Design Principles of Programming Languages

编程语言的设计原理

Haiyan Zhao, Di Wang

赵海燕，王迪

Peking University, Spring Term 2025

Substructural Types

亚结构类型

Motivation

Example (Abstract Type `file` and Its Interface)

```
type file

val open   : string → file option
val read   : file → string * file
val append : file * string → file
val write  : file * string → file
val close  : file → unit
```

The type abstraction encapsulates the **internal representation** of a file and its **invariants**.

Question

Can the type system prevent a file from being **read after it has been closed**, **closed twice**, or **forgotten to close**?

Structural Typing

Observation

The type systems we have seen so far in this course are all **structural**.

Principle (Structural Properties)

Recall that typing contexts can be defined by $\Gamma ::= \varnothing \mid \Gamma, x : T$.
Instead of treating Γ as a **function** from variables to types, we literally view Γ as a **list** $x_1 : T_1, \ldots, x_n : T_n$.

Exchange If $\Gamma_1, x_1 : T_1, x_2 : T_2, \Gamma_2 \vdash t : T$, then $\Gamma_1, x_2 : T_2, x_1 : T_1, \Gamma_2 \vdash t : T$.

Weakening If $\Gamma_1, \Gamma_2 \vdash t : T$, then $\Gamma_1, x_1 : T_1, \Gamma_2 \vdash t : T$.

Contraction If $\Gamma_1, x_2 : T_1, x_3 : T_1, \Gamma_2 \vdash t : T$, then $\Gamma_1, x_1 : T_1, \Gamma_2 \vdash [x_2 \mapsto x_1][x_3 \mapsto x_1]t : T$.

Remark

A structural type system enjoys **all three above properties**, i.e., typing contexts behave as functions.

Structural Typing

Principle (Structural Properties)

Exchange If $\Gamma_1, x_1 : T_1, x_2 : T_2, \Gamma_2 \vdash t : T$, then $\Gamma_1, x_2 : T_2, x_1 : T_1, \Gamma_2 \vdash t : T$.

Weakening If $\Gamma_1, \Gamma_2 \vdash t : T$, then $\Gamma_1, x_1 : T_1, \Gamma_2 \vdash t : T$.

Contraction If $\Gamma_1, x_2 : T_1, x_3 : T_1, \Gamma_2 \vdash t : T$, then $\Gamma_1, x_1 : T_1, \Gamma_2 \vdash [x_2 \mapsto x_1][x_3 \mapsto x_1]t : T$.

Observation

A structural type system **cannot** limit the **number** or **order** of uses of a data structure or operation.

Definition (Substructural Type Systems)

A **substructural type system** is a type system where one or more of the structural properties do **not** hold.

Substructural Typing

Definition (Substructural Type Systems)

A **substructural type system** is a type system where one or more of the structural properties do **not** hold.
That is, a substructural type system allows **fine-grained control of variable use**.

Type System	Intuition	Exchange	Weakening	Contraction
structural	no restriction on variable use	✓	✓	✓
affine	variables are used at most once	✓	✓	✗
relevant	variables are used at least once	✓	✗	✓
linear	variables are used exactly once	✓	✗	✗
ordered	variables are used exactly once and in their introduction order	✗	✗	✗

Question

Can you think of possible scenarios for other combintations, e.g., only weakening or only contraction?

Substructural Typing

Example (Abstract Type `file` and Its Interface)

```
val open   : string → file option
val read   : file → string * file
val append : file * string → file
val write  : file * string → file
val close  : file → unit
```

Question

What would be the consequences if we treat the type `file` as a **linear** type?

Principle

Substructural types are useful for **constraining resource use**, such as files, locks, and memory.

Linear Types

Remark

A **linear** type system allows **Exchange** but forbids **Weakening** and **Contraction**.

Syntax

$$t ::= x \mid \lambda x{:}T.\, t \mid t\ t \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\ t\ \mathtt{then}\ t\ \mathtt{else}\ t \mid \mathtt{0} \mid \mathtt{succ}\ t \mid \mathtt{iszero}\ t \qquad T ::= T \to T \mid \mathtt{Bool} \mid \mathtt{Nat}$$

$$v ::= \lambda x{:}T.\, t \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{0} \mid \mathtt{succ}\ v \qquad \Gamma ::= \varnothing \mid \Gamma, x : T$$

Question

How to formulate the typing relation to enforce **linearity**, i.e., “variables are used exactly once”?

Linear Types

Principle

The typing relation maintains the invariant that **variables are used exactly once along every control-flow path**.

$\Gamma \vdash^{\ell} t : T$: "term t has type T in context Γ with linear typing"

Let us implicitly assume **Exchange** by treating Γ as an unordered list.

$$\frac{}{x : T \vdash^{\ell} x : T}\ \text{T-Var} \qquad \frac{}{\varnothing \vdash^{\ell} \mathtt{true} : \mathtt{Bool}}\ \text{T-True} \qquad \frac{}{\varnothing \vdash^{\ell} \mathtt{false} : \mathtt{Bool}}\ \text{T-False}$$

$$\frac{}{\varnothing \vdash^{\ell} \mathtt{0} : \mathtt{Nat}}\ \text{T-Zero} \qquad \frac{\Gamma \vdash^{\ell} t_1 : \mathtt{Nat}}{\Gamma \vdash^{\ell} \mathtt{succ}\ t_1 : \mathtt{Nat}}\ \text{T-Succ} \qquad \frac{\Gamma \vdash^{\ell} t_1 : \mathtt{Nat}}{\Gamma \vdash^{\ell} \mathtt{iszero}\ t_1 : \mathtt{Bool}}\ \text{T-IsZero}$$

$$\frac{\Gamma_1 \vdash^{\ell} t_1 : \mathtt{Bool} \qquad \Gamma_2 \vdash^{\ell} t_2 : T \qquad \Gamma_2 \vdash^{\ell} t_3 : T}{\Gamma_1, \Gamma_2 \vdash^{\ell} \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : T}\ \text{T-If}$$

Linear Types

$\Gamma \vdash^{\ell} t : T$: "term t has type T in context Γ with linear typing"

$$\frac{\Gamma, x : T_1 \vdash^{\ell} t_2 : T_2}{\Gamma \vdash^{\ell} \lambda x{:}T_1 .\, t_2 : T_1 \to T_2}\ \text{T-Abs} \qquad \frac{\Gamma_1 \vdash^{\ell} t_1 : T_{11} \to T_{12} \qquad \Gamma_2 \vdash^{\ell} t_2 : T_{11}}{\Gamma_1, \Gamma_2 \vdash^{\ell} t_1\ t_2 : T_{12}}\ \text{T-App}$$

Example

$$\dfrac{\dfrac{\dfrac{}{f : \mathsf{Nat} \to \mathsf{Nat} \to T \vdash^{\ell} f : \mathsf{Nat} \to \mathsf{Nat} \to T}\ \text{T-Var} \qquad \dfrac{}{x : \mathsf{Nat} \vdash^{\ell} x : \mathsf{Nat}}\ \text{T-Var}}{f : \mathsf{Nat} \to \mathsf{Nat} \to T, x : \mathsf{Nat} \vdash^{\ell} f\ x : \mathsf{Nat} \to T}\ \text{T-App} \qquad \dfrac{}{y : \mathsf{Nat} \vdash^{\ell} y : \mathsf{Nat}}\ \text{T-Var}}{f : \mathsf{Nat} \to \mathsf{Nat} \to T, x : \mathsf{Nat}, y : \mathsf{Nat} \vdash^{\ell} (f\ x)\ y : T}\ \text{T-App}$$

Question

Let Γ be $f : \mathsf{Nat} \to \mathsf{Nat} \to T, x : \mathsf{Nat}, y : \mathsf{Nat}$. Can we derive $\Gamma \vdash^{\ell} (f\ x)\ \mathtt{0} : T$, $\Gamma \vdash^{\ell} (f\ x)\ x : T$?

Operational Semantics

Question

How to justify if the typing relation maintains **linearity**?

Remark

We need an operational semantics that can **keep track of how variables are used**.
Recall that for references, we introduced **stores** to keep track of how locations are used.

$$\frac{l \notin dom(\mu)}{\texttt{ref}\ v \mid \mu \longrightarrow l \mid (\mu, l \mapsto v)}\ \text{E-RefV} \qquad \frac{\mu(l) = v}{!l \mid \mu \longrightarrow v \mid \mu}\ \text{E-DerefLoc}$$

Fine-Grained Operational Semantics

Let us introduce **value stores** by $V ::= \varnothing \mid V, x \mapsto v$ and evaluation relations as $t \mid V \longrightarrow t' \mid V'$.

$$\frac{V(x) = v}{x \mid V \longrightarrow v \mid (V \setminus x)}\ \text{E-Var} \qquad \frac{y \notin dom(V)}{(\lambda x{:}T_{11}.\, t_{12})\ v_2 \mid V \longrightarrow [x \mapsto y]t_{12} \mid (V, y \mapsto v_2)}\ \text{E-AppAbs}$$

