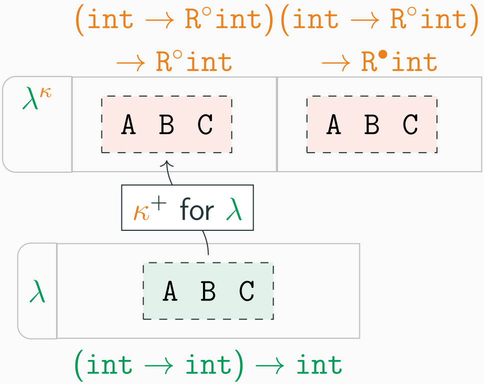
$\texttt{program}\,\mathtt{B} - \lambda \mathtt{f} : \mathtt{int} \to \mathtt{int}.\, \mathtt{f}\,0;\ 1$

$\texttt{program}\,\mathtt{C} - \lambda \mathtt{f} : \mathtt{int} \to \mathtt{int}.\, \mathtt{f}\,0;\ \mathtt{f}\,0;\ 1$



More about equivalences

$$\texttt{program}\,\texttt{A} - \lambda \texttt{f} : \texttt{int} \to \texttt{int}.\, 1$$
$$\texttt{program}\,\texttt{B} - \lambda \texttt{f} : \texttt{int} \to \texttt{int}.\, \texttt{f}\, 0;\ 1$$
$$\texttt{program}\,\texttt{C} - \lambda \texttt{f} : \texttt{int} \to \texttt{int}.\, \texttt{f}\, 0;\ \texttt{f}\, 0;\ 1$$

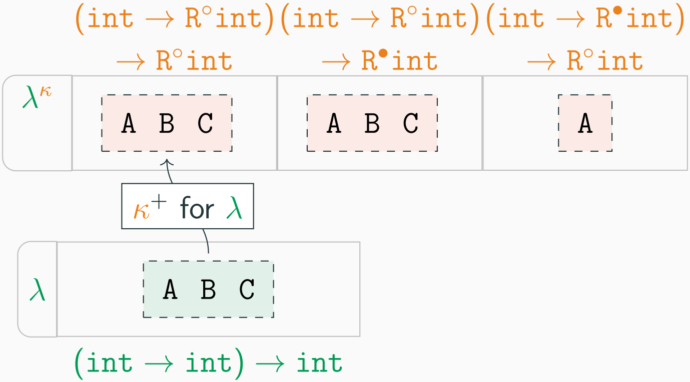


More about equivalences

$\texttt{program A} - \lambda \texttt{f} : \texttt{int} \to \texttt{int}.\ \texttt{1}$

$\texttt{program B} - \lambda \texttt{f} : \texttt{int} \to \texttt{int}.\ \texttt{f 0; 1}$

$\texttt{program C} - \lambda \texttt{f} : \texttt{int} \to \texttt{int}.\ \texttt{f 0; f 0; 1}$

