# A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries

Appendix

July 26, 2024

## Contents

# 1 Results

Table 1: Download counts for each crate where a bug was found, sorted by “All-Time”. The mean download count per day is aggregated across the 6 months between March 20th and September 20th, 2023.

| Crate | Version | Mean ⬇ / Day | ⬇ All-Time | Last Updated | Bug IDs |
|---|:-:|--:|--:|:-:|:-:|
| flate2 | 1.0.27 | 136,292 | 82,637,795 | 2023-08-15 | 38 |
| foreign-types | 0.5.0 | 88,416 | 64,223,888 | 2019-10-13 | 33 |
| bzip2 | 0.4.4 | 23,862 | 13,447,830 | 2023-01-05 | 34 |
| zmq | 0.10.0 | 1,594 | 1,824,174 | 2022-11-04 | 19 |
| lcms2 | 6.0.0 | 363 | 158,000 | 2023-09-02 | 42, 43, 10 |
| dec | 0.4.8 | 237 | 118,377 | 2022-02-05 | 41, 22, 46, 47 |
| tectonic_engine_bibtex | 0.2.1 | 32 | 15,680 | 2023-06-15 | 27 |
| special-fun | 0.2.0 | 28 | 15,176 | 2019-03-15 | 9 |
| littlefs2 | 0.4.0 | 40 | 14,288 | 2023-02-07 | 35, 36 |
| librsync | 0.2.3 | 3 | 12,350 | 2023-03-10 | 39 |
| libhydrogen | 0.4.1 | 23 | 12,264 | 2021-05-16 | 20 |
| bad64 | 0.6.0 | 10 | 6,071 | 2021-12-22 | 26 |
| fluidlite | 0.2.1 | 6 | 5,768 | 2021-08-21 | 17 |
| sgp4-rs | 0.4.0 | 10 | 5,680 | 2023-07-19 | 28 |
| minimap2-sys | 0.1.16[1] | 15 | 3,782 | 2023-09-12 | 45 |
| minimp3_ex-sys | 0.1.1 | 14 | 3,192 | 2020-12-19 | 16 |
| libcmark-sys | 0.1.0 | 1 | 2,396 | 2017-11-30 | 12 |
| dec-number-sys | 0.0.25 | 8 | 1,400 | 2022-11-28 | 23, 32, 7 |
| xxhrs | 2.0.0 | 3 | 1,390 | 2020-09-15 | 48 |
| tetsy-secp256k1 | 0.7.0 | 3 | 1,202 | 2021-02-19 | 29 |
| bchlib | 0.2.1 | 1 | 739 | 2019-05-27 | 11 |
| tree-sitter-svelte | 0.10.2 | 2 | 718 | 2022-04-15 | 14 |
| everrs | 0.2.1 | 2 | 650 | 2020-04-12 | 4 |
| x42ltc-sys | 0.0.5 | 1 | 648 | 2020-09-05 | 15 |
| quickjs_regex | 0.2.3 | 3 | 637 | 2021-11-30 | 5, 24, 24 |
| crypto_pimitives | 0.1.1 | 2 | 535 | 2019-11-17 | 21 |
| klu-rs | 0.4.0 | 2 | 530 | 2022-09-15 | 44 |
| ytnef | 0.2.0 | 6 | 521 | 2021-11-14 | 18 |
| tinyspline-sys | 0.2.0 | 1 | 441 | 2020-06-09 | 13 |
| ms5837 | 0.2.1 | 2 | 377 | 2022-07-30 | 31 |
| spritz_cipher | 0.1.0 | 1 | 324 | 2019-10-16 | 8 |
| mseed | 0.5.0 | 4 | 318 | 2023-08-24 | 25 |
| jh-rs | 0.1.0 | 1 | 206 | 2021-01-31 | 1 |
| lsmlite-rs | 0.1.0 | 2 | 52 | 2023-07-17 | 30 |

[1] 0.1.16+minimap2.2.26

Table 2: Unique bugs detected by our tool, sorted Miri's error label ("Error Category") and our additional classification ("Error Type"). A "-" indicates that our classification is the same as Miri's label. When a commit is listed in the last column, it indicates that the bug has been fixed. If multiple commits were used to fix a bug, we provide the last commit in the series.