Operational Semantics

$t \mid V \longrightarrow t' \mid V'$: "term t with value store V one-step evaluates to term t' with value store V'"

$$\frac{t_1 \mid V \longrightarrow t_1' \mid V'}{t_1\ t_2 \mid V \longrightarrow t_1'\ t_2 \mid V'}\ \text{E-App1} \qquad \frac{t_2 \mid V \longrightarrow t_2' \mid V'}{v_1\ t_2 \mid V \longrightarrow v_1\ t_2' \mid V'}\ \text{E-App2}$$

$$\frac{t_1 \mid V \longrightarrow t_1' \mid V'}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \mid V \longrightarrow \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \mid V'}\ \text{E-If}$$

$$\frac{}{\texttt{if}\ \texttt{true}\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \mid V \longrightarrow t_2 \mid V}\ \text{E-IfTrue} \qquad \frac{}{\texttt{if}\ \texttt{false}\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \mid V \longrightarrow t_3 \mid V}\ \text{E-IfFalse}$$

$$\frac{t_1 \mid V \longrightarrow t_1' \mid V'}{\texttt{succ}\ t_1 \mid V \longrightarrow \texttt{succ}\ t_1' \mid V'}\ \text{E-Succ} \qquad \frac{t_1 \mid V \longrightarrow t_1' \mid V'}{\texttt{iszero}\ t_1 \mid V \longrightarrow \texttt{iszero}\ t_1' \mid V'}\ \text{E-Iszero}$$

$$\frac{}{\texttt{iszero}\ \texttt{0} \mid V \longrightarrow \texttt{true} \mid V}\ \text{E-IszeroZero} \qquad \frac{}{\texttt{iszero}\ \texttt{succ}\ v_1 \mid V \longrightarrow \texttt{false} \mid V}\ \text{E-IszeroSucc}$$

Operational Semantics

Example

$$
\begin{aligned}
(\lambda f.\, f\, 5)\, (\underline{(\lambda x.\, \lambda y.\, x + y)\, 2}) \mid \emptyset &\longrightarrow \underline{(\lambda f.\, f\, 5)\, (\lambda y.\, z_1 + y)} \mid (z_1 \mapsto 2) \\
&\longrightarrow \underline{z_2}\, 5 \mid (z_1 \mapsto 2, z_2 \mapsto \lambda y.\, z_1 + y) \\
&\longrightarrow \underline{(\lambda y.\, z_1 + y)\, 5} \mid (z_1 \mapsto 2) \\
&\longrightarrow \underline{z_1} + z_3 \mid (z_1 \mapsto 2, z_3 \mapsto 5) \\
&\longrightarrow 2 + \underline{z_3} \mid (z_3 \mapsto 5) \\
&\longrightarrow \underline{2 + 5} \mid \varnothing \longrightarrow 7 \mid \varnothing
\end{aligned}
$$

Example

$$
\begin{aligned}
\underline{(\lambda x.\, x + x)\, 2} \mid \varnothing &\longrightarrow \underline{z_1} + z_1 \mid (z_1 \mapsto 2) \\
&\longrightarrow 2 + z_1 \mid \varnothing \not\longrightarrow
\end{aligned}
$$

Soundness

$V : \Gamma$: "value store V confroms with typing context Γ"

$$\frac{}{\varnothing : \varnothing}\ \text{T-Empty} \qquad \frac{V : (\Gamma_1, \Gamma_2) \qquad \Gamma_1 \vdash^{\ell} v : T}{(V, x \mapsto v) : (\Gamma_2, x : T)}\ \text{T-Extend}$$

For example, we can have $(z_1 \mapsto 2, z_2 \mapsto \lambda y.\, z_1 + y) : (z_2 : \mathsf{Nat} \to \mathsf{Nat})$.

Theorem (Soundness)

Progress If $V : \Gamma$ and $\Gamma \vdash^{\ell} t : T$, then $t \mid V \longrightarrow t' \mid V'$ or t is a value.

Preservation If $V : \Gamma$, $\Gamma \vdash^{\ell} t : T$, and $t \mid V \longrightarrow t' \mid V'$, then $V' : \Gamma'$ and $\Gamma' \vdash^{\ell} t' : T$.

Corollary (Linearity)

If $\varnothing \vdash^{\ell} t : T$, $T = \mathsf{Bool}$ or $T = \mathsf{Nat}$, and $t \mid \varnothing \longrightarrow^{*} v \mid V'$, then $V' = \varnothing$.

Proof of Preservation (Sketch)

Proof: Induction on the Derivation of $t \mid V \longrightarrow t' \mid V'$

E-Var Have $x \mid V \longrightarrow V(x) \mid (V \setminus x), x : T \vdash^{\ell} x : T$, and $V : (x : T)$.
Inversion on $V : (x : T)$. Have $(V \setminus x) : \Gamma_1$ and $\Gamma_1 \vdash^{\ell} V(x) : T$.
Conclude by setting $\Gamma' \stackrel{\text{def}}{=} \Gamma_1$.

E-AppAbs Have $(\lambda x{:}T_{11}.\, t_{12})\, v_2 \mid V \longrightarrow [x \mapsto y]t_{12} \mid (V, y \mapsto v_2), \Gamma \vdash^{\ell} (\lambda x{:}T_{11}.\, t_{12})\, v_2 : T_{12}$, and $V : \Gamma$.
Inversion on $\Gamma \vdash^{\ell} (\lambda x{:}T_{11}.\, t_{12})\, v_2 : T_{12}$. Have $\Gamma = \Gamma_1, \Gamma_2$ such that
$\Gamma_1 \vdash^{\ell} \lambda x{:}T_{11}.\, t_{12} : T_{11} \to T_{12}$ and $\Gamma_2 \vdash^{\ell} v_2 : T_{11}$.
Inversion on $\Gamma_1 \vdash^{\ell} \lambda x{:}T_{11}.\, t_{12} : T_{11} \to T_{12}$. Have $\Gamma_1, x : T_{11} \vdash^{\ell} t_{12} : T_{12}$.
Apply **substitution**. Have $\Gamma_1, y : T_{11} \vdash^{\ell} [x \mapsto y]t_{12} : T_{12}$.
Apply **T-Extend** to $V : (\Gamma_1, \Gamma_2)$. Have $(V, y \mapsto v_2) : (\Gamma_1, y : T_{11})$.
Conclude by setting $\Gamma' \stackrel{\text{def}}{=} (\Gamma_1, y : T_{11})$.

Lemma (Substitution)

If $\Gamma_1, x : T \vdash^{\ell} t_1 : T_1$ and $\Gamma_2 \vdash^{\ell} y : T$, then $\Gamma_1, \Gamma_2 \vdash^{\ell} [x \mapsto y]t_1 : T_1$.

Algorithmic Typing

$\Gamma \vdash^{\ell} t : T$: "term t has type T in context Γ with linear typing"

$$\frac{\Gamma, x : T_1 \vdash^{\ell} t_2 : T_2}{\Gamma \vdash^{\ell} \lambda x{:}T_1 .\, t_2 : T_1 \to T_2} \text{ T-Abs} \qquad \frac{\Gamma_1 \vdash^{\ell} t_1 : T_{11} \to T_{12} \qquad \Gamma_2 \vdash^{\ell} t_2 : T_{11}}{\Gamma_1, \Gamma_2 \vdash^{\ell} t_1\ t_2 : T_{12}} \text{ T-App}$$

Observation

Although all rules are syntax-directed, some of them (e.g., T-App) require "guessing" how to **split** a context.

Principle

Rather than trying to split the context, the algorithmic typing can pass the entire context as an input and return the unused portion as the **remainder context**, i.e., the algorithmic typing relation takes the form $\Gamma_{in} \vdash^{\ell} t : T; \Gamma_{out}$.

Algorithmic Typing

$\Gamma_{in} \vdash^{\ell} t : T; \Gamma_{out}$: "given context Γ_{in}, term t has type T with remainder context Γ_{out}"

$$\frac{}{\Gamma, x : T \vdash^{\ell} x : T; \Gamma}\ \text{A-Var} \qquad \frac{}{\Gamma \vdash^{\ell} \mathtt{0} : \mathsf{Nat}; \Gamma}\ \text{A-Zero} \qquad \frac{\Gamma_1 \vdash^{\ell} t_1 : \mathsf{Nat}; \Gamma_2}{\Gamma_1 \vdash^{\ell} \mathtt{succ}\ t_1 : \mathsf{Nat}; \Gamma_2}\ \text{A-Succ}$$

$$\frac{\Gamma_1 \vdash^{\ell} t_1 : \mathsf{Bool}; \Gamma_2 \qquad \Gamma_2 \vdash^{\ell} t_2 : T; \Gamma_3 \qquad \Gamma_2 \vdash^{\ell} t_3 : T; \Gamma_3}{\Gamma_1 \vdash^{\ell} \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : T; \Gamma_3}\ \text{A-If}$$

$$\frac{\Gamma_1, x : T_1 \vdash^{\ell} t_2 : T_2; \Gamma_2 \qquad x \notin dom(\Gamma_2)}{\Gamma_1 \vdash^{\ell} \lambda x{:}T_1.\, t_2 : T_1 \to T_2; \Gamma_2}\ \text{A-Abs} \qquad \frac{\Gamma_1 \vdash^{\ell} t_1 : T_{11} \to T_{12}; \Gamma_2 \qquad \Gamma_2 \vdash^{\ell} t_2 : T_{11}; \Gamma_3}{\Gamma_1 \vdash^{\ell} t_1\ t_2 : T_{12}; \Gamma_3}\ \text{A-App}$$

Theorem (Correctness)

Soundness If $\Gamma \vdash^{\ell} t : T; \varnothing$, then $\Gamma \vdash^{\ell} t : T$.

