Set-theoretic Foundation of Parametric Polymorphism and Subtyping

Giuseppe Castagna[1] and Zhiwu Xu[1,2]

[1]CNRS, Laboratoire Preuves, Programmes et Systèmes, Université Paris Diderot, Paris, France.

[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Science, Beijing, China

ICFP, Tokyo, 19th of September, 2011

Goal

1. Take your favorite **type constructors**

$$\times,\ \rightarrow,\ \{\dots\}, \mathtt{chan}(),\ \dots$$

Goal

1. Take your favorite **type constructors**

$$\times,\ \rightarrow,\ \{\dots\}, \mathtt{chan}(),\ \dots$$

2. add **Boolean connectives**:

$$\vee,\quad \wedge,\quad \neg$$

Goal

1. Take your favorite **type constructors**

$$\times,\ \rightarrow,\ \{\ldots\}, \mathtt{chan}(),\ \ldots$$

2. add **Boolean connectives**:

$$\vee,\quad \wedge,\quad \neg$$

3. add **type variables**

$$\alpha, \beta, \gamma, \ldots$$

Goal

1. Take your favorite **type constructors**

$$\times,\ \rightarrow,\ \{\dots\}, \texttt{chan}(),\ \dots$$

2. add **Boolean connectives**:

$$\vee,\quad \wedge,\quad \neg$$

3. add **type variables**

$$\alpha, \beta, \gamma, \ldots$$

4. give an intuitive (*ie*, set-theoretic) semantics so as to deduce
   - classic distribution laws (for all α, β, γ)

$$((\alpha \vee \beta) \times \gamma) \lesseqgtr (\alpha \times \gamma) \vee (\beta \times \gamma)$$

Goal

1. Take your favorite **type constructors**

$$\times,\ \rightarrow,\ \{\dots\}, \mathtt{chan}(),\ \dots$$

2. add **Boolean connectives**:

$$\vee,\quad \wedge,\quad \neg$$

3. add **type variables**

$$\alpha, \beta, \gamma, \ldots$$

4. give an intuitive (*ie*, set-theoretic) semantics so as to deduce
   - classic distribution laws (for all α, β, γ)

   $$((\alpha\vee\beta)\times\gamma) \lesseqgtr (\alpha\times\gamma) \vee (\beta\times\gamma)$$

   - data structure containments (for all α):

   $$\underbrace{\mu t.(\alpha\times(\alpha\times t))\vee \mathsf{nil}}_{\alpha\text{-lists of even length}} \leq \underbrace{\mu t.(\alpha\times t)\vee \mathsf{nil}}_{\alpha\text{-lists}}$$

Goal

1. Take your favorite **type constructors**

$$\times,\ \rightarrow,\ \{\dots\}, \mathtt{chan}(),\ \dots$$

2. add **Boolean connectives**:

$$\vee,\quad \wedge,\quad \neg$$

3. add **type variables**

$$\alpha, \beta, \gamma, \dots$$

4. give an intuitive (*ie*, set-theoretic) semantics so as to deduce
   - classic distribution laws (for all α, β, γ)

$$((\alpha \vee \beta) \times \gamma) \lesseqgtr (\alpha \times \gamma) \vee (\beta \times \gamma)$$

   - data structure containments (for all α):

$$\underbrace{\mu t.(\alpha \times (\alpha \times t)) \vee \mathrm{nil}}_{\alpha\text{-lists of even length}} \leq \underbrace{\mu t.(\alpha \times t) \vee \mathrm{nil}}_{\alpha\text{-lists}}$$

WHY?

WHY? briefly:

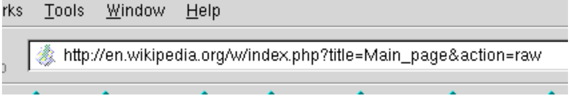
WHY? briefly:

1. **Boolean connectives:**
   - Unions, products and recursive types encode regular trees and therefore **XML**
   - Intersection and negation permit XML typed programming with **overloading** and powerful **pattern matching**.

WHY? briefly:

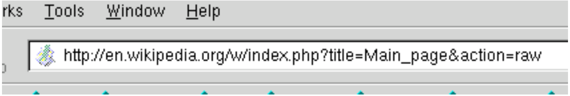
1. **Boolean connectives:**
    - Unions, products and recursive types encode regular trees and therefore **XML**
    - Intersection and negation permit XML typed programming with **overloading** and powerful **pattern matching**.
2. **Type variables:**
    - Parametric polymorphism already demonstrated its worth in practice.
    - Fulfills new needs specific to XML processing (*eg*, SOAP envelopes).
    - Sheds new light on the notion of **parametricity**.

Real case example: active pages



To create a *dynamically* generated page in the *Ocsigen* web development systems:

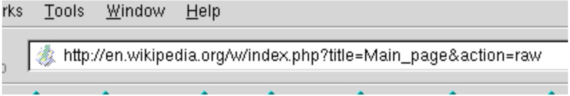
Real case example: active pages



To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to Xhtml:
   `let page_fun(p: {title: string, ...}) : Xhtml = ...`

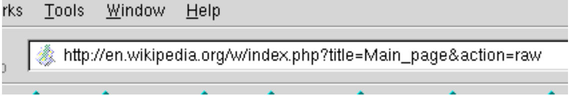
Real case example: active pages



To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to Xhtml:
   `let page_fun(p: {title: string, ...}) : Xhtml = ...`
2. bind `page_fun` to the path `$WEBROOT/w/index` by:
   `register_new_service(page_fun,"w/index")`

Real case example: active pages



To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to Xhtml:
   `let page_fun(p: {title: string, ...}) : Xhtml = ...`
2. bind `page_fun` to the path `$WEBROOT/w/index` by:

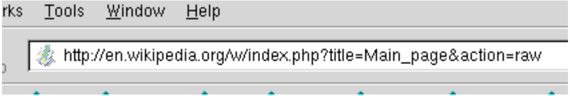
   `register_new_service``(page_fun,"w/index")`

The (wished) type of `register_new_service` is

$$\forall(\mathtt{X} \leq \mathtt{Params}).((\mathtt{X} \to \mathtt{Xhtml}) \times \mathtt{Path}) \to \mathtt{unit}$$

where `Params` is a specification of all possible query strings

Real case example: active pages



To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to Xhtml:
   `let page_fun(p: {title: string, ...}) : Xhtml = ...`
2. bind `page_fun` to the path `$WEBROOT/w/index` by:

   `register_new_service``(page_fun,"w/index")`

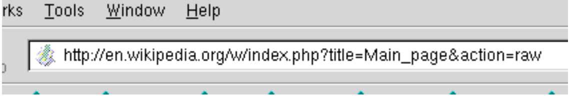
XML types

The (wished) type of `register_new_service` is

$$\forall(\mathtt{X} \leq \mathtt{Params}).((\mathtt{X} \rightarrow \mathtt{Xhtml}) \times \mathtt{Path}) \rightarrow \mathtt{unit}$$

where `Params` is a specification of all possible query strings

Real case example: active pages



To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to Xhtml:
   `let page_fun(p: {title: string, ...}) : Xhtml = ...`
2. bind `page_fun` to the path `$WEBROOT/w/index` by:

   `register_new_service``(page_fun,"w/index")`

XML types

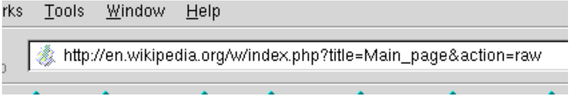
The (wished) type of `register_new_service` is

$$\forall(\mathtt{X} \leq \mathtt{Params}).((\mathtt{X} \rightarrow \mathtt{Xhtml}) \times \mathtt{Path}) \rightarrow \mathtt{unit}$$

higher-order functions

where `Params` is a specification of query strings

Real case example: active pages



To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to Xhtml:
   `let page_fun(p: {title: string, ...}) : Xhtml = ...`
2. bind `page_fun` to the path `$WEBROOT/w/index` by:

   register_new_service`(page_fun,"w/index")`

type variables

XML types

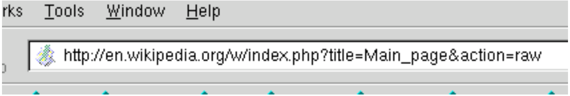
The ... ter_n... ice is

$$\forall(X \le \mathtt{Params}).((X \to \mathtt{Xhtml}) \times \mathtt{Path}) \to \mathtt{unit}$$

higher-order functions

where `Params` is a specificati... query strings

Real case example: active pages



To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to Xhtml:
   `let page_fun(p: {title: string, ...}) : Xhtml = ...`
2. bind `page_fun` to the path `$WEBROOT/w/index` by:

   register_new_service(page_fun,"w/index")

type variables

XML types

The ... ster_ne... ice is

$$\forall(X \leq \texttt{Params}).((X \rightarrow \texttt{Xhtml}) \times \texttt{Path}) \rightarrow \texttt{unit}$$

bounded quantification

higher-order functions

a specificati... query strings

Real case example: …e pages

To crea… developm…

1. de… : Xhtml = ...
2. …to …$WEB… …x by:

register_new_service(page_fun,"w/index")

Since all these features are not available Ocsigen's type system must be unplugged

The (…) type of …ster_n… …ice is

$$\forall(X \leq \mathtt{Params}).((X \rightarrow \mathtt{Xhtml}) \times \mathtt{Path}) \rightarrow \mathtt{unit}$$

type variables

XML types

bounded quantification

higher-order functions

… a specificati… …query strings

Current status

Study of a type system of (recursive/regular) types with

$$t ::= B \mid t{\times}t \mid t{\rightarrow}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$$

type constructors **logical connectives** **type variables**

Current status

Study of a type system of (recursive/regular) types with

$$t ::= B \mid t\times t \mid t\rightarrow t \mid t\vee t \mid t\wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$$

type constructors **logical connectives** type variables

- **Logical connectives:** Well-known how to implement a functional language with pattern-matching, higher-order functions, and *connectives with set theoretic interpretation*.

Current status

Study of a type system of (recursive/regular) types with

$$t ::= B \mid t{\times}t \mid t{\rightarrow}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$$

type constructors **logical connectives** type variables

- **Logical connectives:** Well-known how to implement a functional language with pattern-matching, higher-order functions, and *connectives with set theoretic interpretation*.

Semantic subtyping
(implemented by the language $\mathbb{C}$Duce).

Current status

Study of a type system of (recursive/regular) types with

$$t ::= B \mid t{\times}t \mid t{\to}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \boldsymbol{\alpha}$$

type constructors **logical connectives** **type variables**

- **Logical connectives:** Well-known how to implement a functional language with pattern-matching, higher-order functions, and *connectives with set theoretic interpretation*.

  Semantic subtyping
  (implemented by the language $\mathbb{C}$Duce).

- **Type variables:** A set-theoretic approach was deemed unfeasible or even impossible:

Current status

Study of a type system of (recursive/regular) types with

$$t ::= B \mid t\times t \mid t\rightarrow t \mid t\vee t \mid t\wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$$

type constructors   logical connectives   **type variables**

- **Logical connectives:** Well-known how to implement a functional language with pattern-matching, higher-order functions, and *connectives with set theoretic interpretation*.

  Semantic subtyping
  (implemented by the language ℂDuce).

- **Type variables:** A set-theoretic approach was deemed unfeasible or even impossible:

  This work
  (built on the work of semantic subtyping)

Semantic Subtyping in a nutshell

Semantic subtyping

$$t ::= B \mid t{\times}t \mid t{\rightarrow}t \mid t{\vee}t \mid t{\wedge}t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$

Semantic subtyping

$$t ::= B \mid t\times t \mid t\rightarrow t \mid t\vee t \mid t\wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$

- Constructor subtyping is *easy*:
  constructors do not mix, *eg.*:

$$\frac{s_2 \leq s_1 \qquad t_1 \leq t_2}{s_1\rightarrow t_1 \leq s_2\rightarrow t_2}$$

Semantic subtyping

$$t \ ::= \ B \ | \ t{\times}t \ | \ t{\rightarrow}t \ | \ t{\vee}t \ | \ t{\wedge}t \ | \ \neg t \ | \ \mathbb{0} \ | \ \mathbb{1}$$

- Constructor subtyping is *easy*:
  constructors do not mix, *eg.*:

$$\frac{s_2 \leq s_1 \qquad t_1 \leq t_2}{s_1{\rightarrow}t_1 \leq s_2{\rightarrow}t_2}$$

- Connective subtyping is *harder*:
  connectives distribute over *constructors*, *eg.*

$$(s_1{\vee}s_2){\rightarrow}t \quad \substack{\geq \\ \leq} \quad (s_1{\rightarrow}t){\wedge}(s_2{\rightarrow}t)$$

Semantic subtyping

$$t \ ::= \ B \ | \ t{\times}t \ | \ t{\rightarrow}t \ | \ t{\vee}t \ | \ t{\wedge}t \ | \ \neg t \ | \ \mathbb{0} \ | \ \mathbb{1}$$

- Constructor subtyping is *easy*:
  constructors do not mix, *eg.*:

$$\frac{s_2 \leq s_1 \qquad t_1 \leq t_2}{s_1{\rightarrow}t_1 \leq s_2{\rightarrow}t_2}$$

- Connective subtyping is *harder*:
  connectives distribute over *constructors*, *eg.*

$$(s_1{\vee}s_2){\rightarrow}t \quad \gtreqless \quad (s_1{\rightarrow}t){\wedge}(s_2{\rightarrow}t)$$

Define subtyping semantically: [Hosoya, Pierce]

1. Interpret types as sets (of values)
2. *Define* subtyping as set containment.

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have their natural interpretation:

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$
$$[\![t_1 \to t_2]\!] = \{f \mid f \text{ function from} [\![t_1]\!] \text{ to } [\![t_2]\!]\}$$

