Higher-Order Subtyping for Dependent Types

Andreas Abel

Department of Computer Science
Ludwig-Maximilians-University Munich

Theory Seminar
Institute of Cybernetics, Tallinn, Estonia
24 November 2011

Subtyping

Definition (Subtype)

A is a subtype of B, written $A \leq B$, if a value of type A can be supplied wherever a value of type B is demanded.

Examples:

- $\mathsf{Pos} \leq \mathsf{Nat}$. A function that can handle natural numbers $n \in \mathsf{Nat}$, meaning $n \geq 0$, is not embarrased by a positive number $n \in \mathsf{Pos}$, meaning $n > 0$.
- $\mathsf{List}\,\mathsf{Pos} \leq \mathsf{List}\,\mathsf{Nat}$. If I can process a list of natural numbers, I can process a list of positive numbers.
- $\mathsf{Nat} \to \mathsf{Pos} \leq \mathsf{Pos} \to \mathsf{Nat}$. I want a function that turns positive numbers into natural numbers. I accept a function that handles even all natural numbers and returns positive natural numbers.

Subtyping = information loss.

Higher-Order Subtyping

- Higher-Order subtyping deals with type constructors.

$$\frac{\mathsf{List}^{\neq\emptyset} \leq \mathsf{List}}{\mathsf{List}^{\neq\emptyset}\, A \leq \mathsf{List}\, A}$$

- $\mathsf{List}^{\neq\emptyset}\,\mathsf{Nat} \leq \mathsf{List}\,\mathsf{Nat}$. I want a list of natural numbers. I accept, in particular, non-empty lists.
- Subtyping the list elements:

$$\frac{\mathsf{List}^{\neq\emptyset} \leq \mathsf{List} \qquad \mathsf{Pos} \leq \mathsf{Nat}}{\mathsf{List}^{\neq\emptyset}\,\mathsf{Pos} \leq \mathsf{List}\,\mathsf{Nat}}$$

- Expecting a list of natural numbers, I can handle a non-empty list of positive numbers.

Variances

- The type constructor List is monotone/covariant.

$$\frac{\mathsf{List} : +\mathsf{Set} \to \mathsf{Set} \qquad A \leq B}{\mathsf{List}\, A \leq \mathsf{List}\, B}$$

- In Scala: `class List[+A]`
- What about arrays? Java has the covariant rule:

$$\frac{A \leq B}{A[] \leq B[]}$$

- Unsound! If my array can only store positive numbers, I cannot squeeze in a (fat) 0.
- Can produce ArrayStoreException in Java.

The World Refuses Variances

Google does not want type soundness.

> *The type system [of DART] is unsound, due to the covariance of generic types. This is a deliberate choice (and undoubtedly controversial). Experience has shown that sound type rules for generics fly in the face of programmer intuition.*

http://www.dartlang.org/docs/spec/dartLangSpec.pdf, page 76

Contravariance

- Correct subtyping for arrays (mixed-variance):

$$\frac{\mathsf{Array} : \mathsf{Set} \to \mathsf{Set} \qquad A = B}{\mathsf{Array}\, A \leq \mathsf{Array}\, B}$$

- Subtyping for function spaces:

$$\frac{A' \leq A \qquad B \leq B'}{A \to B \leq A' \to B'}$$

- In Scala: `class Function[-A, +B]`
- In my own language MiniAgda:

$$\to : \; -\mathsf{Set} \to +\mathsf{Set} \to \mathsf{Set}$$

No Subtyping in Agda

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat

data Fin : Nat -> Set where
  zero : {n : Nat}            -> Fin (suc n)
  suc  : {n : Nat} -> Fin n -> Fin (suc n)

-- annoying cast function
weak :   {n : Nat} -> Fin n -> Fin (suc n)
weak  zero    = zero
weak (suc i) = suc (weak i)
```

- In Agda, `weak` is necessary to cast from `Fin n` to `Fin (suc n)`.
- Annoying. Costly at run-time.
- Need smart compiler to eliminate `weak`.
- Better: not require it in the first place $\Longrightarrow$ subtyping!

Rules for Subtyping Fin

- Rules for sub "typing" natural numbers.

$$\frac{}{0 \leq m : \mathsf{Nat}} \qquad \frac{n \leq m : \mathsf{Nat}}{1+n \leq 1+m : \mathsf{Nat}} \qquad \frac{n \leq m : \mathsf{Nat}}{n \leq 1+m : \mathsf{Nat}}$$

- Rule for monotone functions (here : Fin).

$$\frac{\mathsf{Fin} \leq \mathsf{Fin} : +\mathsf{Nat} \to \mathsf{Set} \qquad n \leq m : \mathsf{Nat}}{\mathsf{Fin}\, n \leq \mathsf{Fin}\, m : \mathsf{Set}}$$

- Reflexivity and subsumption rule:

$$\frac{t : T}{t \leq t : T} \qquad \frac{t : T \qquad T \leq T' : \mathsf{Set}}{t : T'}$$

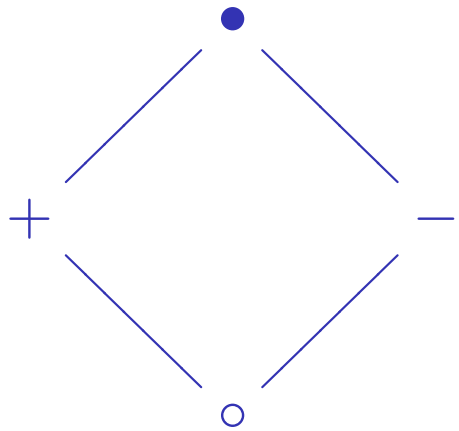
- Seems easy... let's look closer...

Variances/Polarities

- We distinguish four kinds of function types $pU \longrightarrow T$:

  $p = \bullet$ constant (invariant) function
  $p = +$ monotone (covariant) function
  $p = -$ antitone (contravariant) function
  $p = \circ$ arbitrary (mixed-variant) function (no information)

- Variances are ordered by the information they provide:



Examples

- Contravariance:

$$\lambda X.\, X \to \mathsf{Nat} \;:\; -\mathsf{Set} \to \mathsf{Set}$$

- