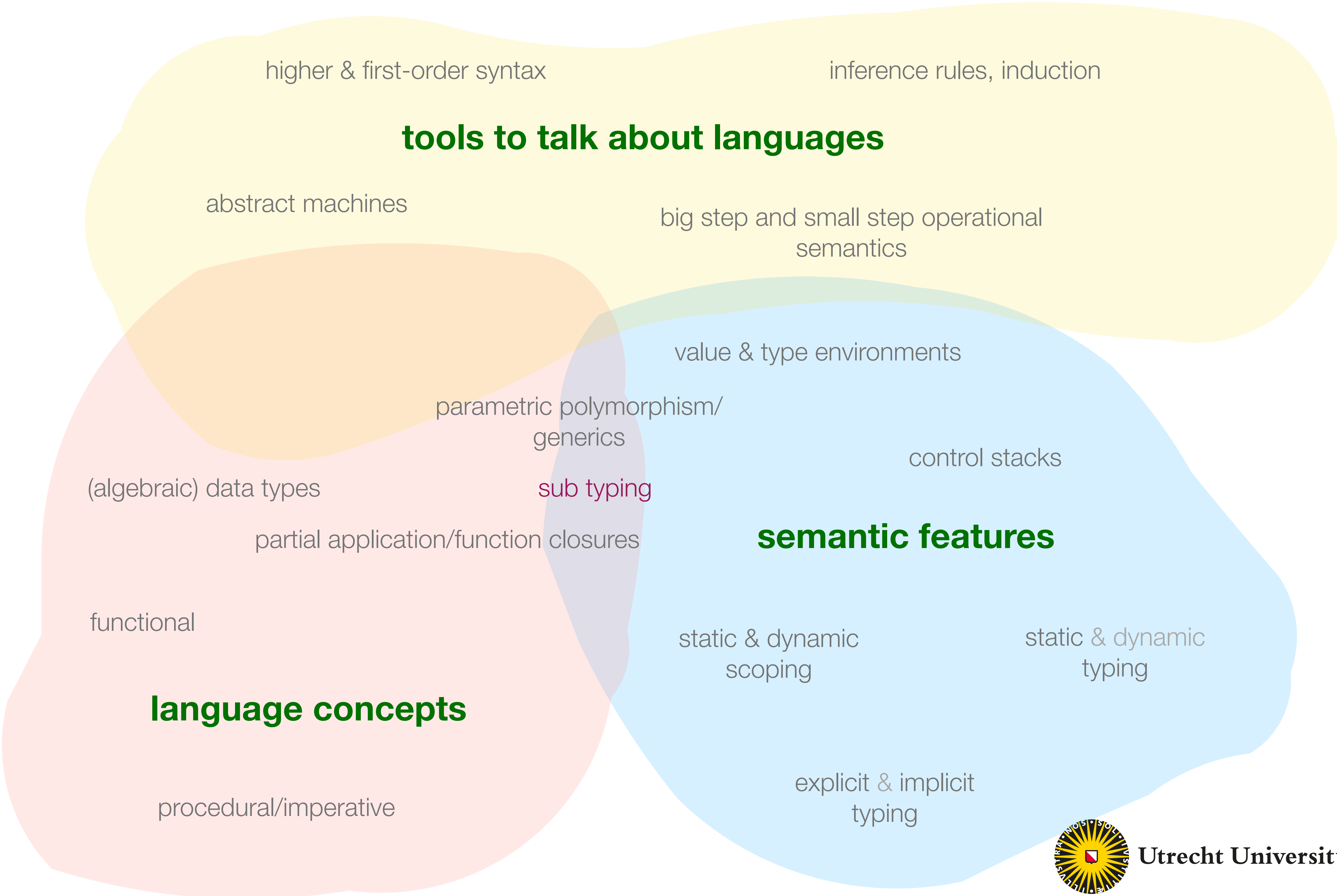
Utrecht University

Concepts of Programming Language Design
Subtyping

Gabriele Keller
Tom Smeding

Overview



Utrecht University

Subtyping

- Why subtyping?
  - eliminates the need to explicitly convert between elements of different types

    Int → $1 \ +_{\texttt{Float}} \ 1.565$ ← Float

  - can be used to express program properties
  - essential in OO-languages: closely related to the subclass-relationship
- Subtype relation
  - $\tau \leq \sigma$: τ is a subtype of σ

    note that we're overloading notation here: we used '≤' for 'is less general' previously!

  - what does it mean for a type τ to be a subtype of another type σ?
    - wherever a value of type σ is required, we can use a value of type instead τ

Utrecht University

Two different forms of subtyping

- Subset interpretation
  - if $\tau \leq \sigma$, then every value τ of is also a value of σ,
  - e.g.,
    - even integers $\leq$ integers
    - non-empty lists $\leq$ lists
    - squares $\leq$ rectangles $\leq$ polygons
- Coercion interpretation
  - if then every value τ of can be coerced to a value of type σ in a unique way
  - e.g.,
    - `Int` $\leq$ `Float` (e.g., `3` to `3.0`)
    - `Char` $\leq$ `String` (e.g., `'w'` to `''w''`)
- The subclass relationship in OO is a special form of subtyping, but we will cover it separately

