Microsoft Research

Microsoft

What We Talk About When We Talk About Types

Nick Benton

Types in everyday programming

```
- fun map f [] = []
    |   map f (x::xs) = f x :: map f xs;
val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

```
public static <T extends Comparable<? super T>>
        T max(Collection<T> coll) { … }
```

```
> [] + []

> [] + {}
[object Object]
> {}+[]
0
> {} + {}
Nan
```

```
CASE Ap:   { LET f, a = eval(H2!x, e), eval(H3!x, e)
             LET bv, body, env = H1!f, H2!f, H3!f
             RESULTIS eval(body, mk3(bv, a, env))
           }
```

We don't talk about types, we argue about them…

typed or untyped? (Or unityped?)

static

nomin

ong?

on-strict?

“Dynamic typing is but a special case of static typing, one that limits, rather than liberates, one that shuts down opportunities, rather than opening up new vistas.” - Harper

inferred or explicit?

latent?

type safe? type sound? memory safe?

For and against

- Static catches errors early, dynamic catches errors
- Static does away with runtime tests
- Static aids other static analysis and optimization
- IDE exploits types for completion, etc.
- Type info for garbage collection
- Can enforce security-critical invariants (e.g. JVM)
- Code can be generated or inferred from types
- Aids evolution, refactoring
- Documenting, communicating interfaces
- Mental scaffolding, blueprint during design
- Static too complex and bureaucractic
- Static too brittle, hinders "loose coupling"
- Too restrictive: "we don't need no stinkin' types"

In programming language conferences

- Polymorphism, modules, dependent types, refinements, overloading, subtyping, classes…
- Effect analysis, information flow, access control, communication protocols, lock usage, reactivity, distribution, data representation, staging, complexity
- Type theory and logic

$$\textsc{Sub-Exists}\quad \dfrac{\begin{array}{c} C\langle P_1,\ \ldots,\ P_m\rangle \text{ is a subclass of } D\langle \overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n\rangle \\ \Gamma, \Gamma \overset{\varnothing}{\longleftarrow} \Gamma' \vdash D\langle \overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_n\rangle[P_1 \mapsto \overset{\exists}{\tau}_1, \ldots, P_m \mapsto \overset{\exists}{\tau}_m] \approx_{\theta_i} D\langle \overset{\exists}{\tau}'_1,\ \ldots,\ \overset{\exists}{\tau}'_n\rangle \\ \text{for all } v' \text{ in } \Gamma',\ \text{exists } i \text{ in } 1 \text{ to } n \text{ with } \theta(v') = \theta_i(v') \\ \text{for all } i \text{ in } 1 \text{ to } n,\ \ \Gamma, \Gamma : \Delta, \Delta \vdash \overset{\exists}{\tau}_i[P_1 \mapsto \overset{\exists}{\tau}_1, \ldots, P_m \mapsto \overset{\exists}{\tau}_m] \cong \overset{\exists}{\tau}'_i[\theta] \\ \text{for all } v <:: \overset{\hat{\exists}}{\tau} \text{ in } \Delta',\ \ \Gamma, \Gamma : \Delta, \Delta \vdash \theta(v) <: \overset{\hat{\exists}}{\tau}[\theta] \\ \text{for all } v ::> \overset{\hat{\exists}}{\tau} \text{ in } \Delta',\ \ \Gamma, \Gamma : \Delta, \Delta \vdash \overset{\hat{\exists}}{\tau}[\theta] <: \theta(v) \end{array}}{\Gamma : \Delta \vdash \exists \Gamma : \Delta(\Delta).\ C\langle \overset{\exists}{\tau}_1,\ \ldots,\ \overset{\exists}{\tau}_m\rangle <: \exists \Gamma' : \Delta'(\Delta').\ D\langle \overset{\exists}{\tau}'_1,\ \ldots,\ \overset{\exists}{\tau}'_n\rangle}$$

The logical, proof-theoretic view, and propositions as types

$$\Gamma, x : A \vdash x : A \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.\, M : A \to B}$$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N\ : B}$$

propositions = types
proofs = terms

Logic	Types
$\supset$	$\to$
$\wedge$	$\times$
$\vee$	$+$

Proof normalization

- Simplify (identify) proofs by removal of "detours"
- Substitution lemma:

$$\frac{\Gamma, x{:}A \vdash M{:}B \qquad \Gamma \vdash N{:}A}{\Gamma \vdash M[N/x]{:}B}$$

- Now reduce intro/elim pairs

$$\frac{\dfrac{\Gamma, x{:}A \vdash M{:}B}{\Gamma \vdash \lambda x{:}A.\,M{:}A \to B} \qquad \Gamma \vdash N{:}A}{\Gamma \vdash (\lambda x{:}A.\,M)N{:}B}$$

- reduces to $\Gamma \vdash M[N/x]{:}B$
- proof simplification = beta reduction

Discussion

- Subject reduction (reduction preserves types)
- Strong normalization (all reduction sequences terminate, logical consistency)
- Sequent calculus presentations too (cut elimination)
- Very syntactic. Rules of the game given by beautiful symmetries, etc.
