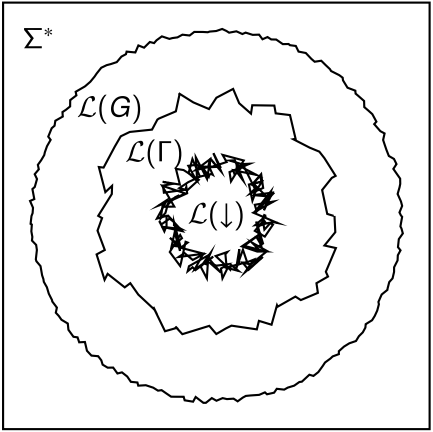
A Relevance Sampler for $\mu\text{Rust}_{\text{sl}}$

Breandan Mark Considine

December 13, 2025

Programming language [in]approximability

- Σ^*: all words over Σ
- $\mathcal{L}(G)$: syntactically valid
- $\mathcal{L}(\Gamma)$: type-safe programs
- $\mathcal{L}(\downarrow)$: halting programs
- Most LLMs: $\sigma \leftsquigarrow \Sigma^*$
- Typesafe: $\sigma \leftsquigarrow \mathcal{L}(\Gamma)$
- Tighter approximations require ever-increasing expressive power
- Volumes are not to scale



High-level plan

Morally, we want to generate type-safe Rust code. We will focus on a straight-line fragment with nominal relevance, lower this into a context-free grammar (CFG) using a fixed-parameter tractable embedding, and then decode the CFG by slice sampling.

1. Consider μRust, a tiny sublangauge of Rust.
2. Restrict to straight-line fragment, $\mu\text{Rust}_{\text{SL}} \subset \mu\text{Rust}$.
3. Embed $\mu\text{Rust}_{\text{SL}}$ into a CFG, $G := \langle \Sigma, V, P, S \rangle$.
4. Sample words from $\sigma \leftsquigarrow \mathcal{L}(G) \cap \Sigma^n$.

Substructural type systems

	$\frac{\Gamma,\Delta \vdash A}{\Delta,\Gamma \vdash A}$	$\frac{\Gamma,A,A \vdash B}{\Gamma,A \vdash B}$	$\frac{\Gamma \vdash C}{\Gamma,A \vdash C}$	
	Exchange	**Contraction**	**Weakening**	Usage
Ordered	✗	✗	✗	$= 1$
Linear	✓	✗	✗	$= 1$
Affine	✓	✗	✓	≤ 1
Relevant	✓	✓	✗	≥ 1
Classical	✓	✓	✓	≥ 0

Table: Each typing discipline enforces different usage obligations. **Exchange** allows permutation of assumptions, **contraction** allows duplication or merging, and **weakening** allows discarding assumptions.

Context-free grammars (CFG)

Definition. A CFG is a quadruple $G := \langle \Sigma, V, P, S \rangle$ where:

- Σ is a finite set of *terminals*,
- V is a finite set of *nonterminals*, disjoint from Σ,
- $P \subseteq V \times (V \cup \Sigma)^*$ is a finite set of *productions* $W \to \sigma$,
- $S \in V$ is the *start* symbol.

Chomsky Normal Form (CNF). A CFG is in CNF if every $p \in P$ is either

$$W \to XZ \quad \text{or} \quad W \to a,$$

with $W, X, Z \in V$, $a \in \Sigma$.

- Every CFG G has an equivalent CFG G' in CNF s.t. $\mathcal{L}(G') = \mathcal{L}(G)$. The conversion has polynomial worst case blowup.