| ID | Crate | Version | Error Category | Error Type | Fix | Error | Issue(s) | Pull(s) | Commit(s) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | jh-rs | 0.1.0 | Alignment | Invalid Transmutation | Rust | Rust | #1 | | |
| 2 | quickjs_regex | 0.2.3 | Cross-Language Free | - | LLVM | Rust | #2 | | |
| 3 | tree-sitter | 0.20.3 | Dangling Int Pointer | Null Pointer Dereference | LLVM | LLVM | #8 | | 4676cd4 |
| 4 | everrs | 0.2.1 | Incorrect Binding | Incorrect Integer Width | Binding | Binding | #1 | | |
| 5 | quickjs_regex | 0.2.3 | Incorrect Binding | Incorrect Integer Width | Binding | Binding | | #1 | |
| 6 | secp256k1 | 0.28.0 | Incorrect Binding | Incorrect Integer Width | Binding | Binding | #669 | #670 | 60a5e36 |
| 7 | dec-number-sys | 0.0.25 | Incorrect Binding | Missing Return Type | Binding | Binding | | #2 | |
| 8 | spritz_cipher | 0.1.0 | Incorrect Binding | Missing Return Type | Binding | Binding | | #1 | |
| 9 | special-fun | 0.2.0 | Incorrect Binding | Missing Return Type | Binding | Binding | #14 | #13 | ded37f8 |
| 10 | lcms2 | 6.0.0 | Invalid Enum Tag | Logical Error | Rust | Rust | | | 85218b6 |
| 11 | bchlib | 0.2.1 | Memory Leaked | Missing C Destructor | Rust | Rust | #1 | | |
| 12 | libcmark-sys | 0.1.0 | Memory Leaked | Missing C Destructor | Rust | Rust | #3 | | |
| 13 | tinyspline-sys | 0.2.0 | Memory Leaked | Missing C Destructor | Rust | Rust | #1 | | |
| 14 | tree-sitter-svelte | 0.10.2 | Memory Leaked | Missing C Destructor | Rust | Rust | #46 | | |
| 15 | x42ltc-sys | 0.0.5 | Memory Leaked | Missing C Destructor | Rust | Rust | #1 | #2 | 1c594f2 |
| 16 | minimp3_ex-sys | 0.1.1 | Memory Leaked | Missing C Destructor | Rust | Rust | | #5 | 33bea0d |
| 17 | fluidlite | 0.2.1 | Memory Leaked | Missing from Raw | Rust | Rust | #15 | | |
| 18 | ytnef | 0.2.0 | Memory Leaked | Missing from Raw | Rust | Rust | #1 | | |
| 19 | zmq | 0.10.0 | Memory Leaked | Missing from Raw | Rust | Rust | #387 | #388 | |
| 20 | libhydrogen | 0.4.1 | Memory Leaked | Missing from Raw | Rust | Rust | #11 | | bddff45 |
| 21 | crypto_pimitives | 0.1.1 | Out of Bounds Access | - | LLVM | Rust | #1 | | |
| 22 | dec | 0.4.8 | Out of Bounds Access | - | LLVM | LLVM | #76 | | |
| 23 | dec-number-sys | 0.0.25 | Out of Bounds Access | - | LLVM | LLVM | #76 | | |
| 24 | quickjs_regex | 0.2.3 | Out of Bounds Access | - | LLVM | Rust | | #1 | |
| 25 | mseed | 0.5.0 | Out of Bounds Access | - | LLVM | Rust | | | 0cfede1 |
| 26 | bad64 | 0.6.0 | Out of Bounds Access | - | LLVM | LLVM | | | 6dbd961 |
| 27 | tectonic_engine_bibtex | 0.2.1 | Tree Borrows | Freeing Through `&mut T` | LLVM | Rust | | #1129 | c64e524 |
| 28 | sgp4-rs | 0.4.0 | Tree Borrows | Incorrect Integer Width | LLVM | Binding | #29 | | |
| 29 | tetsy-secp256k1 | 0.7.0 | Tree Borrows | Incorrect `const` | LLVM | Rust | #3 | | |
| 30 | lsmlite-rs | 0.1.0 | Tree Borrows | Incorrect `const` | LLVM | Binding | | #5 | 2e0cf90 |
| 31 | ms5837 | 0.2.1 | Tree Borrows | Incorrect `const` | LLVM | Rust | | #26 | 7be05c1 |
| 32 | dec-number-sys | 0.0.25 | Tree Borrows | Incorrect `const` | LLVM | Binding | #1 | #2 | 4a12cce |
| 33 | foreign-types | 0.5.0 | Tree Borrows | Phantom `UnsafeCell<T>` | LLVM | Rust | #24 | | |
| 34 | bzip2 | 0.4.4 | Tree Borrows | Sharing `&mut T` | LLVM | Rust | #94 | | |
| 35 | littlefs2 | 0.4.0 | Tree Borrows | Sharing `&mut T` | LLVM | Rust | | #54 | |