- Types are *intrinsic, prescriptive, synthetic* - "*Church style*". Ill-typed terms aren't considered.
- Hugely successful, influential approach
  - Generalizes to lots of other propositional logics (linear, S4 for staged computation, S5 for distribution, lax logic for monads, classical logic and control,…)
  - Also to richer logics, program extraction in dependent type theory
- Not so easy to extend to "real" PL type systems
- Analogy between proof simplification and operational semantics imperfect

Intrinsic models "bottom up"

- Interpret types as sets (objects)
- $[\![x_1 : A_1, \ldots, x_n : A_n]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$ (product)
- $[\![A \to B]\!] = [\![B]\!]^{[\![A]\!]}$ (set of functions, exponential)
- Interpret terms as functions (morphisms)
- $[\![\Gamma \vdash M : A]\!] : \; [\![\Gamma]\!] \to [\![A]\!]$
- Semantics models equations induced by beta & eta - equivalent proofs/terms interpreted by equal morphisms
- Denotational models in this general style do work for richer languages, even when the logical, proof-theoretic story breaks down

Extrinsic semantics & "well-typed programs don't go wrong"

- Quite different approach: programs come first
  - Give semantics to all type-free programs, which may involve some notion of dynamic error
- Types are *extrinsic, descriptive, analytic* properties of programs
- e.g. Milner starts with semantics of untyped CBV lambda calculus in a universal domain

$$V \cong \mathbb{N} + (V \to V_\perp) + \{wrong\}$$

- $[\![M\ N]\!]\rho = let\ f = [\![M]\!]\rho; v = [\![N]\!]\rho\ in\ apply(f, v)$
- where, e.g. $apply(in_1(n), v) = [in_3(wrong)]$
- Carves out meanings of types as certain subsets of V

Extrinsic models "top down"

- $[\![nat]\!] = \{in_1(n) | n \in \mathbb{N}\}$
- $[\![A \to B]\!] = \{in_2(f) | \forall v \in [\![A]\!], f v \in [\![B]\!]_\perp\}$
- Then not all elements have a type, some have more than one type (e.g. identity function)
- Give Curry-style type assignment for type-free terms

$$\frac{\Gamma, x: A \vdash M: B}{\Gamma \vdash \lambda x. B : A \to B}$$

- If $\Gamma \vdash M: A$ and $\rho \in [\![\Gamma]\!]$then $[\![M]\!]\rho \in [\![A]\!]_\perp$
- In particular, well-typed programs don't go wrong

Syntactic type soundness

- Can construct extrinsic models of types over operational semantics too
- but Wright and Felleisen ('94) came up with something simpler
  - Work with small step operational semantics
  - Define `proper' values (fully evaluated expressions)
  - Instead of explicitly saying $(3\ true) \rightarrow wrong$ just allow the semantics to get *stuck*, so $(3\ true) \nrightarrow$
  - Prove preservation, if $M: A$ and $M \rightarrow M'$ then $M': A$ (cf. subject reduction)
  - Prove progress, if $M: A$ and M is not a value, then $\exists M', M \rightarrow M'$
  - Hence, well-typed programs don't get stuck
- This is widely held to be *the* definition of type safety/soundness

Discussion

- It *is* simple, and superficially natural for simple types (think of writing an interpreter in ML)
- Only talks about specific type rules and internal details of specific operational semantics
- Have to extend typing rules to objects that only appear in operational semantics (heaps, stacks, pointers, configurations)
- For fancier types (effects, locks,…) have to *instrument* operational semantics, introducing new, fictitious stuck states that weren't there before
  - Gets silly, e.g. for TAL - machine code programs *don't* go wrong
- Never says what types *mean*, fails to capture compositional role as interface contracts (functions = functions?)
