Fundamentals of Substructural Type Systems

Frank Pfenning

Computer Science Department
Carnegie Mellon University

January 19, 2025
POPL Tutorialfest

Apologies for impressionistic style and lack of references
Some of the more recent ideas joint with
Sophia Roshal, Junyoung Jang, Brigitte Pientka

Why Substructural Types?

- Memory management (Rust)
- Race-free concurrency (Oxidized OCaml)
- Session types for communication (*many libraries*)
- Effect handlers (Koka, Effekt)
- Efficient program reasoning (Verus, linear Dafny)
- Implicit computational complexity
- Quantum computing
- Sharpening general benefits of static type systems
  - Modularity and compositionality
  - Static error detection
  - Verifiable documentation

Modal Types

- Substructural types are part of a larger family of modal types
- Comonadic types
  - Quotation and metaprogramming
  - Phase distinction
- Monadic types
  - Explicating or isolating effects
  - Advanced program structure

Why Not Substructural Type Systems

- Too difficult to understand or use effectively
- Infectious in programs
- Too many ad hoc designs
- Insufficient benefits
- The PL community is making progress on all of these!

This Tutorial

- Focus on fundamental principles of substructural type systems
- Avoid specializing to particular applications
- A step towards mitigating their shortcomings?

Outline

- Linear types (what are they?)
- Fundamental properties and algorithms
  - Statics (type checking)
  - Dynamics (computation)
- Other substructural types (wait, there are more?)
- Integrating type systems (how?)
- Modal types (where do they fit?)
- Principal modes

Positive Linear Types

- Negative types: observe behavior of values by interaction

$A \multimap B$	(linear) functions
$A \mathbin{\&} B$	(lazy) pairs

- Positive types: directly observe structure of values

$A \otimes B$	(eager) pairs	(v, w)
$\mathbf{1}$	unit value	$(\,)$
$\oplus_{\ell \in L}\{\ell : A_\ell\}$	injections	$k(v)$

- Rules for judgment $v : A$ (closed values = observables)

$$\frac{v : A \quad w : B}{(v, w) : A \otimes B} \qquad \frac{}{(\,) : \mathbf{1}} \qquad \frac{(k \in L) \quad v : A_k}{k(v) : \oplus_{\ell \in L}\{\ell : A_\ell\}}$$

Examples of Types and Values

Programs

- Expressions e, organized by type
- Typing judgment (linear natural deduction)

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\text{stand for values}} \quad \vdash \quad \underbrace{e : C}_{\text{computes to value}}$$

- Dynamics
  - Properties of substructural types should be evident
  - Otherwise as high level as possible
  - Use global environment

Pairs, Introduction

- Introduction rule

$$\frac{\Gamma \vdash e_1 : A \quad \Delta \vdash e_2 : B}{\Gamma \,;\, \Delta \vdash (e_1, e_2) : A \otimes B} \otimes I$$

- Context join $\Gamma \,;\, \Delta$ (often written as Γ, Δ)

$$\begin{array}{rcl} (\Gamma, x : A) \;;\; \Delta & = & (\Gamma \,;\, \Delta), x : A \quad (x \notin \Delta) \\ \Gamma \;;\; (\Delta, x : A) & = & (\Gamma \,;\, \Delta), x : A \quad (x \notin \Gamma) \\ (\cdot) \;;\; (\cdot) & = & (\cdot) \end{array}$$

error otherwise

- Bottom-up: distribute variables between premises

Pairs, Elimination

- Elimination rule

$$\frac{\Gamma \vdash e : A \otimes B \quad \Delta, x : A, y : B \vdash e' : C}{\Gamma \,;\, \Delta \vdash \textbf{match}\ e\ ((x, y) \Rightarrow e') : C}\ \otimes E$$

- Substitution-based reduction obscures linearity

$$\textbf{match}\ (v, w)\ ((x, y) \Rightarrow e'(x, y)) \quad \longrightarrow \quad e'(v, w)$$

Substructural Operational Semantics (SSOS)

