Polymorphic set-theoretic types for functional languages

type inference · gradual typing · non-strict languages

Tommaso Petrucciani

DIBRIS, Università di Genova, Italy · IRIF, Université Paris Diderot, France

14 March 2019

SUBJECT

Type systems for functional programming languages
using **polymorphic set-theoretic types**
and **semantic subtyping**

GOAL

Show how to use set-theoretic types
for a wider variety of programming languages features

type inference · **gradual typing** · **non-strict languages**

Set-theoretic types & semantic subtyping

TYPE SYSTEMS

- Static **type systems** check programs for type safety before they are executed

- **Sound:** well-typed programs do not "go wrong"
  but some correct programs are rejected too

- Goal for much research:
  make type systems more expressive and less intrusive

POLYMORPHISM

- It allows expressions to have more than one type
- **parametric polymorphism** $\lambda x.\, x \;:\; \forall \alpha.\, \alpha \to \alpha$
- **ad-hoc polymorphism (overloading)**
  $+ \;:\; \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$
  $+ \;:\; \mathsf{String} \times \mathsf{String} \to \mathsf{String}$
- **subtype polymorphism**

$$\{x : 3, y : \mathsf{true}\} \;:\; \{x : \mathsf{Int}, y : \mathsf{Bool}\} \le \{x : \mathsf{Int}\}$$
$$\{x : 2, z : 4\} \;:\; \{x : \mathsf{Int}, z : \mathsf{Int}\} \le \{x : \mathsf{Int}\}$$

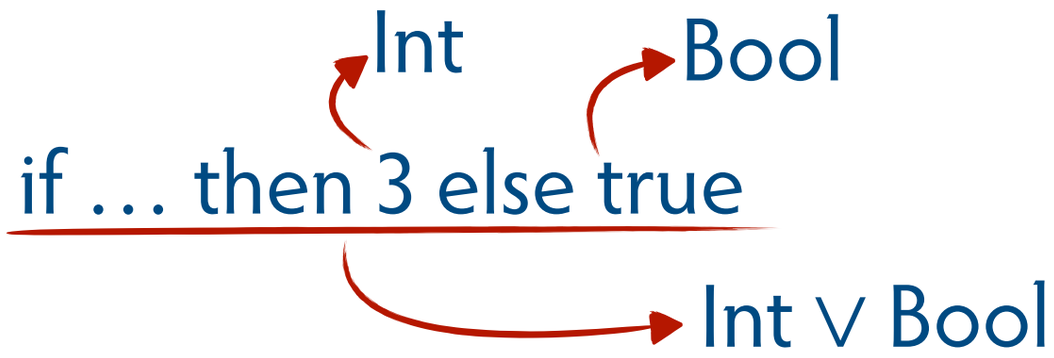
SET-THEORETIC TYPES

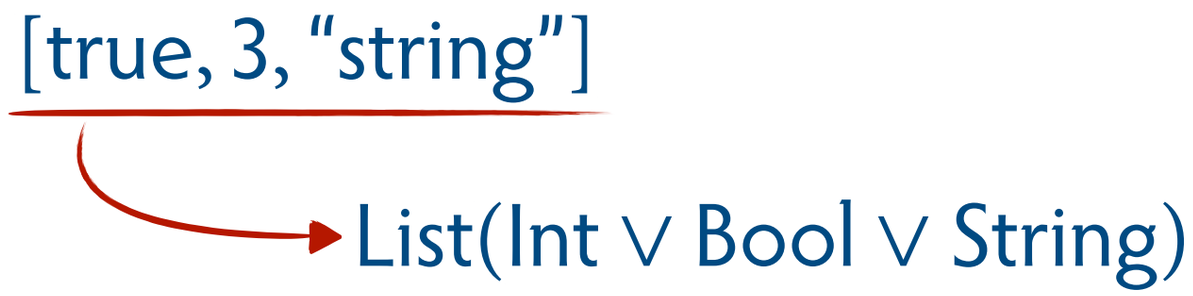
$$t ::= \cdots \mid t \vee t \mid t \wedge t \mid \neg t$$

Types which correspond to the operations of union, intersection, and complementation of sets:

union	$t \vee s$	either t or s
intersection	$t \wedge s$	both t and s
negation	$\neg t$	not t

