Lambda-calcul et Catégories

RustBelt

Neven Villani

2021-11-24

Introduction

Efficient languages need side effects and memory accesses, which pure λ-calculus lacks

Unwilling to compromise safety and high-level constructs

Goals of λ_{Rust}

Guarantee (at the type level) the absence of

- data races
- use-after-free
- dangling or null pointers

Tools

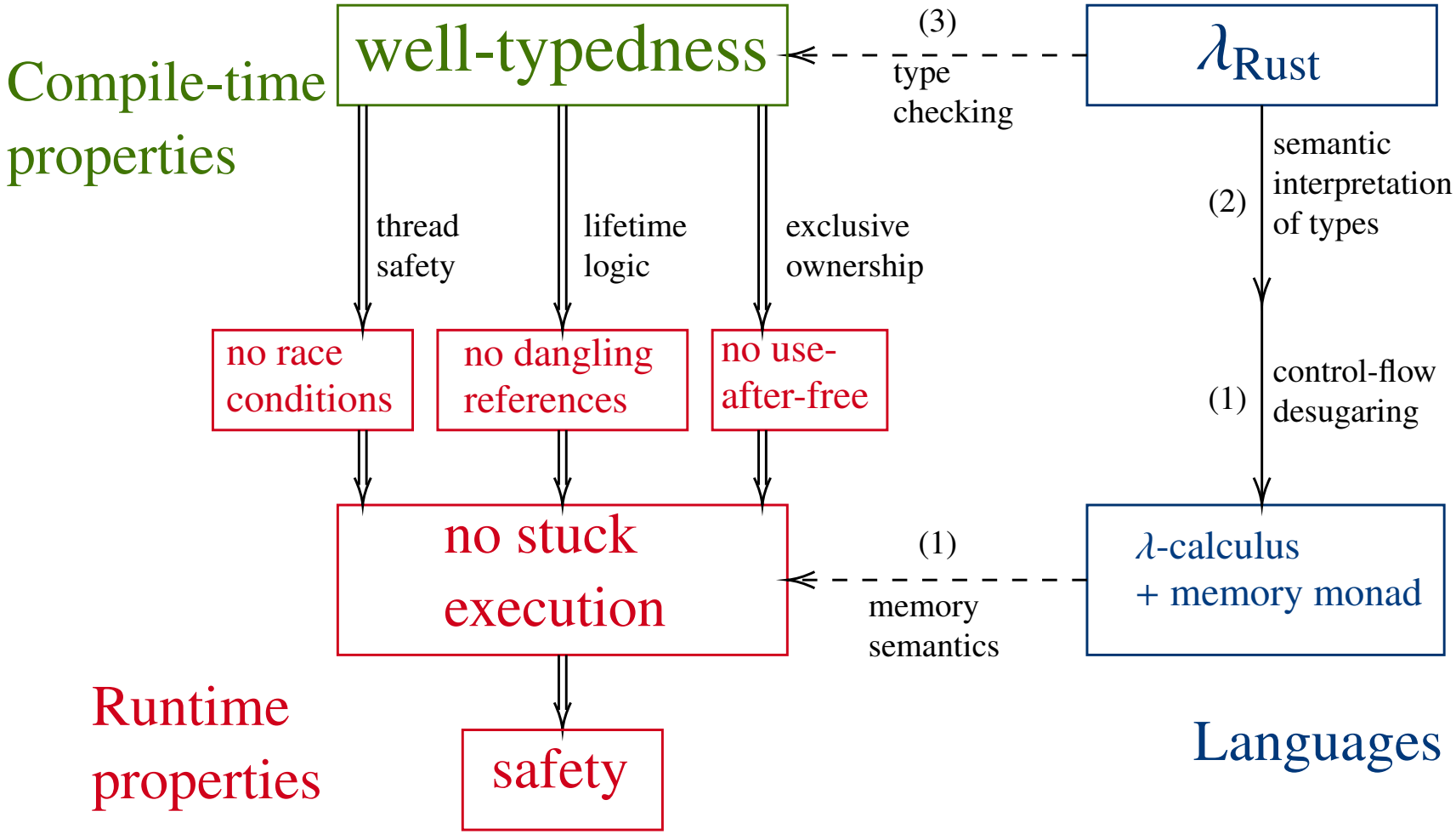
Within the language

- exception monad
- product and sum types
- continuation-style

External

- memory monad
- integers

Structure


Compile-time properties
well-typedness
(3) type checking
λ_{Rust}
thread safety
lifetime logic
exclusive ownership
(2) semantic interpretation of types
(1) control-flow desugaring
no race conditions
no dangling references
no use-after-free
no stuck execution
(1) memory semantics
λ-calculus + memory monad
Runtime properties
safety
Languages

Translation to pure λ-calculus (continuation-style)

$\texttt{funrec}\ f(\overline{x})\ \texttt{ret}\ k\ \texttt{:=}\ F$	$Y\ (\lambda f.\ \lambda \overline{x}.\ \lambda k.\ F)$
$\texttt{if}\ p\ \texttt{then}\ F_1\ \texttt{else}\ F_2$	$p\ F_1\ F_2$
$\texttt{jump}\ k(\overline{x})$	$k\ \overline{x}$
$\texttt{call}\ f(\overline{x})\ \texttt{ret}\ k$	$f\ \overline{x}\ k$

Memory monad

$$\text{Memory} = \mathbb{N} \times \mathbb{N} \to_{\text{fin}} \text{LockSt} \times \text{Val}$$

$$\text{LockSt} = \{\texttt{reading}\, n \mid n \in \mathbb{N}\} \cup \{\texttt{writing}\}$$

$$M(\text{Instr}) = \text{Memory} \times \text{Instr}$$

Memory monad