Continued on next page

**Table 2 – continued from previous page**

| ID | Crate | Version | Error Category | Error Type | Fix | Error | Issue(s) | Pull(s) | Commit(s) |
|---|---|---|---|---|---|---|---|---|---|
| 36 | littlefs2 | 0.4.0 | Tree Borrows | Sharing `&mut T` | LLVM | Rust | | #54 | |
| 37 | spng | 0.2.0-alpha.2 | Tree Borrows | Sharing `&mut T` | Rust | Rust | #11 | #12 | |
| 38 | flate2 | 1.0.27 | Tree Borrows | Sharing `&mut T` | LLVM | Rust | #392 | #394 | 0a584f4 |
| 39 | librsync | 0.2.3 | Tree Borrows | `&T as *mut T` | Rust | Rust | #23 | | |
| 40 | blitsort-sys | 0.1.0 | Tree Borrows | `&T as *mut T` | LLVM | Rust | #1 | #2 | |
| 41 | dec | 0.4.8 | Tree Borrows | `&T as *mut T` | LLVM | Rust | #74 | #2 | ece7d84 |
| 42 | lcms2 | 6.0.0 | Tree Borrows | `&T as *mut T` | LLVM | Rust | | #18 | 28626ed |
| 43 | lcms2 | 6.0.0 | Tree Borrows | `&T as *mut T` | LLVM | Rust | | | 5d3b648 |
| 44 | klu-rs | 0.4.0 | Tree Borrows | `&T as *mut T` | LLVM | Rust | | #1 | c5e89d1 |
| 45 | minimap2-sys | 0.1.16[2] | Uninitialized Memory | Erroneous Failure | Rust | Rust | | | 2ac2a6d |
| 46 | dec | 0.4.8 | Uninitialized Memory | Incomplete Initialization | Rust | Rust | #76 | #77 | 3545623 |
| 47 | dec | 0.4.8 | Uninitialized Memory | Incomplete Initialization | LLVM | Rust | #76 | #77 | 3545623 |
| 48 | xxhrs | 2.0.0 | Uninitialized Memory | Uninitialized Padding | Rust | Rust | | #10 | def77e5 |

Table 3: Test results across each of the three evaluation modes. In the "Zeroed" mode, all stack and heap memory from LLVM is zero-initialized. In the "Uninitialized" mode, LLVM is allowed to read uninitialized bytes without throwing an error.

| | **Zeroed** | | **Uninitialized** | |
|---|---|---|---|---|
| **Error Type** | *Stacked Borrows* | *Tree Borrows* | *Stacked Borrows* | *Tree Borrows* |
| *Borrowing Violation* | 2.7% (245) | 2% (184) | 2.7% (245) | 2% (183) |
| *Using Uninitialized Memory* | 2.2% (197) | 2.2% (202) | 2.2% (200) | 2.2% (205) |
| *Other Error* | 5.4% (495) | 5.5% (501) | 5.2% (479) | 5.4% (490) |
| *Passed* | 18.7% (1706) | 18.9% (1724) | 18.6% (1695) | 18.7% (1710) |
| *Timeout* | 9.7% (890) | 10.6% (968) | 9.6% (873) | 10.4% (953) |
| *Unsupported Operation* | 61.3% (5597) | 60.8% (5551) | 61.8% (5638) | 61.2% (5589) |

[2] +minimap2.2.26

# 2 Semantics

🟥 Rust 🟦 LLVM

## 2.1 Domains

$b \in \textsc{Bytes}$ *(bytes)*

$m, n \in \mathbb{N} \cup \{0\}$ *(sizes)*

$\ell \in \textsc{Locations} : \mathcal{P}(\textsc{Bytes})$ *(heap locations)*

$t \in \textsc{Tags}$ *(access tags)*

## 2.2 Type Syntax

$\tau ::= \mathsf{int}(n) \mid \mathsf{ptr} \mid \overline{\tau}$ *(LLVM types)*

$\tau ::= \mathsf{int}(n) \mid *\tau \mid \tau^p$ *(Rust types)*

$\tau^p ::= \overline{\langle \tau, n \rangle}^m$ *(Rust products)*

$\tau ::= \tau \mid \tau$ *(Types)*

## 2.3 Value Syntax

$v_b ::= \overline{b} \mid \langle \ell, \varrho \rangle$ *(Base Values)*

$v ::= v_b \mid \langle \overline{v} \rangle$ *(LLVM Values)*

$v ::= v_b$ *(Rust Values)*

$\varrho ::= t \mid * \mid \cdot$ *(Provenance)*

## 2.4 Environments

$\mu \in \textsc{Mem} : \textsc{Loc} \to (\textsc{Bytes} \times \textsc{Tag})$ *(memory)*

$\sigma \in \textsc{TagSet} : \mathcal{P}(\textsc{Tag})$ *(exposed tags)*

## 2.5 Conversion

$$\boxed{\mu; \sigma \vdash v : \tau \leftrightsquigarrow v : \tau \dashv \mu'; \sigma'}$$