- Configuration of semantic objects
  - eval e (evaluate e)
  - retn v (return value v)
  - susp f (wait on value to be returned)
  - bind x v (bind x to v)
- Reduction matches the left-hand side of a rule and replaces it by the right-hand side. For example:

$$\text{bind } x\ v \cdot \text{eval } x \quad \longrightarrow \quad \text{retn } v$$

- Binding of x to v is deallocated!
- Configuration is ordered (for sequential computation), except for bindings
- Theorem:
  eval $e \longrightarrow^*$ retn v *iff* e *evaluates to* v
  (e.g., under substitution-based semantics)
- Significant: no bindings in the final configuration

Pairs, Dynamics (selected rules)

$$\text{eval}\ (\textbf{match}\ e\ ((x,y) \Rightarrow e'))$$
$$\longrightarrow \text{eval}\ e \cdot \text{susp}\ (\textbf{match}\ \_\ ((x,y) \Rightarrow e'))$$

$$\text{retn}\ (v,w) \cdot \text{susp}\ (\textbf{match}\ \_\ ((x,y) \Rightarrow e'))$$
$$\longrightarrow \text{bind}\ x\ v \cdot \text{bind}\ y\ w \cdot \textbf{eval}\ e' \qquad (x,y \text{ “fresh”})$$

Injections

- Statics

$$\frac{(k \in L) \quad \Gamma \vdash e : A_k}{\Gamma \vdash k(e) : \oplus_{\ell \in L}\{\ell : A_\ell\}} \oplus I$$

$$\frac{\Gamma \vdash e : \oplus_{\ell \in L}\{\ell : A_\ell\} \quad (\Delta, x_\ell : A_\ell \vdash e_\ell : C) \quad (\forall \ell \in L)}{\Gamma \,;\, \Delta \vdash \mathbf{match}\ e\ \{\ell(x_\ell) \Rightarrow e_\ell\}_{\ell \in L} : C} \oplus E$$

- Dynamics (selected)

$$\begin{array}{l} \text{eval}\ (\mathbf{match}\ e\ (\ell(x_\ell) \Rightarrow e_\ell)_{\ell \in L}) \\ \qquad \longrightarrow \text{eval}\ e \cdot \text{susp}\ (\mathbf{match}\ \_\ (\ell(x_\ell) \Rightarrow e_\ell)_{\ell \in L}) \end{array}$$

$$\begin{array}{l} \text{retn}\ k(v) \cdot \text{susp}\ (\mathbf{match}\ \_\ (\ell(x_\ell) \Rightarrow e_\ell)_{\ell \in L}) \\ \qquad \longrightarrow \text{bind}\ x_\ell\ v \cdot \mathbf{eval}\ e_k \qquad (x_\ell\ \text{“fresh”}) \end{array}$$

Intuitionistic Linear Logic

- Coheres with (intuitionistic) linear logic

$$\frac{}{A \vdash A}\ \text{hyp}$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma\ ;\ \Delta \vdash A \otimes B}\ \otimes I \qquad \frac{\Gamma \vdash A \otimes B \quad \Delta, A, B \vdash C}{\Gamma\ ;\ \Delta \vdash C}\ \otimes E$$

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Gamma \vdash \mathbf{1} \quad \Delta \vdash C}{\Gamma\ ;\ \Delta \vdash C}\ \mathbf{1}E$$

$$\frac{(k \in L) \quad \Gamma \vdash A_k}{\Gamma \vdash \oplus_{\ell \in L}\{\ell : A_\ell\}}\ \oplus I \qquad \frac{\Gamma \vdash \oplus_{\ell \in L}\{\ell : A_\ell\} \quad (\Delta, A_\ell \vdash C) \quad (\forall \ell \in L)}{\Gamma\ ;\ \Delta \vdash C}\ \oplus E$$

- But: types may be recursive, though contractive
- Exponential $!A$ to be discussed later

Top-Level Recursive Definitions

- Support mutually recursive top-level definitions

  decl $F\ (x_1 : A_1)\ \ldots\ (x_n : A_n) : C$
  defn $F\ x_1\ \ldots\ x_n = e$
  $e ::= \ldots \mid F\ e_1\ \ldots\ e_n$
- (Partially) internalize the typing judgment
- Technically: Linear Contextual Modal Type Theory