Completeness If $\Gamma \vdash^{\ell} t : T$, then $\Gamma \vdash^{\ell} t : T; \varnothing$.

Affine Types

Remark

An **affine** type system allows **Exchange** and **Weakening** but forbids **Contraction**.
That is, it allows variables to be used **at most once**.

$\Gamma \vdash^a t : T$: "term t has type T in context Γ with affine typing"

(all rules for linear types)

$$\frac{\Gamma \vdash^a t : T}{\Gamma, x_1 : T_1 \vdash^a t : T}\ \text{T-Weak}$$

Example

$$\dfrac{\dfrac{\dfrac{}{f : \mathsf{Nat} \to \mathsf{Nat} \to T \vdash^a f : \mathsf{Nat} \to \mathsf{Nat} \to T}\ \text{T-Var} \quad \dfrac{}{x : \mathsf{Nat} \vdash^a x : \mathsf{Nat}}\ \text{T-Var}}{f : \mathsf{Nat} \to \mathsf{Nat} \to T, x : \mathsf{Nat} \vdash^a f\,x : \mathsf{Nat} \to T}\ \text{T-App} \quad \dfrac{\dfrac{}{\varnothing \vdash^a 0 : \mathsf{Nat}}\ \text{T-Zero}}{y : \mathsf{Nat} \vdash^a 0 : \mathsf{Nat}}\ \text{T-Weak}}{f : \mathsf{Nat} \to \mathsf{Nat} \to T, x : \mathsf{Nat}, y : \mathsf{Nat} \vdash^a (f\,x)\,0 : T}\ \text{T-App}$$

Syntax-Directed Typing

Question

The structural rule (T-Weak) is **not** syntax-directed, which may complicate soundness proof.

Observation

One can always "push" uses of the weakening rule down to the **leaves** of the derivation tree.
Thus, we can update the **axiom-like** rules to allow immediate weakening.

Syntax-Directed Affine Typing

$$\frac{}{\Gamma, x : T \vdash^{a} x : T}\ \text{T-Var} \qquad \frac{}{\Gamma \vdash^{a} \mathtt{true} : \mathtt{Bool}}\ \text{T-True} \qquad \frac{}{\Gamma \vdash^{a} \mathtt{false} : \mathtt{Bool}}\ \text{T-False} \qquad \frac{}{\Gamma \vdash^{a} \mathtt{0} : \mathtt{Nat}}\ \text{T-Zero}$$

Relevant Types

Remark

A relevant type system allows **Exchange** and **Contraction** but forbids **Weakening**.
That is, it allows variables to be used **at least once**.

$\Gamma \vdash^r t : T$: "term t has type T in context Γ with relevant typing"

(all rules for linear types)

$$\frac{\Gamma, x_2 : T_1, x_3 : T_1 \vdash^r t : T}{\Gamma, x_1 : T_1 \vdash^r [x_2 \mapsto x_1][x_3 \mapsto x_1]t : T}\ \text{T-Contract}$$

Example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : \mathsf{Nat} \to \mathsf{Nat} \to T \vdash^r f : \mathsf{Nat} \to \mathsf{Nat} \to T}\ \text{T-Var} \qquad \dfrac{}{y : \mathsf{Nat} \vdash^r y : \mathsf{Nat}}\ \text{T-Var}}{f : \mathsf{Nat} \to \mathsf{Nat} \to T, y : \mathsf{Nat} \vdash^r f\ y : \mathsf{Nat} \to T}\ \text{T-App} \qquad \dfrac{}{z : \mathsf{Nat} \vdash^r z : \mathsf{Nat}}\ \text{T-Var}}{f : \mathsf{Nat} \to \mathsf{Nat} \to T, y : \mathsf{Nat}, z : \mathsf{Nat} \vdash^r (f\ y)\ z : T}\ \text{T-App}}{f : \mathsf{Nat} \to \mathsf{Nat} \to T, x : \mathsf{Nat} \vdash^r (f\ x)\ x : T}\ \text{T-Contract}$$

Syntax-Directed Typing

Observation

The structural rule (T-Contract) is **not** syntax-directed, which may complicate soundness proof.

Observation

We can avoid the explicit substitution by "duplicate" variables in the **cases that need to split the context**.
That is, when splitting Γ into Γ_1, Γ_2, we allow them to **overlap**.

$\Gamma \curlyvee (\Gamma_1 ; \Gamma_2)$: "context Γ can be split into Γ_1 and Γ_2 with overlapping allowed"

$$\frac{}{\varnothing \curlyvee (\varnothing ; \varnothing)} \qquad \frac{\Gamma \curlyvee (\Gamma_1 ; \Gamma_2)}{\Gamma, x : T \curlyvee (\Gamma_1, x : T ; \Gamma_2)} \qquad \frac{\Gamma \curlyvee (\Gamma_1 ; \Gamma_2)}{\Gamma, x : T \curlyvee (\Gamma_1 ; \Gamma_2, x : T)} \qquad \frac{\Gamma \curlyvee (\Gamma_1 ; \Gamma_2)}{\Gamma, x : T \curlyvee (\Gamma_1, x : T ; \Gamma_2, x : T)}$$

Syntax-Directed Typing

Syntax-Directed Relevant Typing

$$\frac{\Gamma \curlyvee (\Gamma_1 ; \Gamma_2) \qquad \Gamma_1 \vdash^r t_1 : \texttt{Bool} \qquad \Gamma_2 \vdash^r t_2 : T \qquad \Gamma_2 \vdash^r t_3 : T}{\Gamma \vdash^r \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T}\ \text{T-If}$$

$$\frac{\Gamma \curlyvee (\Gamma_1 ; \Gamma_2) \qquad \Gamma_1 \vdash^r t_1 : T_{11} \to T_{12} \qquad \Gamma_2 \vdash^r t_2 : T_{11}}{\Gamma \vdash^r t_1\ t_2 : T_{12}}\ \text{T-App}$$

Example

Let $\Gamma \overset{\text{def}}{=} (f : \mathsf{Nat} \to \mathsf{Nat} \to T, x : \mathsf{Nat}]$, $\Gamma_1 \overset{\text{def}}{=} (f : \mathsf{Nat} \to \mathsf{Nat} \to T, x : \mathsf{Nat})$, $\Gamma_{11} \overset{\text{def}}{=} (f : \mathsf{Nat} \to \mathsf{Nat} \to T)$, $\Gamma_{12} \overset{\text{def}}{=} (x : \mathsf{Nat})$, and $\Gamma_2 \overset{\text{def}}{=} (x : \mathsf{Nat})$.

$$\dfrac{\dfrac{\cdots}{\Gamma \curlyvee (\Gamma_1 ; \Gamma_2)} \qquad \dfrac{\dfrac{\cdots}{\Gamma_1 \curlyvee (\Gamma_{11} ; \Gamma_{12})} \qquad \dfrac{}{\Gamma_{11} \vdash^r f : \mathsf{Nat} \to \mathsf{Nat} \to T} \qquad \dfrac{}{\Gamma_{12} \vdash^r x : \mathsf{Nat}}}{\Gamma_1 \vdash^r f\ x : \mathsf{Nat} \to T} \qquad \dfrac{}{\Gamma_2 \vdash^r x : \mathsf{Nat}}}{\Gamma \vdash^r (f\ x)\ x : T}$$

Syntax-Directed Typing

Another Approach for Syntax-Directed Relevant Typing

We can also augment the language with a new term to **explicitly "duplicate" a program variable**.

$$t ::= \ldots \mid \mathtt{share}\ t_1\ \mathtt{as}\ x_1, x_2\ \mathtt{in}\ t_2$$

$$\frac{\Gamma_1 \vdash^r t_1 : T_1 \qquad \Gamma_2, x_1 : T_1, x_2 : T_1 \vdash^r t_2 : T_2}{\Gamma_1, \Gamma_2 \vdash^r \mathtt{share}\ t_1\ \mathtt{as}\ x_1, x_2\ \mathtt{in}\ t_2 : T_2}\ \text{T-Share}$$

$$\frac{y_1, y_2 \notin dom(V)}{\mathtt{share}\ v_1\ \mathtt{as}\ x_1, x_2\ \mathtt{in}\ t_2 \mid V \longrightarrow [x_1 \mapsto y_1][x_2 \mapsto y_2]t_2 \mid (V, y_1 \mapsto v_1, y_2 \mapsto v_1)}\ \text{E-ShareV}$$

Example

$$\dfrac{\dfrac{}{x : \mathsf{Nat} \vdash^r x : \mathsf{Nat}}\ \text{T-Var} \qquad \dfrac{\dfrac{}{y : \mathsf{Nat} \vdash^r y : \mathsf{Nat}}\ \text{T-Var} \qquad \dfrac{}{z : \mathsf{Nat} \vdash^r z : \mathsf{Nat}}\ \text{T-Var}}{y : \mathsf{Nat}, z : \mathsf{Nat} \vdash^r y + z : \mathsf{Nat}}}{x : \mathsf{Nat} \vdash^r \mathtt{share}\ x\ \mathtt{as}\ y, z\ \mathtt{in}\ y + z : \mathsf{Nat}}\ \text{T-Share}$$

LFPL: A Linearly-Typed Functional Programming Language

Let us augment the linearly-typed lambda calculus with useful extensions such as pairs and lists.