UNION TYPES


Int
Bool
if … then 3 else true
Int ∨ Bool


[true, 3, “string”]
List(Int ∨ Bool ∨ String)

INTERSECTION TYPES

$$f \equiv \lambda x.\,(x \in \mathsf{Int})\ ?\ (x+1)\ :\ \neg x$$

$$(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool}) \leq (\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \vee \mathsf{Bool})$$

$$f\ 3 : \mathsf{Int}$$

$$f\ \mathsf{true} : \mathsf{Bool}$$

$$f\ (\text{if } \ldots \text{ then } 3 \text{ else true}) : \mathsf{Int} \vee \mathsf{Bool}$$

UNION, INTERSECTION, & NEGATION

$$\lambda x.\,(x \in \text{String})\ ?\ \text{string2int}(x)\ :\ x$$

$$\forall\alpha.\,(\text{String} \to \text{Int}) \wedge ((\alpha \setminus \text{String}) \to (\alpha \setminus \text{String}))$$

$$\forall\alpha.\,\alpha \to (\text{Int} \vee (\alpha \setminus \text{String}))$$

$(t \setminus s \stackrel{def}{=} t \wedge \neg s)$

e.g. red-black tree balancing function (Castagna et al., 2015):

$$\forall\alpha, \beta.\,(\text{Unbalanced}(\alpha) \to \text{Rtree}(\alpha)) \wedge (\beta \setminus \text{Unbalanced}(\alpha) \to \beta \setminus \text{Unbalanced}(\alpha))$$

SUBTYPING FOR SET-THEORETIC TYPES

- To explain intuitively set-theoretic types and subtyping, we can see types as sets of values

  e.g. from the documentation of the Flow type checker:

> A type like number, boolean, or string describes a set of possible values. […] If we want to know whether one type is the subtype of another, we need to look at all the possible values for both types and figure out if the other has a subset of the values.

but then, in Flow: $t_0 \wedge (t_1 \vee t_2) \gtrless (t_0 \wedge t_1) \vee (t_0 \wedge t_2)$

- **Semantic subtyping:** define subtyping using this intuition

SEMANTIC SUBTYPING

To define subtyping for set-theoretic types:

- define an **interpretation** of types as sets (of values)

$$[\![\cdot]\!] : \text{Types} \to \mathcal{P}(\text{Domain})$$

e.g.

$$[\![\text{Int}]\!] = \mathbb{Z} \qquad [\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$

$$[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!] \qquad [\![\neg t]\!] = \text{Domain} \setminus [\![t]\!]$$

- define **subtyping as set containment**

$$t_1 \leq t_2 \overset{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$$

USING SEMANTIC SUBTYPING

- First application: XML-processing functional languages
- Later: adapted to
  - more general-purpose language features
    (higher-order functions, parametric polymorphism)
  - different paradigms
    (object-oriented languages, process calculi)
- **Goal:** make them applicable to type various languages including dynamic ones like JavaScript

Implicit typing & type inference

- semantic subtyping for implicit typing
- type inference
- type inference + annotations

Partially based on work with G. Castagna and K. Nguyễn, presented at ICFP 2016.

IMPLICITLY TYPED LANGUAGE

- Previous work: languages with explicitly typed functions
- Here: an implicitly typed language

$$
\begin{aligned}
e ::= {} & x \mid \lambda x.\, e \mid e\ e \\
& \mid c \mid (e, e) \mid \pi_i\, e \\
& \mid (e \in \mathbf{t}\ ?\ e : e) \\
& \mid \text{let } x = e \text{ in } e
\end{aligned}
$$

typecase (on ground, non-arrow types)

- Standard call-by-value operational semantics

$$v ::= \lambda x.\, e \mid c \mid (v, v)$$

TYPE SYSTEM

$$t ::= \alpha \mid b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathsf{Empty} \mid \mathsf{Any}$$
$$b ::= \mathsf{Int} \mid \mathsf{Bool} \mid \cdots$$

$$\frac{\Gamma, x : t' \vdash e : t}{\Gamma \vdash \lambda x.\, e : t' \to t} \qquad \frac{\Gamma \vdash e_1 : t' \to t \quad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1\ e_2 : t}$$

$$\frac{}{\Gamma \vdash x : t[\vec{t}/\vec{\alpha}]}\ \Gamma(x) = \forall \vec{\alpha}.\, t \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x : \forall \vec{\alpha}.\, t_1 \vdash e_2 : t}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t}\ \vec{\alpha}\ \text{not in}\ \Gamma$$

$$\frac{\Gamma \vdash e : t'}{\Gamma \vdash e : t}\ t' \leq t \qquad (\wedge\text{-in})\ \frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2}$$