  $F :: (\Delta \vdash C)$
  $F = (\overline{x}.\, e)$
  $e ::= \ldots \mid F[\eta] \quad$ (for a substitution $\eta : \Delta$)
- F closed, so may be reused even in a linear type theory
- First-order linear programs = positive types + metavariables
- Examples

Some Observations

- No garbage ($=$ bind semantic objects) [Girard & Lafont'87]
  - eval $e \longrightarrow^* \mathsf{retn}\ v$ (or diverges)
  - Proof is not difficult
- Drop and copy can be defined for any positive type
- Weakening and contraction are logically admissible
- Mostly, sharing is better than copying

Outline

- Linear types (what are they?)
- Fundamental properties and algorithms
  - Statics (type checking)
  - Dynamics (computation)
- Other substructural types (wait, there are more?)
- Integrating type systems (how?)
- Modal types (where do they fit?)
- Principal modes

Typechecking

- Two principal ideas
- Bidirectional typing

$$\Gamma \vdash e : A \quad \rightsquigarrow \quad \begin{cases} \Gamma \vdash e \Leftarrow A & e \text{ checks against } A \\ \Gamma \vdash e \Rightarrow A & e \text{ synthesizes } A \end{cases}$$

  - $\Gamma \vdash e \Leftarrow A$ assumes Γ, e, and A are given
  - $\Gamma \vdash e \Rightarrow A$ assume Γ, e are given, A synthesized

- Context generation ("additive resource management")

$$\Gamma \vdash e \Leftrightarrow A \quad \rightsquigarrow \quad \Gamma \vdash e \Leftrightarrow A \;/\; \Delta$$

In $\Gamma \vdash e \Leftrightarrow A \;/\; \Delta$

- Γ contains all variables lexically in scope
- Δ picks out the variables actually used in e
- $\Gamma \vdash e : A \;/\; \Delta$ implies $\Gamma \supseteq \Delta$ and $\Delta \vdash e : A$

Some Interesting Rules

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A \;/\; (x : A)}\ \text{var} \qquad \frac{\Gamma \vdash x \Rightarrow A' \;/\; \Delta \quad A' = A}{\Gamma \vdash x \Leftarrow A \;/\; \Delta}\ {\Rightarrow}/{\Leftarrow}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow A \;/\; \Delta_1 \quad \Gamma \vdash e_2 \Leftarrow B \;/\; \Delta_2}{\Gamma \vdash (e_1, e_2) \Leftarrow A \otimes B \;/\; \Delta_1 \;;\; \Delta_2}\ {\otimes}I$$

$$\frac{\Gamma \vdash e \Rightarrow A \otimes B \;/\; \Delta_1 \quad \Gamma, x : A, y : B \vdash e' \Leftarrow C \;/\; \Delta_2}{\Gamma \vdash \textbf{match}\ e\ ((x, y) \Rightarrow e') \Leftarrow C \;/\; \Delta_1 \;;\; (\Delta_2 \setminus x \setminus y)}\ {\otimes}E$$

- $\Delta \setminus x$ checks that x is in Δ and removes it
- Ensure variables are actually used

$$\begin{array}{lcl} (\Delta, x : A) \setminus x & = & \Delta \\ (\Delta, y : B) \setminus x & = & (\Delta \setminus x), y : B \quad (x \neq y) \\ (\cdot) \setminus x & = & \text{error} \end{array}$$

Some More Interesting Rules

$$\frac{(k \in L) \quad \Gamma \vdash e \Leftarrow A_k \;/\; \Delta}{\Gamma \vdash k(e) \Leftarrow \oplus_{\ell \in L}\{\ell : A_\ell\} \;/\; \Delta} \;\oplus I$$

$$\frac{\Gamma \vdash e \Rightarrow \oplus_{\ell \in L}\{\ell : A_\ell\} \;/\; \Delta \quad (\Gamma, x_\ell : A_\ell \vdash e_\ell \Leftarrow C \;/\; \Delta_\ell \quad \Delta_\ell \setminus x_\ell = \Delta') \quad (\forall \ell \in L)}{\Gamma \vdash \textbf{match}\; e\; \{\ell(x_\ell) \Rightarrow e_\ell\}_{\ell \in L} \Leftarrow C \;/\; \Delta \;;\; \Delta'} \;\oplus E$$