μRust: Syntax and semantics

Consider a singly-typed language with the following terms:

$$
\begin{array}{lcl}
\texttt{FUN} & ::= & \texttt{fn f0 (PRM) -> } \mathbb{T} \texttt{ \{ BDY \}} \\
\texttt{PRM} & ::= & \texttt{PID : } \mathbb{T} \mid \texttt{PRM , PID : } \mathbb{T} \\
\texttt{BDY} & ::= & \texttt{INV} \mid \texttt{STM BDY} \\
\texttt{STM} & ::= & \texttt{let PID = INV ;} \\
\texttt{INV} & ::= & \texttt{FID (ARG)} \\
\texttt{ARG} & ::= & \texttt{PID} \mid \texttt{ARG , ARG} \\
\texttt{PID} & ::= & \texttt{p1} \mid \ldots \mid \texttt{pk} \\
\texttt{FID} & ::= & \texttt{f0} \mid \texttt{f1} \mid \ldots \mid \texttt{fm}
\end{array}
$$

Assume an ambient context, Γ, consisting of $\texttt{f1}, \ldots, \texttt{fm}$:

$$\Gamma \quad ::= \quad \varnothing \quad \mid \quad \Gamma, \texttt{f\_} : (\tau_1, \ldots, \tau_k) \to \tau$$

The unrestricted semantics are conventional.

μRust: Examples

This admits straight line programs (SLPs) of the following shape,

```
fn f0(p1 : T, p2 : T) -> T {
    let p3 = mul(p1, p2);
    let p4 = add(p1, p1);
    let p5 = mul(p1, p3);
    let p6 = add(p3, p1);
    add(p3, p5)
}
```

```
> Warning: p4, p6 are unused!
```

however unused resources, i.e., names may remain after returning.

Ambient context

Now, let us interpolate `f0` as a string inside the following context:

```
[forbid(unused_variables)]
[derive(Clone, Copy, Debug)]
[must_use]
pub struct T(i128);
fn add(_ : T, _ : T) -> T { T(0) }
fn mul(_ : T, _ : T) -> T { T(0) }
...
fn fm(_ : T, ..., _ : T) -> T { T(0) }
fn f0(p1 : T, ..., pk : T) -> T { <...> }
fn main() { println!("{:?}", f0(T(0), T(1))) }
```

$\mu\text{Rust}_{\text{SL}}$: Relevance semantics

Obligations. For `f0` with parameters $p_1 : \tau_1, \ldots, p_k : \tau_k$, initialize $\Phi = \{p_1, \ldots, p_k\}$. Each bound name must be used *at least once*. Locals introduced by `let` also carry obligations. Body is well-typed iff all obligations are discharged, i.e., $\Gamma, \Delta \vdash \mathtt{BDY} : \tau \mid \Phi' \Rightarrow \emptyset$.

$$\text{Judgments:} \quad \Gamma, \Delta \;\vdash\; t : \tau \mid \Phi \Rightarrow \Phi'.$$

$$\frac{p : \tau \in \Gamma \qquad p \in \Phi}{\Gamma, \Delta \vdash p : \tau \mid \Phi \Rightarrow \Phi \setminus \{p\}}\ (\text{VAR}) \qquad \frac{p : \tau \in \Gamma \qquad p \notin \Phi}{\Gamma, \Delta \vdash p : \tau \mid \Phi \Rightarrow \Phi}\ (\text{VAR}_{\notin})$$

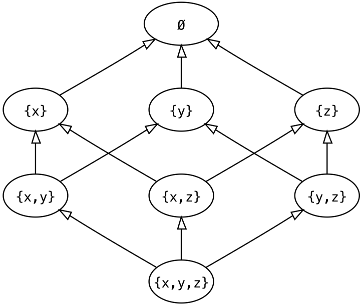
$$\frac{\Gamma \vdash \mathtt{f\_} : (\tau_1, \ldots, \tau_m) \to \tau \qquad \Gamma, \Delta \vdash e_i : \tau_i \mid \Phi_{i-1} \Rightarrow \Phi_i \ \forall i \in [1, m]}{\Gamma, \Delta \vdash \mathtt{f\_}\ \mathtt{(}\ e_1\ \mathtt{,}\ \ldots\ \mathtt{,}\ e_m\ \mathtt{)} : \tau \mid \Phi_0 \Rightarrow \Phi_m}\ (\text{INV})$$

