# 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 2026

# 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 $\Gamma$ as a **function** from variables to types, we literally view $\Gamma$ 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

$$t ::= x \mid \lambda x{:}T.\, t \mid t\ t \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \mid \texttt{0} \mid \texttt{succ}\ t \mid \texttt{iszero}\ t$$

$$v ::= \lambda x{:}T.\, t \mid \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid \texttt{succ}\ v$$

$$T ::= T \to T \mid \texttt{Bool} \mid \texttt{Nat}$$

$$\Gamma ::= \varnothing \mid \Gamma, x : T$$

## 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 $\Gamma$ with linear typing"

Let us implicitly assume **Exchange** by treating $\Gamma$ as an unordered list.

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

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

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

# Linear Types

$\Gamma \vdash^{\ell} t : T$: "term $t$ has type $T$ in context $\Gamma$ 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*

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

## Question

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

# 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 $V'$"

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

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

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

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

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

# Operational Semantics

## *Example*

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

## *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 $V$ conforms with typing context $\Gamma$"

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

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

## 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 $\varnothing \vdash^{\ell} t : T$, $T = \texttt{Bool}$ or $T = \texttt{Nat}$, 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 $\Gamma$ 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 $\Gamma_{in}$, term $t$ has type $T$ with remainder context $\Gamma_{out}$"

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

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

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

## 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 $\Gamma$ 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*

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

# 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

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

# 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 $\Gamma$ 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*

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

# 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 $\Gamma$ into $\Gamma_1, \Gamma_2$, we allow them to **overlap**.

$\Gamma \curlyvee (\Gamma_1; \Gamma_2)$: "context $\Gamma$ can be split into $\Gamma_1$ and $\Gamma_2$ 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

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

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

## *Example*

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

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

# 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**.

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

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

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

## *Example*

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

# LFPL: A Linearly-Typed Functional Programming Language

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

## Syntax

$$t ::= x \mid \lambda x{:}T_1.\, t_2 \mid t_1\ t_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \mid \{t_1, t_2\} \mid \texttt{letp}\ \{x_1, x_2\} = t_1\ \texttt{in}\ t_2$$
$$\mid\ \texttt{nil}[T_1] \mid \texttt{cons}[T_1]\ t_1\ t_2 \mid \texttt{iter}_L\ t_1\ \texttt{with}\ \texttt{nil} \Rightarrow t_2 \mid \texttt{cons}(x, \_)\ \texttt{of}\ y \Rightarrow t_3$$
$$T ::= T_1 \to T_2 \mid \texttt{Bool} \mid T_1 \times T_2 \mid \texttt{List}\ T_1$$

## Linear Typing for Pairs

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

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

## Question

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

# LFPL: A Linearly-Typed Functional Programming Language

## Linear Typing for Lists

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

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

## 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*

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

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

# LFPL: A Linearly-Typed Functional Programming Language

## Structural Rules

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

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

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

## *Example*

$$\textit{append2} : \texttt{List Bool} \to \texttt{List Bool} \to \texttt{List Bool}$$

$$\textit{append2} \equiv \lambda l_1 : \texttt{List Bool}.\ \lambda l_2 : \texttt{List Bool}.$$

$$\texttt{iter}_{\texttt{L}}\ l_1\ \texttt{with nil} \Rightarrow l_2$$

$$\mid \texttt{cons}(x, \_)\ \texttt{of}\ y \Rightarrow \texttt{cons[Bool]}\ x\ (\texttt{cons[Bool]}\ x\ y)$$

# 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 $\texttt{cons}[T_1]\ t_1\ t_2$.
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 `cons`-constructors.

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

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

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

# Non-Size-Increasing Functions

## *Example*

$$
\begin{aligned}
&\mathit{append} : \mathtt{List\ T} \to \mathtt{List\ T} \to \mathtt{List\ T} \\
&\mathit{append} \equiv \lambda \mathtt{l}_1 : \mathtt{List\ T}.\ \lambda \mathtt{l}_2 : \mathtt{List\ T}. \\
&\qquad \mathtt{iter}_{\mathtt{L}}\ \mathtt{l}_1\ \mathtt{with\ nil} \Rightarrow \mathtt{l}_2 \\
&\qquad\qquad \mid \mathtt{cons}(\mathtt{x}_{\mathtt{d}}, \mathtt{x}, \_)\ \mathtt{of\ y} \Rightarrow \mathtt{cons}[\mathtt{T}]\ \mathtt{x}_{\mathtt{d}}\ \mathtt{x\ y}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{failed\text{-}append2} : \mathtt{List\ Bool} \to \mathtt{List\ Bool} \to \mathtt{List\ Bool} \\
&\mathit{failed\text{-}append2} \equiv \lambda \mathtt{l}_1 : \mathtt{List\ Bool}.\ \lambda \mathtt{l}_2 : \mathtt{List\ Bool}. \\
&\qquad \mathtt{iter}_{\mathtt{L}}\ \mathtt{l}_1\ \mathtt{with\ nil} \Rightarrow \mathtt{l}_2 \\
&\qquad\qquad \mid \mathtt{cons}(\mathtt{x}_{\mathtt{d}}, \mathtt{x}, \_)\ \mathtt{of\ y} \Rightarrow \mathtt{cons}[\mathtt{Bool}]\ \mathtt{x}_{\mathtt{d}}\ \mathtt{x}\ (\mathtt{cons}[\mathtt{Bool}]\ \mathtt{x}_{\mathtt{d}}\ \mathtt{x\ y})
\end{aligned}
$$