- Δ' must be the same in all branches
- $L \neq \{\,\}$ is significant in $\oplus E$ so Δ' is defined

Summary So Far

- Positive linear types + metavariables
- Variables must be used exactly once
- Algorithmic type-checking
  - Bidirectional typing
  - Context generation
  - Join $\Delta_1 ; \Delta_2$ and removal $\Delta \setminus x$ operations
- Evaluation with global environment
  - Bindings are deallocated when read
  - No "garbage" (semantic objects bind x v)
  - Related version with global heap has related property

Outline

- Linear types (what are they?)
- Fundamental properties and algorithms
  - Statics (type checking)
  - Dynamics (computation)
- Other substructural types (wait, there are more?)
- Integrating type systems (how?)
- Modal types (where do they fit?)
- Principal modes

Affine Types

- Rust is based on affine types
- Variables can be used at most once
- Ideas carry over surprisingly easily
- The expressions of the language do not change at all!

Join and Remove Revisited

- Join remains the same

$$
\begin{array}{rcl}
(\Gamma, x : A) \; ; \; \Delta & = & (\Gamma \; ; \Delta), x : A \quad (x \notin \Delta) \\
\Gamma \; ; \; (\Delta, x : A) & = & (\Gamma \; ; \Delta), x : A \quad (x \notin \Gamma) \\
(\cdot) \; ; \; (\cdot) & = & (\cdot)
\end{array}
$$

error otherwise

- Remove allows variables not to be used

$$
\begin{array}{rcl}
(\Delta, x : A) \setminus x & = & \Delta \\
(\Delta, y : B) \setminus x & = & (\Delta \setminus x), y : B \quad (x \neq y) \\
(\cdot) \setminus x & = & (\cdot)
\end{array}
$$

no longer error

Least Upper Bounds for Branches

- Variables may be used in some branches but not others

$$\frac{\Gamma \vdash e \Rightarrow \oplus_{\ell \in L}\{\ell : A_\ell\} \;/\; \Delta \quad (\Gamma, x_\ell : A_\ell \vdash e_\ell \Leftarrow C \;/\; \Delta_\ell \quad \Delta_\ell \setminus x_\ell = \Delta'_\ell) \quad (\forall \ell \in L)}{\Gamma \vdash \textbf{match}\ e\ \{\ell(x_\ell) \Rightarrow e_\ell\}_{\ell \in L} \Leftarrow C \;/\; \Delta \;;\; (\bigsqcup_{\ell \in L} \Delta'_\ell)} \oplus E$$

- Variable is used if used in at least one branch

$$\begin{array}{rcllll}
(\Gamma, x : A) & \sqcup & (\Delta, x : A) & = & (\Gamma \sqcup \Delta), x : A & \\
(\Gamma, x : A) & \sqcup & \Delta & = & (\Gamma \sqcup \Delta), x : A & (x \notin \Delta) \\
\Gamma & \sqcup & (\Delta, x : A) & = & (\Gamma \sqcup \Delta), x : A & (x \notin \Gamma) \\
(\cdot) & \sqcup & (\cdot) & = & (\cdot) &
\end{array}$$

error otherwise

Examples

Dynamically

- All bindings are introduced as semantic objects [bind x v] which need not be used
- Variable rule

$$[\text{bind } x\ v], \text{eval } x \quad \longrightarrow \quad \text{retn } v$$

- Theorem

$$\text{eval } e \longrightarrow^* [\overline{\text{bind } x_i\ w_i}] \cdot \text{retn } v \textit{ iff } e \textit{ evaluates to } v$$

- We can map affine to linear types and explicitly deallocate
  - At the end of scopes where variables are unused $(\Delta \setminus x)$
  - At the end of branches where variables are unused $(\bigsqcup_{\ell \in L} \Delta_\ell)$

Strict Types

- The Haskell compiler performs strictness analysis (for efficiency)
- Annoying warnings about unused variables in ML
- A variable is strict if it is used at least once
    - Dynamically, when the program runs
- Again, ideas carry over surprisingly easily
- The language of expressions does not change at all