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

  such that
  - **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

  - **Constructors** have their natural interpretation:

$$\begin{aligned} [\![t_1 \times t_2]\!] &= [\![t_1]\!] \times [\![t_2]\!] \\ [\![t_1 \to t_2]\!] &= \{f \mid f \text{ function from } [\![t_1]\!] \text{ to } [\![t_2]\!]\} \end{aligned}$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \quad \overset{def}{\iff} \quad [\![s]\!] \subseteq [\![t]\!]$$

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have their natural interpretation:

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \qquad \mathcal{D}^2 \subseteq \mathcal{D}$$
$$[\![t_1 \to t_2]\!] = \{f \mid f \text{ function from } [\![t_1]\!] \text{ to } [\![t_2]\!]\} \qquad \mathcal{D}^{\mathcal{D}} \subseteq \mathcal{D}$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \overset{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

  such that

  - **Connectives** have their set-theoretic interpretation:

    $[\![\mathbb{0}]\!] = \varnothing$ $\qquad$ $[\![t_1 \vee t_2$

    $[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!]$ $\qquad$ $[\![t_1 \wedge t_2$

  - **Constructors** have their natural interpretation:

$$\begin{array}{lcl} [\![t_1 \times t_2]\!] & = & [\![t_1]\!] \times [\![t_2]\!] \\ [\![t_1 \to t_2]\!] & = & \{ f \mid f \text{ function from } [\![t_1]\!] \text{ to } [\![t_2]\!] \} \end{array}$$

cardinality problem

$\mathcal{D}^{\mathcal{D}} \subseteq \mathcal{D}$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \ \stackrel{def}{\iff} \ [\![s]\!] \subseteq [\![t]\!]$$

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$[\![\mathbb{0}]\!] = \varnothing$ $\qquad$ $[\![t_1 \vee t_2$

$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!]$ $\qquad$ $[\![t_1 \wedge t_2$

cardinality problem

- **Constructors** have their natural interpretation:

$$\begin{array}{lcl} [\![t_1 \times t_2]\!] & = & [\![t_1]\!] \times [\![t_2]\!] \\ [\![t_1 \to t_2]\!] & = & \{ f \mid f \text{ function from} [\![t_1]\!] \text{ to } [\![t_2]\!] \} \end{array}$$

$\mathcal{D}^{\mathcal{D}} \subseteq \mathcal{D}$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \quad \stackrel{def}{\Longleftrightarrow} \quad [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types are
define how they are related

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\;]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$

$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have their natural interpretation:

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$

$$[\![t_1 \to t_2]\!] = \{f \mid f \text{ function from } [\![t_1]\!] \text{ to } [\![t_2]\!]\}$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \overset{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types are
define how they are related

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

  - **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

  - **Constructors** have their natural interpretation:

$$\begin{array}{lcl} [\![t_1 \times t_2]\!] & = & [\![t_1]\!] \times [\![t_2]\!] \\ [\![t_1 \to t_2]\!] & = & \{f \subseteq \mathcal{D}^2 \mid (d_1, d_2) \in f, d_1 \in [\![t_1]\!] \Rightarrow d_2 \in [\![t_2]\!]\} \end{array}$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \stackrel{def}{\Longleftrightarrow} [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types are
define how they are related

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$

$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have their natural interpretation:

$$\begin{aligned} [\![t_1 \times t_2]\!] &= [\![t_1]\!] \times [\![t_2]\!] \\ [\![t_1 \to t_2]\!] &= \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}}) \end{aligned}$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \stackrel{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types are
define how they are related

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have **their natural interpretation**:

$$\begin{array}{rcl} [\![t_1 \times t_2]\!] & = & [\![t_1]\!] \times [\![t_2]\!] \\ [\![t_1 \to t_2]\!] & = & \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}}) \end{array}$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \quad \overset{def}{\Longleftrightarrow} \quad [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types are
define how they are related

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

  such that

  - **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$

$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

  - **Constructors** have **the same** $\subseteq$ **as** their natural interpretation:

$$[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$

$$[\![t_1 \to t_2]\!] = \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \quad \stackrel{def}{\iff} \quad [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types are
define how they are related

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have **the same $\subseteq$ as** their natural interpretation:

$$[\![s_1 \times s_2]\!] \subseteq [\![t_1 \times t_2]\!] \iff \underline{[\![s_1]\!] \times [\![s_2]\!] \subseteq [\![t_1]\!] \times [\![t_2]\!]}$$
$$[\![s_1 \to s_2]\!] \subseteq [\![t_1 \to t_2]\!] \iff \mathcal{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}}) \subseteq \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \overset{def}{\iff} [\![s]\!] \subseteq [\![t]\!]$$

Key idea

Do not define what types are
define how they are related

Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})$$

such that

- **Connectives** have their set-theoretic interpretation:

$$[\![\mathbb{0}]\!] = \varnothing \qquad [\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$$
$$[\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$$

- **Constructors** have **the same** $\subseteq$ **as their natural interpretation**:

$$[\![s_1 \times s_2]\!] \subseteq [\![t_1 \times t_2]\!] \iff [\![s_1]\!] \times [\![s_2]\!] \subseteq [\![t_1]\!] \times [\![t_2]\!]$$
$$[\![s_1 \to s_2]\!] \subseteq [\![t_1 \to t_2]\!] \iff \overline{\mathcal{P}([\![s_1]\!] \times \overline{[\![s_2]\!]})} \subseteq \overline{\mathcal{P}([\![t_1]\!] \times \overline{[\![t_2]\!]})}$$

- **Then** *define* the **subtyping relation** as set-containment.

$$s \leq t \quad \overset{def}{\iff} \quad [\![s]\!] \subseteq [\![t]\!]$$

Semantic subtyping [Benzaken, Castagna, Frisch]

1. Gives an interpretation satisfying the above constraints;
2. Gives an algorithm to decide the induced subtyping relation.