$$\eta_{\text{Instr}} : \text{Instr} \to M(\text{Instr})$$
$$i \mapsto ([], i)$$
$$\mu_{\text{Instr}} : M(M(\text{Instr})) \to M(\text{Instr})$$
$$(m', (m, i)) \mapsto (m \| m', i)$$

where

$$\text{dom}(m \| m') = \text{dom}(m') \cup \text{dom}(m)$$

$$(m' \| m)(x) = \begin{cases} m(x) & \text{if } x \in \text{dom}(m) \\ m'(x) & \text{otherwise} \end{cases}$$

Explicit rules

$$\frac{\text{preconditions}}{(\text{memory}|\text{instruction}) \to (\text{memory}'|\text{result})}$$

Execution error if stuck

Otherwise reduces to the empty continuation $\lambda x.\ x$

Locks: reading

$$\frac{h(l) = (\texttt{reading}\, n, v)}{(h \mid {}^{*} l) \to (h[l \leftarrow \texttt{reading}\, n + 1, v] \mid {}^{*\prime} l)}$$

$$\frac{h(l) = (\texttt{reading}\, n + 1, v)}{(h \mid {}^{*\prime} l) \to (h[l \leftarrow \texttt{reading}\, n, v] \mid v)}$$

Locks: writing

$$\frac{h(l) = (\texttt{reading}\,0, v')}{(h \mid l :=\! v) \to (h[l \leftarrow \texttt{writing}, v') \mid l :=\! v)}$$

$$\frac{h(l) = (\texttt{writing}, v')}{(h \mid l :=\! v) \to (h[l \leftarrow \texttt{reading}\,0, v) \mid \text{☣})}$$

λ_{Rust}'s type system

Features

- sum and product types
- recursive types
- lifetimes

Memory management is implicit and determined by lifetimes

Any program that is correctly typed cannot be stuck

Type constructs

$\texttt{own}_n\,\tau$	owned pointer
$\&^{\alpha}_{\mu}\tau$	μ-reference to τ with lifetime α
	$\mu \in \{\texttt{mut}, \texttt{shr}\}$
$\mu T.\,\tau$	recursive type
$\Pi\overline{\tau}, \Sigma\overline{\tau}$	products and sums