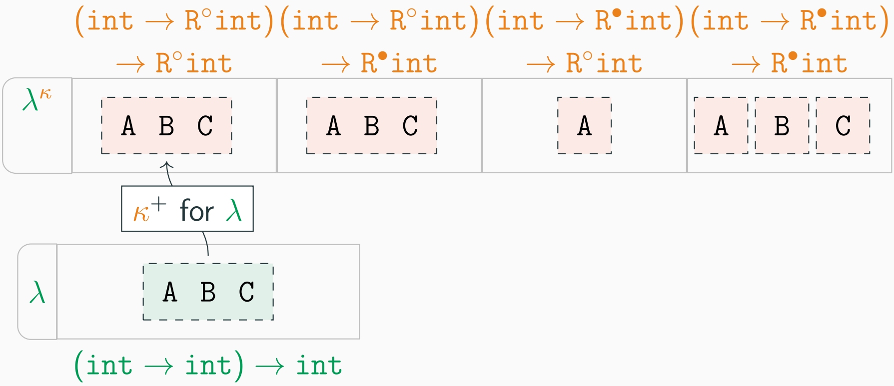


More about equivalences

program A — $\lambda f : \mathtt{int} \to \mathtt{int}.\ 1$

program B — $\lambda f : \mathtt{int} \to \mathtt{int}.\ f\ 0;\ 1$

program C — $\lambda f : \mathtt{int} \to \mathtt{int}.\ f\ 0;\ f\ 0;\ 1$

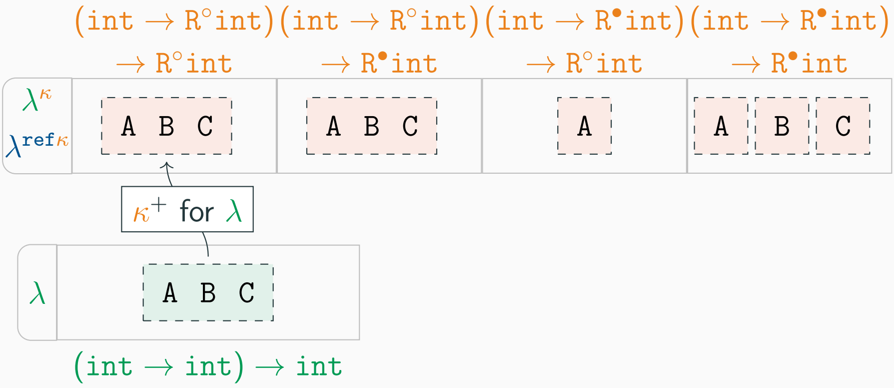


More about equivalences

$$\texttt{program}\,A - \lambda f : \texttt{int} \to \texttt{int}.\, 1$$

$$\texttt{program}\,B - \lambda f : \texttt{int} \to \texttt{int}.\, f\,0;\, 1$$

$$\texttt{program}\,C - \lambda f : \texttt{int} \to \texttt{int}.\, f\,0;\, f\,0;\, 1$$

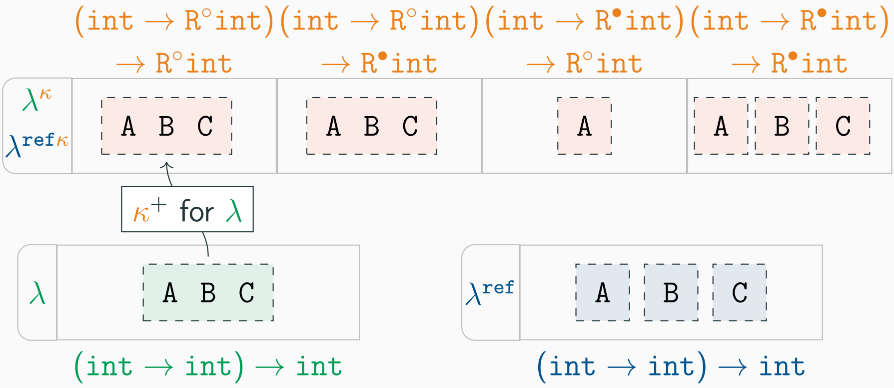


More about equivalences

$\mathtt{program}\,\mathsf{A} - \lambda \mathtt{f} : \mathtt{int} \rightarrow \mathtt{int}.\, \mathsf{1}$

$\mathtt{program}\,\mathsf{B} - \lambda \mathtt{f} : \mathtt{int} \rightarrow \mathtt{int}.\, \mathtt{f}\,\mathsf{0};\ \mathsf{1}$

$\mathtt{program}\,\mathsf{C} - \lambda \mathtt{f} : \mathtt{int} \rightarrow \mathtt{int}.\, \mathtt{f}\,\mathsf{0};\ \mathtt{f}\,\mathsf{0};\ \mathsf{1}$

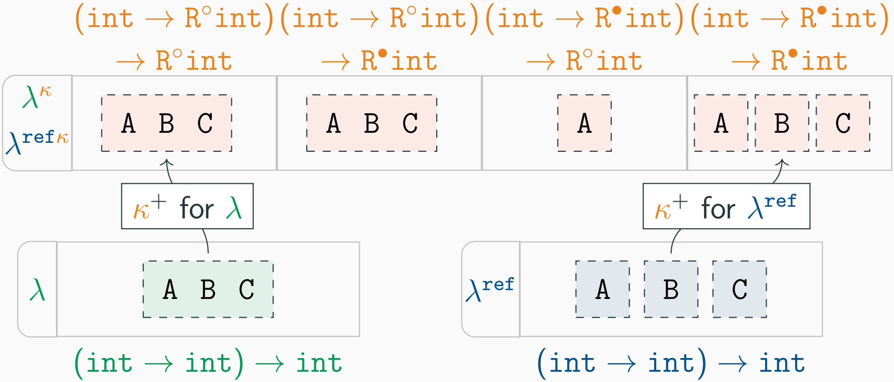


More about equivalences

$\texttt{program}\,\mathsf{A} - \lambda \mathtt{f} : \mathtt{int} \to \mathtt{int}.\,1$

$\texttt{program}\,\mathsf{B} - \lambda \mathtt{f} : \mathtt{int} \to \mathtt{int}.\,\mathtt{f}\,0;\ 1$

$\texttt{program}\,\mathsf{C} - \lambda \mathtt{f} : \mathtt{int} \to \mathtt{int}.\,\mathtt{f}\,0;\ \mathtt{f}\,0;\ 1$



Linking types: back to example

$$\lambda \mathtt{c}\colon \mathtt{unit} \rightarrow \mathtt{int}.\, \mathtt{c}() \approx^{ctx}_{\lambda^{\kappa}} \lambda \mathtt{c}\colon \mathtt{unit} \rightarrow \mathtt{int}.\, \mathtt{c}(); \mathtt{c}()$$