Join and Remove Revisited

- Only need to reconsider join, remove, least upper bound
- Join changes:

$$
\begin{array}{rclll}
(\Gamma, x : A) & ; & (\Delta, x : A) & = & (\Gamma ; \Delta), x : A \quad \text{new!} \\
(\Gamma, x : A) & ; & \Delta & = & (\Gamma ; \Delta), x : A \quad (x \notin \Delta) \\
\Gamma & ; & (\Delta, x : A) & = & (\Gamma ; \Delta), x : A \quad (x \notin \Gamma) \\
(\cdot) & ; & (\cdot) & = & (\cdot)
\end{array}
$$

- Remove reverts: variables must be used

$$
\begin{array}{lcl}
(\Delta, x : A) \setminus x & = & \Delta \\
(\Delta, y : B) \setminus x & = & (\Delta \setminus x), y : B \quad (x \neq y) \\
(\cdot) \setminus x & & \text{error}
\end{array}
$$

Least Upper Bound Revisited

- Least upper bound no longer allows weakening

$$\begin{array}{rcccl}
(\Gamma, x : A) & \sqcup & (\Delta, x : A) & = & (\Gamma \sqcup \Delta), x : A \\
(\Gamma, x : A) & \sqcup & \Delta & & \text{error for } x \notin \Delta \\
\Gamma & \sqcup & (\Delta, x : A) & & \text{error for } x \notin \Gamma \\
(\cdot) & \sqcup & (\cdot) & = & (\cdot)
\end{array}$$

Examples

Dynamics

- Bindings are introduced as required. For example

$$\begin{array}{ll} & \mathsf{retn}\ (v, w) \cdot \mathsf{susp}\ (\mathbf{match}\ \_\ ((x, y) \Rightarrow e')) \\ \longrightarrow & \mathsf{bind}\ x\ v \cdot \mathsf{bind}\ y\ w \cdot \mathbf{eval}\ e' \qquad (x, y\ \text{“fresh”}) \end{array}$$

- Become provisional, once read

$$\mathsf{bind}\ x\ v \cdot \mathsf{eval}\ x \quad \longrightarrow \quad [\mathsf{bind}\ x\ v] \cdot \mathsf{retn}\ x$$

- Theorem
  $\mathsf{eval}\ e \longrightarrow^{*} [\overline{\mathsf{bind}\ x_i\ w_i}] \cdot \mathsf{retn}\ v$ *iff* e *evaluates to* v

- Implies each variable is read at least once

Outline

- Linear types (what are they?)
- Fundamental properties and algorithms
  - Statics (type checking)
  - Dynamics (computation)
- Other substructural types (wait, there are more?)
- Integrating type systems (how?)
- Modal types (where do they fit?)
- Principal modes

Combining Systems

- Linear logic [Girard'87]
  - Embed by translation $A \to B \triangleq\ !A \multimap B$
  - Elegant theoretically, but difficult to work with
- LNL [Benton'94]
  - Linear and nonliner logics combined by adjoint operators
  - We can *natively* program in linear or nonlinear modes
  - And switch between them
- Generalize LNL to a set of related modes
- Adjoint types [Reed'09] [Pruiksma et al.'18] [Jang et al.'24]

Modes

- Assume a set of modes and a preorder $n \geq m$ between them
- Each mode has an intrinsic set of structural properties $\sigma(m)$
  - $\mathsf{W} \in \sigma(m)$ means weakening (variables need not be used)
  - $\mathsf{C} \in \sigma(m)$ means contraction (variable may be reused)
  - Exchange is always assumed (variable order is irrelevant)
- $n \geq m$ implies $\sigma(n) \supseteq \sigma(m)$
  - Necessary so structural rules don't sneak in through the back door