Syntax

$$\begin{aligned} t ::= {} & x \mid \lambda x{:}T_1 . t_2 \mid t_1\ t_2 \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 \mid \{t_1, t_2\} \mid \mathtt{let}\ \{x_1, x_2\} = t_1\ \mathtt{in}\ t_2 \\ \mid {} & \mathtt{nil}[T_1] \mid \mathtt{cons}[T_1]\ t_1\ t_2 \mid \mathtt{iter}_L\ t_1\ \mathtt{with}\ \mathtt{nil} \Rightarrow t_2 \mid \mathtt{cons}(x, \_)\ \mathtt{of}\ y \Rightarrow t_3 \end{aligned}$$

$$T ::= T_1 \to T_2 \mid \mathtt{Bool} \mid T_1 \times T_2 \mid \mathtt{List}\ T_1$$

Linear Typing for Pairs

$$\frac{\Gamma_1 \vdash t_1 : T_1 \qquad \Gamma_2 \vdash t_2 : T_2}{\Gamma_1, \Gamma_2 \vdash \{t_1, t_2\} : T_1 \times T_2}\ \text{T-Pair}$$

$$\frac{\Gamma_1 \vdash t_1 : T_{11} \times T_{12} \qquad \Gamma_2, x_1 : T_{11}, x_2 : T_{12} \vdash t_2 : T_2}{\Gamma_1, \Gamma_2 \vdash \mathtt{let}\ \{x_1, x_2\} = t_1\ \mathtt{in}\ t_2 : T_2}\ \text{T-LetP}$$

Question

Instead of `letp`, can we use `fst` and `snd` as elimination forms?

LFPL: A Linearly-Typed Functional Programming Language

Linear Typing for Lists

$$\frac{}{\varnothing \vdash \mathtt{nil}[T_1] : \mathtt{List}\ T_1}\ \text{T-Nil} \qquad \frac{\Gamma_1 \vdash t_1 : T_1 \qquad \Gamma_2 \vdash t_2 : \mathtt{List}\ T_1}{\Gamma_1, \Gamma_2 \vdash \mathtt{cons}[T_1]\ t_1\ t_2 : \mathtt{List}\ T_1}\ \text{T-Cons}$$

$$\frac{\Gamma_1 \vdash t_1 : \mathtt{List}\ T_{11} \qquad \Gamma_2 \vdash t_2 : S \qquad x : T_{11}, y : S \vdash t_3 : S}{\Gamma_1, \Gamma_2 \vdash \mathtt{iter}_L\ t_1\ \mathtt{with}\ \mathtt{nil} \Rightarrow t_2 \mid \mathtt{cons}(x, \_)\ \mathtt{of}\ y \Rightarrow t_3 : S}\ \text{T-IterL}$$

Question

Why does the third premise of (T-IterL) allow **only x and y** in the context?

Observation

Recall that substructural typing constrains **resource use**.

The third premise stands for the **iteration** case, which can be executed for **multiple times**.

LFPL: A Linearly-Typed Functional Programming Language

Example

$$
\begin{aligned}
&\mathit{append} : \mathsf{List}\ \mathsf{T} \to \mathsf{List}\ \mathsf{T} \to \mathsf{List}\ \mathsf{T} \\
&\mathit{append} \equiv \lambda l_1 : \mathsf{List}\ \mathsf{T}.\ \lambda l_2 : \mathsf{List}\ \mathsf{T}. \\
&\qquad\qquad \mathtt{iter}_{\mathsf{L}}\ l_1\ \mathtt{with}\ \mathtt{nil} \Rightarrow l_2 \\
&\qquad\qquad\qquad\qquad \mid \mathtt{cons}(x, \_)\ \mathtt{of}\ y \Rightarrow \mathtt{cons}[\mathsf{T}]\ x\ y
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reverse} : \mathsf{List}\ \mathsf{T} \to \mathsf{List}\ \mathsf{T} \\
&\mathit{reverse} \equiv \lambda l : \mathsf{List}\ \mathsf{T}. \\
&\qquad (\mathtt{iter}_{\mathsf{L}}\ l\ \mathtt{with}\ \mathtt{nil} \Rightarrow (\lambda a{:}\mathsf{List}\ \mathsf{T}.\ a) \\
&\qquad\qquad\qquad \mid \mathtt{cons}(x, \_)\ \mathtt{of}\ y \Rightarrow (\lambda a{:}\mathsf{List}\ \mathsf{T}.\ y\ (\mathtt{cons}[\mathsf{T}]\ x\ a))) \\
&\qquad \mathtt{nil}[\mathsf{T}]
\end{aligned}
$$

LFPL: A Linearly-Typed Functional Programming Language

Structural Rules

It is reasonable to allow **Weakening**; as well as **Contraction** for "copyable" types.

$$\frac{\Gamma \vdash t : T}{\Gamma, x_1 : T_1 \vdash t : T} \text{ T-Weak} \qquad \frac{\Gamma, x_2 : T_1, x_3 : T_1 \vdash t : T \qquad T_1 \text{ copyable}}{\Gamma, x_1 : T_1 \vdash [x_2 \mapsto x_1][x_3 \mapsto x_1]t : T} \text{ T-Contract}$$

$$\frac{}{\texttt{Bool} \text{ copyable}} \qquad \frac{T_1 \text{ copyable} \qquad T_2 \text{ copyable}}{(T_1 \times T_2) \text{ copyable}}$$

Example

$$\begin{aligned}
&\mathit{append2} : \texttt{List Bool} \to \texttt{List Bool} \to \texttt{List Bool} \\
&\mathit{append2} \equiv \lambda l_1 : \texttt{List Bool}.\ \lambda l_2 : \texttt{List Bool}. \\
&\qquad \texttt{iter}_L\ l_1\ \texttt{with nil} \Rightarrow l_2 \\
&\qquad\qquad \mid \texttt{cons}(x, _)\ \texttt{of}\ y \Rightarrow \texttt{cons[Bool]}\ x\ (\texttt{cons[Bool]}\ x\ y)
\end{aligned}$$

Non-Size-Increasing Functions

Question

Let us become more **expicit** about "resource," such as memory.
In LFPL, the only data structure that consumes memory is $\mathtt{cons}[T_1]\ t_1\ t_2$.
Can we extend the type system to **explicitly** account for such memory consumption?

The Diamond Type

Let us introduce a type $\Diamond$ (called **diamond**), whose inhabitants are **memory cells** that hold $\mathtt{cons}$-constructors.

$$t ::= \ldots \mid \mathtt{nil}[T] \mid \mathtt{cons}[T_1]\ t_d\ t_1\ t_2 \mid \mathtt{iter_L}\ t_1\ \mathtt{with\ nil} \Rightarrow t_2 \mid \mathtt{cons}(x_d, x, \_)\ \mathtt{of}\ y \Rightarrow t_3$$

$$\frac{\Gamma_d \vdash t_d : \Diamond \qquad \Gamma_1 \vdash t_1 : T_1 \qquad \Gamma_2 \vdash t_2 : \mathtt{List}\ T_1}{\Gamma_d, \Gamma_1, \Gamma_2 \vdash \mathtt{cons}[T_1]\ t_d\ t_1\ t_2 : \mathtt{List}\ T_1}\ \text{T-Cons}$$

$$\frac{\Gamma_1 \vdash t_1 : \mathtt{List}\ T_{11} \qquad \Gamma_2 \vdash t_2 : S \qquad x_d : \Diamond, x : T_{11}, y : S \vdash t_3 : S}{\Gamma_1, \Gamma_2 \vdash \mathtt{iter_L}\ t_1\ \mathtt{with\ nil} \Rightarrow t_2 \mid \mathtt{cons}(x_d, x, \_)\ \mathtt{of}\ y \Rightarrow t_3 : S}\ \text{T-IterL}$$

Non-Size-Increasing Functions

Example

$$
\begin{aligned}
&\textit{append} : \mathtt{List}\ \mathsf{T} \to \mathtt{List}\ \mathsf{T} \to \mathtt{List}\ \mathsf{T} \\
&\textit{append} \equiv \lambda l_1 : \mathtt{List}\ \mathsf{T}.\ \lambda l_2 : \mathtt{List}\ \mathsf{T}. \\
&\qquad \mathtt{iter}_{\mathsf{L}}\ l_1\ \mathtt{with}\ \mathtt{nil} \Rightarrow l_2 \\
&\qquad\qquad \mid \mathtt{cons}({\color{red}x_d}, x, \_)\ \mathtt{of}\ y \Rightarrow \mathtt{cons}[\mathsf{T}]\ {\color{red}x_d}\ x\ y
\end{aligned}
$$

$$
\begin{aligned}
&\textit{failed-append2} : \mathtt{List}\ \mathtt{Bool} \to \mathtt{List}\ \mathtt{Bool} \to \mathtt{List}\ \mathtt{Bool} \\
&\textit{failed-append2} \equiv \lambda l_1 : \mathtt{List}\ \mathtt{Bool}.\ \lambda l_2 : \mathtt{List}\ \mathtt{Bool}. \\
&\qquad \mathtt{iter}_{\mathsf{L}}\ l_1\ \mathtt{with}\ \mathtt{nil} \Rightarrow l_2 \\
&\qquad\qquad \mid \mathtt{cons}({\color{red}x_d}, x, \_)\ \mathtt{of}\ y \Rightarrow \mathtt{cons}[\mathtt{Bool}]\ {\color{red}x_d}\ x\ (\mathtt{cons}[\mathtt{Bool}]\ {\color{red}x_d}\ x\ y)
\end{aligned}
$$

Question

The diamond type $\Diamond$ is **not** copyable. Any workaround?

