

Theory exercises

Program verification with types and logic (NWI-IMC060)

Week 8

1. To guarantee safe concurrency, Rust uses the following traits:

- A type is **Send** if it is safe to send it to another thread.
- A type is **Sync** if it is safe to share between threads (**T** is **Sync** if and only if **&T** is **Send**).

This exercise is about the **Send** and **Sync** traits.

- (a) Give an example of a type is both **Send** and **Sync**, and explain why.
 - (b) Give an example of a type is neither **Send** nor **Sync**, and explain why.
 - (c) Give an example of a type is **Send**, but not **Sync**, and explain why.
2. We consider the semantics of a simply-typed language with **spawn** and **join** similar to those operations in Rust. The formal syntax of our language is as follows:

$$\begin{aligned} e \in \text{Expr} &::= x \mid n \mid tid \mid \lambda x. e \mid e_1 e_2 \mid \text{spawn } e \mid \text{join } e \\ A \in \text{Type} &::= \text{nat} \mid A_1 \rightarrow A_2 \mid \text{joinhandle } A \end{aligned}$$

We let variables $x \in \text{string}$, numerals $n \in \mathbb{N}$, and thread IDs $tid \in \mathbb{N}$.

Aside from the usual typing rules of the simply-typed lambda calculus with natural numbers, the type system has the following typing rules for **spawn** and **join**:

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \text{spawn } e : \text{joinhandle } A} \quad \frac{\Gamma \vdash e : \text{joinhandle } A}{\Gamma \vdash \text{join } e : A}$$

The intuitive semantics is that **spawn** e creates new thread that runs expression e , and returns the thread's join handle (represented as a thread ID). The construct **join** e will wait for the spawned thread e to terminate. Since the type system is unrestricted (*i.e.*, not substructural), one can use a join handler zero or multiple times, possibly in different threads. The idea is that each **join** will spin until the associated thread has terminated, and then obtains the return value of that thread.

We let $\text{Val} \subseteq \text{Expr}$ denote the subset of expressions that are values, *i.e.*, numerals n , thread IDs tid , and functions $\lambda x. e$. Configurations σ are represented as lists of threads $e_1 \dots e_n$. The main thread is the first element in the configuration, followed by all other threads in the order they were spawned. Threads that have terminated (*i.e.*, have reduced to a value) are kept in the

configuration. The small-step reduction $\sigma \Rightarrow_i \sigma'$ says that thread i in configuration σ can step to configuration σ' :

$$\frac{\sigma(i) = k((\lambda x. e) v) \quad \text{ctx } k}{\sigma \Rightarrow_i \sigma[i := k(\text{subst } x v e)]} \quad \frac{\sigma(i) = k(\text{spawn } e) \quad tid = \text{length } \sigma \quad \text{ctx } k}{\sigma \Rightarrow_i (\sigma[i := k tid]) ++ [e]}$$

$$\frac{\sigma(i) = k(\text{join } tid) \quad \sigma(tid) = e \quad e \notin \text{Val} \quad \text{ctx } k}{\sigma \Rightarrow_i \sigma}$$

$$\frac{\sigma(i) = k(\text{join } tid) \quad \sigma(tid) = v \quad v \in \text{Val} \quad \text{ctx } k}{\sigma \Rightarrow_i \sigma[i := k v]}$$

We represent evaluation contexts k as functions from expressions to expressions. The judgment $\text{ctx } k$ says that k is a valid evaluation context. We use the notation $\sigma(i)$ to look up the i th element of a list (*i.e.*, the assertion $\sigma(i) = e$ means that the index i is within bounds of the list σ , and the value e is stored at position i in the list σ), and the notation $\sigma[i := e]$ to overwrite the i th element of the list σ with value e .

The step relation $\sigma \Rightarrow \sigma'$ non-deterministically lets a thread take a step:

$$\frac{\sigma \Rightarrow_i \sigma' \quad i < \text{length } \sigma}{\sigma \Rightarrow \sigma'}$$

Our goal is to prove type safety, that is: If $\emptyset \vdash e : A$, then $\text{safe } [e]$. Safety is defined as follows:

$$\text{safe } \sigma \triangleq \forall \sigma', (\sigma \Rightarrow^* \sigma') \rightarrow \forall i < \text{length } \sigma'. (\sigma'(i) \in \text{Val}) \vee (\exists \sigma''. \sigma' \Rightarrow_i \sigma'')$$

Here, \Rightarrow^* is the reflexive-transitive closure of \Rightarrow .

(a) To prove type safety, we need to define a run-time typing judgment $\vdash_{\text{cfg}} \sigma : A$ for configurations. We then prove the following properties of the run-time typing judgment:

- Initialization: $\emptyset \vdash e : A$ implies $\vdash_{\text{cfg}} [e] : A$
- Preservation: $\vdash_{\text{cfg}} \sigma : A$ and $\sigma \Rightarrow \sigma'$ implies $\vdash_{\text{cfg}} \sigma' : A$
- Progress: $\vdash_{\text{cfg}} \sigma : A$ implies $\forall i < \text{length } \sigma. (\sigma(i) \in \text{Val}) \vee (\exists \sigma'. \sigma \Rightarrow_i \sigma')$

Explain how these three properties imply type safety.

- (b) To define the run-time typing judgment $\vdash_{\text{cfg}} \sigma : A$ for configurations, we first define a run-time typing judgment $\Sigma \mid \Gamma \vdash e : A$ for expressions. Here, Σ is a *thread typing* represented as a list containing the types of all threads in the configuration. Give all inference rules for the run-time typing judgment $\Sigma \mid \Gamma \vdash e : A$ required to prove type safety.
- (c) Give a definition of the run-time typing judgment $\vdash_{\text{cfg}} \sigma : A$ for configurations and briefly indicate why the three properties in Exercise a hold. You do not need to give a formal proof of these properties.
- (d) Our definition of type safety does not rule out configurations that are deadlocked *i.e.*, configurations consisting of threads that spin indefinitely long via `join` to wait on each other. Give an example of a configuration σ that is deadlocked, but for which $\text{safe } \sigma$ holds. Explain your answer.
- (e) Does the type system rule out deadlocks? If yes, give an intuition why. If not, give an example of a program that is well-typed, but that deadlocks.