Linking types: back to example

$$\lambda \mathtt{c}\colon \mathtt{unit} \rightarrow \mathtt{R}^{\circ}\,\mathtt{int}.\, \mathtt{c}() \approx^{ctx}_{\lambda^{\kappa}} \lambda \mathtt{c}\colon \mathtt{unit} \rightarrow \mathtt{R}^{\circ}\,\mathtt{int}.\, \mathtt{c}();\mathtt{c}()$$

Linking types: back to example

$$\lambda \mathtt{c}\colon \mathtt{unit} \to \mathtt{R}^{\circ}\,\mathtt{int}.\,\mathtt{c}() \approx^{ctx}_{\lambda^{\kappa}} \lambda \mathtt{c}\colon \mathtt{unit} \to \mathtt{R}^{\circ}\,\mathtt{int}.\,\mathtt{c}();\mathtt{c}()$$

$$\begin{aligned} \mathtt{C}^{\mathtt{counter}} = \quad & \mathtt{let}\,\mathtt{v} = \mathtt{ref}\,0\,\mathtt{in} \\ & \mathtt{let}\,\mathtt{c}'\,() = \mathtt{v} := !\mathtt{v} + 1;\ !\mathtt{v}\,\mathtt{in} \\ & [\cdot]\,\mathtt{c}' \end{aligned}$$

Linking types: back to example

$$\lambda \mathtt{c} : \mathtt{unit} \to \mathtt{R}^{\circ}\, \mathtt{int}.\, \mathtt{c}() \approx^{ctx}_{\lambda^{\kappa}} \lambda \mathtt{c} : \mathtt{unit} \to \mathtt{R}^{\circ}\, \mathtt{int}.\, \mathtt{c}(); \mathtt{c}()$$

$$\begin{aligned} \mathtt{C}^{\mathtt{counter}} = \quad & \mathtt{let}\ \mathtt{v} = \mathtt{ref}\ 0\ \mathtt{in} \\ & \mathtt{let}\ \mathtt{c}'\,() = \mathtt{v} := !\mathtt{v} + 1;\ !\mathtt{v}\ \mathtt{in} \\ & [\cdot]\ \mathtt{c}' \end{aligned}$$

$$\mathtt{c}' : \mathtt{unit} \to \mathtt{R}^{\bullet}\, \mathtt{int}$$

Linking types: back to example

$$\lambda \mathtt{c} : \mathtt{unit} \to \mathtt{R}^{\bullet}\,\mathtt{int}.\ \mathtt{c}() \not\approx^{ctx}_{\lambda^{\kappa}} \lambda \mathtt{c} : \mathtt{unit} \to \mathtt{R}^{\bullet}\,\mathtt{int}.\ \mathtt{c}(); \mathtt{c}()$$

$$\begin{aligned} \mathtt{C}^{\mathtt{counter}} = \quad & \mathtt{let}\ \mathtt{v} = \mathtt{ref}\ 0\ \mathtt{in} \\ & \mathtt{let}\ \mathtt{c}'\,() = \mathtt{v} := !\mathtt{v} + 1;\ !\mathtt{v}\ \mathtt{in} \\ & [\cdot]\ \mathtt{c}' \end{aligned}$$

$$\mathtt{c}' : \mathtt{unit} \to \mathtt{R}^{\bullet}\,\mathtt{int}$$

Linking types: back to example

$$\lambda \mathtt{c} : \mathtt{unit} \to \mathtt{R}^{\bullet}\, \mathtt{int}.\, \mathtt{c}() \not\approx^{ctx}_{\lambda^{\kappa}} \lambda \mathtt{c} : \mathtt{unit} \to \mathtt{R}^{\bullet}\, \mathtt{int}.\, \mathtt{c}(); \mathtt{c}()$$

$$\begin{aligned} \mathtt{C}^{\mathtt{counter}} = \quad & \mathtt{let}\, \mathtt{v} = \mathtt{ref}\, 0\, \mathtt{in} \\ & \mathtt{let}\, \mathtt{c}'\, () = \mathtt{v} := !\mathtt{v} + 1;\ !\mathtt{v}\ \mathtt{in} \\ & [\cdot]\, \mathtt{c}' \end{aligned}$$

$$\mathtt{c}' : \mathtt{unit} \to \mathtt{R}^{\bullet}\, \mathtt{int}$$

$$\mathtt{C}^{\mathtt{counter}}[\lambda \mathtt{c}.\, \mathtt{c}()] \Downarrow 1 \qquad \mathtt{C}^{\mathtt{counter}}[\lambda \mathtt{c}.\, \mathtt{c}(); \mathtt{c}()] \Downarrow 2$$

With **linking types**, the programmer specifies the equivalences she wants, in a language she can understand.