- Reduces static types to dynamic types

Intensional versus extensional

- What do we think we're doing when we write an operational semantics?
  - We're defining a language, but do we take the intermediate configurations seriously?
  - The compiler only cares about observable behaviour. If performs optimizing transformations and then emits machine code whose traces bear only a loose similarity with the original operational semantics
- Milner's semantics is mostly extensional. There are terms that inhabit a semantic type without being typable in the original system
  $if\ true\ then\ 3\ else\ false : nat$
- But still assumes dynamic test on summands of universal type

Parametricity and abstraction

- "Type structure is a syntactic discipline for enforcing levels of abstraction" - Reynolds
- Collection of techniques originating in study of abstract datatypes, representation independence and parametric polymorphism
  - What does it mean to say complex numbers are an abstract type?
  - When are two implementations of complex numbers equivalent?
  - In what sense do polymorphic functions behave "uniformly"?
- Central idea: go from types as subsets to types as relations (and type operators as operators on relations)

Free theorems

- Any function f of type $\forall X. X \to X$ is the identity
- $\forall A, B, R \subseteq A \times B, (\mathrm{a}, \mathrm{b}) \in R, (f_A a, f_B b) \in R$
  - Write $(f_A, f_B) \in R \to R$
- Any function of type $\forall X. List\ X \to List\ X$ just reorganizes its input in a fixed way
- $\forall A, B, R \subseteq A \times B, (as, bs) \in List\ R, (f_A as, f_B bs) \in List\ R$
- $\forall A, B, as{:}\ List\ A, h{:}\ A \to B, f_B (map\ h\ as) = map\ h\ (f_A as)$

Top-down relational models of types

- Carve out meanings of types as relations over an untyped model (these days, often just operational semantics)
- $[\![A \to B]\!] = [\![A]\!] \to [\![B]\!]$ (n.b. relational →!)
- $[\![\forall X. A]\!]\rho = \cap_R [\![A]\!]\rho[X \mapsto R]$
- Want type meanings to be *partial equivalence relations* (PERs)
- So subset of values together with a coarser notion of equality
- Defined together as values inhabiting compound types must respect equality on components

Discussion