"*Under the store $\mu$ and tag set $\sigma$, Rust values $v$ of type $\tau$ and LLVM values $v$ of type $\tau$ are interconvertible, producing the updated store $\mu'$ and tag set $\sigma'$.*"

$$\textsc{C-Pointer}\quad \frac{}{\mu; \sigma \vdash \langle \ell, \varrho \rangle : \tau \leftrightsquigarrow \langle \ell, \varrho \rangle : \mathsf{ptr} \dashv \mu; \sigma}$$

$$\textsc{C-Int}\quad \mu; \sigma \vdash \overline{b}^n : \mathsf{int}(n) \leftrightsquigarrow \overline{b}^n : \mathsf{int}(n) \dashv \mu; \sigma$$

$$\textsc{C-Product}\quad \frac{\mathsf{fields}(\langle \ell, \varrho \rangle : \overline{\tau^p}^n) = \overline{v : \tau^p}^n \qquad \forall i \in [1, n].\ \mu_{i-1}; \sigma_{i-1} \vdash v : \tau^p \leftrightsquigarrow v : \tau_i \dashv \mu_i; \sigma_i}{\mu_0, \sigma_0 \vdash \langle \ell, \varrho \rangle : \overline{\tau^p}^n \leftrightsquigarrow \langle \overline{v} \rangle : \overline{\tau}^n \dashv \mu_n; \sigma_n}$$

$$\boxed{\mu;\sigma \vdash \textcolor{red}{v} : \textcolor{red}{\tau} \rightsquigarrow \textcolor{blue}{v} : \textcolor{blue}{\tau} \dashv \mu;\sigma'}$$

“*Under the store $\mu$ and tag set $\sigma$, Rust values $\textcolor{red}{v}$ of type $\textcolor{red}{\tau}$ can be converted to LLVM values $\textcolor{blue}{v}$ of type $\textcolor{blue}{\tau}$, producing the updated tag set $\sigma'$*”

C-PointerToInt

$$\frac{\ell \triangleq \overline{b} \qquad \mathsf{expose}(\sigma, \varrho) = \sigma'}{\mu;\sigma \vdash \langle \ell, \varrho \rangle : *\textcolor{red}{\tau} \rightsquigarrow \overline{b} : \mathsf{int}(n_{ptr}) \dashv \mu;\sigma'}$$

C-FieldToScalar

$$\frac{\begin{array}{c}\mu;\sigma \vdash \mathsf{read}(\ell, \textcolor{red}{\tau}) = v_b \dashv \sigma'' \\ \mu;\sigma'' \vdash v_b : \textcolor{red}{\tau} \rightsquigarrow v_b : \textcolor{blue}{\tau} \dashv \mu;\sigma'\end{array}}{\mu;\sigma \vdash \langle \ell, \varrho \rangle : \langle \textcolor{red}{\tau}, 0 \rangle \rightsquigarrow v_b : \textcolor{blue}{\tau} \dashv \mu;\sigma'}$$

C-ProductToInt

$$\frac{\begin{array}{c}\mathsf{sizeof}(\textcolor{red}{\tau^p}) = q \\ \mu;\sigma \vdash \mathsf{read}(\ell, \mathsf{int}(q)) = \overline{b} \dashv \sigma'\end{array}}{\mu;\sigma \vdash \langle \ell, \varrho \rangle : \textcolor{red}{\tau^p} \rightsquigarrow \overline{b} : \mathsf{int}(q) \dashv \mu;\sigma'}$$

$$\boxed{\mu;\sigma \vdash \textcolor{red}{v} : \textcolor{red}{\tau} \leftsquigarrow \textcolor{blue}{v} : \textcolor{blue}{\tau} \dashv \mu;\sigma'}$$

“*Under the store $\mu$ and tag set $\sigma$, LLVM values $\textcolor{red}{v}$ of type $\textcolor{blue}{\tau}$ can be converted to Rust values $\textcolor{blue}{v}$ of type $\textcolor{red}{\tau}$, producing the updated store $\mu'$*”

C-PointerFromInt

$$\frac{\ell \triangleq \overline{b}}{\mu;\sigma \vdash \langle \ell, * \rangle : *\textcolor{red}{\tau} \leftsquigarrow \overline{b} : \mathsf{int}(n_{ptr}) \dashv \mu;\sigma}$$

C-FieldFromScalar

$$\frac{\begin{array}{c}\mu'' \vdash \mathsf{write}(\ell, v_b) \dashv \mu' \\ \mu;\sigma \vdash v_b : \textcolor{red}{\tau} \leftsquigarrow v_b : \textcolor{blue}{\tau} \dashv \mu'';\sigma\end{array}}{\mu;\sigma \vdash \langle \ell, \varrho \rangle : \langle \textcolor{red}{\tau}, 0 \rangle \leftsquigarrow v_b : \textcolor{blue}{\tau} \dashv \mu';\sigma}$$

C-ProductFromInt

$$\frac{\mathsf{sizeof}(\textcolor{red}{\tau^p}) = q \qquad \mu \vdash \mathsf{write}(\ell, \overline{b}) \dashv \mu'}{\mu;\sigma \vdash \langle \ell, \varrho \rangle : \textcolor{red}{\tau^p} \leftsquigarrow \overline{b} : \mathsf{int}(q) \dashv \mu';\sigma}$$

## 2.6 Store Operations

$$\boxed{\mu(\ell) = \langle b, \varrho \rangle}$$

“*The store $\mu$ maps the location $\ell$ to the byte $b$ with provenance $\varrho$.*”

