THE UNIVERSITY of EDINBURGH
informatics

Compiler Intermediate Representations

SPLV 2020 — Michel Steuwer

Outline of Lectures over the week

- **Tuesday:** Functional Intermediate Representations
  - Lambda Calculus and the Lambda Cube
  - Implementation Strategies for System F (ADTs across different PLs)
  - Compiler transformations as rewrite rules
- **Wednesday:** Imperative Intermediate Representations
  - Foundations of Single Static Assignment (SSA)
  - LLVM IR
  - Control-Flow Graphs
  - Data-flow analysis
- **Thursday:** Domain-Specific Intermediate Representations
  - MLIR — a compiler infrastructure for building domain-specific intermediate representations
  - Dataflow graphs — TensorFlow
  - Pattern-based (and functional) — RISE

Lamda Calculus

$\rightarrow$ *(untyped)*

Syntax

$\mathtt{t}$	::=		*terms:*
		$\mathtt{x}$	*variable*
		$\lambda\mathtt{x.t}$	*abstraction*
		$\mathtt{t\ t}$	*application*
$\mathtt{v}$	::=		*values:*
		$\lambda\mathtt{x.t}$	*abstraction value*

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(E-App2)}$$

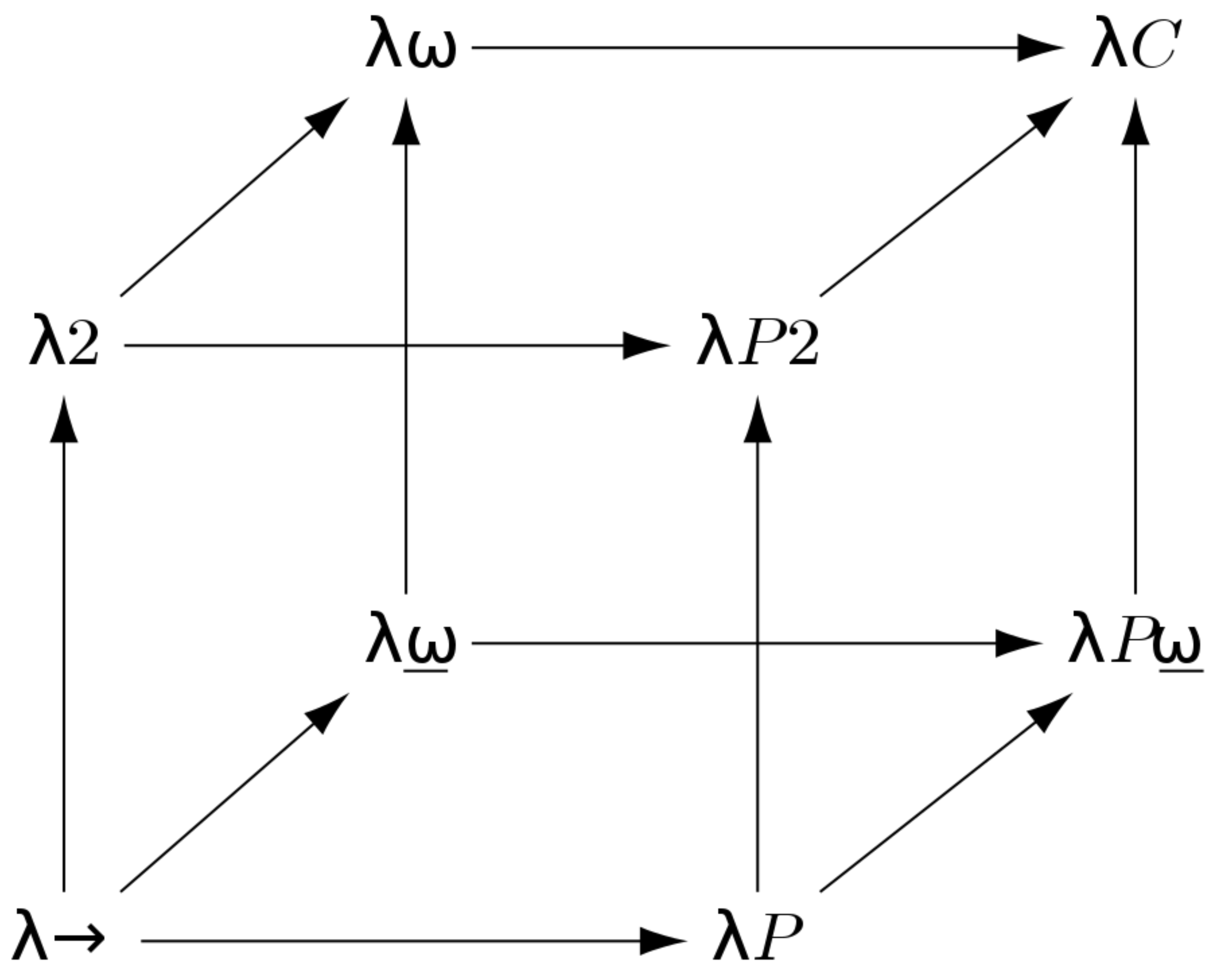
$$(\lambda\mathtt{x.t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \quad \text{(E-AppAbs)}$$