$$\frac{\Gamma, \Delta \vdash s_1 : \mathtt{unit} \mid \Phi_0 \Rightarrow \Phi_1 \qquad \Gamma, \Delta \vdash s_2 : \mathtt{unit} \mid \Phi_1 \Rightarrow \Phi_2}{\Gamma, \Delta \vdash s_1\ \mathtt{;}\ s_2 : \mathtt{unit} \mid \Phi_0 \Rightarrow \Phi_2}\ (\text{SEQ})$$

$$\frac{\Gamma, \Delta \vdash e : \tau \mid \Phi_0 \Rightarrow \Phi_1 \qquad \Gamma, \Delta \vdash x : \tau \mid \Phi_1 \cup \{x\} \Rightarrow \Phi_2}{\Gamma, \Delta \vdash \mathtt{let}\ x\ \mathtt{=}\ e : \mathtt{unit} \mid \Phi_0 \Rightarrow \Phi_2}\ (\text{LET})$$

CFG embedding: intution

To express all $\Phi \Rightarrow \Phi'$ possible transitions, we must construct a Hasse digram, H_k, e.g., for $k = 3$ parameters $\{x, y, z\}$,



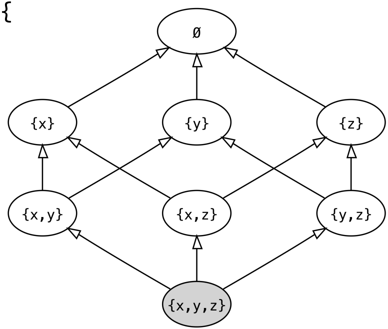
for all relevant productions. This will be tractable for $k \lesssim 10$.

$$\left|\{v, v' \in H_k \mid v \subset v'\}\right| = \sum_{i=0}^{k} \binom{k}{i} \left(2^{k-i} - 1\right) = 3^k - 2^k.$$

Path enumeration

Our grammar will need to express all possible transition paths, e.g.,

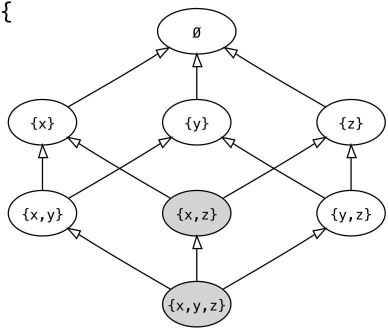
```
fn f0(x : T, y : T, z : T) -> T {
    // Unused: {x,y,z}
}
```



Path enumeration

Our grammar will need to express all possible transition paths, e.g.,

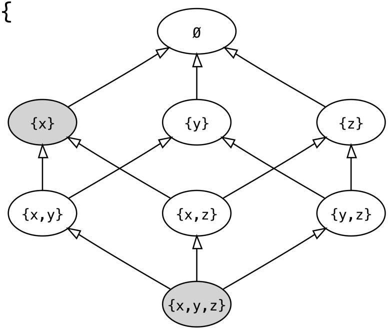
```
fn f0(x : T, y : T, z : T) -> T {
   let _ = mul(y, y); // {x,z}
   // BDY | {x,y,z} => {x,z}
}
```



Path enumeration

Our grammar will need to express all possible transition paths, e.g.,

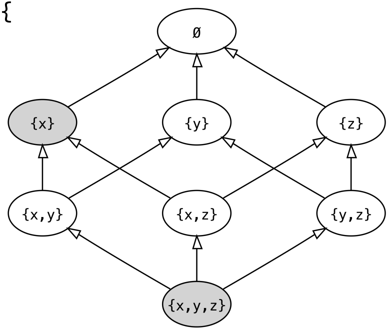
```
fn f0(x : T, y : T, z : T) -> T {
    let _ = mul(y, y); // {x,z}
    let _ = add(y, z); // {x}
    // BDY | {x,y,z}=>{x}
}
```