Non-Size-Increasing Functions

Example

$$
\begin{aligned}
&\mathit{append2} : \mathtt{List}\ (\mathtt{Bool} \times \Diamond) \to \mathtt{List\ Bool} \to \mathtt{List\ Bool} \\
&\mathit{append2} \equiv \lambda l_1 : \mathtt{List}\ (\mathtt{Bool} \times \Diamond).\ \lambda l_2 : \mathtt{List\ Bool}. \\
&\qquad\qquad \mathtt{iter}_{\mathtt{L}}\ l_1\ \mathtt{with\ nil} \Rightarrow l_2 \\
&\qquad\qquad\qquad\qquad \mid \mathtt{cons}(x_{d,1}, x, \_)\ \mathtt{of}\ y \Rightarrow \\
&\qquad\qquad\qquad\qquad\quad \mathtt{letp}\ \{z, x_{d,2}\} = x\ \mathtt{in} \\
&\qquad\qquad\qquad\qquad\quad \mathtt{cons}[\mathtt{Bool}]\ x_{d,1}\ z\ (\mathtt{cons}[\mathtt{Bool}]\ x_{d,2}\ z\ y)
\end{aligned}
$$

Question

Implement a function triple : $\mathtt{List\ Bool} \to \mathtt{List\ Bool}$ that essentially concatenates three copies of its input. You will need to update the type to insert $\Diamond$ accordingly.

Non-Size-Increasing Functions

Observation

LFPL with the diamond type $\Diamond$ captures **non-size-increasing** computation, where an instance of $\Diamond$ has **size one**.
For *append* : $\texttt{List}\ T \to \texttt{List}\ T \to \texttt{List}\ T$, we have $|\mathit{append}\ l_1\ l_2| \leqslant |l_1| + |l_2|$.
For *append2* : $\texttt{List}\ (\texttt{Bool} \times \Diamond) \to \texttt{List Bool} \to \texttt{List Bool}$, we also have $|\mathit{append2}\ l_1\ l_2| \leqslant |l_1| + |l_2|$.

Definition

Let us define an **interpretation** $[\![T]\!]$ of types and then the **size function** $s_T(v)$.
Let us introduce a distinguished value $\blacklozenge$ fro the diamond type $\Diamond$.

$$
\begin{aligned}
[\![\texttt{Bool}]\!] &= \{\texttt{true}, \texttt{false}\} & s_{\texttt{Bool}}(v) &= 0 \\
[\![\Diamond]\!] &= \{\blacklozenge\} & s_{\Diamond}(\blacklozenge) &= 1 \\
[\![T_1 \times T_2]\!] &= [\![T_1]\!] \times [\![T_2]\!] & s_{T_1 \times T_2}(\langle v_1, v_2 \rangle) &= s_{T_1}(v_1) + s_{T_2}(v_2) \\
[\![\texttt{List}\ T_1]\!] &= \{[v_1, \dots, v_n] \mid v_i \in [\![T_1]\!]\} & s_{\texttt{List}\ T_1}([v_1, \dots, v_n]) &= n + \textstyle\sum_{i=1}^{n} s_{T_1}(v_i) \\
[\![T_1 \to T_2]\!] &= [\![T_1]\!] \to [\![T_2]\!] & s_{T_1 \to T_2}(f) &= \min\{c \mid \forall v \in [\![T_1]\!].\ s_{T_2}(f(v)) \leqslant c + s_{T_1}(v)\}
\end{aligned}
$$

A function $f \in [\![T_1 \to T_2]\!]$ is said to be **non-size-increasing** if $s_{T_1 \to T_2}(f) = 0$.

Non-Size-Increasing Functions

Theorem

If $\varnothing \vdash v : T$, then $s_T(\llbracket v \rrbracket) = 0$, where $\llbracket v \rrbracket$ encodes a denotational interpretation of v.

We can define binary numbers using lists of Booleans:

$$\widehat{0} = \mathtt{nil}[\mathsf{Bool}] \qquad \widehat{2n+1} = \mathtt{cons}[\mathsf{Bool}] \blacklozenge \mathtt{false}\ \widehat{n} \qquad 2\widehat{(n+1)} = \mathtt{cons}[\mathsf{Bool}] \blacklozenge \mathtt{true}\ \widehat{n}$$

Theorem

We say that a function $h : \mathbb{N}^k \to \mathbb{N}$ is **definable** in LFPL if there exists a term $t : \mathtt{List\ Bool} \to \ldots \to \mathtt{List\ Bool}$ such that $t\ \widehat{n_1}\ \ldots\ \widehat{n_k} \longrightarrow^* \widehat{h(n_1, \ldots, n_k)}$ for all $n_1, \ldots, n_k$.
A function $h : \mathbb{N}^k \to \mathbb{N}$ is definable in LFPL **if and only if** h is in FP and $|h(n_1, \ldots, n_k)| \leqslant \sum_{i=1}^{k} |n_i|$.

Remark

Ref: M. Hofmann. 1999. Linear types and non-size-increasing polynomial time computation. In *Logic in Computer Science* (LICS'99), 464–473. doi: 10.1109/LICS.1999.782641.

Other Extensions: References

Question

Suppose that we store a linear resource in a reference. Can we perform arbitrary **dereferences** or **assignments**?

Syntax and Typing

$$t ::= \ldots \mid \mathtt{ref}\ t_1 \mid !t_1 \mid \mathtt{swap}\ t_1\ t_2 \qquad T ::= \ldots \mid \mathsf{Ref}\ T_1$$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \mathtt{ref}\ t_1 : \mathsf{Ref}\ T_1}\ \text{T-Ref} \qquad \frac{\Gamma \vdash t_1 : \mathsf{Ref}\ T_{11}}{\Gamma \vdash !t_1 : T_{11}}\ \text{T-Deref} \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{Ref}\ T_{11} \quad \Gamma_2 \vdash t_2 : T_{11}}{\Gamma_1, \Gamma_2 \vdash \mathtt{swap}\ t_1\ t_2 : \mathsf{Ref}\ T_{11} \times T_{11}}\ \text{T-Swap}$$

Operational Semantics

Let us allow value stores V to support both **variables** and **locations**.

$$\frac{\ell \notin dom(V)}{\mathtt{ref}\ v \mid V \longrightarrow \ell \mid (V, \ell \mapsto v)} \qquad \frac{V(\ell) = v}{!\ell \mid V \longrightarrow v \mid (V \setminus \ell)} \qquad \frac{V(\ell) = v_1}{\mathtt{swap}\ \ell\ v_2 \mid V \longrightarrow \{\ell, v_1\} \mid (V \setminus \ell, \ell \mapsto v_2)}$$

Other Extensions: Arrays

Question

Can you follow the approach for supporting references to arrays?

Syntax and Typing

$$\mathsf{t} ::= \ldots \mid \texttt{array}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \mid \texttt{free}\ \mathsf{t}_1\ \texttt{with}\ x.\ \mathsf{t}_2 \mid \texttt{swap}\ \mathsf{t}_1 [\mathsf{t}_2]\ \mathsf{t}_3 \qquad \mathsf{T} ::= \ldots \mid \texttt{Array}\ \mathsf{T}_1$$

$$\frac{\Gamma_i \vdash \mathsf{t}_i : \mathsf{T}_1 \quad (\text{for } 1 \leqslant i \leqslant n)}{\Gamma_1, \ldots, \Gamma_n \vdash \texttt{array}(\mathsf{t}_1, \ldots, \mathsf{t}_n) : \texttt{Array}\ \mathsf{T}_1}\ \text{T-Array} \qquad \frac{\Gamma \vdash \mathsf{t}_1 : \texttt{Array}\ \mathsf{T}_{11} \qquad x : \mathsf{T}_{11} \vdash \mathsf{t}_2 : \texttt{Unit}}{\Gamma \vdash \texttt{free}\ \mathsf{t}_1\ \texttt{with}\ x.\ \mathsf{t}_2 : \texttt{Unit}}\ \text{T-Free}$$

$$\frac{\Gamma_1 \vdash \mathsf{t}_1 : \texttt{Array}\ \mathsf{T}_{11} \qquad \Gamma_2 \vdash \mathsf{t}_2 : \texttt{Nat} \qquad \Gamma_3 \vdash \mathsf{t}_3 : \mathsf{T}_{11}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \texttt{swap}\ \mathsf{t}_1 [\mathsf{t}_2]\ \mathsf{t}_3 : \texttt{Array}\ \mathsf{T}_{11} \times \mathsf{T}_{11}}\ \text{T-ArraySwap}$$

Question (Exercise)

Formulate the operational semantics for arrays.

Other Extensions: Reference Counting

Question

In LFPL, data structures only have two modes: (i) **copyable**, or (ii) **linear**.
Can you think of mechanisms to support some modes **between the two ends of the spectrum**?

Principle

Reference counting is a **dynamic** technique that allows any number of pointers to an object and keeps track of that number dynamically. When the reference count gets zero, the object is deallocated automatically.

Syntax

$$\mathsf{t} ::= \ldots \mid \texttt{ref}\ \mathsf{t}_1 \mid \texttt{inc}\ \mathsf{t}_1 \mid \texttt{dec}\ \mathsf{t}_1\ \texttt{with}\ x.\ \mathsf{t}_2 \mid \texttt{swap}\ \mathsf{t}_1\ \mathsf{t}_2 \qquad \mathsf{T} ::= \ldots \mid \mathsf{Rc}\ \mathsf{T}_1$$

R.C. increment: $\texttt{inc}\ \mathsf{t}$ evaluates t to a pointer, increments the ref count, and returns **two copies** of the pointer.