Linking types: next steps

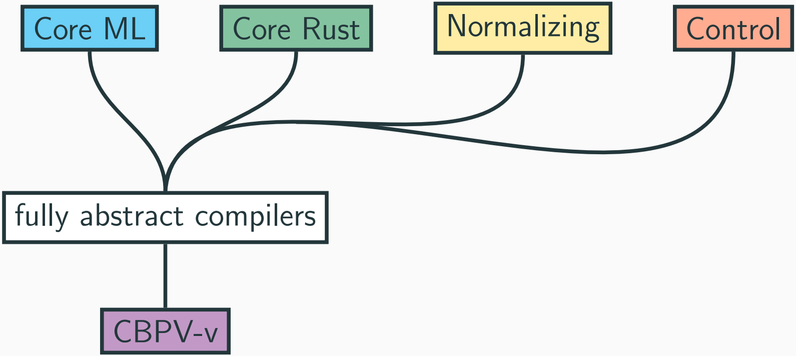
Core ML

Core Rust

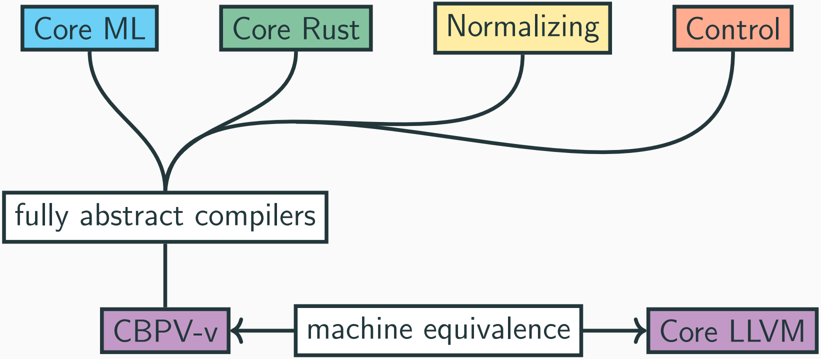
Normalizing

Control

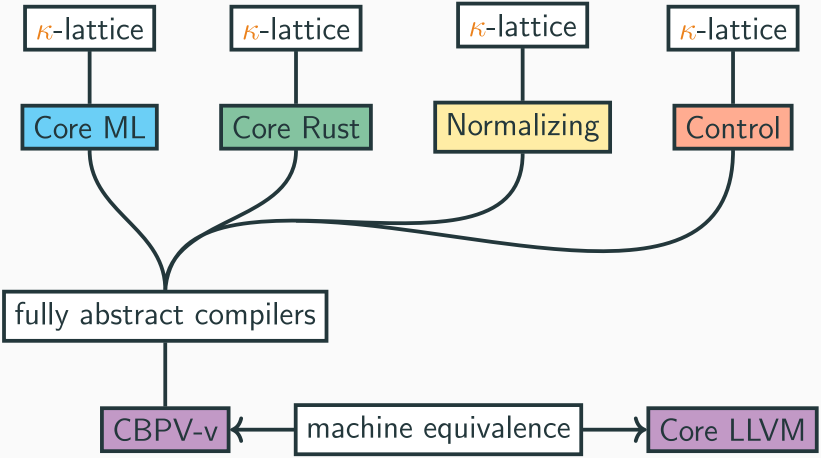
Linking types: next steps



Linking types: next steps



Linking types: next steps



Linking types make realizing fully abstract compilation possible in multi-language systems

Learn more:
https://dbp.io/pubs/2017/linking-types-snapl-submission.pdf

Linking types make realizing fully abstract compilation possible in multi-language systems (which are the only systems that really exist).

Learn more:
https://dbp.io/pubs/2017/linking-types-snapl-submission.pdf

Linking types: extra content

Backwards Compatibility

- Annotations are optional for programmers.
- To change target, only need to fully abstractly compile old target to the new.

Linking types: extra content

Backwards Compatibility

- Annotations are optional for programmers.
- To change target, only need to fully abstractly compile old target to the new.

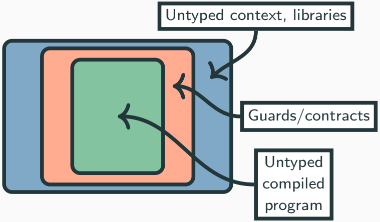
Developer Tooling

- With typed target, type translations can guide if two components can be linked together.
- Type translate to target, attempt reverse type translation to other language. Enables *cross-language type errors*.

Full abstraction: different approaches

Dynamic enforcement

- Devriese *et al.*, POPL16
- Patrignani *et al.*, TOPLAS15
- Patrignani, 2015 Dissertation
- *etc.*



Typed target languages

- New *et al.*, ICFP16
- Ahmed, Blume, ICFP11
- *etc.*