Path enumeration

Our grammar will need to express all possible transition paths, e.g.,

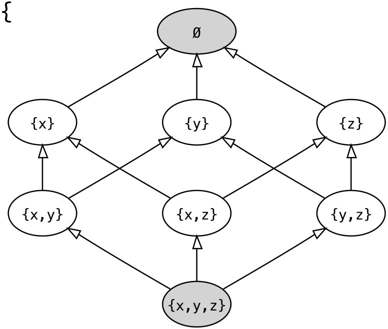
```
fn f0(x : T, y : T, z : T) -> T {
    let _ = mul(y, y); // {x,z}
    let _ = add(y, z); // {x}
    let _ = mul(z, y); // {x}
    // BDY | {x,y,z}=>{x}
}
```



Path enumeration

Our grammar will need to express all possible transition paths, e.g.,

```
fn f0(x : T, y : T, z : T) -> T {
    let _ = mul(y, y); // {x,z}
    let _ = add(y, z); // {x}
    let _ = mul(z, y); // {x}
    let _ = mul(x, z); // {}
    // BDY | {x,y,z}=>{}
}
```



CFG embedding

We want to permit only functions with no outstanding obligations. Construct a CFG, $G_\Gamma = \langle \Sigma, V_\Gamma, P_\Gamma, S_\Gamma \rangle$:

$$\frac{\vec{\tau} : \mathbb{T}^{0..k} \qquad \Phi \Rightarrow \Phi' = \{p_i, \ldots, p_k\} \Rightarrow \varnothing}{\left(S_\Gamma \to \texttt{fn}\ \texttt{f0}\ \texttt{(}\ \mathop{\large ,}_{i=1}^{|\vec{\tau}|} \left(p_i\ \texttt{:}\ \vec{\tau}_i\right) \texttt{)}\ \texttt{:}\ \tau\ \texttt{=}\ \mathtt{BDY}[\tau, \Phi \Rightarrow \Phi']\right) \in P_\Gamma}\ \mathtt{FUN}_\varphi$$

We will decorate nonterminals with a pair of (1) the expression's local return type (τ), and (2) relevance obligations ($\Phi \Rightarrow \Phi'$):

$$\frac{\Gamma \vdash \texttt{f\_} : (\tau_1, \ldots, \tau_m) \to \tau \qquad \Phi' \subseteq \Phi \qquad \Phi \setminus \Phi' = \bigcup_{i=1}^{m} \{p_i\}}{\left(\mathtt{INV}[\tau, \Phi \Rightarrow \Phi']\ \to\ \texttt{f\_}\ \texttt{(}\ \mathop{\large ,}_{i=1}^{m} p_i\ \texttt{)}\right) \in P_\Gamma}\ \mathtt{INV}_\varphi$$

where $\mathop{\large ,}(\cdot)$ denotes a macro for a comma-separated list, i.e.,

$$\mathop{\large ,}_{i=1}^{m}(x_i) := x_1\ \texttt{,}\ \ldots\ x_m \text{ if } m > 1 \text{ else } x_1 \text{ if } m = 1 \text{ else } \varepsilon$$

μRust$_{\mathrm{SL}}$: Sequencing and binding

Sequencing. A sequence s_1 `;` s_2 composes obligation contexts:

$$[\![s_1 \; \mathtt{;} \; s_2]\!] \;=\; [\![s_2]\!] \circ [\![s_1]\!].$$

$$\frac{\Gamma, \Delta \vdash s_1 : \mathtt{unit} \mid \Phi_0 \Rightarrow \Phi_1 \qquad \Gamma, \Delta \vdash s_2 : \mathtt{unit} \mid \Phi_1 \Rightarrow \Phi_2}{\Gamma, \Delta \vdash s_1 \; \mathtt{;} \; s_2 : \mathtt{unit} \mid \Phi_0 \Rightarrow \Phi_2} \text{(SEQ)}$$