## Question

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

# Non-Size-Increasing Functions

## *Example*

$$
\begin{aligned}
&\mathit{append2} : \mathtt{List}\ (\mathtt{Bool} \times \Diamond) \to \mathtt{List}\ \mathtt{Bool} \to \mathtt{List}\ \mathtt{Bool} \\
&\mathit{append2} \equiv \lambda \mathtt{l}_1 : \mathtt{List}\ (\mathtt{Bool} \times \Diamond).\ \lambda \mathtt{l}_2 : \mathtt{List}\ \mathtt{Bool}. \\
&\qquad \mathtt{iter}_{\mathtt{L}}\ \mathtt{l}_1\ \mathtt{with}\ \mathtt{nil} \Rightarrow \mathtt{l}_2 \\
&\qquad\qquad \mid \mathtt{cons}(\mathtt{x}_{\mathtt{d},1}, \mathtt{x}, \_)\ \mathtt{of}\ \mathtt{y} \Rightarrow \\
&\qquad\qquad\quad \mathtt{letp}\ \{\mathtt{z}, \mathtt{x}_{\mathtt{d},2}\} = \mathtt{x}\ \mathtt{in} \\
&\qquad\qquad\quad \mathtt{cons}[\mathtt{Bool}]\ \mathtt{x}_{\mathtt{d},1}\ \mathtt{z}\ (\mathtt{cons}[\mathtt{Bool}]\ \mathtt{x}_{\mathtt{d},2}\ \mathtt{z}\ \mathtt{y})
\end{aligned}
$$

## Question

Implement a function $\mathtt{triple} : \mathtt{List}\ \mathtt{Bool} \to \mathtt{List}\ \mathtt{Bool}$ 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* : $\texttt{List (Bool} \times \Diamond\texttt{)} \to \texttt{List Bool} \to \texttt{List Bool}$, 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 $\Diamond$.

$$[\![\texttt{Bool}]\!] = \{\texttt{true}, \texttt{false}\} \qquad s_{\texttt{Bool}}(v) = 0$$

$$[\![\Diamond]\!] = \{\blacklozenge\} \qquad s_{\Diamond}(\blacklozenge) = 1$$

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

$$[\![\texttt{List}\ T_1]\!] = \{[v_1, \ldots, v_n] \mid v_i \in [\![T_1]\!]\} \qquad s_{\texttt{List}\ T_1}([v_1, \ldots, v_n]) = n + \textstyle\sum_{i=1}^{n} s_{T_1}(v_i)$$

$$[\![T_1 \to T_2]\!] = [\![T_1]\!] \to [\![T_2]\!] \qquad s_{T_1 \to T_2}(f) = \min\{c \mid \forall v \in [\![T_1]\!].\, s_{T_2}(f(v)) \leqslant c + s_{T_1}(v)\}$$

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 $s_T([\![v]\!]) = 0$, where $[\![v]\!]$ encodes a denotational interpretation of $v$.

We can define binary numbers using lists of Booleans:

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

**Theorem**

We say that a function $h : \mathbb{N}^k \to \mathbb{N}$ is **definable** in LFPL if there exists a term $t : \texttt{List Bool} \to \ldots \to \texttt{List Bool}$ 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

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

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \texttt{ref}\ t_1 : \texttt{Ref}\ T_1}\ \text{T-Ref} \qquad \frac{\Gamma \vdash t_1 : \texttt{Ref}\ T_{11}}{\Gamma \vdash !t_1 : T_{11}}\ \text{T-Deref}$$

$$\frac{\Gamma_1 \vdash t_1 : \texttt{Ref}\ T_{11} \qquad \Gamma_2 \vdash t_2 : T_{11}}{\Gamma_1, \Gamma_2 \vdash \texttt{swap}\ t_1\ t_2 : \texttt{Ref}\ T_{11} \times T_{11}}\ \text{T-Swap}$$

## Operational Semantics

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

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

# Other Extensions: Arrays

## Question

Can you follow the approach for supporting references to arrays?