SUBJECT REDUCTION PROOF: DIFFICULTY

- Soundness: often proven as progress + **subject reduction**

$$\left.\begin{array}{l} \Gamma \vdash e : t \\ e \mapsto e' \end{array}\right\} \implies \Gamma \vdash e' : t$$

ground ground

- In our setting, subject reduction would require

$$\forall v \in \text{Values}, t \in \text{Types}. \quad \vdash v : t \;\text{ or }\; \vdash v : \neg t$$

ground

but: ~~$\not\vdash 3 : \alpha$~~ ~~$\not\vdash 3 : \neg\alpha$~~

$\not\vdash \lambda x.\, x : \text{Int} \to \text{Bool}$

$\not\vdash \lambda x.\, x : \neg(\text{Int} \to \text{Bool})$

$$\dfrac{\dfrac{\Gamma \vdash \lambda x.\,(x,x) : \big(t \to (t \times t)\big) \wedge \big(\neg t \to (\neg t \times \neg t)\big)}{\Gamma \vdash \lambda x.\,(x,x) : \big(t \vee \neg t\big) \to \big((t \times t) \vee (\neg t \times \neg t)\big)} \qquad \Gamma \vdash v : t \vee \neg t}{\Gamma \vdash \big(\lambda x.\,(x,x)\big)\, v : (t \times t) \vee (\neg t \times \neg t)}$$

FUNCTIONS & NEGATIONS OF ARROW TYPES

- We must ensure:

well-typed closed *ground*

$$\forall(\lambda x.\, e), (t_1 \to t_2).$$

$$\vdash \lambda x.\, e : t_1 \to t_2 \quad \text{or} \quad \vdash \lambda x.\, e : \neg(t_1 \to t_2)$$

- If functions had an explicit type $\mathcal{T}$
  - $\mathcal{T} \leq t_1 \to t_2$
  - $\mathcal{T} \not\leq t_1 \to t_2$

- Here we must know when a function **cannot** have a type

$$\frac{}{\Gamma \vdash \lambda x.\, e : \neg(t_1 \to t_2)} \quad \Gamma \not\vdash \lambda x.\, e : t_1 \to t_2$$

but this doesn't make sense!
solution: a stratification of type systems with a system for each size of expression

TYPE INFERENCE

- The previous presentation of the type system does not give an algorithm directly

- **Goal:** define a type inference algorithm for the implicitly typed language
  - build on previous work on constraint solving for set-theoretic types (Castagna et al., 2015)
  - correct a mistake in previous work on type inference (Castagna, Petrucciani, & Nguyễn, 2016)

TYPE INFERENCE: DIFFICULTIES

- Generalization for let is based on comparing variables that appear in types

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : \forall \vec{\alpha}. t_1 \vdash e_2 : t}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t} \; \vec{\alpha} \text{ not in } \Gamma$$

- This does not work well with semantic subtyping

$$[\![\text{Empty}]\!] = [\![\text{Empty} \wedge \alpha]\!] = [\![\alpha \setminus \alpha]\!]$$

- **Solution:** use a different type system, based on work by Dolan and Mycroft (2017)