Store

$$\frac{\ell \mapsto \langle b, \varrho \rangle \in \mu}{\mu(\ell) = \langle b, \varrho \rangle}$$

$$\boxed{\mu(\ell, m) = \overline{\langle b, \varrho \rangle}^{m}}$$

“*Reading a value of size $m$ from location $\ell$ produces a list of $m$ pairs of bytes and provenance values.*”

Store-Slice

$$\frac{\mu(\ell), \ldots, \mu(\ell + m - 1) = \overline{\langle b, \varrho \rangle}^{m}}{\mu(\ell, m) = \overline{\langle b, \varrho \rangle}^{m}}$$

$$\boxed{\mathsf{expose}(\sigma, \varrho) = \sigma'}$$

“*Exposing the tag $\varrho$ produces the updated tag set $\sigma'$.*”

Ex-Tag

$$\mathsf{expose}(\sigma, t) = \sigma \cup \{t\}$$

Ex-Null

$$\mathsf{expose}(\sigma, \cdot) = \sigma$$

Ex-Wild

$$\mathsf{expose}(\sigma, *) = \sigma$$

$\boxed{\mu \vdash \mathsf{write}(\ell, v) \dashv \mu'}$

"*Writing the value $v$ to the store $\mu$ at location $\ell$ produces the updated store $\mu'$.*"

$$\textsc{W-Bytes}\quad \frac{\begin{array}{c}\ell \in \mathsf{dom}(\mu_0) \\ \forall i \in [0, n-1].\mu_{i+1} = \mu_i[\ell + i \mapsto \langle b_i, \cdot\rangle]\end{array}}{\mu_0 \vdash \mathsf{write}(\ell, \overline{b}^n) \dashv \mu_n}$$

$$\textsc{W-Ptr}\quad \frac{\begin{array}{c}\ell \in \mathsf{dom}(\mu_0) \qquad \ell \triangleq \overline{b}^{n_{ptr}} \\ \forall i \in [0, n_{ptr}-1].\mu_{i+1} = \mu_i[\ell + i \mapsto \langle b_i, \varrho\rangle]\end{array}}{\mu_0 \vdash \mathsf{write}(\ell, \langle \ell, \varrho\rangle) \dashv \mu_n}$$

$\boxed{\mu; \sigma \vdash \mathsf{read}(\ell, \tau) = v \dashv \sigma'}$

"*Reading a rust value $v$ of type $\tau$ from the store $\mu$ at location $\ell$ produces the updated tag set $\sigma'$.*"

$$\textsc{R-Int}\quad \frac{\begin{array}{c}\mu[\ell, n] = \overline{\langle b, \varrho\rangle}^n \\ \forall i \in [1, n].\mathsf{expose}(\sigma_{i-1}, \varrho_i) = \sigma_i\end{array}}{\mu; \sigma_0 \vdash \mathsf{read}(\ell, \mathsf{int}(n)) = \overline{b}^n \dashv \sigma_n}$$

$$\textsc{R-Ptr}\quad \frac{\begin{array}{c}\mu[\ell, n_{ptr}] = \overline{\langle b, \varrho\rangle}^{n_{ptr}} \\ \ell' \triangleq \overline{b}^{n_{ptr}} \qquad \forall i \in [1, n_{ptr}].\varrho_i = \varrho'\end{array}}{\mu; \sigma \vdash \mathsf{read}(\ell, *\tau) = \langle \ell', \varrho'\rangle \dashv \sigma}$$

## 2.7 Metafunctions

$\boxed{\mathsf{sizeof}(\tau) = n}$

"*The type $\tau$ has size $n$.*"

$$\textsc{TS-Int}\quad \mathsf{sizeof}(\mathsf{int}(n)) = n \qquad \textsc{TS-R-Ptr}\quad \mathsf{sizeof}(*\tau) = n_{ptr} \qquad \textsc{TS-L-Ptr}\quad \mathsf{sizeof}(\mathsf{ptr}) = n_{ptr}$$

$$\textsc{TS-R-Field}\quad \frac{\mathsf{sizeof}(\tau) + m = n}{\mathsf{sizeof}(\langle \tau, m\rangle) = n} \qquad \textsc{TS-R-Prod}\quad \frac{\Sigma_{i=1}^{m}(\mathsf{sizeof}({\tau^p}_i)) = n}{\mathsf{sizeof}(\overline{\tau^p}^m) = n} \qquad \textsc{TS-L-Prod}\quad \frac{\Sigma_{i=1}^{m}(\mathsf{sizeof}(\tau_i)) = n}{\mathsf{sizeof}(\overline{\tau}^m) = n}$$

$\boxed{\mathsf{scalar}(\tau)}$

"*The type $\tau$ is a scalar.*"

$$\mathsf{scalar}(\mathsf{int}(n)) \qquad \mathsf{scalar}(*\tau) \qquad \mathsf{scalar}(\mathsf{ptr})$$