R.C. decrement: $\texttt{dec}\ \mathsf{t}_1\ \texttt{with}\ x.\ \mathsf{t}_2$ evaluates t_1 to a pointer, decrements the ref count, and **apply** $\lambda x.\ \mathsf{t}_2$ **to the object if ref count gets zero**.

Other Extensions: Reference Counting

Operational Semantics

$$\frac{\ell \notin dom(V)}{\texttt{ref}\ v_1 \mid V \longrightarrow \ell \mid (V, \ell \mapsto (v, 1))}\ \text{E-RefV}$$

$$\frac{V(\ell) = (v, k)}{\texttt{inc}\ \ell \mid V \longrightarrow \{\ell, \ell\} \mid (V \setminus \ell, \ell \mapsto (v, k+1))}\ \text{E-IncLoc}$$

$$\frac{V(\ell) = (v, k) \qquad k > 1}{\texttt{dec}\ \ell\ \texttt{with}\ x.\ t_2 \mid V \longrightarrow \texttt{unit} \mid (V \setminus \ell, \ell \mapsto (v, k-1))}\ \text{E-DecNonZero}$$

$$\frac{V(\ell) = (v, 1) \qquad y \notin dom(V)}{\texttt{dec}\ \ell\ \texttt{with}\ x.\ t_2 \mid V \longrightarrow [x \mapsto y]t_2 \mid (V \setminus \ell, y \mapsto v)}\ \text{E-DecZero}$$

$$\frac{V(\ell) = (v_1, k)}{\texttt{swap}\ \ell\ v_2 \mid V \longrightarrow \{\ell, v_1\} \mid (V \setminus \ell, \ell \mapsto (v_2, k))}\ \text{E-Swap}$$

Other Extensions: Reference Counting

Typing

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \mathtt{ref}\ t_1 : \mathtt{Rc}\ T_1}\ \text{T-Ref} \qquad \frac{\Gamma \vdash t_1 : \mathtt{Rc}\ T_{11}}{\Gamma \vdash \mathtt{inc}\ t_1 : \mathtt{Rc}\ T_1 \times \mathtt{Rc}\ T_{11}}\ \text{T-Inc}$$

$$\frac{\Gamma \vdash t_1 : \mathtt{Rc}\ T_{11} \qquad x : T_{11} \vdash t_2 : \mathtt{Unit}}{\Gamma \vdash \mathtt{dec}\ t_1\ \mathtt{with}\ x.\ t_2 : \mathtt{Unit}}\ \text{T-Dec} \qquad \frac{\Gamma_1 \vdash t_1 : \mathtt{Rc}\ T_{11} \qquad \Gamma_2 \vdash t_2 : T_{11}}{\Gamma_1, \Gamma_2 \vdash \mathtt{swap}\ t_1\ t_2 : \mathtt{Rc}\ T_{11} \times T_{11}}\ \text{T-Swap}$$

Question

Are any of `Ref`, `Array`, or `Rc` types **copyable**?

Question (Exercise)

Try to prove the soundness of LFPL with reference counting.
What **invariants** should the operational semantics maintain?

Resource Bounds with LFPL

Remark

Recall that LFPL with the diamond type $\Diamond$ captures **non-size-increasing** computation.

$$
\begin{aligned}
&\mathit{double} : \mathtt{List}\ (\mathtt{Bool} \times \Diamond) \to \mathtt{List\ Bool} \\
&\mathit{double} \equiv \lambda l : \mathtt{List}\ (\mathtt{Bool} \times \Diamond). \\
&\qquad\qquad \mathtt{iter}_{\mathtt{L}}\ l\ \mathtt{with\ nil} \Rightarrow \mathtt{nil}[\mathtt{Bool}] \\
&\qquad\qquad\qquad \mid \mathtt{cons}(x_{d,1}, x, \_)\ \mathtt{of}\ y \Rightarrow \\
&\qquad\qquad\qquad\quad \mathtt{letp}\ \{z, x_{d,2}\} = x\ \mathtt{in\ cons}[\mathtt{Bool}]\ x_{d,1}\ z\ (\mathtt{cons}[\mathtt{Bool}]\ x_{d,2}\ z\ y)
\end{aligned}
$$

Observation

The diamonds in the input (i.e., the size of the input) are an **upper bound** on the number of `cons`-operations.
Can we adapt the type system to derive **upper bounds** on **other kinds of resource**?

Principle

We can separate diamonds from lists and add a **new term `tick`** that consumes one diamond.

Resource Bounds with LFPL

Principle

A `tick` then stands for **one units of resource use**!

$$\frac{\Gamma_1 \vdash t_1 : \Diamond \qquad \Gamma_2 \vdash t_2 : T_2}{\Gamma_1, \Gamma_2 \vdash \mathtt{tick}(t_1); t_2 : T_2}\ \text{T-Tick} \qquad \frac{\Gamma_1 \vdash t_1 : T_1 \qquad \Gamma_2 \vdash t_2 : \mathtt{List}\ T_1}{\Gamma_1, \Gamma_2 \vdash \mathtt{cons}[T_1]\ t_1\ t_2 : \mathtt{List}\ T_1}\ \text{T-Cons}$$

$$\frac{\Gamma_1 \vdash t_1 : \mathtt{List}\ T_{11} \qquad \Gamma_2 \vdash t_2 : S \qquad x : T_{11}, y : S \vdash t_3 : S}{\Gamma_1, \Gamma_2 \vdash \mathtt{iter}_L\ t_1\ \mathtt{with\ nil} \Rightarrow t_2 \mid \mathtt{cons}(x, \_)\ \mathtt{of}\ y \Rightarrow t_3 : S}\ \text{T-IterL}$$

Example

Let us derive a bound on the number of constructors used by an inefficient identity function.

$$\begin{aligned}
&\mathit{id} : \mathtt{List}\ (\Diamond \times \mathtt{Bool}) \to \Diamond \to \mathtt{List\ Bool} \\
&\mathit{id} \equiv \lambda l : \mathtt{List}\ (\Diamond \times \mathtt{Bool}).\ \lambda d_1 : \Diamond. \\
&\qquad \mathtt{iter}\ l\ \mathtt{with\ nil} \Rightarrow \mathtt{tick}(d_1); \mathtt{nil}[\mathtt{Bool}] \\
&\qquad\qquad \mid \mathtt{cons}(x, _)\ \mathtt{of}\ y \Rightarrow \mathtt{letp}\ \{d_2, z\} = x\ \mathtt{in\ tick}(d_2); \mathtt{cons}[\mathtt{Bool}]\ z\ y
\end{aligned}$$

Resource Bounds with LFPL

Question

The type $id : \texttt{List}\ (\lozenge \times \texttt{Bool}) \to \lozenge \to \texttt{List Bool}$ indicates a resource bound of $id(\ell)$ to be $|\ell| + 1$.
However, there are a few restrictions that make LFPL **inconvenient** to use:

- The linear nature prevents using some variables multiple times (even with contraction for copyable types).
- The diamonds "pollute" the code and we have to manage which diamond to pay for a tick.

Any workaround?

Principle (Type-Level Resource-Bound Analysis)

We can remove diamonds from terms and **only keep track of them in the types**.

Ref: M. Hofmann and S. Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Princ. of Prog. Lang.* (POPL'03), 185–197. doi: 10.1145/604131.604148.
Ref: J. Hoffmann and S. Jost. 2022. Two Decades of Automatic Amortized Resource Analysis. *Math. Struct. Comp. Sci.*, 32, 729–759, 6. doi: 10.1017/S0960129521000487.

Type-Level Linear-Resource-Bound Analysis

Remark

The “linear” in “linear-resource-bound” means the bounds are **analytically linear** functions of the input **sizes**.

Syntax

$$
\begin{aligned}
t ::= {} & x \mid \mathtt{fun}\ f\ x.\ t_2 \mid t_1\ t_2 \mid \mathtt{unit} \mid \{t_1, t_2\} \mid \mathtt{let}\ \{x_1, x_2\} = t_1\ \mathtt{in}\ t_2 \\
\mid {} & \mathtt{nil} \mid \mathtt{cons}\ t_1\ t_2 \mid \mathtt{case}_L\ t_1\ \mathtt{of}\ \mathtt{nil} \Rightarrow t_2 \mid \mathtt{cons}(x_1, x_2) \Rightarrow t_3 \mid \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 \\
\mid {} & \mathtt{tick}\ q \quad (q \in \mathbb{Q}) \mid \mathtt{share}\ t_1\ \mathtt{as}\ x_1, x_2\ \mathtt{in}\ t_2
\end{aligned}
$$

Resource-Aware Types

We define **types** and **resource-aware types** as follows:

$$
\begin{aligned}
T &::= A \to B \mid \mathtt{Unit} \mid T_1 \times T_2 \mid \mathtt{List}\ A \\
A, B &::= T^q \quad (q \in \mathbb{Q}_{\geq 0})
\end{aligned}
$$

Resource-Aware Types and Potentials

Principle (The Potential Method for Amortized Complexity Analysis)

Consider a transition system $\langle S, \to, s_0 \rangle$, where $s_0 \in S$ is the initial state and $\to \subseteq S \times S \times \mathbb{Q}$ is the **resource-aware** transition relation.