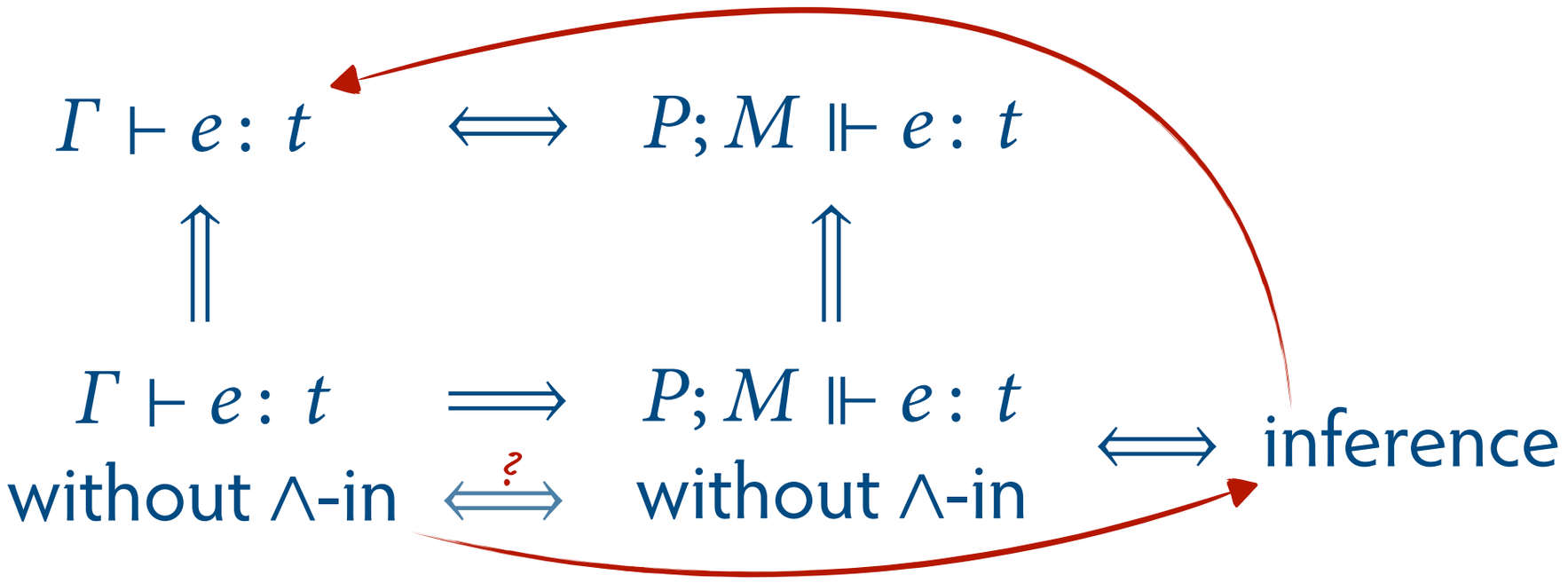
$$\lambda x. \text{let } f = \lambda y. (y, x) \text{ in } \ldots$$

(x : α, y : β)

~~$\forall \beta. \beta \to (\beta \times \alpha)$~~

$$\forall \alpha, \beta. \langle x : \alpha \rangle (\beta \to (\beta \times \alpha))$$

TYPE INFERENCE: OVERVIEW



Main ingredients

$\langle\!\langle e : t \rangle\!\rangle$

generates structured constraints: the conditions for e to have type t

$P; M; \sigma \Vdash C$

constraint satisfaction: semantics of constraints

structured constraints

simple constraints

$P \vdash C \rightsquigarrow D \mid M$

algorithmic constraint solving

reuses the existing "tallying" algorithm to solve

$D = \{(t_1^1 \mathrel{\dot{\leqslant}} t_1^2), \ldots, (t_n^1 \mathrel{\dot{\leqslant}} t_n^2)\}$

$\langle\!\langle e_1\ e_2 : t \rangle\!\rangle$

$\Vert$

$\exists \alpha.\ \langle\!\langle e_1 : \alpha \to t \rangle\!\rangle \wedge \langle\!\langle e_2 : \alpha \rangle\!\rangle$

TYPE INFERENCE + ANNOTATIONS

- Allow programmers to supply type annotations to obtain more precise types

- For example:

$$\lambda x.\, x \in \mathsf{Int}\ ?\ x + 1 : \neg x$$

$$\downarrow$$

$$(\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \vee \mathsf{Bool})$$

$$(\lambda x.\, x \in \mathsf{Int}\ ?\ x + 1 : \neg x) :: (\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$$

$$\downarrow$$

$$(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$$

- **Difficulty:** type annotations can contain variables and make the problems with generalization harder

SUMMARY & FUTURE WORK

- **Results:**
  - the type system is sound
  - type inference is sound and complete (except for ∧-in)
  - annotations make type inference more powerful

- **Open problems and future work:**
  - completeness of type inference with annotations
  - study language extensions,
    especially with record types and operations

Gradual typing

With G. Castagna, V. Lanvin, and J. Siek. Presented at POPL 2019.