$\boxed{\mathsf{fields}(v : \tau^p) = \overline{v : \tau}}$

"*The rust product value $v : \tau^p$ can be represented as a list of field values $v : \tau$*"

$$\frac{\forall i \in [1, n].o_i = \Sigma_{j=1}^{i-1}(\mathsf{sizeof}({\tau^p}_j))}{\mathsf{fields}(\langle \ell, \varrho\rangle : \overline{\tau^p}^n) = \overline{\langle \ell + o_i, \varrho\rangle : {\tau^p}_i}^n}$$

$\boxed{\mathsf{homogeneous}(\tau)}$

"*The type $\tau$ is a homogeneous aggregate.*"

$$\mathsf{homogeneous}(\tau_b) \qquad \mathsf{homogeneous}(*\tau) \qquad \frac{\exists \tau.\forall \langle \tau', n\rangle \in \tau^p.n = 0 \land \tau' = \tau \land \mathsf{homogeneous}(\tau)}{\mathsf{homogeneous}(\tau^p)}$$

$\boxed{\mathsf{equivalent}(\tau) = \tau'}$

"*The type* $\tau$ *is equivalent to the type* $\tau'$"

$$\mathsf{equivalent}(\tau_b) = \tau_b \qquad \mathsf{equivalent}(*\tau) = \mathsf{ptr} \qquad \mathsf{equivalent}(\mathsf{ptr}) = *\tau$$

## 2.8 Well-formedness

$\boxed{\vdash v : \tau}$

"*The typed value* $v : \tau$ *is well-formed.*"

$$\textsc{WF-Int}\ \ \vdash \overline{b}^{n} : \mathsf{int}(n) \qquad \textsc{WF-LLVMPtr}\ \ \vdash \langle \ell, \varrho \rangle : \mathsf{ptr} \qquad \textsc{WF-RustPtr}\ \ \vdash \langle \ell, \varrho \rangle : *\tau$$

$$\textsc{WF-RustProd}\ \ \vdash \langle \ell, \varrho \rangle : \tau^{p} \qquad \textsc{WF-LLVMProd}\ \ \frac{\forall i \in [1, n] \vdash v_i : \tau_i}{\vdash \langle \overline{v}^{n} \rangle : \overline{\tau}^{n}}$$

## 2.9 Proofs

**Lemma 2.9.1** (Canonical Forms)**.** For all values $v$, if $\vdash v : \tau$, then

1. If $v \triangleq \overline{b}^{n}$, then $\tau \triangleq \mathsf{int}(n)$.
2. If $v \triangleq \langle \ell, \varrho \rangle$, then $\tau$ is either $*\tau$ or $\tau^{p}$ in Rust, or $\mathsf{ptr}$ in LLVM.
3. If $v \triangleq \langle \overline{v}^{n} \rangle$ then $\tau$ is an LLVM product type $\overline{\tau}^{n}$

*Proof.* By inspection of $\vdash v : \tau$. □

**Lemma 2.9.2** (Compatible Forms)**.** For all Rust typed values $v : \tau$ and LLVM typed values $v : \tau$, if $\mu; \sigma \vdash v : \tau \rightsquigarrow v : \tau \dashv \mu; \sigma'$ then $v : \tau$ determines the possible forms of $v : \tau$.

1. If $v : \tau \triangleq \overline{b}^{n} : \mathsf{int}(n)$, then $v : \tau \triangleq v : \tau$
2. If $v : \tau \triangleq \langle \ell, \varrho \rangle : \tau$, then $v : \tau$ is either:
   (a) An opaque pointer of the form $\langle \ell, \varrho \rangle : \mathsf{ptr}$.
   (b) An integer of the form $\overline{b}^{n} : \mathsf{int}(n)$.
   (c) An LLVM product type $\langle \overline{v} \rangle : \overline{\tau}$.

Similarly, if $\mu; \sigma \vdash v : \tau \leftsquigarrow v : \tau \dashv \mu'; \sigma$
then the form of $v : \tau$ determines the possible forms of $v : \tau$.

1. If $v : \tau \triangleq \overline{b}^{n} : \mathsf{int}(n)$, then $v : \tau$ is either:
   (a) An integer value of the same form.
   (b) A pointer value of the form $\langle \ell, * \rangle : *\tau$
   (c) A Rust product value $\langle \ell, \varrho \rangle : \tau^{p}$ stored at some valid location $\ell$.
2. If $v : \tau \triangleq \langle \ell, \varrho \rangle : \mathsf{ptr}$, then $v : \tau$ is either:
   (a) A Rust product value $\langle \ell, \varrho \rangle : \tau^{p}$ for some $\tau^{p}$
   (b) A Rust pointer value $\langle \ell, \varrho \rangle : *\tau$ for some $\tau$.
3. If $v : \tau \triangleq \langle \overline{\tau} \rangle : \overline{\tau}$ then $v : \tau$ must be a Rust product value $\langle \ell, \varrho \rangle : \tau^{p}$ for some $\tau^{p}$.