- No need to ever talk about errors
- Relational semantics neither stronger nor weaker than syntactic safety
  - Syntactically untypable expressions can inhabit semantic types
  - Syntactically type-safe operations that break abstraction are ruled out
  - $\lambda f\!: nat \to nat.\, if\ f = (\lambda x.\, x)\ then\ 3\ else\ 4 \notin [\![(nat \to nat) \to nat]\!]$
- We get equational rules as well as inhabitation
- Traditionally started with system then looked for model, but these are the properties we wanted all along

Example: Information flow

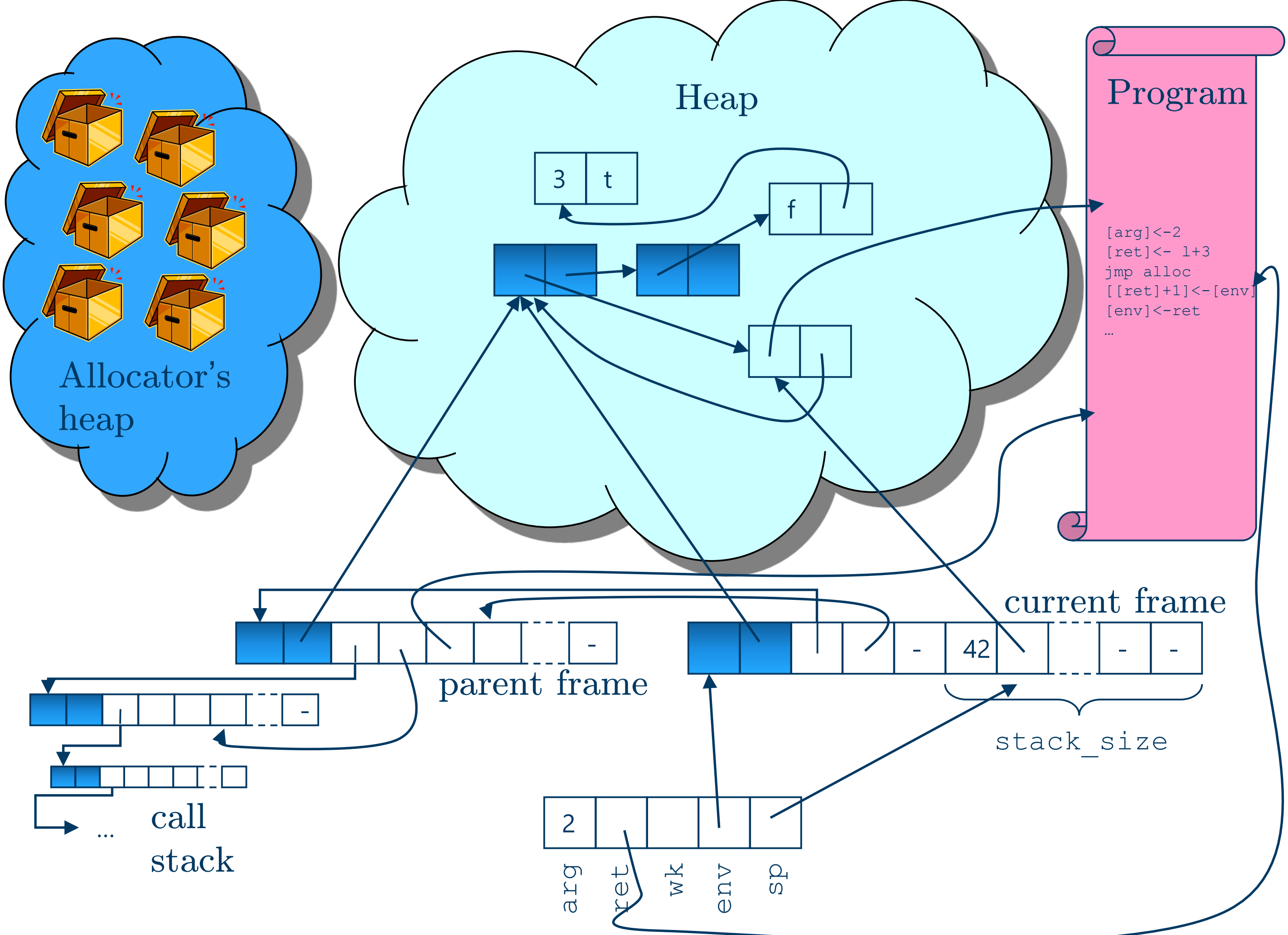
- Want to ensure no information flows from high-security variables to low-security ones
- This is not something one can naturally even explain in terms of runtime errors
- $[\![C]\!]: \mathbb{N}_h \times \mathbb{N}_l \to \mathbb{N}_\mathrm{h} \times \mathbb{N}_l$
- $\forall (n_h, n_l), (n'_h, n'_l), if\ n_l = n'_l\ then\ \pi_2 [\![C]\!](n_h, n_l) = \pi_2 [\![C]\!](n'_h, n'_l)$
- $([\![C]\!], [\![C]\!]) \in T \times \Delta \to T \times \Delta$
- There's a very natural relational logic that captures this and many other static analyses *and* the transformations they enable

Dimensions, etc.

- Kennedy. Incorporate physical dimensions (mass, length, time) into polymorphic type system that checks for dimensional consistency
  - real<d> is reals indexed by dimension d
  - Purely syntactically, this is interesting because there are equations on dimension expressions
  - Implemented in F#
- But what does it mean?
  - Nature doesn't carry dimension tags around and raise an exception if they don't match up
  - Essence of dimensional correctness is extensional, and rather beautiful: *invariance under scaling*
  - If $f: real < a > \to real < a^2 >$ then $\forall k > 0, x, f(k * x) = k^2 * f(x)$
- Relational semantics also gives (non)definability results
- Generalizes to e.g. geometry (invariance under transformations, AJK)
- And even to physics (laws of motion from conservation laws, Atkey)!

Compositional type soundness of compilers

- Express meaning of high-level types as relational, extensional constraints on the behaviour of compiled code
- What does it mean to say a word in memory contains an integer, Boolean, code pointer, data structure pointer?