INTRODUCTION TO GRADUAL TYPING

- Combine the advantages of static and dynamic typing
- **Goals:**
  - Fully untyped programs are allowed
  - Fully typed programs are allowed and proved safe
  - Typing the program can be a gradual process
  - (Runtime type errors are traced back to untyped code)

INTRODUCTION: TYPING

- The type "?" represents types that are **unknown** statically
- **Consistency** "$\sim$" is used to relax the type system

$$\frac{\Gamma \vdash e_1 : t' \to t \quad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1\ e_2 : t}$$

$$? \sim \mathsf{Int}$$
$$\mathsf{Int} \sim ?$$
$$\mathsf{Int} \not\sim \mathsf{Bool}$$
$$(? \to \mathsf{Int}) \sim (\mathsf{Bool} \to ?)$$

$$\frac{\Gamma \vdash e_1 : t' \to t \quad \Gamma \vdash e_2 : t''}{\Gamma \vdash e_1\ e_2 : t}\ t'' \sim t' \qquad \frac{\Gamma \vdash e_1 : ? \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\ e_2 : ?}$$

- Typing a program also produces a compiled version with runtime type tests

GRADUAL TYPING WITH SET-THEORETIC TYPES

- **Goal:** combine gradual typing + polymorphic set-theoretic types + type inference

- **Difficulty:** we cannot interpret as a set the type "?"
  - In particular, we need $? \wedge \neg ?$ to be non-empty

- **Our approach:** rely on existing definitions for polymorphic, non-gradual types
  - This also gives a novel approach to gradual typing for systems without set-theoretic types

OUR RESULTS: OVERVIEW

- Uniform approach to add gradual typing to
  1. Hindley-Milner polymorphism
  2. Hindley-Milner + simple subtyping
  3. Hindley-Milner + semantic subtyping

- We study:
  - declarative type systems and compilation
  - type inference for 1 and 3 — *sound & complete* / *sound*
  - semantics for the language with runtime type tests

OUR APPROACH: DECLARATIVE SYSTEM

For ~~subtyping~~ *gradual typing*,
we distinguish **declarative** and **algorithmic** systems

DECLARATIVE

$$\frac{\Gamma \vdash e_1 : t' \to t \quad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1\, e_2 : t} \qquad \frac{\Gamma \vdash e : t' \quad t' \sim t}{\Gamma \vdash e : t} \; t' \leq t$$

ALGORITHMIC

$$\frac{\Gamma \vdash e_1 : t' \to t \quad \Gamma \vdash e_2 : t''}{\Gamma \vdash e_1\, e_2 : t} \; t'' \sim t' \quad t'' \leq t'$$

OUR APPROACH: DECLARATIVE SYSTEM

- Consistency cannot be used in a structural rule because it's not transitive

- We use **precision** $\sqsubseteq$ instead to give a declarative system

$$\frac{}{? \sqsubseteq t} \qquad \frac{}{\mathsf{Int} \sqsubseteq \mathsf{Int}} \qquad \frac{t_1 \sqsubseteq t_1' \quad t_2 \sqsubseteq t_2'}{t_1 \to t_1' \sqsubseteq t_2 \to t_2'}$$

- Adding one rule is enough to make a type system gradual

$$\frac{\Gamma \vdash e : t' \quad \textit{compiles to } E}{\Gamma \vdash e : t} \; t' \sqsubseteq t \quad \textit{compiles to } E\langle t' \Rightarrow t \rangle$$

$$\frac{\Gamma \vdash e_1 : t' \to t \quad \Gamma \vdash e_2 : t''}{\Gamma \vdash e_1\, e_2 : t} \; t' \sim t'' \quad \left\{ \begin{array}{l} t' \sqsubseteq t''' \\ t'' \sqsubseteq t''' \end{array} \right\}$$

OUR APPROACH: SUBTYPING

- Semantic subtyping must be extended to consider "?"
- In some previous work, subtyping treats "?" as a base type

$$? \leq ? \qquad ? \to \mathsf{Nat} \leq ? \to \mathsf{Int} \qquad ? \not\leq \mathsf{Int}$$