Shifts

- Each type has an intrinsic mode A_m
- Shifts $\uparrow_k^m A_k$ and $\downarrow_m^n A_n$ transition between modes
- Typical example $\mathsf{U}, \mathsf{A}, \mathsf{L}$ with

$$\begin{array}{lcll}\sigma(\mathsf{U}) & = & \{\mathsf{W}, \mathsf{C}\} & \text{(Unrestricted)} \\ \sigma(\mathsf{A}) & = & \{\mathsf{W}\} & \text{(Affine)} \\ \sigma(\mathsf{L}) & = & \{\,\} & \text{(Linear)}\end{array}$$

  with

$$\mathsf{U} > \mathsf{A} > \mathsf{L}$$

- Syntax defaults back to a more generic notation

$$\begin{array}{lrcl}\text{Positive types} & A_m & ::= & A_m \times B_m \mid \mathbf{1} \mid +_{\ell \in L}\{\ell : A_m^\ell\} \mid \downarrow_m^n A_n \\ \text{Negative types} & A_m & ::= & A_m \to B_m \mid \&_{\ell \in L}\{\ell : A_m^\ell\} \mid \uparrow_k^m A_k\end{array}$$

Typing

- Typing changes subtly, but fundamentally
- Account for $m \geq k$
- Define $\Delta \geq m$ if $n \geq m$ for all $y : B_n$ in Δ
- Independence principle
  - $\Delta \vdash e : A_m$ presupposes $\Delta \geq m$
  - Maintain that for $\Gamma \vdash e \Leftrightarrow A_m \;/\; \Delta$ we have $\Delta \geq m$

Expressions

- New construct for $\downarrow A$ (positive) and $\uparrow A$ (negative)
- $\downarrow A$ is observable, a "pointer" (address) $\langle v \rangle$

$$\frac{v : A_n}{\langle v \rangle : \downarrow^n_m A_n} \downarrow I \qquad \frac{\Gamma \vdash e \Leftarrow A_n \;/\; \Delta}{\Gamma \vdash \langle e \rangle \Leftarrow \downarrow^n_m A_n \;/\; \Delta} \downarrow I$$

$$\frac{\Gamma \vdash e \Rightarrow \downarrow^n_m A_n \;/\; \Delta \quad (m \geq r) \quad \Gamma, x : A_n \vdash e' \Leftarrow C_r \;/\; \Delta'}{\Gamma \vdash \textbf{match}\; e\; (\langle x \rangle \Rightarrow e') \Leftarrow C_r \;/\; \Delta \;;\; (\Delta' \setminus x_n)} \downarrow E$$

- $\uparrow A$ represents a suspension
  - Expressions suspend **susp** e and force e.**force**

Typing

- In each elimination, we need to enforce independence
- For example

$$\frac{\Gamma \vdash e_1 \Leftarrow A_m \;/\; \Delta_1 \quad \Gamma \vdash e_2 \Leftarrow B_m \;/\; \Delta_2}{\Gamma \vdash (e_1, e_2) \Leftarrow A_m \times B_m \;/\; \Delta_1 \;;\; \Delta_2} \times I$$

$$\frac{\Gamma \vdash e \Rightarrow A_m \times B_m \;/\; \Delta_1 \quad (m \geq r) \quad \Gamma, x : A_m, y : B_m \vdash e' \Leftarrow C_r \;/\; \Delta_2}{\Gamma \vdash \textbf{match}\ e\ ((x, y) \Rightarrow e') \Leftarrow C_r \;/\; \Delta_1 \;;\; (\Delta_2 \setminus x_m \setminus y_m)} \times E$$

Context Operators Discriminate on Modes

- Join

$$
\begin{array}{rcl}
(\Gamma, x : A_m) \;;\; (\Delta, x : A_m) & = & (\Gamma \,;\, \Delta), x : A_m \quad \text{provided } C \in \sigma(m) \\
(\Gamma, x : A_m) \;;\; \Delta & = & (\Gamma \,;\, \Delta), x : A_m \quad (x \notin \Delta) \\
\Gamma \;;\; (\Delta, x : A_m) & = & (\Gamma \,;\, \Delta), x : A_m \quad (x \notin \Gamma) \\
(\cdot) \;;\; (\cdot) & = & (\cdot)
\end{array}
$$

- Removal

$$
\begin{array}{rcl}
(\Delta, x : A_m) \setminus x_m & = & \Delta \\
(\Delta, y : B_k) \setminus x_m & = & (\Delta \setminus x_m), y : B_k \quad (x \neq y) \\
(\cdot) \setminus x_m & = & (\cdot) \quad \text{provided } W \in \sigma(m)
\end{array}
$$