Standard examples

- `Option<`τ`>`

$$() + \texttt{own}\,\tau$$

- `Cow<`α`,`τ`>`

$$\texttt{own}\,\tau + \&^{\alpha}_{\texttt{shr}}\tau$$

What is in a type

- a finite size
  - $[\![int]\!].\texttt{size} = [\![\texttt{own}_n\, t]\!].\texttt{size} = \left[\!\left[\&^{\alpha}_{\mu}\tau\right]\!\right].\texttt{size} = 1$
  - $[\![\Pi\overline{\tau}]\!].\texttt{size} = \sum_i [\![\overline{\tau}_i]\!].\texttt{size}$
  - $[\![\Sigma\overline{\tau}]\!].\texttt{size} = 1 + \max_i [\![\overline{\tau}_i]\!].\texttt{size}$
- an ownership predicate
  - $[\![int]\!].\texttt{own}(t, \overline{v}) = \exists z.\, \overline{v} = [z]$
  - $[\![\&^{\kappa}_{\texttt{mut}}\tau]\!].\texttt{own}(t, \overline{v}) = \exists l.\, \overline{v} = [l] * \&^{\kappa}_{\texttt{full}} \exists \overline{u}.\, l \mapsto \overline{u} * [\![\tau]\!].\texttt{own}(t, \overline{u})$
- a sharing predicate
  - $[\![\Pi\overline{\tau}]\!].\texttt{shr}(\kappa, t, l) = \ast_i [\![\overline{\tau}_i]\!].\texttt{shr}(k, t, l + \sum_{j<i} \left[\!\left[\overline{\tau}_j\right]\!\right].\texttt{size})$
  - $[\![int]\!].\texttt{shr}(\kappa, t, l) = \exists \overline{v}.\, \&^{\kappa}_{\texttt{frac}}(\lambda q.\, l \mapsto^{q} \overline{v}) * [\![int]\!].\texttt{own}(t, \overline{v})$

Safety

Thread safety for a type is a property of its ownership and sharing predicates

- τ is `Send` if $[\![\tau]\!]\,.\texttt{own}(t, \overline{v})$ is independent of t
- τ is `Sync` if $[\![\tau]\!]\,.\texttt{shr}(t, \overline{v}, l)$ is independent of t

Type judgements: structure

$$\Gamma \mid E; L \mid K; T \vdash F$$

$$\Gamma \mid E; L \mid K; T \vdash I \dashv x.\, T'$$

Where

Γ	variable bindings $x : \texttt{val}$ or $\alpha : \texttt{lft}$ or $T : \texttt{Type}$
E	external lifetime inclusions $\kappa \sqsubseteq_e \alpha$
L	local lifetime inclusions $\kappa \sqsubseteq_l \overline{\kappa'}$
K	external type constraints $x \lhd \tau$ (incl. continuations)
T	local type constraints

Lifetimes

$$\frac{\Gamma \mid E; L \vdash \kappa \sqsubseteq \kappa' \qquad \Gamma \mid E; L \vdash \kappa' \sqsubseteq \kappa''}{\Gamma \mid E; L \vdash \kappa \sqsubseteq \kappa''}$$

$$\frac{\Gamma \mid E; L \vdash \kappa \sqsubseteq \kappa' \qquad \Gamma \mid E; L \vdash \kappa \text{ alive}}{\Gamma \mid E; L \vdash \kappa' \text{ alive}}$$

Borrowing

$$\Gamma \mid E; L \vdash p \lhd \mathtt{own}\,\tau \Rightarrow p \lhd \&^{\kappa}_{\mathtt{mut}}\,\tau, p \lhd^{\dagger\kappa} \mathtt{own}\,\tau$$

Memory accesses

Reads can move data, writes can erase it

$$\text{old} \circ\!\!-^{\text{read}} \text{new} \qquad \text{old} -\!\!\circ^{\text{write}} \text{new}$$

Summary

- semantics defined by a translation to λ-terms
- all safety expressed in the type system
- theorems prove a correspondance between the two:
  any execution of a translation of a well-typed function body cannot get stuck
- not an equivalence: some safe programs are rejected
  `unsafe` escape hatch: looser typing but the interface must be proven safe