Let-binding. A local binding introduces a fresh obligation that must be subsequently discharged:

$$\Gamma, \Delta \vdash \mathtt{let}\; x = e \quad \text{acts as} \quad \Phi_0 \xrightarrow{e} \Phi_1 \xrightarrow{\cup\{x\}} \Phi_2$$

$$\frac{\Gamma, \Delta \vdash e : \tau \mid \Phi_0 \Rightarrow \Phi_1 \qquad \Gamma, \Delta \vdash x : \tau \mid \Phi_1 \cup \{x\} \Rightarrow \Phi_2}{\Gamma, \Delta \vdash \mathtt{let}\; x \;\mathtt{=}\; e : \mathtt{unit} \mid \Phi_0 \Rightarrow \Phi_2} \text{(LET)}$$

μRust$_{\text{SL}}$ embedding: sequencing and binding

Sequencing. Recall the (SEQ) rule, which $\mathtt{BDY}_\varphi$ will mirror:

$$\big(\mathtt{BDY}[\tau, \Phi_0 \Rightarrow \Phi_2] \;\rightarrow\; \mathtt{STM}[\mathtt{unit}, \Phi_0 \Rightarrow \Phi_1]\ \mathtt{;}\ \mathtt{BDY}[\tau, \Phi_1 \Rightarrow \Phi_2]\big) \in P_\Gamma,$$

$$\big(\mathtt{BDY}[\tau, \Phi \Rightarrow \varnothing] \;\rightarrow\; \mathtt{INV}[\tau, \Phi \Rightarrow \varnothing]\big) \in P_\Gamma$$

for all possible obligation states Φ_0, Φ_1, Φ_2, s.t. $\Phi_2 \subseteq \Phi_1 \subseteq \Phi_0$.

Let-binding. $\mathtt{STM}_\varphi$ generates a set of $\mathtt{STM}$ productions. Whenever,

$$\Gamma, \Delta \vdash e : \tau \mid \Phi_0 \Rightarrow \Phi_1 \quad \text{and} \quad \Gamma, \Delta \vdash x : \tau \mid \Phi_1 \cup \{x\} \Rightarrow \Phi_2,$$

we will add the corresponding production:

$$\big(\mathtt{STM}[\mathtt{unit}, \Phi_0 \Rightarrow \Phi_2] \;\rightarrow\; \mathtt{let}\ x\ \mathtt{=}\ \mathtt{INV}[\tau, \Phi_0 \Rightarrow \Phi_1]\big) \in P_\Gamma,$$

These rules ensure every word $\sigma \in \mathcal{L}(\mathtt{BDY}[\tau, \Phi \Rightarrow \varnothing])$ corresponds to a well-typed relevant μRust$_{\text{SL}}$ fragment.

Future work

- More compact embeddings and asymptotic complexity
- Laziness: instantiate CFG productions during parsing
- Extend to richer type systems, e.g., polymorphism, higher-order functions, subtyping, nested scope
- "A Word Sampler for Well-Typed Functions" (Considine, 2025) explores a first order, simply typed langauge with finite types
- " A Tree Sampler for Bounded CFLs" (Considine, 2024) describes a uniform sampler for finite CFL intersections
- Other FPT embeddings. Open to suggestions!
- Applications to program synthesis and repair
- Feasible matrix multiplication algorithms
- Try it yourself at: https://tidyparse.github.io/cnf.html

Acknowledgements

Thank you!

- George Morgan
- Tali Benyon
- Ori Roth
- Chuta Sano
- Brigitte Pientka
- David Bieber
- David Chiang
- David Yu-Tung Hui
- Margaret Considine
- Mark Considine

Lastly, thank you to the MWPLS organizers!