The map $\Phi : S \to \mathbb{Q}_{\geqslant 0}$ is said to be a **potential function** if for any $s \xrightarrow{q} s'$, it holds that $\Phi(s) \geqslant q + \Phi(s')$.

Then for any $p_0 \geqslant \Phi(s_0)$ and transition sequence $s_0 \xrightarrow{q_1} s_1 \xrightarrow{q_2} \cdots \xrightarrow{q_n} s_n$, it holds that $p_0 - \sum_{i=1}^{n} q_i \geqslant \Phi(s_n)$, i.e., $\Phi(s_0)$ is an **upper bound** on the resource use of the transition system.

Type-Defined Potential Functions: $\Phi_T : [\![T]\!] \to \mathbb{Q}_{\geqslant 0}$

$$[\![T^q]\!] = [\![T]\!] \qquad \Phi_{T^q}(v) = \Phi_T(v) + q$$

$$[\![\mathtt{Unit}]\!] = \{\mathtt{unit}\} \qquad \Phi_{\mathtt{Unit}}(\mathtt{unit}) = 0$$

$$[\![T_1 \times T_2]\!] = [\![T_1]\!] \times [\![T_2]\!] \qquad \Phi_{T_1 \times T_2}(\langle v_1, v_2 \rangle) = \Phi_{T_1}(v_1) + \Phi_{T_2}(v_2)$$

$$[\![\mathtt{List}\ A]\!] = \{[v_1, \ldots, v_n] \mid v_i \in [\![A]\!]\} \qquad \Phi_{\mathtt{List}\ A}([v_1, \ldots, v_n]) = \sum_{i=1}^{n} \Phi_A(v_i)$$

$$[\![A \to B]\!] = [\![A]\!] \to [\![B]\!] \qquad \Phi_{A \to B}(f) = 0$$

Resource-Aware Types and Potentials

Example

$$
\begin{aligned}
&\mathit{id} : (\mathsf{List\ Unit}^{2})^{1} \to (\mathsf{List\ Unit}^{0})^{0} \\
&\mathit{id} \equiv \mathsf{fun}\ f\ l.\ \mathsf{case}_L\ l\ \mathsf{with\ nil} \Rightarrow \mathsf{let}\ \_ = \mathsf{tick}\ 1\ \mathsf{in\ nil} \\
&\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}(h, t) \Rightarrow \mathsf{let}\ \_ = \mathsf{tick}\ 2\ \mathsf{in\ cons}\ h\ (f\ t)
\end{aligned}
$$

The potential function $\Phi_{\mathsf{List}\ T^q}([v_1, \ldots, v_n]) = \sum_{i=1}^{n} \Phi_{T^q}(v_i) = q \cdot n + \sum_{i=1}^{n} \Phi_T(v_i)$.
Thus, $\Phi_{(\mathsf{List\ Unit}^2)^1}(\ell) = \Phi_{\mathsf{List\ Unit}^2}(\ell) + 1 = 2 \cdot |\ell| + \sum_{i=1}^{|\ell|} \Phi_{\mathsf{Unit}}(\ldots) + 1 = 2 \cdot |\ell| + 1$.
That is, $2 \cdot |\ell| + 1$ is **an upper bound on the amount of ticks** of executing $\mathit{id}(\ell)$.

Example

A function can have **more than one** resource-aware types. Such a property is useful when **composing** functions.

$$
\begin{aligned}
\mathit{id} &: (\mathsf{List\ Unit}^{2})^{1} \to (\mathsf{List\ Unit}^{0})^{0} \\
\mathit{id} &: (\mathsf{List\ Unit}^{4})^{6} \to (\mathsf{List\ Unit}^{2})^{5} \\
\mathit{id} &: (\mathsf{List\ Unit}^{4})^{6} \to (\mathsf{List\ Unit}^{0})^{0}
\end{aligned}
$$

Resource-Aware Typing

$\Gamma; q \vdash t : A$: "term t has resource-aware type A under context Γ and potential q ($q \in \mathbb{Q}_{\geq 0}$)"

The typing context Γ is defined as $\Gamma ::= \varnothing \mid \Gamma, x : T$.

$$\frac{}{x : T; q \vdash x : T^q} \text{ T-Var} \qquad \frac{}{\varnothing; q \vdash \texttt{unit} : \texttt{Unit}^q} \text{ T-Unit} \qquad \frac{p = q + r}{\varnothing; p \vdash \texttt{tick}\ q : \texttt{Unit}^r} \text{ T-Tick}$$

$$\frac{\Gamma_1; q_0 \vdash t_1 : T_1^{q_1} \qquad \Gamma_2; q_1 \vdash t_2 : T_2^{q_2}}{\Gamma_1, \Gamma_2; q_0 \vdash \{t_1, t_2\} : (T_1 \times T_2)^{q_2}} \text{ T-Pair}$$

$$\frac{\Gamma_1; q_0 \vdash t_1 : (T_{11} \times T_{12})^{q_1} \qquad \Gamma_2, x_1 : T_{11}, x_2 : T_{12}; q_1 \vdash t_2 : A}{\Gamma_1, \Gamma_2; q_0 \vdash \texttt{let}\ \{x_1, x_2\} = t_1\ \texttt{in}\ t_2 : A} \text{ T-LetP}$$

$$\frac{\Gamma_1; q_0 \vdash t_1 : T_1^{q_1} \qquad \Gamma_2, x : T_1; q_1 \vdash t_2 : A}{\Gamma_1, \Gamma_2; q_0 \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : A} \text{ T-Let}$$

Resource-Aware Typing

$\Gamma; q \vdash t : A$: "term t has resource-aware type A under context Γ and potential q ($q \in \mathbb{Q}_{\geqslant 0}$)"

How to handle lists?

$$
\begin{aligned}
\Phi_{\texttt{List } T^p}(\texttt{nil}) &= 0 \\
\Phi_{\texttt{List } T^p}(\texttt{cons } v_1\ v_2) &= \Phi_{T^p}(v_1) + \Phi_{\texttt{List } T^p}(v_2) \\
&= p + \Phi_T(v_1) + \Phi_{\texttt{List } T^p}(v_2)
\end{aligned}
$$

$$
\frac{}{\varnothing; q \vdash \texttt{nil} : (\texttt{List } A)^q}\ \text{T-Nil}
\qquad
\frac{\Gamma_1; q_0 \vdash t_1 : T_1^{q_1} \qquad \Gamma_2; q_1 \vdash t_2 : (\texttt{List } T_1^p)^{q_2+p}}{\Gamma_1, \Gamma_2; q_0 \vdash \texttt{cons } t_1\ t_2 : (\texttt{List } T_1^p)^{q_2}}\ \text{T-Cons}
$$

$$
\frac{\Gamma_1; q_0 \vdash t_1 : (\texttt{List } T_{11}^p)^{q_1} \qquad \Gamma_2; q_1 \vdash t_2 : A \qquad \Gamma_2, x_1 : T_{11}, x_2 : \texttt{List } T_{11}^p; q_1 + p \vdash t_3 : A}{\Gamma_1, \Gamma_2; q_0 \vdash \texttt{case}_L\ t_1 \texttt{ of nil} \Rightarrow t_2 \mid \texttt{cons}(x_1, x_2) \Rightarrow t_3 : A}\ \text{T-CaseL}
$$

Resource-Aware Typing

$\Gamma; q \vdash t : A$: "term t has resource-aware type A under context Γ and potential q ($q \in \mathbb{Q}_{\geq 0}$)"

How to handle functions?

$$\frac{\Gamma, f : T_1^{q_1} \to T_2^{q_2}, x : T_1; q_1 \vdash t_2 : T_2^{q_2}}{\Gamma; p \vdash \text{fun } f\ x.\ t_2 : (T_1^{q_1} \to T_2^{q_2})^{p}} \text{ T-Fun?}$$

$$\frac{\Gamma_1; q_0 \vdash t_1 : (T_{11}^{q_{11}} \to T_{12}^{q_{12}})^{q_1} \qquad \Gamma_2; q_1 \vdash t_2 : T_{11}^{q_{11}+r}}{\Gamma_1, \Gamma_2; q_0 \vdash t_1\ t_2 : T_{12}^{q_{12}+r}} \text{ T-App}$$

Question

Is the rule (T-Fun?) sound?

$\texttt{let}\ z = \texttt{cons unit (cons unit nil) in}$ $\qquad z : \texttt{List Unit}^2; 2$

$\texttt{let}\ f = \texttt{fun}\ f\ x.\ (\texttt{case}_L\ x\ \texttt{of nil} \Rightarrow \texttt{unit}$

$\qquad \mid \texttt{cons}(\_, t) \Rightarrow \texttt{let}\ \_ = \mathit{id}(z)\ \texttt{in}\ f\ t)\ \texttt{in}$ $\qquad f : (\texttt{List Unit}^1)^0 \to (\texttt{List Unit}^0)^0; 2$

$f\ (\texttt{cons unit (cons unit nil)})$

Resource-Aware Typing

Observation

Because fun $f\ x.\ t_2$ defines a **recursive** function, the rule below might use the resources in Γ for **multiple times**.

$$\frac{\Gamma, f : T_1^{q_1} \to T_2^{q_2}, x : T_1; q_1 \vdash t_2 : T_2^{q_2}}{\Gamma; p \vdash \text{fun } f\ x.\ t_2 : (T_1^{q_1} \to T_2^{q_2})^p} \text{ T-Fun-Unsound}$$

How about the rule below?

$$\frac{f : T_1^{q_1} \to T_2^{q_2}, x : T_1; q_1 \vdash t_2 : T_2^{q_2}}{\varnothing; p \vdash \text{fun } f\ x.\ t_2 : (T_1^{q_1} \to T_2^{q_2})^p} \text{ T-Fun-Attempt}$$

It is sound, but might be too restrictive as it requires every function to be a **closed term**.