Polymorphic extension: adding type variables

Naive solution

$$t ::= B \mid t \times t \mid t \to t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$$

Naive solution

$$t \ ::= \ B \ \mid \ t{\times}t \ \mid \ t{\rightarrow}t \ \mid \ t{\vee}t \ \mid \ t{\wedge}t \ \mid \ \neg t \ \mid \ \mathbb{0} \ \mid \ \mathbb{1} \ \mid \ \alpha$$

Naive solution

$$t \ ::= \ B \ | \ t{\times}t \ | \ t{\rightarrow}t \ | \ t{\vee}t \ | \ t{\wedge}t \ | \ \neg t \ | \ \mathbb{0} \ | \ \mathbb{1} \ | \ \alpha$$

Idea: Use the previous relation since is defined for "ground types"

Let σ : **Vars** $\rightarrow$ **ClosedTypes** denote ground substitutions. Define:

$$s \leq t \quad \overset{def}{\Longleftrightarrow} \quad \forall\sigma \,.\, s\sigma \leq t\sigma$$

Naive solution

$$t ::= B \mid t \times t \mid t \rightarrow t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$$

Idea: Use the previous relation since is defined for "ground types"

Let σ : **Vars** $\rightarrow$ **ClosedTypes** denote ground substitutions. Define:

$$s \leq t \quad \stackrel{def}{\Longleftrightarrow} \quad \forall \sigma \,.\, s\sigma \leq t\sigma$$

or equivalently

$$s \leq t \quad \stackrel{def}{\Longleftrightarrow} \quad \forall \sigma . [\![s\sigma]\!] \subseteq [\![t\sigma]\!]$$

Naive solution

$$t \ ::= \ B \ | \ t\times t \ | \ t\rightarrow t \ | \ t\vee t \ | \ t\wedge t \ | \ \neg t \ | \ \mathbb{0} \ | \ \mathbb{1} \ | \ \alpha$$

Idea: Use the previous relation since is defined for "ground types"

Let σ : **Vars** $\rightarrow$ **ClosedTypes** denote ground substitutions. Define:

$$\cancel{s \leq t \quad \stackrel{def}{\iff} \quad \forall\sigma\,.\,s\sigma \leq t\sigma}$$

or equivalently

$$\cancel{s \leq t \quad \stackrel{def}{\iff} \quad \forall\sigma.[\![s\sigma]\!] \subseteq [\![t\sigma]\!]}$$

THIS IS A WRONG WAY: TOO MANY PROBLEMS

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\,s\sigma \leq t\sigma$ is *at least* as hard as solving Diophantine equations

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\,s\sigma \leq t\sigma$ is *at least* as hard as solving Diophantine equations

2. **It breaks parametricity:**

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma \,.\, s\sigma \leq t\sigma$ is *at least* as hard as solving Diophantine equations

2. **It breaks parametricity:**

$$(t\times\alpha) \quad\leq\quad (t\times\neg t)\vee(\alpha\times t) \qquad (1)$$

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\,s\sigma \leq t\sigma$ is *at least* as hard as solving Diophantine equations

2. **It breaks parametricity:**

$$(t{\times}\alpha) \quad \leq \quad (t{\times}\neg t)\vee(\alpha{\times}t) \qquad (1)$$

This inclusion holds if and only if t is an *indivisible* type (*eg.*, a singleton or a basic type):

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma\,.\,s\sigma \leq t\sigma$ is *at least* as hard as solving Diophantine equations

2. **It breaks parametricity:**

$$(t{\times}\alpha) \quad \leq \quad (t{\times}\neg t)\vee(\alpha{\times}t) \tag{1}$$

This inclusion holds if and only if t is an *indivisible* type (*eg.*, a singleton or a basic type):

Property of indivisible types

If t is an *indivisible type*, then for all possible interpretations of α

$$t \leq \alpha \qquad \text{or} \qquad \alpha \leq \neg t$$

holds.

Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall\sigma \,.\, s\sigma \leq t\sigma$ is *at least* as hard as solving Diophantine equations

2. **It breaks parametricity:**

$$(t\times\alpha) \quad\leq\quad (t\times\neg t)\vee(\alpha\times t) \qquad (1)$$

This inclusion holds if and only if t is an *indivisible* type (*eg*., a singleton or a basic type):

Property of indivisible types

If t is an *indivisible type*, then for all possible interpretations of α

$$t\leq\alpha \qquad \text{or} \qquad \alpha\leq\neg t$$

holds.

- If $\alpha\leq\neg t$ then the left element of the union in (18) suffices;
- If $t\leq\alpha$, then $\alpha=(\alpha\backslash t)\vee t$. Thus $(t\times\alpha)=(t\times(\alpha\backslash t))\vee(t\times t)$. This union is contained component-wise in the one in (18).

Problems with the naive solution

The fact that

$$(t{\times}\alpha) \quad\leq\quad (t{\times}\neg t)\vee(\alpha{\times}t)$$

holds if and only if t is *indivisible* is really catastrophic:

Problems with the naive solution

The fact that

$$(t\times\alpha) \quad \leq \quad (t\times\neg t)\vee(\alpha\times t)$$

holds if and only if t is *indivisible* is really catastrophic:

- Deciding subtyping needs deciding indivisibility ... which is very hard.

Problems with the naive solution

The fact that

$$(t \times \alpha) \quad \leq \quad (t \times \neg t) \vee (\alpha \times t)$$

holds if and only if t is *indivisible* is really catastrophic:

- Deciding subtyping needs deciding indivisibility ... which is very hard.
- **This subtyping relation breaks parametricity**:
by subsumption a function generic in its first argument, becomes generic on its second argument.

Problems with the naive solution

The fact that

$$(t\times\alpha) \quad\leq\quad (t\times\neg t)\vee(\alpha\times t)$$

holds if and only if t is *indivisible* is really catastrophic:

- Deciding subtyping needs deciding indivisibility ... which is very hard.
- **This subtyping relation breaks parametricity**:
  by subsumption a function generic in its first argument, becomes generic on its second argument.

- A semantic solution was deemed unfeasible (even w/o arrows)
- Problem eschewed by resorting to syntactic solutions: [Hosoya, Frisch, Castagna: POPL 05], [Vouillon: POPL 06].

Problems with the naive solution

The fact that

$$(t\times\alpha) \quad\leq\quad (t\times\neg t)\vee(\alpha\times t)$$

holds if and only if t is *indivisible* is really catastrophic:

- Deciding subtyping needs deciding indivisibility ... which is very hard.
- **This subtyping relation breaks parametricity**: by subsumption a function generic in its first argument, becomes generic on its second argument.

- A semantic solution was deemed unfeasible (even w/o arrows)
- Problem eschewed by resorting to syntactic solutions: [Hosoya, Frisch, Castagna: POPL 05], [Vouillon: POPL 06].

A SEMANTIC SOLUTION IS POSSIBLE

A semantic solution

A faint intuition

The loss of parametricity is only due to the interpretation of indivisible types, all the rest works (more or less) smoothly

A semantic solution

A faint intuition

The loss of parametricity is only due to the interpretation of indivisible types, all the rest works (more or less) smoothly

The crux of the problem is that for an indivisible type $\boldsymbol{i}$

$$\boldsymbol{i} \leq \alpha \quad \text{or} \quad \alpha \leq \neg \boldsymbol{i}$$

validity can **stutter** from one formula to another, missing in this way the uniformity typical of parametricity

A semantic solution

A faint intuition

The loss of parametricity is only due to the interpretation of indivisible types, all the rest works (more or less) smoothly

The crux of the problem is that for an indivisible type $\mathbf{i}$

$$\mathbf{i} \leq \alpha \quad \text{or} \quad \alpha \leq \neg\mathbf{i}$$

validity can **stutter** from one formula to another, missing in this way the uniformity typical of parametricity

The *leitmotif* of this work

A semantic characterization of models where *stuttering* is absent, should yield a subtyping relation that is:

1. Semantic
2. Intuitive for the programmer
3. Decidable

A semantic solution

Rough idea

Make indivisible types "splittable" so that type variables can range over strict subsets of every type, indivisible types included.
[intuition: interpret all non-empty types into infinite sets]

A semantic solution

Rough idea

Make indivisible types "splittable" so that type variables can range over strict subsets of every type, indivisible types included.
[intuition: interpret all non-empty types into infinite sets]

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignments:

$$\eta : \textbf{Vars} \to \mathcal{P}(\mathcal{D})$$

A semantic solution

Rough idea

Make indivisible types "splittable" so that type variables can range over strict subsets of every type, indivisible types included.
[intuition: interpret all non-empty types into infinite sets]

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignments:

$$\eta : \textbf{Vars} \to \mathcal{P}(\mathcal{D})$$

and now the interpretation function takes an extra parameter

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})^{\textbf{Vars}} \to \mathcal{P}(\mathcal{D})$$

A semantic solution

Rough idea

Make indivisible types "splittable" so that type variables can range over strict subsets of every type, indivisible types included.
[intuition: interpret all non-empty types into infinite sets]

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignments:

$$\eta : \textbf{Vars} \to \mathcal{P}(\mathcal{D})$$

and now the interpretation function takes an extra parameter

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})^{\textbf{Vars}} \to \mathcal{P}(\mathcal{D})$$

with

$$\begin{array}{lcl} [\![\alpha]\!]\eta & = & \eta(\alpha) \\ [\![t_1 \vee t_2]\!]\eta & = & [\![t_1]\!]\eta \cup [\![t_2]\!]\eta \\ [\![\mathbb{0}]\!]\eta & = & \varnothing \end{array} \qquad \begin{array}{lcl} [\![\neg t]\!]\eta & = & \mathcal{D} \backslash [\![t]\!]\eta \\ [\![t_1 \wedge t_2]\!]\eta & = & [\![t_1]\!]\eta \cap [\![t_2]\!]\eta \\ [\![\mathbb{1}]\!]\eta & = & \mathcal{D} \end{array}$$

A semantic solution

Rough idea

Make indivisible types "splittable" so that type variables can range over strict subsets of every type, indivisible types included.

[intuition: interpret all non-empty types into infinite sets]

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignments:

$$\eta : \textbf{Vars} \to \mathcal{P}(\mathcal{D})$$

and now the interpretation function takes an extra parameter

$$[\![\ ]\!] : \textbf{Types} \to \mathcal{P}(\mathcal{D})^{\textbf{Vars}} \to \mathcal{P}(\mathcal{D})$$

with

$$\begin{array}{lclclcl} [\![\alpha]\!]\eta & = & \eta(\alpha) & \quad & [\![\neg t]\!]\eta & = & \mathcal{D}\setminus[\![t]\!]\eta \\ [\![t_1{\vee}t_2]\!]\eta & = & [\![t_1]\!]\eta \cup [\![t_2]\!]\eta & & [\![t_1{\wedge}t_2]\!]\eta & = & [\![t_1]\!]\eta \cap [\![t_2]\!]\eta \\ [\![\mathbb{0}]\!]\eta & = & \varnothing & & [\![\mathbb{1}]\!]\eta & = & \mathcal{D} \end{array}$$

and such that it satisfies:

$$[\![t_1{\to}s_1]\!]\eta \subseteq [\![t_2{\to}s_2]\!]\eta \iff \mathcal{P}(\overline{[\![t_1]\!]\eta \times \overline{[\![s_1]\!]\eta}}) \subseteq \mathcal{P}(\overline{[\![t_2]\!]\eta \times \overline{[\![s_2]\!]\eta}})$$

Subtyping relation

In this framework the natural definition of subtyping is

$$s \leq t \quad \overset{def}{\Longleftrightarrow} \quad \forall \eta \,.\, [\![s]\!]\eta \subseteq [\![t]\!]\eta$$

It "just" remains to find the uniformity condition to avoid stuttering and recover parametricity.

The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.(\llbracket t_1\rrbracket\eta{=}\varnothing \text{ or } \llbracket t_2\rrbracket\eta{=}\varnothing) \iff (\forall\eta.\llbracket t_1\rrbracket\eta{=}\varnothing) \text{ or } (\forall\eta.\llbracket t_2\rrbracket\eta{=}\varnothing)$$

The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.(\llbracket t_1\rrbracket\eta{=}\varnothing \text{ or } \llbracket t_2\rrbracket\eta{=}\varnothing) \iff (\forall\eta.\llbracket t_1\rrbracket\eta{=}\varnothing) \text{ or } (\forall\eta.\llbracket t_2\rrbracket\eta{=}\varnothing)$$

- It avoids stuttering: $\forall\eta.(\llbracket t{\wedge}\neg\alpha\rrbracket\eta{=}\varnothing \text{ or } \llbracket t{\wedge}\alpha\rrbracket\eta{=}\varnothing)$ —that is, $(t \leq \alpha \text{ or } \alpha \leq \neg t)$— holds if and only if t is empty.

The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.(\llbracket t_1\rrbracket\eta{=}\varnothing \text{ or } \llbracket t_2\rrbracket\eta{=}\varnothing) \iff (\forall\eta.\llbracket t_1\rrbracket\eta{=}\varnothing) \text{ or } (\forall\eta.\llbracket t_2\rrbracket\eta{=}\varnothing)$$

- It avoids stuttering: $\forall\eta.(\llbracket t{\wedge}\neg\alpha\rrbracket\eta{=}\varnothing$ or $\llbracket t{\wedge}\alpha\rrbracket\eta{=}\varnothing)$ —that is, $(t \leq \alpha$ or $\alpha \leq \neg t)$— holds if and only if t is empty.
- There are natural models: all models that map all non-empty types into infinite sets satisfy it [our initial intuition].

The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } [\![t_2]\!]\eta{=}\varnothing) \iff (\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } (\forall\eta.[\![t_2]\!]\eta{=}\varnothing)$$

- It avoids stuttering: $\forall\eta.([\![t\wedge\neg\alpha]\!]\eta{=}\varnothing$ or $[\![t\wedge\alpha]\!]\eta{=}\varnothing)$ —that is, $(t \leq \alpha$ or $\alpha \leq \neg t)$— holds if and only if t is empty.
- There are natural models: all models that map all non-empty types into infinite sets satisfy it [our initial intuition].
- A sound, complete, and terminating decision algorithm: the condition gives us exactly the right conditions needed to reuse the subtyping algorithm devised for ground types.