- It's a constraint on what information code that uses it is allowed to depend on
- This way of doing things supports cross-language linking


Allocator's heap
Heap
3
t
f
Program
[arg]<-2
[ret]<- 1+3
jmp alloc
[[ret]+1]<-[env]
[env]<-ret
…
current frame
parent frame
42
stack_size
call stack
2
arg
ret
wk
env
sp

```
Fixpoint semantics_of_types (t:ExpType) (Ra:stateRel) ptr ptr' struct t :=
  match t with
  | Int P ⇒ lift (P ptr ∧ (ptr = ptr'))
  | Bool P ⇒ lift (P (n2b ptr) ∧ (n2b ptr = n2b ptr'))
  | a ∗ b  ⇒ Ex value, Ex value2, Ex value', Ex value2',
          (ptr,ptr'↦value,value') ×
           (ptr+1,ptr'+1↦value2,value2') × ⟦b⟧ Ra value value' × ⟦a⟧ Ra value2 value2')
  | a ⟶ b  ⇒ Ex Rprivate,
          (ptr,ptr' ↦ Later ( Perp (Pre_arrow Rprivate ptr ptr' Ra (⟦a⟧) (⟦b⟧))) × Rprivate)
  end
  where "'⟦' t '⟧'" := (semantics_of_types t ).


Definition Post_arrow b (Ra Rc: stateRel) Rc_cloud (n n' stack_ptr stack_ptr': nat):=
  Ex ptr_result, Ex ptr_result',
    (stack_ptr,stack_ptr' ↦ ptr_result,ptr_result') ⊗ (stack_ptr+1,stack_ptr'+1↦-) ⊗
    ((b Ra ptr_result ptr_result') × Rc_cloud) ⊗ Ra ⊗ Rc ⊗ (spreg↦ stack_ptr,stack_ptr') ⊗
    (envreg↦ n,n') ⊗ unused_space.

Definition Pre_arrow R_private ptr_function ptr_function' Ra a b:=
  Ex Rc, Ex Rc_cloud, Ex n, Ex n', Ex ptr_arg, Ex ptr_arg', Ex stack_ptr, Ex stack_ptr',
    (stack_ptr,stack_ptr'↦ ptr_arg,ptr_arg') ⊗
    (stack_ptr+1,stack_ptr'+1↦ ptr_function,ptr_function')
    ⊗ (R_private × a Ra ptr_arg ptr_arg' × Rc_cloud) ⊗
    ((n+4,n'+4 ↦ Later (Perp (Post_arrow b Ra Rc Rc_cloud n n' stack_ptr stack_ptr'))) × Rc) ⊗
    Ra ⊗ (spreg↦ stack_ptr+1,stack_ptr'+1) ⊗ (envreg↦ n,n') ⊗ unused_space.
```

Effect systems

$$\frac{\Theta, x : X \vdash M : T_\varepsilon Y}{\Theta \vdash \lambda x : U(X).M : X \to T_\varepsilon Y} \qquad \frac{\Theta \vdash V_1 : X \to T_\varepsilon Y \quad \Theta \vdash V_2 : X}{\Theta \vdash V_1\, V_2 : T_\varepsilon Y}$$

$$\frac{\Theta \vdash V : X}{\Theta \vdash \mathtt{val}\ V : T_\emptyset X} \qquad \frac{\Theta \vdash M : T_\varepsilon X \quad \Theta, x : X \vdash N : T_{\varepsilon'} Y}{\Theta \vdash \mathtt{let}\ x \Leftarrow M\ \mathtt{in}\ N : T_{\varepsilon \cup \varepsilon'} Y}$$

$$\frac{\Theta \vdash V : \mathtt{bool} \quad \Theta \vdash M : T_\varepsilon X \quad \Theta \vdash N : T_\varepsilon X}{\Theta \vdash \mathtt{if}\ V\ \mathtt{then}\ M\ \mathtt{else}\ N : T_\varepsilon X}$$

$$\frac{}{\Theta \vdash \mathtt{read}(\ell) : T_{\{\mathtt{r}_\ell\}}(\mathtt{int})} \qquad \frac{\Theta \vdash V : \mathtt{int}}{\Theta \vdash \mathtt{write}(\ell, V) : T_{\{\mathtt{w}_\ell\}}(\mathtt{unit})}$$

$$\frac{\Theta \vdash V : X \quad X \cdot X'}{\Theta \vdash V : X'} \qquad \frac{\Theta \vdash M : T_\varepsilon X \quad T_\varepsilon X \cdot T_{\varepsilon'} X'}{\Theta \vdash M : T_{\varepsilon'} X'}$$

Semantics of refined types

$$
\begin{aligned}
[\![X]\!] &\subseteq [\![U(X)]\!] \times [\![U(X)]\!] \\
[\![\texttt{int}]\!] &= \Delta_{\mathbb{Z}} \\
[\![\texttt{bool}]\!] &= \Delta_{\mathbb{B}} \\
[\![\texttt{unit}]\!] &= \Delta_{1} \\
[\![X \times Y]\!] &= [\![X]\!] \times [\![Y]\!] \\
[\![X \to T_\varepsilon Y]\!] &= [\![X]\!] \to [\![T_\varepsilon Y]\!] \\
[\![T_\varepsilon X]\!] &= \bigcap_{R \in \mathcal{R}_\varepsilon} R \to R \times [\![X]\!]