Figure 5-3: Untyped lambda-calculus (λ)

Types and Programming Languages, *B. Pierce*

Typed Lambda Calculus

What type system (or logical foundation) do you want?



Simply typed λ-calculus

Simply Typed Lambda Calculus

→ (typed) *Based on λ (5-3)*

Syntax

$\mathtt{t}$	::=		*terms:*
		$\mathtt{x}$	*variable*
		$\lambda\mathtt{x}{:}\mathtt{T}.\mathtt{t}$	*abstraction*
		$\mathtt{t}\;\mathtt{t}$	*application*
$\mathtt{v}$	::=		*values:*
		$\lambda\mathtt{x}{:}\mathtt{T}.\mathtt{t}$	*abstraction value*
$\mathtt{T}$	::=		*types:*
		$\mathtt{T}\to\mathtt{T}$	*type of functions*
Γ	::=		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma, \mathtt{x}{:}\mathtt{T}$	*term variable binding*

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\;\mathtt{t}_2 \longrightarrow \mathtt{t}_1'\;\mathtt{t}_2} \quad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\;\mathtt{t}_2 \longrightarrow \mathtt{v}_1\;\mathtt{t}_2'} \quad \text{(E-App2)}$$

$$(\lambda\mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_{12})\;\mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \quad \text{(E-AppAbs)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-Var)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1\to\mathtt{T}_2} \quad \text{(T-Abs)}$$