The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } [\![t_2]\!]\eta{=}\varnothing) \iff (\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } (\forall\eta.[\![t_2]\!]\eta{=}\varnothing)$$

- It avoids stuttering: $\forall\eta.([\![t{\wedge}\neg\alpha]\!]\eta{=}\varnothing \text{ or } [\![t{\wedge}\alpha]\!]\eta{=}\varnothing)$ —that is, $(t \leq \alpha \text{ or } \alpha \leq \neg t)$— holds if and only if t is empty.
- There are natural models: all models that map all non-empty types into infinite sets satisfy it [our initial intuition].
- A sound, complete, and terminating decision algorithm: the condition gives us exactly the right conditions needed to reuse the subtyping algorithm devised for ground types.
- An intuitive relation: the algorithm returns intuitive results (actually, it helps to better understand twisted examples)

The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

$$\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } [\![t_2]\!]\eta{=}\varnothing) \iff (\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } (\forall\eta.[\![t_2]\!]\eta{=}\varnothing)$$

- It avoids stuttering: $\forall\eta.([\![t{\wedge}\neg\alpha]\!]\eta{=}\varnothing$ or $[\![t{\wedge}\alpha]\!]\eta{=}\varnothing)$ —that is, $(t \leq \alpha$ or $\alpha \leq \neg t)$— holds if and only if t is empty.
- **There are natural models:** all models that map all non-empty types into infinite sets satisfy it [our initial intuition].
- **A sound, complete, and terminating decision algorithm:** the condition gives us exactly the right conditions needed to reuse the subtyping algorit nd types.
- An intuitive relation tuitive results (actually, it helps t twisted examples)

Examples of subtyping relations

Examples

We can internalize properties such as:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \sim \alpha \vee \beta \to \gamma$$

Examples

We can internalize properties such as:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \;\sim\; \alpha \vee \beta \to \gamma$$

or distributivity laws:

$$(\alpha \vee \beta \times \gamma) \;\sim\; (\alpha \times \gamma) \vee (\beta \times \gamma)$$

Examples

We can internalize properties such as:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \sim \alpha{\vee}\beta \to \gamma$$

or distributivity laws:

$$(\alpha{\vee}\beta \times \gamma) \sim (\alpha{\times}\gamma) \vee (\beta{\times}\gamma)$$

and combining them deduce:

$$(\alpha{\times}\gamma \to \delta_1) \wedge (\beta{\times}\gamma \to \delta_2) \leq (\alpha{\vee}\beta \times \gamma) \to \delta_1 \vee \delta_2$$

Examples

We can internalize properties such as:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \sim \alpha \vee \beta \to \gamma$$

or distributivity laws:

$$(\alpha \vee \beta \times \gamma) \sim (\alpha \times \gamma) \vee (\beta \times \gamma)$$

and combining them deduce:

$$(\alpha \times \gamma \to \delta_1) \wedge (\beta \times \gamma \to \delta_2) \leq (\alpha \vee \beta \times \gamma) \to \delta_1 \vee \delta_2$$

Of course the problematic relation never holds, whatever the t:

$$(t \times \alpha) \not\leq (t \times \neg t) \vee (\alpha \times t)$$

We can prove relevant relations on infinite types, *eg*., for the type of generic α-lists:

$$\alpha\text{-list} = \mu z.(\alpha \times z) \vee \text{nil}$$

We can prove relevant relations on infinite types, *eg*., for the type of generic α-lists:

$$\alpha\text{-list} = \mu z.(\alpha \times z) \vee \mathrm{nil}$$

we can prove that it contains both the α-lists of even length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee \mathrm{nil}}_{\alpha\text{-lists of even length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \mathrm{nil}}_{\alpha\text{-lists}}$$

and the α-lists with of odd length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee (\alpha \times \mathrm{nil})}_{\alpha\text{-lists of odd length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \mathrm{nil}}_{\alpha\text{-lists}}$$

We can prove relevant relations on infinite types, *eg*., for the type of generic α-lists:

$$\alpha\text{-list} = \mu z.(\alpha \times z) \vee \mathtt{nil}$$

we can prove that it contains both the α-lists of even length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee \mathtt{nil}}_{\alpha\text{-lists of even length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \mathtt{nil}}_{\alpha\text{-lists}}$$

and the α-lists with of odd length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee (\alpha \times \mathtt{nil})}_{\alpha\text{-lists of odd length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \mathtt{nil}}_{\alpha\text{-lists}}$$

and that it is itself contained in the union of the two, that is:

$$\alpha\text{-list} \sim (\mu z.(\alpha \times (\alpha \times z)) \vee \mathtt{nil}) \vee (\mu z.(\alpha \times (\alpha \times z)) \vee (\alpha \times \mathtt{nil}))$$

We can prove relevant relations on infinite types, *eg.*, for the type of generic α-lists:

$$\alpha\text{-list} = \mu z.(\alpha \times z) \vee \text{nil}$$

we can prove that it contains both the α-lists of even length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee \text{nil}}_{\alpha\text{-lists of even length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \text{nil}}_{\alpha\text{-lists}}$$

and the α-lists with of odd length

$$\underbrace{\mu z.(\alpha \times (\alpha \times z)) \vee (\alpha \times \text{nil})}_{\alpha\text{-lists of odd length}} \leq \underbrace{\mu z.(\alpha \times z) \vee \text{nil}}_{\alpha\text{-lists}}$$

and that it is itself contained in the union of the two, that is:

$$\alpha\text{-list} \sim (\mu z.(\alpha \times (\alpha \times z)) \vee \text{nil}) \vee (\mu z.(\alpha \times (\alpha \times z)) \vee (\alpha \times \text{nil}))$$

And we can prove far more complicated relations (see paper).

Subtyping algorithm

Subtyping Algorithm: $t_1 \leq t_2$

Step 1: Transform the subtyping problem into an emptiness decision problem:

$t_1 \leq t_2 \iff \forall\eta.[\![t_1]\!]\eta \subseteq [\![t_2]\!]\eta \iff \forall\eta.[\![t_1{\wedge}\neg t_2]\!]\eta{=}\varnothing \iff t_1{\wedge}\neg t_2 \leq \mathbb{0}$

Subtyping Algorithm: $t_1 \leq t_2$

Step 1: **Transform the subtyping problem into an emptiness decision problem:**

$t_1 \leq t_2 \iff \forall\eta.[\![t_1]\!]\eta \subseteq [\![t_2]\!]\eta \iff \forall\eta.[\![t_1{\wedge}\neg t_2]\!]\eta{=}\varnothing \iff t_1{\wedge}\neg t_2 \leq \mathbb{0}$

Step 2: **Put the type whose emptiness is to be decided in disjunctive normal form.**

$$\bigvee_{i\in I}\bigwedge_{j\in J}\ell_{ij}$$

where $a ::= b \mid t \times t \mid t \rightarrow t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$ and $\ell ::= a \mid \neg a$

Subtyping Algorithm: $t_1 \leq t_2$

Step 1: Transform the subtyping problem into an emptiness decision problem:

$t_1 \leq t_2 \iff \forall\eta.[\![t_1]\!]\eta \subseteq [\![t_2]\!]\eta \iff \forall\eta.[\![t_1{\wedge}\neg t_2]\!]\eta{=}\varnothing \iff t_1{\wedge}\neg t_2 \leq \mathbb{0}$

Step 2: Put the type whose emptiness is to be decided in disjunctive normal form.

$$\bigvee_{i\in I}\bigwedge_{j\in J}\ell_{ij}$$

where $a ::= b \mid t \times t \mid t \to t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha$ and $\ell ::= a \mid \neg a$

Step 3: Simplify mixed intersections:

Consider each summand of the union: cases such as $t_1{\times}t_2 \wedge t_1{\to}t_2$ or $t_1{\times}t_2 \wedge \neg(t_1{\to}t_2)$ are straightforward.