\end{aligned}
$$

Values of base type are related just to themselves (diagonal relation)

Functions are related in the usual "logical" fashion: related arguments → related results

Computations are related if they preserve all state relations that respect the effect

$$
\begin{aligned}
\mathcal{R}_\varepsilon, \mathcal{R}_e &\subseteq \mathbb{P}(S \times S) \\
\mathcal{R}_\varepsilon &= \bigcap_{e \in \varepsilon} \mathcal{R}_e \\
\mathcal{R}_{\texttt{r}_\ell} &= \{R \mid \forall (s, s') \in R,\ s\,\ell = s'\,\ell\} \\
\mathcal{R}_{\texttt{w}_\ell} &= \{R \mid \forall (s, s') \in R,\ \forall n \in \mathbb{Z}.\,(s[\ell \mapsto n], s'[\ell \mapsto n]) \in R\}
\end{aligned}
$$

Effect-dependent equivalences (1)

Dead Computation:

$$\frac{\Theta \vdash M : T_\varepsilon X \quad \Theta \vdash N : T_{\varepsilon'} Y}{\Theta \vdash \texttt{let}\, x \Leftarrow M \,\texttt{in}\, N = N : T_{\varepsilon'} Y}\; x \notin \Theta, \mathrm{wrs}(\varepsilon) = \emptyset$$

Duplicated Computation:

$$\frac{\Theta \vdash M : T_\varepsilon X \quad \Theta, x : X, y : X \vdash N : T_{\varepsilon'} Y}{\Theta \vdash \begin{array}{ll} & \texttt{let}\, x \Leftarrow M \,\texttt{in}\, \texttt{let}\, y \Leftarrow M \,\texttt{in}\, N \\ = & \texttt{let}\, x \Leftarrow M \,\texttt{in}\, N[x/y] \end{array} : T_{\varepsilon \cup \varepsilon'} Y}\; \mathrm{rds}(\varepsilon) \cap \mathrm{wrs}(\varepsilon) = \emptyset$$

Effect-dependent equivalences (2)

Commuting Computations:

$$\frac{\Theta \vdash M_1 : T_{\varepsilon_1} X_1 \quad \Theta \vdash M_2 : T_{\varepsilon_2} X_2 \quad \Theta, x_1 : X_1, x_2 : X_2 \vdash N : T_{\varepsilon'} Y}{\Theta \vdash \begin{array}{rl} & \texttt{let}\ x_1 \Leftarrow M_1\ \texttt{in}\ \texttt{let}\ x_2 \Leftarrow M_2\ \texttt{in}\ N \\ = & \texttt{let}\ x_2 \Leftarrow M_2\ \texttt{in}\ \texttt{let}\ x_1 \Leftarrow M_1\ \texttt{in}\ N \end{array} : T_{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon'} Y} \quad \begin{array}{l} \mathrm{rds}(\varepsilon_1) \cap \mathrm{wrs}(\varepsilon_2) = \emptyset \\ \mathrm{wrs}(\varepsilon_1) \cap \mathrm{rds}(\varepsilon_2) = \emptyset \\ \mathrm{wrs}(\varepsilon_1) \cap \mathrm{wrs}(\varepsilon_2) = \emptyset \end{array}$$

Pure Lambda Hoist:

$$\frac{\Theta \vdash M : T_{\{\}} Z \quad \Theta, x : X, y : Z \vdash N : T_{\varepsilon} Y}{\Theta \vdash \begin{array}{rl} & \texttt{val}\ (\lambda x : U(X).\texttt{let}\ y \Leftarrow M\ \texttt{in}\ N) \\ = & \texttt{let}\ y \Leftarrow M\ \texttt{in}\ \texttt{val}\ (\lambda x : U(X).N) \end{array} : T_{\{\}}(X \to T_{\varepsilon} Y)}$$

Summary

- Please stop doing syntactic type soundness proofs!