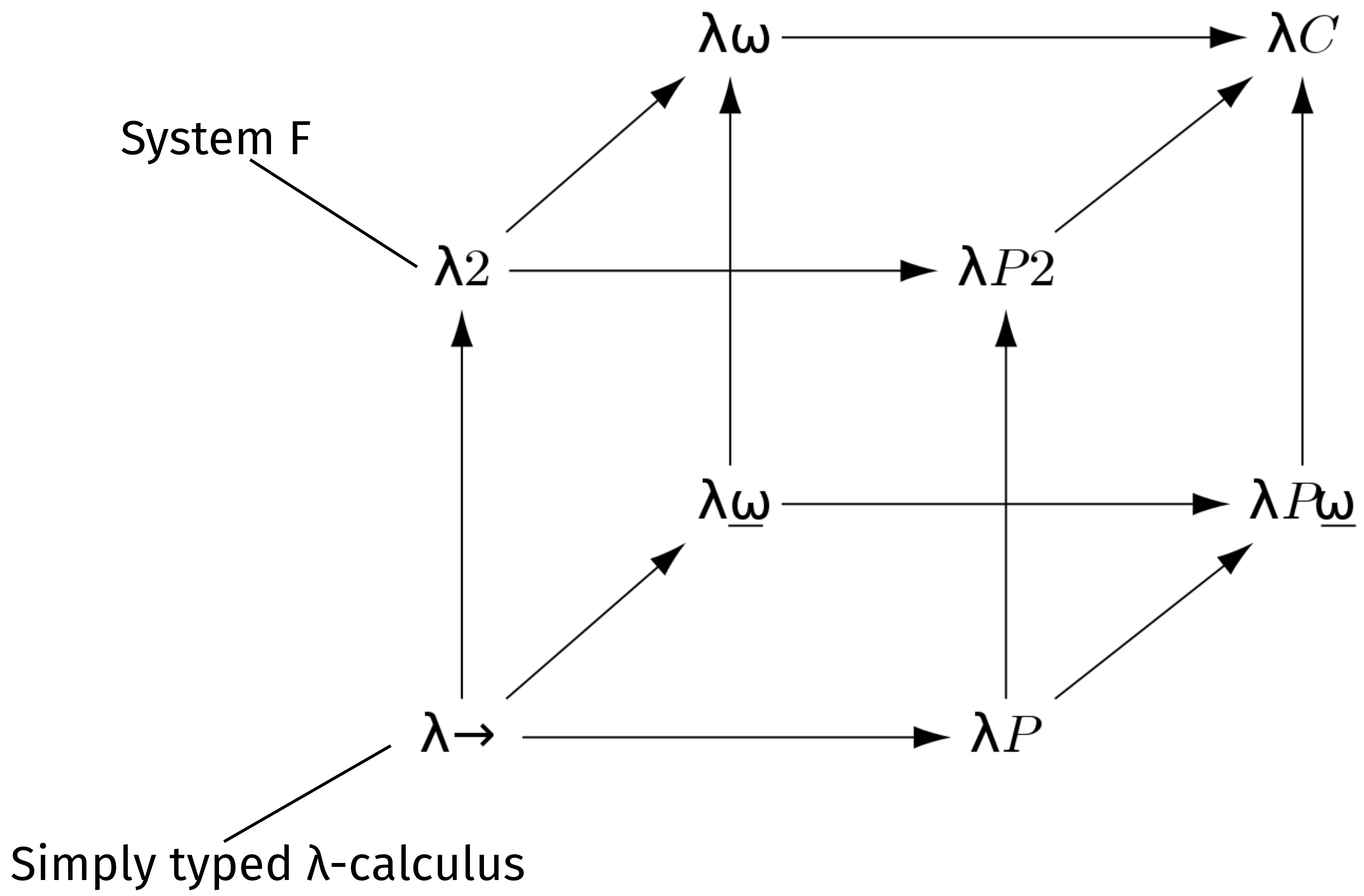
$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11}\to\mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\;\mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-App)}$$

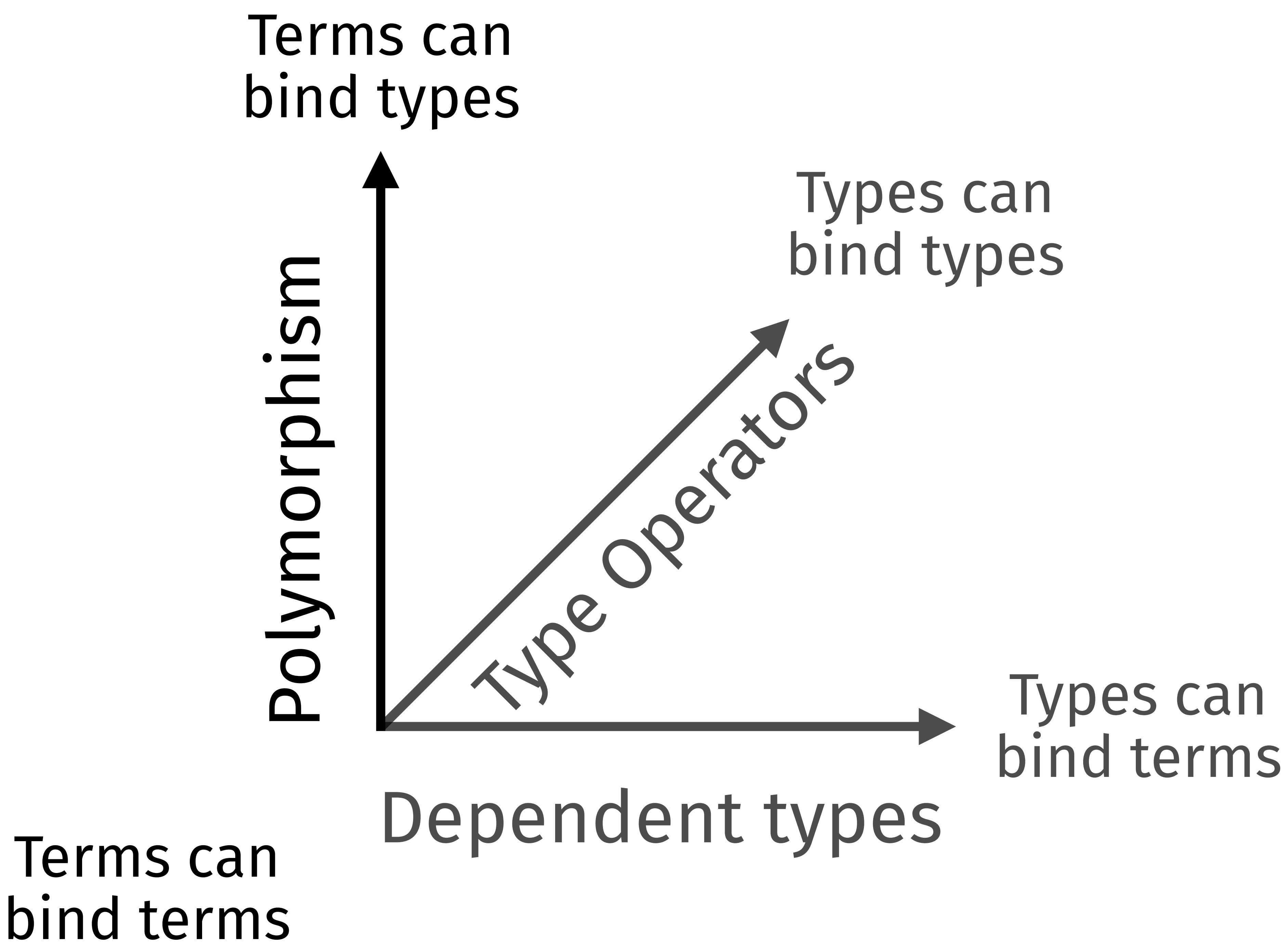
Figure 9-1: Pure simply typed lambda-calculus ($\lambda_\to$)