Solve:
$$\bigwedge_{i\in I} a_i \bigwedge_{j\in J} \neg a'_j \bigwedge_{h\in H} \alpha_h \bigwedge_{k\in K} \neg\beta_k$$

where all a are of the same kind.

Step 4: Eliminate toplevel negative variables.,

$$\forall\eta.[\![t]\!]\eta = \varnothing \iff \forall\eta.[\![t\{{}^{\neg\alpha}/_{\alpha}\}]\!]\eta = \varnothing$$

so replace $\neg\beta_k$ for β_k (forall $k \in K$)

Solve: $$\bigwedge_{i\in I} a_i \bigwedge_{j\in J} \neg a'_j \bigwedge_{h\in H} \alpha_h$$

Step 4: Eliminate toplevel negative variables.,

$$\forall\eta.[\![t]\!]\eta = \varnothing \iff \forall\eta.[\![t\{{}^{\neg\alpha}/_{\alpha}\}]\!]\eta = \varnothing$$

so replace $\neg\beta_k$ for β_k (forall $k \in K$)

Solve: $$\bigwedge_{i\in I} a_i \bigwedge_{j\in J} \neg a'_j \bigwedge_{h\in H} \alpha_h$$

Step 5: Eliminate toplevel variables.

$$\bigwedge_{t_1\times t_2\in P} t_1\times t_2 \bigwedge_{h\in H} \alpha_h \quad\leq\quad \bigvee_{t'_1\times t'_2\in N} t'_1\times t'_2$$

holds if and only if

$$\bigwedge_{t_1\times t_2\in P} t_1\sigma \times t_2\sigma \bigwedge_{h\in H} \gamma^1_h \times \gamma^2_h \quad\leq\quad \bigvee_{t'_1\times t'_2\in N} t'_1\sigma \times t'_2\sigma$$

where $\sigma = \{{}^{(\gamma^1_h\times\gamma^2_h)\vee\alpha_h}/_{\alpha_h}\}_{h\in H}$ (similarly for arrows)

Step 6: Eliminate toplevel constructors, memoize, and recurse.
Thanks to *convexity* and (set-theoretic) product decomposition rules

$$\bigwedge_{t_1\times t_2\in P} t_1\times t_2 \quad\leq\quad \bigvee_{t_1'\times t_2'\in N} t_1'\times t_2' \qquad (2)$$

is equivalent to

$$\forall N'\subseteq N.\left(\bigwedge_{t_1\times t_2\in P} t_1 \leq \bigvee_{t_1'\times t_2'\in N'} t_1'\right) \text{ or } \left(\bigwedge_{t_1\times t_2\in P} t_2 \leq \bigvee_{t_1'\times t_2'\in N\setminus N'} t_2'\right)$$

(similarly for arrows)

Conclusion and New Directions

Conclusion

- We presented the first known solution to the problem of defining a semantic subtyping relation for a polymorphic regular tree types.
- A solution to this problem was considered unfeasible or even impossible.
- Our solution immediately applies to functional XML processing, but the potential fields of application seem much more numerous.
- Finally, our work opens both *practical* and *theoretical* new directions of research.

Practical problems

New typing possibilities:

```
fun even =
    | Int -> (x mod 2) == 0
    |  _  -> x
```

Intuitively we want to type it by

$$(\texttt{Int}{\longrightarrow}\texttt{Bool}) \wedge (\alpha\backslash\texttt{Int} \rightarrow \alpha\backslash\texttt{Int})$$

Practical problems

New typing possibilities:

```
fun even =
    | Int -> (x mod 2) == 0
    |  _  -> x
```

Intuitively we want to type it by

$$(\mathtt{Int} \to \mathtt{Bool}) \wedge (\alpha \backslash \mathtt{Int} \to \alpha \backslash \mathtt{Int})$$

Local type inference:

Let **map** : $(\alpha \to \beta) \to \alpha$ list $\to \beta$ list,

then for **map even** we wish to deduce the following type:

$(\ \mathtt{Int}\ \text{list} \to \mathtt{Bool}\ \text{list}\ ) \wedge$
$(\ (\alpha \backslash \mathtt{Int})\ \text{list} \to (\alpha \backslash \mathtt{Int})\ \text{list}\ ) \wedge$
$(\ \alpha\ \text{list} \to ((\alpha \backslash \mathtt{Int}) \vee \mathtt{Bool})\ \text{list}\ )$

Practical problems

New typing possibilities:

```
fun even =
    | Int -> (x mod 2) == 0
    |  _  -> x
```

Intuitively we want to type it by

$$(\texttt{Int}\to\texttt{Bool}) \wedge (\alpha\setminus\texttt{Int} \to \alpha\setminus\texttt{Int})$$

Local type inference:

Let **map** : $(\alpha \to \beta) \to \alpha$ list $\to \beta$ list,
then for **map even** we wish to deduce the following type:

$(\,\texttt{Int}$ list $\to \texttt{Bool}$ list $) \wedge$ int lists return bool lists
$(\,(\alpha\setminus\texttt{Int})$ list $\to (\alpha\setminus\texttt{Int})$ list $) \wedge$
$(\,\alpha$ list $\to ((\alpha\setminus\texttt{Int})\vee\texttt{Bool})$ list $)$

Practical problems

New typing possibilities:

```
fun even =
    | Int -> (x mod 2) == 0
    |  _  -> x
```

Intuitively we want to type it by

$$(\mathtt{Int}{\to}\mathtt{Bool}) \wedge (\alpha\backslash\mathtt{Int} \to \alpha\backslash\mathtt{Int})$$

Local type inference:

Let **map** : $(\alpha \to \beta) \to \alpha$ list $\to \beta$ list,

then for **map even** we wish to deduce the following type:

$(\,\mathtt{Int}\text{ list} \to \mathtt{Bool}\text{ list}\,) \wedge$ int lists return bool lists

$(\,(\alpha\backslash\mathtt{Int})\text{ list} \to (\alpha\backslash\mathtt{Int})\text{ list}\,) \wedge$ lists w/o ints return the same type

$(\,\alpha\text{ list} \to ((\alpha\backslash\mathtt{Int}){\vee}\mathtt{Bool})\text{ list}\,)$

Practical problems

New typing possibilities:

```
fun even =
    | Int -> (x mod 2) == 0
    |  _  -> x
```

