LINKING TYPES

SPECIFYING SAFE INTEROPERABILITY AND EQUIVALENCES

Daniel Patterson
dbp@ccs.neu.edu

We should not force programmers to use ad-hoc FFIs (1) or write entire programs in a single "general-purpose" language (2).

(1) Consider languages $\mathcal{S}$ and $\mathcal{R}$ that compile to $\mathcal{T}$. Linking $\mathcal{S}$ and $\mathcal{R}$ components requires the programmer understand:

- the $\mathcal{S}$-to-$\mathcal{T}$ and $\mathcal{R}$-to-$\mathcal{T}$ compilers.
- how resulting $\mathcal{T}$ components interact.

We believe understanding $\mathcal{S}$ and $\mathcal{R}$ should be enough.

(2) General purpose languages are usually either:

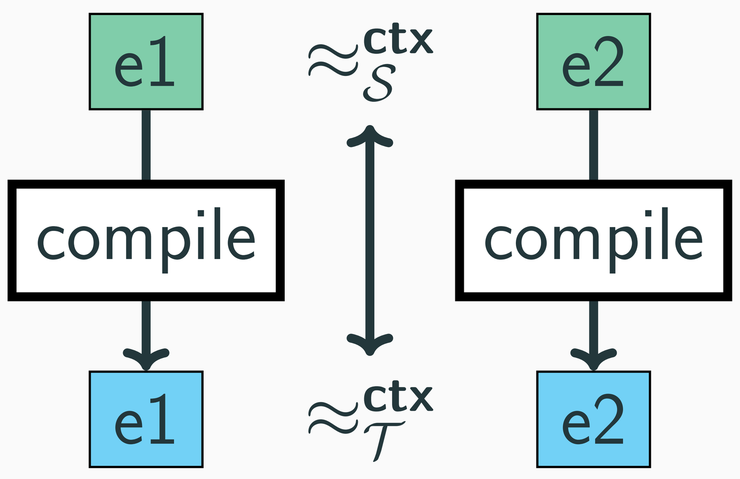
- **too low level** (increasing incidental complexity).
- **too feature rich** (harder to understand).

Embedded DSLs aren't enough, since they require understanding (usually complex) host languages.

To preserve source-level equational reasoning, we should not merely have correct (3) compilers, but fully abstract (4) compilers.

(3) A correct compiler, like CompCert or CakeML, guarantees that **compilation preserves source semantics**. But when building multi-language software, **we have no (multi-language) source semantics** to preserve.

(4) Fully abstract compilers preserve & reflect equivalences.



For e_1, e_2 in a language $\mathcal{L}$, $e_1 \approx^{ctx}_{\mathcal{L}} e_2$ iff for all contexts C in $\mathcal{L}$, $C[e_1]$ and $C[e_2]$ coterminate.

But fully abstract compilers prevent linking with more expressive languages, because such expressivity can violate equivalences (5).

(5) In a pure language λ (a simply-typed lambda calculus): $\lambda c.\, c() \approx^{ctx}_{\lambda} \lambda c.\, c(); c() : (\mathtt{unit} \to \mathtt{int}) \to \mathtt{int}$

But a *more expressive* impure language $\lambda^{\mathtt{ref}}$ can distinguish these two programs with a context that uses a counter.

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

problem ⟶ $\boxed{\mathtt{c} : \underbrace{\mathtt{unit} \to \mathtt{int}}_{\text{pure}\uparrow} \neq \mathtt{c}' : \underbrace{\mathtt{unit} \to \mathtt{int}}_{\text{impure}\uparrow}}$

linking prevented by compiler

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

Linking types allow writing types (6) for behavior *inexpressible* in a language, enabling fully abstract compilers that allow linking (7).

(6) A linking types extension for a language λ is three parts:

- an **extended language** λ^{κ}, which has types and *representative* terms that reflect new behavior.0

  $\tau ::= \mathtt{unit} \mid \mathtt{int} \mid \mathtt{ref}\ \tau \mid \boxed{\tau \to \mathrm{R}^{\circ}\, \tau \mid \tau \to \mathrm{R}^{\bullet}\, \tau}$

  computation type ⟶ (pure↑ impure↑)

- a λ-to-λ^{κ} type function κ^{+}, the default embedding, that **preserves equivalences from** λ. i.e.,

  $\forall e_1, e_2.\ e_1 \approx^{ctx}_{\lambda} e_2 : \tau \implies e_1 \approx^{ctx}_{\lambda^{\kappa}} e_2 : \kappa^{+}(\tau)$

$$\kappa^{+}(\mathtt{unit}) = \mathtt{unit}$$
$$\kappa^{+}(\mathtt{int}) = \mathtt{int}$$
$$\kappa^{+}(\tau_1 \to \tau_2) = \kappa^{+}(\tau_1) \to \mathrm{R}^{\circ}\, \kappa^{+}(\tau_2)$$

- a λ^{κ}-to-λ type function κ^{-} that we use to require **all** λ^{κ} **programs are** λ **programs**. i.e.,

  $\forall \tau.\ e : \tau \implies e : \kappa^{-}(\tau)$

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

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

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

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

linking allowed by compiler

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

(*) Linking types are all 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 — $\lambda f : \mathtt{int} \to \mathtt{int}.\ f\ 0;\ f\ 0;\ 1$