One Workaround

We can require Γ to **carry zero units of potenial** in (T-Fun), i.e., $\Gamma = |\Gamma|$.

$$|A \to B| = A \to B \qquad |\texttt{Unit}| = \texttt{Unit} \qquad |T_1 \times T_2| = |T_1| \times |T_2| \qquad |\texttt{List}\ T^p| = \texttt{List}\ |T|^0$$

Resource-Aware Typing

$\Gamma; q \vdash t : A$: "term t has resource-aware type A under context Γ and potential q ($q \in \mathbb{Q}_{\geq 0}$)"

How to handle contraction via the `share` terms?

$$\frac{\Gamma_1; q_0 \vdash t_1 : T_1^{q_1} \qquad \Gamma_2, x_1 : T_1, x_2 : T_1; q_1 \vdash t_2 : A}{\Gamma_1, \Gamma_2; q_0 \vdash \texttt{share}\ t_1\ \texttt{as}\ x_1, x_2\ \texttt{in}\ t_2 : A} \text{ T-Share?}$$

Question

Is the rule (T-Share?) sound?

$\texttt{let}\ z = \texttt{cons unit}\ (\texttt{cons unit nil})\ \texttt{in}$ $\qquad z : \texttt{List Unit}^2; 2$

$\texttt{share}\ z\ \texttt{as}\ x_1, x_2\ \texttt{in}$ $\qquad x_1 : \texttt{List Unit}^2, x_2 : \texttt{List Unit}^2; 2$

$\{id(x_1), id(x_2)\}$

Resource-Aware Typing

Observation

Because T_1 can **carry non-zero units of potential**, the contraction in the rule below is unsound.

$$\frac{\Gamma_1; q_0 \vdash t_1 : T_1^{q_1} \qquad \Gamma_2, x_1 : T_1, x_2 : T_1; q_1 \vdash t_2 : A}{\Gamma_1, \Gamma_2; q_0 \vdash \mathtt{share}\ t_1\ \mathtt{as}\ x_1, x_2\ \mathtt{in}\ t_2 : A}\ \text{T-Share-Unsound}$$

To make the rule sound, we need a notion of **splitting up the potential in a type**.

$T \curlyvee (T_1, T_2)$: "T can be split into T_1 and T_2 and T's potential is the sum of T_1's and T_2's"

That is, if $T \curlyvee (T_1, T_2)$, then $\Phi_T(v) = \Phi_{T_1}(v) + \Phi_{T_2}(v)$.

$$\frac{}{A \to B \curlyvee (A \to B, A \to B)} \qquad \frac{}{\mathtt{Unit} \curlyvee (\mathtt{Unit}, \mathtt{Unit})} \qquad \frac{T_1 \curlyvee (T_{11}, T_{12}) \qquad T_2 \curlyvee (T_{21}, T_{22})}{T_1 \times T_2 \curlyvee (T_{11} \times T_{21}, T_{12} \times T_{22})}$$

$$\frac{T \curlyvee (T_1, T_2) \qquad q = q_1 + q_2}{\mathtt{List}\ T^q \curlyvee (\mathtt{List}\ T_1^{q_1}, \mathtt{List}\ T_2^{q_2})}$$

Resource-Aware Typing

$\Gamma; q \vdash t : A$: "term t has resource-aware type A under context Γ and potential q ($q \in \mathbb{Q}_{\geq 0}$)"

$$\frac{\Gamma_1; q_0 \vdash t_1 : T_1^{q_1} \qquad T_1 \curlyvee (T_{11}, T_{12}) \qquad \Gamma_2, x_1 : T_{11}, x_2 : T_{12}; q_1 \vdash t_2 : A}{\Gamma_1, \Gamma_2; q_0 \vdash \texttt{share } t_1 \texttt{ as } x_1, x_2 \texttt{ in } t_2 : A} \text{ T-Share}$$

Example

We have $\texttt{List Unit}^4 \curlyvee (\texttt{List Unit}^2, \texttt{List Unit}^2)$.

$$\begin{array}{ll} \texttt{let } z = \texttt{cons unit (cons unit nil) in} & z : \texttt{List Unit}^4; 2 \\ \texttt{share } z \texttt{ as } x_1, x_2 \texttt{ in} & x_1 : \texttt{List Unit}^2, x_2 : \texttt{List Unit}^2; 2 \\ \{id(x_1), id(x_2)\} & \end{array}$$

Question (Exercise)

Extend $\curlyvee$ to context-level, i.e., $\Gamma \curlyvee (\Gamma_1, \Gamma_2)$. Use $\curlyvee$ to reformulate the relation $\Gamma = |\Gamma|$.

Resource-Aware Typing

$\Gamma; q \vdash t : A$: "term t has resource-aware type A under context Γ and potential q ($q \in \mathbb{Q}_{\geq 0}$)"

How about weakening?

$$\frac{\Gamma; q \vdash t : A}{\Gamma, x_1 : T_1; q + r \vdash t : A} \text{ T-Weak}$$

Question

Any more structural rules?

$$\dfrac{\dfrac{}{\varnothing; 0 \vdash \mathtt{nil} : (\mathtt{List\ Unit}^0)^0} \quad \dfrac{}{\varnothing; 0 \vdash \mathtt{tick}\ -1 : \mathtt{Unit}^1} \quad \dfrac{\dfrac{}{\varnothing; 0 \vdash \mathtt{tick}\ -2 : \mathtt{Unit}^2}}{\_ : \mathtt{Unit}, \_ : \mathtt{List\ Unit}^0; 0 \vdash \mathtt{tick}\ -2 : \mathtt{Unit}^2}}{\varnothing; 0 \vdash \mathtt{case}_L\ \mathtt{nil}\ \mathtt{of}\ \mathtt{nil} \Rightarrow \mathtt{tick}\ -1 \mid \mathtt{cons}(\_, \_) \Rightarrow \mathtt{tick}\ -2 : ???}$$

Resource-Aware Typing

Observation

Because a value can have multiple types, e.g., $\mathtt{Unit}^1$ and $\mathtt{Unit}^2$ for a `unit`, we need to some form of **subtyping**.

Question

Which direction is sound for resource-bound analysis? $\mathtt{Unit}^1 <: \mathtt{Unit}^2$ or $\mathtt{Unit}^2 <: \mathtt{Unit}^1$?

$T_1 <: T_2$: "T_1 is a subtype of T_2 in the sense that T_1's potential is not less than T_2's"

$$\frac{A_2 <: A_1 \qquad B_1 <: B_2}{A_1 \to B_1 <: A_2 \to B_2} \qquad \frac{}{\mathtt{Unit} <: \mathtt{Unit}} \qquad \frac{T_{11} <: T_{21} \qquad T_{12} <: T_{22}}{T_{11} \times T_{12} <: T_{21} \times T_{22}} \qquad \frac{A_1 <: A_2}{\mathtt{List}\ A_1 <: \mathtt{List}\ A_2}$$

$$\frac{T_1 <: T_2 \qquad {\color{red} q_1 \geqslant q_2}}{T_1^{q_1} <: T_2^{q_2}}$$

Resource-Aware Typing

$\Gamma; q \vdash t : A$: "term t has resource-aware type A under context Γ and potential q ($q \in \mathbb{Q}_{\geqslant 0}$)"

$$\frac{\Gamma; q \vdash t : A \qquad A <: B}{\Gamma; q \vdash t : B}\ \text{T-Sub}$$

Example

Because $\mathtt{Unit}^2 <: \mathtt{Unit}^1$, we obtain the following derivation.

$$\dfrac{\dfrac{}{\varnothing; 0 \vdash \mathtt{nil} : (\mathtt{List\ Unit}^0)^0} \qquad \dfrac{}{\varnothing; 0 \vdash \mathtt{tick}\ {-1} : \mathtt{Unit}^1} \qquad \dfrac{\dfrac{\dfrac{}{\varnothing; 0 \vdash \mathtt{tick}\ {-2} : \mathtt{Unit}^2}}{\_ : \mathtt{Unit}, \_ : \mathtt{List\ Unit}^0; 0 \vdash \mathtt{tick}\ {-2} : \mathtt{Unit}^2}}{\_ : \mathtt{Unit}, \_ : \mathtt{List\ Unit}^0; 0 \vdash \mathtt{tick}\ {-2} : \mathtt{Unit}^1}\ \text{T-Sub}}{\varnothing; 0 \vdash \mathtt{case_L}\ \mathtt{nil}\ \mathtt{of}\ \mathtt{nil} \Rightarrow \mathtt{tick}\ {-1} \mid \mathtt{cons}(\_,\_) \Rightarrow \mathtt{tick}\ {-2} : \mathtt{Unit}^1}$$

Design Principles of Programming Languages

编程语言的设计原理

Key Takeaways

Principle

- The uses of type systems **go far beyond** their role in detecting errors.
- Type systems offer **crucial support** for programming: **abstraction, safety, efficiency, ...**
- Language design shall go **hand-in-hand** with type-system design.