*Proof.* By Lemma 2.9.1 and inspection of the syntax for the value conversion judgement. □

**Lemma 2.9.3.** For all well-formed, typed scalar values $v : \tau$ and all valid heap locations $\ell$, we have:

$$\mu \vdash \mathsf{write}(\ell, v) \dashv \mu' \quad \Rightarrow \quad \mu'; \sigma \vdash \mathsf{read}(\ell, \tau) = v \dashv \sigma'$$

*Proof.* By inversion, guided by the structure of $v : \tau$. Since $v : \tau$ is a well-formed, scalar-typed value, we have $\vdash v : \tau$ and $\mathsf{scalar}(\tau)$. It follows that $v$ is either a byte string $\overline{b}$ or a pointer $\langle \ell', \varrho \rangle$ to some location $\ell$ with some provenance $\varrho$.

**Case 1:** $v \triangleq \overline{b}^n$

By Lemma 2.9.1 we have that $\tau \triangleq \mathsf{int}(n)$. By inversion of W-Bytes and for $i \in [0, n-1]$, the store $\mu'$ maps each location $\ell + i$ to the tuple $\langle b_{i+1}, \cdot \rangle$. By Store and Store-List, we have that

$$\mu'(\ell), \ldots, \mu'(\ell + n - 1) = \mu'(\ell, n) = \overline{\langle b, \cdot \rangle}^n$$

Exposing the null provenance of each byte leave $\sigma$ unchanged (Ex-Null). We can now apply R-Int to read the original value $\overline{b}$ back from the store.

**Case 2:** $v \triangleq \langle \ell, \varrho \rangle$

By Lemma 2.9.1 and since $\mathsf{scalar}(\tau)$, we have that $\tau$ is either $*\tau$ of $\mathsf{ptr}$. Each are treated equivalently. We can implicitly convert the location $\ell$ into the byte string, $\overline{b}^n_{ptr}$, so we proceed as in the first case. However, instead of the null provenance, we have:

$$\mu'(\ell, n_{ptr}) = \overline{\langle b, \varrho \rangle}^{n_{ptr}}$$

Each $\varrho_i$ is equivalent to the provenance $\varrho$ of the pointer value. Now, we can apply R-Ptr to reach our goal by reading the original value $\langle \ell, \varrho \rangle$ back from the store.

□

**Theorem 2.9.1** (Conversion is semi-functional)**.** For all well-typed values $v : \tau$ and $v : \tau$, there exists some heaps $\mu, \mu'$ and tag sets $\sigma, \sigma'$ such that

$$\mu; \sigma \vdash v : \tau \leftsquigarrow v : \tau \dashv \mu'\sigma \Rightarrow \mu'; \sigma \vdash v : \tau \rightsquigarrow v : \tau \dashv \mu'\sigma'$$

Converting an LLVM value to Rust may affect the heap, but it will not change the tag set. Likewise, converting a Rust value to LLVM may affect the tag set but it will not change the contents of the heap.

**Case 1:** $v : \tau \triangleq \overline{b} : \tau$

By Lemma 2.9.2, $v : \tau$ can take one of the following forms:

*Subcase 1:* $v : \tau \triangleq \overline{b} : \tau_b$

Both typed values are interconvertible by C-Int.

*Subcase 2:* $v : \tau \triangleq \langle \ell, * \rangle : \tau$

By inversion of C-PtrFromInt, we have $\ell \triangleq \overline{b}$. We can now apply C-PtrToInt to achieve our goal.

*Subcase 3:* $v : \tau \triangleq \langle \ell, * \rangle : \langle \tau, 0 \rangle$ and $\mathsf{scalar}(\tau)$

By inversion of C-FieldFromScalar, we have:

$$\mu; \sigma \vdash v_b : \tau \leftsquigarrow v_b : \tau \dashv \mu''; \sigma \qquad \mu'' \vdash \mathsf{write}(\ell, \overline{b}) \dashv \mu' \qquad \mathsf{scalar}(\tau)$$

By the induction hypothesis and Lemma 2.9.3, we have:

$$\mu'\sigma \vdash \mathsf{read}(\ell, \tau) \dashv \mu'; \sigma'' \qquad \mu'; \sigma'' \vdash \tau \rightsquigarrow v_b \dashv \mu'; \sigma'$$

Now we can apply C-FieldToScalar to reach our goal.

*Subcase 4:* $v : \tau \triangleq \langle \ell, * \rangle : \tau^p$

By inversion of C-ProdFromInt, we have:

$$\mathsf{sizeof}(\tau^p) = q \qquad \mu \dashv \mathsf{write}(\ell, \overline{b}, \dashv)\mu'$$

By Lemma 2.9.3, we have:

$$\mu'\sigma \vdash \mathsf{read}(\ell, \tau) \dashv \mu'; \sigma'$$

We can apply C-ProdToInt to reach our goal.