- To achieve the same result, we transform gradual types to polymorphic types

$$? \leq ? \quad \textit{because} \quad \alpha \leq \alpha$$

$$? \to \mathsf{Nat} \leq ? \to \mathsf{Int} \quad \textit{because} \quad \alpha \to \mathsf{Nat} \leq \alpha \to \mathsf{Int}$$

$$? \not\leq \mathsf{Int} \quad \textit{because} \quad \alpha \not\leq \mathsf{Int}$$

but we don't want

$$? \wedge \neg ? \leq \mathsf{Empty}$$

SUMMARY & FUTURE WORK

- **Contributions:**
  - new declarative presentation of gradual typing
  - type inference reusing work for non-gradual systems
  - subtyping for gradual set-theoretic types
- **Future work for the system with set-theoretic types:**
  - show (a form of) completeness for type inference
  - add the rule $\wedge$-in

Semantic subtyping for non-strict languages

With D. Ancona, G. Castagna, and E. Zucca.
Submitted for publication in the post-proceedings of TYPES 2018.

STRICT VS NON-STRICT LANGUAGES

- **Strict** and **non-strict** (including **lazy evaluation**) are different strategies to evaluate programs

- In non-strict/lazy languages:

  - function arguments are evaluated only when needed

$$(\lambda x.\, x + 3)\ (1 + 2) \overset{\text{strict}}{\rightsquigarrow} (\lambda x.\, x + 3)\ 3 \rightsquigarrow 3 + 3$$
$$(\lambda x.\, x + 3)\ (1 + 2) \overset{\text{non-s.}}{\rightsquigarrow} (1 + 2) + 3 \rightsquigarrow 3 + 3$$

  - data can be accessed before it's fully evaluated

$$\pi_2\ (1 + 2, 4) \overset{\text{strict}}{\rightsquigarrow} \pi_2\ (3, 4) \rightsquigarrow 4$$
$$\pi_2\ (1 + 2, 4) \overset{\text{non-s.}}{\rightsquigarrow} 4$$

THE PROBLEM

- Semantic subtyping is **unsound** for non-strict languages
- The problem is the Empty type of diverging expressions

$$[\![\text{Empty}]\!] \stackrel{\text{def}}{=} \varnothing$$

- Let $\bar{e}$ be an (predictably) diverging expression

$$\frac{\dfrac{\vdash (\bar{e}, 3) : \text{Empty} \times \text{Int}}{\vdash (\bar{e}, 3) : \text{Empty} \times \text{Bool}}}{\vdash \pi_2\ (\bar{e}, 3) : \text{Bool}} \qquad \frac{\dfrac{\vdash (\lambda x.\, 3) : \text{Empty} \to \text{Int}}{\vdash (\lambda x.\, 3) : \text{Empty} \to \text{Bool}}}{\vdash (\lambda x.\, 3)\ \bar{e} : \text{Bool}}$$

because

$$[\![\text{Empty} \times \text{Int}]\!] = [\![\text{Empty} \times \text{Bool}]\!] = \varnothing$$

all types

$$\text{Empty} \to t$$

are equivalent

OUR APPROACH

- We can't use Empty as the type of diverging expressions
- We introduce a new type $\bot$

$$[\![\text{Empty}]\!] = \varnothing$$

no expression has this type (it's only used to describe subtyping)

$$[\![\bot]\!] = \{\bot\}$$

for diverging expressions

- Possibly diverging expressions have types like $\text{Int} \vee \bot$
- The rules assume that most expressions could diverge

$$\frac{\Gamma, x : t' \vdash e : t}{\Gamma \vdash \lambda x.\, e : t' \to t} \qquad \frac{\Gamma \vdash e_1 : (t' \to t) \vee \bot \quad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1\ e_2 : t \vee \bot}$$

CHALLENGES & RESULTS

- Set-theoretic types make the soundness proof non-trivial:
  - we need to add a union disjunction rule
  - we work with call-by-need
  - soundness for call-by-name does not hold if there is non-determinism
- **Results and limitations:**
  - sound type system based on semantic subtyping
  - the interpretation is not really “semantic”

Overall results

- Using semantic subtyping for implicitly typed languages
  - Type system and soundness proof
  - Type inference
  - Type inference with annotations
- Giving a new perspective on gradual typing
  - … for Hindley-Milner type systems
  - … for semantic subtyping
- Adapting semantic subtyping for non-strict semantics