- Least upper bound

$$
\begin{array}{rcl}
(\Gamma, x : A_m) \sqcup (\Delta, x : A_m) & = & (\Gamma \sqcup \Delta), x : A_m \\
(\Gamma, x : A_m) \sqcup \Delta & & \text{for } x \notin \Delta \text{ provided } W \in \sigma(m) \\
\Gamma \sqcup (\Delta, x : A_m) & & \text{for } x \notin \Gamma \text{ provided } W \in \sigma(m) \\
(\cdot) \sqcup (\cdot) & = & (\cdot)
\end{array}
$$

Adjoint Dynamics

- Bindings based on modes
- Provisional if allowing weakening
- Kept if allowing contraction
- Theorem (as before)

  $\text{eval } e \longrightarrow^* [\overline{\text{bind } x_i \; w_i}] \cdot \text{retn } v$ *iff e evaluates to v*

Examples

- A given expression may have multiple types and modes
- Allow overloading
- Different instance may be compiled to different code
- Live code

Summary

- Adjoint type system with linear, affine, strict, unrestricted modes
- Subject to monotonic preorder and independence principle
- We have a full (identical) language of expressions at each mode
- Elegant, mutually conservative integration
- See [Jang,Roshal,Pf.,Pientka'24] for negatives and empty sums lazy products

Outline

- Linear types (what are they?)
- Fundamental properties and algorithms
  - Statics (type checking)
  - Dynamics (computation)
- Other substructural types (wait, there are more?)
- Integrating type systems (how?)
- Modal types (where do they fit?)
- Principal modes

Recovering Comonads

- A special cases where a layer is impoverished
- Define $!A_{\mathsf{L}} = {\downarrow}^{\mathsf{U}}_{\mathsf{L}}{\uparrow}^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}}$ the exponential of linear logic
  - U contains only ${\uparrow}^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}}$
- Define $\mathsf{V} > \mathsf{U}$ with $\sigma(\mathsf{V}) = \sigma(\mathsf{U}) = \{\mathsf{W}, \mathsf{C}\}$
- Define $\Box A_{\mathsf{U}} = {\downarrow}^{\mathsf{V}}_{\mathsf{U}}{\uparrow}^{\mathsf{V}}_{\mathsf{U}} A_{\mathsf{U}}$ the necessity of intuitionistic S4
  - V contains only ${\uparrow}^{\mathsf{V}}_{\mathsf{U}} A_{\mathsf{U}}$
- Additional metaprogramming patterns can be expressed

Recovering Strong Monads

- Define $\mathsf{U} > \mathsf{X}$ with $\sigma(\mathsf{U}) = \sigma(\mathsf{X}) = \{\mathsf{W}, \mathsf{C}\}$
- Define $\bigcirc A_{\mathsf{U}} = \uparrow_{\mathsf{X}}^{\mathsf{U}} \downarrow_{\mathsf{X}}^{\mathsf{U}} A_{\mathsf{U}}$ the strong monad of lax logic
  - X contains only $\downarrow_{\mathsf{X}}^{\mathsf{U}} A_{\mathsf{X}}$

Outline

- Linear types (what are they?)
- Fundamental properties and algorithms
  - Statics (type checking)
  - Dynamics (computation)
- Other substructural types (wait, there are more?)
- Integrating type systems (how?)
- Modal types (where do they fit?)
- Principal modes

Principal Modes

- Expressions do not mention types
- Rules are uniform
  - Types are given, but not modes
  - Collect constraints $\mathsf{W} \in \sigma(m)$, $\mathsf{C} \in \sigma(m)$, $m \geq k$, $m = k$
  - Solve constraints for most general solution
  - Any instance is a valid typing and vice versa
  - Recursion requires a fixed point iteration
- Recent, unpublished work [Roshal & Pf.'24]

Examples

Summary

- Substructural types from very few principles
  - Modes with intrinsic structural properties
  - Monotonic preorder with independence principle
  - Shifts to go between them
  - Uniform set of typing rules
  - Uniform set of computation rules
- Extends to (some) modal types
- Implemented in the Snax compiler
  - See OPLSS'24 lectures
  - Other optimizations like static memory reuse