Utrecht University

Adding Subtyping to MinHs

- MinHs extensions:
  - Add type `Float`
  - Add operations $+_{\texttt{Float}}$, $*_{\texttt{Float}}$ and so on
- We want to be able to write
  - `1` $+_{\texttt{Float}}$ `1.1232312`
  - `1.7` $+_{\texttt{Float}}$ `5`
  - `1` $+_{\texttt{Float}}$ `5`
- How can we implement a coercion interpretation?
  - dynamic resolution
    - floating point operation *dynamically* checks whether operands can be coerced to `Float`, and coerces on the fly
  - static resolution
    - type checker inserts coercions *at compile time*

Properties of Subtyping

- Coercion vs subset interpretation
  - coercion interpretation is more expressive than subset interpretation
  - we discuss coercion interpretation in more detail
- Soundness of subtyping
  - the subtyping relation needs to meet certain formal properties
  - otherwise, type safety will be compromised

Properties of Subtyping

- Required properties: reflexivity and transitivity

$$\frac{}{\tau \leq \tau}$$

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

- Subset interpretation
  - reflexivity and transitivity follow from properties of subset relation
- Coercion interpretation
  - reflexivity (coercion function is the identity), transitivity (coercion function by composing the two coercion functions)

Properties of Subtyping

- Coherence: a subtyping relationship must be coherent
  - this means, the coerced value has to be unique
  - if a value can be coerced in two ways, both must yield the same result
  - example:
    - assume: `Int` ≤ `Float`, `Int` ≤ `String`, `Float` ≤ `String`
    - consider `print (3::Int)`
    - what might go wrong here?
      - direct coercion: `"3"`
      - coercion via `Float`: `"3.0"`

Problems with automatic coercion

- Coercions might hide actual programming bugs
- Coercion behaviour unexpected, may differ from language to language

Examples:

- JS vs PHP:

```
(‘‘5’’ > ‘‘11’’ ? ‘‘TRUE’’ : ‘‘FALSE’’)
```

JS: compares strings alph. : “True”

PHP: converts, then compares: “False"

```
(‘‘0’’          ? ‘‘TRUE’’ : ‘‘FALSE’’)
```

JS: non-null object: “True”

PhP:converts to 0: “False”

Subsumption

- The rule of subsumption - implicit subtyping

$$\frac{\Gamma \vdash e : \tau \qquad \tau \leq \sigma}{\Gamma \vdash e : \sigma}$$

- The rule of subsumption - explicit subtyping (with cast expression (σ))

$$\frac{\Gamma \vdash e : \tau \qquad \tau \leq \sigma}{\Gamma \vdash (\sigma)\ e : \sigma}$$

Composite Types and Subtyping

- If `Int` $\leq$ `Float`, what is then then relationship between the following types:

```
(Int   * Int)
(Float * Int)
(Int   * Float)
(Float * Float)
```

- How about sums?
- Subtyping rules for products and sums:

$$\frac{\tau_1 \leq \sigma_1 \quad \tau_2 \leq \sigma_2}{(\tau_1 * \tau_2) \leq (\sigma_1 * \sigma_2)}$$

$$\frac{\tau_1 \leq \sigma_1 \quad \tau_2 \leq \sigma_2}{(\tau_1 + \tau_2) \leq (\sigma_1 + \sigma_2)}$$

Composite Types and Subtyping

- What is the relationship between the following types:

```
  Int → Int
Float → Int
  Int → Float
Float → Float
```

- Given a coercion function `intToFloat:: Int → Float`, can we define coercion functions of the following types:

```
(Int → Int)     → (Float → Int)
(Float → Int)   → (Int → Float)
(Int → Float)   → (Float → Float)
(Float → Float) → (Int → Int)
```

• • • •

Composite Types and Subtyping

- What is the relationship between the following types:

```
  Int → Int
Float → Int
  Int → Float
Float → Float
```

- Subtyping rules for function types

$$\frac{\sigma_1 \leq \tau_1 \quad \tau_2 \leq \sigma_2}{(\tau_1 \rightarrow \tau_2) \leq (\sigma_1 \rightarrow \sigma_2)}$$

Composite Types and Subtyping

- Rules specifying how a type constructor interacts with subtyping are called variance principles
- If a constructor preserves subtyping, it is called co-variant
  - the sum and product constructor are co-variant in both arguments
- If a constructor inverts subtyping, it is called contra-variant
  - the function type constructor is contra-variant in the first argument
  - and co-variant in the second argument

Variance

- What about array/reference types?

```
newIORef   :: a              -> IO (IORef a)
writeIORef :: a -> IORef a -> IO ()
readIORef  ::      IORef a -> IO a
```

- Is **`IORef`** co- or contra-variant?

```
Int ≤ Float
fRef :: IORef Float
iRef :: IORef Int

readIORef fRef        to get a FLOAT
readIORef iRef        we can read from iRef, and convert Int to Float

writeIORef  (5.2423 :: Float) fRef        to store a Float
writeIORef  (5.2423 :: Float) iRef        we cannot use an iRef to store a Float!

writeIORef  (5 :: Int) iRef        to store an Int
writeIORef  (5 :: Int) fRef        we can convert 5 to a Float, and store it in fRef

readIORef iRef        to get an Int
readIORef fRef        we cannot use an fRef to get an Int!
```

Variance

- What about array/reference types?

```
newIORef   :: a              -> IO (IORef a)
writeIORef :: a -> IORef a -> IO ()
readIORef  ::      IORef a -> IO a
```

- Is `IORef` co- or contra-variant?

  it's neither co- nor contra-variant!

- If a constructor is neither co- nor contra-variant, it is called invariant

- Java and C# have arrays as co-variant type constructors
  - how is it handled?