Types and Programming Languages, *B. Pierce*

Typed Lambda Calculus

What type system (or logical foundation) do you want?





λ2 (aka SystemF)

→ ∀ *Based on λ→ (9-1)*

Syntax

t ::=		*terms:*
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
	λX.t	*type abstraction*
	t [T]	*type application*
v ::=		*values:*
	λx:T.t	*abstraction value*
	λX.t	*type abstraction value*
T ::=		*types:*
	X	*type variable*
	T→T	*type of functions*
	∀X.T	*universal type*
Γ ::=		*contexts:*
	∅	*empty context*
	Γ, x:T	*term variable binding*
	Γ, X	*type variable binding*

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TApp)}$$

$$(\lambda X.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TappTabs)}$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_{12}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TApp)}$$

Figure 23-1: Polymorphic lambda-calculus (System F)

Types and Programming Languages, *B. Pierce*

Haskell Core is build on SystemF*

Haskell

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Core

```
map :: forall a b. (a -> b) -> [a] -> [b]
map =
  \ (@ a) (@ b) (f :: a -> b) (xs :: [a]) ->
    case xs of _ {
      []     -> GHC.Types.[] @ b;
      : y ys -> GHC.Types.: @ b (f y) (map @ a @ b f ys)
    }
```

From http://www.scs.stanford.edu/11au-cs240h/notes/ghc-slides.html#(16)

* Haskell is actually build on an extension called System F_C:
https://www.microsoft.com/en-us/research/wp-content/uploads/2007/01/tldi22-sulzmann-with-appendix.pdf

Implementing SystemF

- GHC Core Implementation:
  https://gitlab.haskell.org/ghc/ghc/-/blob/a1f34d37b47826e86343e368a5c00f1a4b1f2bce/compiler/GHC/Core.hs#L140
- Nice in-depth introductions into Haskell Core:
  https://www.youtube.com/watch?v=uR_VzYxvbxg
  http://www.scs.stanford.edu/11au-cs240h/notes/ghc-slides.html
- Many textbook implementations on GitHub
- E.g. https://github.com/Zepheus/SystemF/blob/master/systemf.hs

Algebraic Data Types across different PLs

```haskell
data Term =
  -- Simply typed lambda calculus:
  Var Symbol |
  Lambda Symbol Type Term |
  App Term Term |
  -- System F
  TLambda Type Term |
  TApp Term Type
  deriving (Show,Eq)
```