## Syntax and Typing

$$t ::= \ldots \mid \texttt{array}(t_1, \ldots, t_n) \mid \texttt{free}\ t_1\ \texttt{with}\ x.\ t_2 \mid \texttt{swap}\ t_1\,[t_2]\ t_3 \qquad T ::= \ldots \mid \texttt{Array}\ T_1$$

$$\frac{\Gamma_i \vdash t_i : T_1 \quad (\text{for } 1 \leqslant i \leqslant n)}{\Gamma_1, \ldots, \Gamma_n \vdash \texttt{array}(t_1, \ldots, t_n) : \texttt{Array}\ T_1}\ \text{T-Array} \qquad \frac{\Gamma \vdash t_1 : \texttt{Array}\ T_{11} \qquad x : T_{11} \vdash t_2 : \texttt{Unit}}{\Gamma \vdash \texttt{free}\ t_1\ \texttt{with}\ x.\ t_2 : \texttt{Unit}}\ \text{T-Free}$$

$$\frac{\Gamma_1 \vdash t_1 : \texttt{Array}\ T_{11} \qquad \Gamma_2 \vdash t_2 : \texttt{Nat} \qquad \Gamma_3 \vdash t_3 : T_{11}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \texttt{swap}\ t_1\,[t_2]\ t_3 : \texttt{Array}\ T_{11} \times T_{11}}\ \text{T-ArraySwap}$$

## 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

$t ::= \ldots \mid \texttt{ref}\ t_1 \mid \texttt{inc}\ t_1 \mid \texttt{dec}\ t_1\ \texttt{with}\ x.\ t_2 \mid \texttt{swap}\ t_1\ t_2 \qquad T ::= \ldots \mid \texttt{Rc}\ T_1$

R.C. increment `inc` $t$ evaluates $t$ to a pointer, increments the ref count, and returns **two copies** of the pointer.

R.C. decrement `dec` $t_1$ `with` $x.\ t_2$ evaluates $t_1$ to a pointer, decrements the ref count, and **apply** $\lambda x.\ t_2$ **to the object if ref count gets zero**.

# Other Extensions: Reference Counting

## Operational Semantics

$$\frac{\ell \notin dom(V)}{\texttt{ref}\ v_1 \mid V \longrightarrow \ell \mid (V, \ell \mapsto (v, 1))}\ \text{E-RefV} \qquad \frac{V(\ell) = (v, k)}{\texttt{inc}\ \ell \mid V \longrightarrow \{\ell, \ell\} \mid (V \setminus \ell, \ell \mapsto (v, k+1))}\ \text{E-IncLoc}$$

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

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

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

# Other Extensions: Reference Counting

## Typing

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

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

## 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.

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

## 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.

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

# Resource Bounds with LFPL

## Question

The type $id : \texttt{List}\ (\Diamond \times \texttt{Bool}) \to \Diamond \to \texttt{List}\ \texttt{Bool}$ 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

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

## Resource-Aware Types

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

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

# 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: $\Phi_T : [\![T]\!] \to \mathbb{Q}_{\geqslant 0}$

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

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

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

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

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

# Resource-Aware Types and Potentials

## *Example*

$$id : (\texttt{List Unit}^{2})^{1} \to (\texttt{List Unit}^{0})^{0}$$
$$id \equiv \texttt{fun f l. case}_{\texttt{L}}\ \texttt{l with nil} \Rightarrow \texttt{let \_} = \texttt{tick 1 in nil}$$
$$\mid \texttt{cons}(h, t) \Rightarrow \texttt{let \_} = \texttt{tick 2 in cons h (f t)}$$

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

## *Example*

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

$$id : (\texttt{List Unit}^{2})^{1} \to (\texttt{List Unit}^{0})^{0}$$
$$id : (\texttt{List Unit}^{4})^{6} \to (\texttt{List Unit}^{2})^{5}$$
$$id : (\texttt{List Unit}^{4})^{6} \to (\texttt{List Unit}^{0})^{0}$$

# Resource-Aware Typing

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

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

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

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

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

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

# Resource-Aware Typing

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

How to handle functions?

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

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

## Question

Is the rule (T-Fun?) sound?

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

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

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

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

# Resource-Aware Typing

## Observation

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

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

How about the rule below?

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

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

## One Workaround

We can require $\Gamma$ 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 $\Gamma$ and potential $q$ ($q \in \mathbb{Q}_{\geqslant 0}$)"

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?

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

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

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

# Resource-Aware Typing

## Observation

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

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

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 $\Phi_T(v) = \Phi_{T_1}(v) + \Phi_{T_2}(v)$.

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

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

# Resource-Aware Typing

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

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

## *Example*

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

```
let z = cons unit (cons unit nil) in
share z as x1, x2 in
{id(x1), id(x2)}
```

$z : \texttt{List Unit}^4; 2$

$x_1 : \texttt{List Unit}^2, x_2 : \texttt{List Unit}^2; 2$

## 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 $\Gamma$ and potential $q$ ($q \in \mathbb{Q}_{\geqslant 0}$)"

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?

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

# Resource-Aware Typing

## Observation

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

## Question

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

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

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

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

# Resource-Aware Typing

$\Gamma; q \vdash t : A$: "term $t$ has resource-aware type $A$ under context $\Gamma$ 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 $\texttt{Unit}^2 <: \texttt{Unit}^1$, we obtain the following derivation.

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


# 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.