- Types are about abstractions not about errors
- Can make that precise using relational parametricity
- All types are abstract, all type systems about information flow
- This way of doing things works at multiple levels of abstraction, from source to machine code
- Recent work on relations for languages with store, control, polymorphism, generativity, concurrency
- Approach yields useful, deep results, including contextual equational laws

Thank you

Standard typing rules

$$\frac{\Gamma \vdash V_1 : A \quad \Gamma \vdash V_2 : B}{\Gamma \vdash (V_1, V_2) : A \times B} \qquad \frac{\Gamma \vdash V : A_1 \times A_2}{\Gamma \vdash \pi_i\, V : A_i}$$

$$\frac{\Gamma, x : A \vdash M : TB}{\Gamma \vdash \lambda x : A.M : A \to TB} \qquad \frac{\Gamma \vdash V_1 : A \to TB \quad \Gamma \vdash V_2 : A}{\Gamma \vdash V_1\, V_2 : TB}$$

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathtt{val}\ V : TA} \qquad \frac{\Gamma \vdash M : TA \quad \Gamma, x : A \vdash N : TB}{\Gamma \vdash \mathtt{let}\ x \Leftarrow M\ \mathtt{in}\ N : TB}$$

$$\frac{\Gamma \vdash V : \mathtt{bool} \quad \Gamma \vdash M : TA \quad \Gamma \vdash N : TA}{\Gamma \vdash \mathtt{if}\ V\ \mathtt{then}\ M\ \mathtt{else}\ N : TA}$$

$$\frac{}{\Gamma \vdash \mathtt{read}(\ell) : T\mathtt{int}} \qquad \frac{\Gamma \vdash V : \mathtt{int}}{\Gamma \vdash \mathtt{write}(\ell, V) : T\mathtt{unit}}$$

Base semantics in Set

$$
\begin{aligned}
S &= \mathrm{Locs} \to \mathbb{Z} \\
[\![\texttt{unit}]\!] &= 1 \\
[\![\texttt{int}]\!] &= \mathbb{Z} \\
[\![\texttt{bool}]\!] &= \mathbb{B} \\
[\![A \times B]\!] &= [\![A]\!] \times [\![B]\!] \\
[\![A \to TB]\!] &= [\![A]\!] \to [\![TB]\!] \\
[\![TA]\!] &= S \to S \times [\![A]\!]
\end{aligned}
$$

Refined types and subtyping

- Subtyping
  - Types

$$
\begin{array}{rcl}
X, Y & := & \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid X \times Y \mid X \to T_\varepsilon Y \\
\Theta & := & x_1 : X_1, \ldots, x_n : X_n \\
\varepsilon & \subseteq & \bigcup_{\ell \in \mathcal{L}} \{\mathtt{r}_\ell, \mathtt{w}_\ell\}
\end{array}
$$

$$
\frac{}{X \cdot \quad X} \qquad \frac{X \cdot \quad Y \qquad Y \cdot \quad Z}{X \cdot \quad Z} \qquad \frac{X \cdot \quad X' \qquad Y \cdot \quad Y'}{X \times Y \cdot \quad X' \times Y'}
$$

$$
\frac{X' \cdot \quad X \qquad T_\varepsilon Y \cdot \quad T_{\varepsilon'} Y'}{(X \to T_\varepsilon Y) \cdot \quad (X' \to T_{\varepsilon'} Y')} \qquad \frac{\varepsilon \subseteq \varepsilon' \qquad X \cdot \quad X'}{T_\varepsilon X \cdot \quad T_{\varepsilon'} X'}
$$

Results

- Soundness of subtyping: If $X \cdot \quad Y$ then $[\![X]\!] \subseteq [\![Y]\!]$.
  - Fundamental theorem:

    If $\Theta \vdash V : X, (\rho, \rho') \in [\![\Theta]\!]$
    then $([\![U(\Theta) \vdash V : U(X)]\!]\,\rho,\ [\![U(\Theta) \vdash V : U(X)]\!]\,\rho') \in [\![X]\!]$.
  - Meaning of top effect: $[\![G(A)]\!] = \Delta_{[\![A]\!]}$.
  - Equivalences
    - Effect-independent: congruence rules, β, η rules, commuting conversions
    - Effect-dependent: dead computation, duplicated computation, commuting computations, pure lambda hoist
    - Reasoning is quite intricate, involving construction of specific effect-respecting relations.