Haskell

??

```cpp
class AST {
  Node *root;
  VariablePool *varPool;
public:
  AST(Node *root);
  virtual ~AST();
  …
};
```

C++

From: https://github.com/omelkonian/lambda-calculus-interpreter/blob/master/abstract_syntax_tree/AST.h

System F in modern C++

- Use `std::variant` as our sum type
- Use `structs` as our product type
- Use `std::visit` to fake pattern matching

```
struct Var;
struct Lambda;
struct Apply;
struct TLambda;
struct TApply;

using Expr = std::variant<
    Var,
    Lambda,
    Apply,
    TLambda,
    TApply
    >;
```

- Caveat: fairly inefficient implementation …
  … but it's fun (and useful) to see the functional concepts shine through.

https://github.com/michel-steuwer/systemF_in_Cpp

Compiler transformations as rewrite rules

```
map f (map g xs) ≡ map (f . g) xs
```

```
{-# RULES
 “map/map” formal f g xs.
           map f (map g xs) = map (f . g) XS
#-}
```

Playing by the Rules: Rewriting a practical optimisation technique in GHC, *Simon P. Jones, Andrew Tolmach, Tony Hoare*

Compiler transformations as rewrite rules

- In which order apply the rules?
- Will the rewriting terminate? Is it confluence?
- Are the rules correct?

Haskell doesn't check this.

Proofing of rewrite rules not too difficult:

```
mapSplit : (n: ℕ) → {m: ℕ} → {s t: Set} → (f: s → t) → (xs: Vec s (m * n)) →
           map (map f) (split n {m} xs) ≡ split n {m} (map f xs)
simplification : (n: ℕ) → {m: ℕ} → {t: Set} → (xs: Vec t (m*n)) → (join ∘ split n {m}) xs ≡ xs
{- Split-join rule proof -}
splitJoin : {m: ℕ} → {s: Set} → {t: Set} → (n: ℕ) → (f: s → t) → (xs: Vec s (m * n)) →
            (join ∘ map (map f) ∘ split n {m}) xs ≡ map f xs
splitJoin {m} n f xs =
    begin
      (join ∘ map (map f) ∘ split n {m}) xs
    ≡⟨⟩
      join (map (map f) (split n {m} xs))
    ≡⟨ cong join (mapSplit n {m} f xs) ⟩
      join (split n {m} (map f xs))
    ≡⟨ simplification n {m} (map f xs) ⟩
      map f xs
    ■
```

Achieving High-Performance the Functional Way, *B. Hagedorn, J. Lenfers, T. Koehler, X. Qin, S. Gorlatch, M. Steuwer*

https://github.com/XYUnknown/individual-project/blob/master/src/lift/

References

- *Benjamin Pierce, Types and Programming Language*
- Martin Sulzmann, Manuel Chakravarty, Simon P. Jones, Kevin Donnelly, *System F with Type Equality Coercions* https://www.microsoft.com/en-us/research/wp-content/uploads/2007/01/tldi22-sulzmann-with-appendix.pdf
- Simon P Jones , *Into the Core - Squeezing Haskell into Nine Constructors* https://www.youtube.com/watch?v=uR_VzYxvbxg
- David Terei, *A Haskell Compiler* http://www.scs.stanford.edu/11au-cs240h/notes/ghc-slides.html#(1)
- Ben Deane, CppCon 2016: *Using Types Effectively* https://www.youtube.com/watch?v=ojZbFIQSdl8
- Tamir Bahar, *That `overloaded` Trick: Overloading Lambdas in C++17* https://dev.to/tmr232/that-overloaded-trick-overloading-lambdas-in-c17
- Simon P. Jones, Andrew Tolmach, Tony Hoare, *Playing by the Rules: Rewriting a practical optimisation technique in GHC* https://www.microsoft.com/en-us/research/wp-content/uploads/2001/09/rules.pdf
- B. Hagedorn, J. Lenfers, T. Koehler, X. Qin, S. Gorlatch, M. Steuwer, *Achieving High-Performance the Functional Way* https://bastianhagedorn.github.io/files/publications/2020/ICFP-2020.pdf