**Case 2:** $v : \tau \triangleq \langle \ell, \varrho \rangle : \mathsf{ptr}$

By Lemma 2.9.2, $v : \tau$ must take the following forms:

*Subcase 1:* $v : \tau \triangleq \langle \ell, \varrho \rangle : *\tau$ for some $\tau$

Both typed values are interconvertible by C-PointerFromPointer and C-AnyToPointer.

*Subcase 2:* $v : \tau \triangleq \langle \ell, \varrho \rangle : \langle \tau, 0 \rangle$ for some $\tau$

Equivalent to Case 1, Subcase 3.

**Case 3:** $v : \tau \triangleq \langle \overline{v} \rangle : \overline{\tau}$

By Lemma 2.9.2, $v : \tau$ must be equivalent to $\langle \ell, \varrho \rangle : \tau^p$ for some $\tau^p$, which is interconvertible by C-Product and the induction hypothesis.

**Theorem 2.9.2** (Equal size is required)**.** Value conversion will succeed if and only if Rust and LLVM values have the same size. For all well-typed values $v : \tau$ and $v : \tau$, if there exists some heaps $\mu, \mu'$ and tag sets $\sigma, \sigma'$ such that either

$$\mu; \sigma \vdash v : \tau \leftsquigarrow v : \tau \dashv \mu'; \sigma \qquad \text{or} \qquad \mu; \sigma \vdash v : \tau \rightsquigarrow v : \tau \dashv \mu; \sigma'$$

Then either $\mathsf{sizeof}(\tau) = \mathsf{sizeof}(\tau)$ or $\tau = \mathsf{ptr}$. That is, conversion will get "stuck" (which is reported as undefined behavior) if the types on either side of the boundary have unequal size, unless the LLVM type is an opaque pointer.

*Proof.* By induction on value conversion. Cases in either direction are equivalent; here, we consider the forwards case ($\rightsquigarrow$) of a Rust value $v : \tau$ being converted into an LLVM value. By Lemma 2.9.2, if the Rust value is an integer such that $v : \tau \triangleq \overline{b} : \tau_b$, then the LLVM value has the same type, so size is preserved. The remaining cases involve products and pointers, where the Rust value takes the form $\langle \ell, \varrho \rangle$ for some $\tau$.

**Case 1:** $\tau \triangleq *\tau$

Then the the LLVM value is either an opaque pointer (C-Pointer) or an integer with a size equal to the size of a pointer (C-PointerToInt).

**Case 2:** $\tau \triangleq \langle \tau^p, 0 \rangle$

By inversion of FieldToScalar, the induction hypothesis, and TS-R-Field.

**Case 3:** $\tau \triangleq \tau^p, \tau : v \triangleq \overline{b} : \mathsf{int}(q)$

By inversion of C-ProductToInt we have $\mathsf{sizeof}(\tau^p) = q$, which is equal to the size of the value read from memory.

**Case 4:** $\tau \triangleq \tau^p, \tau : v \triangleq \langle \overline{v} \rangle : \overline{\tau}$

By inversion of C-Product and the induction hypothesis, size is preserved for each field, so size is preserved for the entire product.

□

## 2.10 Parameter Passing

**Algorithm 1:** Converting a list of LLVM arguments to Rust arguments.

```
// A list of typed values provided by Rust
$R \leftarrow [\overline{v, \tau}^n]$;
// A list of LLVM types.
$L \leftarrow [\overline{\tau}^n]$;
// A calling convention; either ‘static’ or ‘variable’.
$C \leftarrow c$;
// The list of converted arguments
$A \leftarrow []$;
// The initial store and tag set.
$\mathcal{S} \leftarrow \mu; \sigma$;
while $R$ is not empty do
    $v_i : \tau_i \leftarrow \mathsf{next}(R)$;
    if $L$ is not empty then
        $\tau_j \leftarrow \mathsf{next}(L)$;
        if $\mathsf{sizeof}(\tau_i) = \mathsf{sizeof}(\tau_j)$ then
            $\mathcal{S} \leftarrow \mathcal{S}'$ where $\mathcal{S} \vdash v_i : \tau_i \rightsquigarrow v_j : \tau_j \dashv \mathcal{S}'$;
            $A \leftarrow A \mathbin{++} [v_j : \tau_j]$;
            continue
        else
            /* We only expand homogeneous aggregates when converting from Rust; in
               the other direction, we skip directly to an error.                   */
            if $\tau \triangleq \overline{\tau^p}^n$ and $\mathsf{homogeneous}(\tau)$ then
                if $\mathsf{len}(L) \geq n + \mathsf{len}(R)$ then
                    $R \leftarrow R \mathbin{++} [\mathsf{fields}(v_i : \tau_i)]$;
                    continue
                end
            end
        end
    else
        if $L$ is empty and $C = \mathsf{variable}$ and $\mathsf{scalar}(\tau_i)$ then
            $L \leftarrow L \mathbin{++} [\mathsf{equivalent}(\tau_i)]$; continue
        end
    end
    error()
end
```