Intuitively we want to type it by

$$(\texttt{Int}{\to}\texttt{Bool}) \wedge (\alpha\backslash\texttt{Int} \to \alpha\backslash\texttt{Int})$$

Local type inference:

Let **map** : $(\alpha \to \beta) \to \alpha$ list $\to \beta$ list,

then for **map even** we wish to deduce the following type:

$(\,\texttt{Int}\ \text{list} \to \texttt{Bool}\ \text{list}\,)\ \wedge$ int lists return bool lists

$(\,(\alpha\backslash\texttt{Int})\ \text{list} \to (\alpha\backslash\texttt{Int})\ \text{list}\,)\ \wedge$ lists w/o ints return the same type

$(\,\alpha\ \text{list} \to ((\alpha\backslash\texttt{Int})\vee\texttt{Bool})\ \text{list}\,)$ ints in the argument are replaced by bools

Practical problems

New typing possibilities:

```
fun even =
    | Int -> (x mod 2) == 0
    |  _   -> x
```

Intuitively we want to type it by

$$(\mathtt{Int}\rightarrow\mathtt{Bool}) \wedge (\alpha\backslash\mathtt{Int} \rightarrow \alpha\backslash\mathtt{Int})$$

Local type inference:

Let **map** : $(\alpha \rightarrow \beta) \rightarrow \alpha$ list $\rightarrow \beta$ list,

then for **map even** we wish to deduce the following type:

$(\ \mathtt{Int}\ \text{list} \rightarrow \mathtt{Bool}\ \text{list}\ ) \wedge$	int lists return bool lists
$(\ (\alpha\backslash\mathtt{Int})\ \text{list} \rightarrow (\alpha\backslash\mathtt{Int})\ \text{list}\ ) \wedge$	lists w/o ints return the same type
$(\ \alpha\ \text{list} \rightarrow ((\alpha\backslash\mathtt{Int})\vee\mathtt{Bool})\ \text{list}\ )$	ints in the argument are replaced by bools

Cannot be obtained by just instantiating the type of `map`

Practical problems

New typing possibilities:

```
fun even =
    | Int -> (x mod 2) == 0
    |  _  -> x
```

Intuitively we want to type it by

$$(\texttt{Int}\to\texttt{Bool}) \wedge (\alpha\backslash\texttt{Int} \to \alpha\backslash\texttt{Int})$$

Local type inference:

Let **map** : $(\alpha \to \beta) \to \alpha$ list $\to \beta$ list,

then for **map even** we wish to deduce the following type:

$(\,\texttt{Int}\text{ list} \to \texttt{Bool}\text{ list}\,) \wedge$ — int lists return bool lists

$(\,(\alpha\backslash\texttt{Int})\text{ list} \to (\alpha\backslash\texttt{Int})\text{ list}\,) \wedge$ — lists w/o ints return the same type

$(\,\alpha\text{ list} \to ((\alpha\backslash\texttt{Int})\vee\texttt{Bool})\text{ list}\,)$ — ints in the argument are replaced by bools

Cannot be obtained by just instantiating the type of `map`

No principal typing (needs infinite connectives)

Practical problems

New typing possibilities: **new language design**

```
fun even =
    | Int -> (x mod 2) == 0
    |  _  -> x
```

Intuitively we want to type it by

$$(\texttt{Int}\to\texttt{Bool}) \wedge (\alpha\backslash\texttt{Int} \to \alpha\backslash\texttt{Int})$$

Local type inference: **subtyping + instantiation**

Let **map** : $(\alpha \to \beta) \to \alpha$ list $\to \beta$ list,

then for **map even** we wish to deduce the following type:

$(\,\texttt{Int}\text{ list} \to \texttt{Bool}\text{ list}\,) \wedge$ int lists return bool lists
$(\,(\alpha\backslash\texttt{Int})\text{ list} \to (\alpha\backslash\texttt{Int})\text{ list}\,) \wedge$ lists w/o ints return the same type
$(\,\alpha\text{ list} \to ((\alpha\backslash\texttt{Int})\vee\texttt{Bool})\text{ list}\,)$ ints in the argument are replaced by bools

Cannot be obtained by just instantiating the type of `map`
No principal typing (needs infinite connectives)

Convexity and parametricity?

In reality, the condition to be used is the generalization to n types:

$$\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } \cdots \text{ or } [\![t_n]\!]\eta{=}\varnothing)$$
$$\Longleftrightarrow$$
$$(\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } \cdots \text{ or } (\forall\eta.[\![t_n]\!]\eta{=}\varnothing)$$

Convexity and parametricity?

In reality, the condition to be used is the generalization to n types:

$$\forall\eta.(\llbracket t_1\rrbracket\eta{=}\varnothing \text{ or } \cdots \text{ or } \llbracket t_n\rrbracket\eta{=}\varnothing)$$
$$\iff$$
$$(\forall\eta.\llbracket t_1\rrbracket\eta{=}\varnothing) \text{ or } \cdots \text{ or } (\forall\eta.\llbracket t_n\rrbracket\eta{=}\varnothing)$$

The big question

What is the relation of the condition above with parametricity?
Is it a language-independent semantic characterization of it?

Convexity and parametricity?

In reality, the condition to be used is the generalization to n types:

$$\forall\eta.(\llbracket t_1\rrbracket\eta{=}\varnothing \text{ or } \cdots \text{ or } \llbracket t_n\rrbracket\eta{=}\varnothing)$$
$$\Longleftrightarrow$$
$$(\forall\eta.\llbracket t_1\rrbracket\eta{=}\varnothing) \text{ or } \cdots \text{ or } (\forall\eta.\llbracket t_n\rrbracket\eta{=}\varnothing)$$

The big question

What is the relation of the condition above with parametricity?
Is it a language-independent semantic characterization of it?

Two examples of uniformity:

- $(t_1\times\ldots\times t_n)$ is empty if and only if exists at least one t_i empty
- Definability in the second-order typed λ-calculus harnesses expressions to behave uniformity. Similarly, convexity *semantically* harnesses the denotations of expressions and forces them to behave uniformly.

Convexity and parametricity?

In reality, the condition to be used is the generalization to n types:

$$\forall\eta.(\llbracket t_1 \rrbracket\eta{=}\varnothing \text{ or } \cdots \text{ or } \llbracket t_n \rrbracket\eta{=}\varnothing)$$
$$\Longleftrightarrow$$
$$(\forall\eta.\llbracket t_1 \rrbracket\eta{=}\varnothing) \text{ or } \cdots \text{ or } (\forall\eta.\llbracket t_n \rrbracket\eta{=}\varnothing)$$

The big question

What is the relation of the condition above with parametricity?
Is it a language-independent semantic characterization of it?

Two examples of uniformity:

- $(t_1{\times}\ldots{\times}t_n)$ is empty if and only if exists at least one t_i empty
- Definability in the second-order typed λ-calculus harnesses expressions to behave uniformity. Similarly, convexity *semantically* harnesses the denotations of expressions and forces them to behave uniformly.

... we have strong flavors of parametricity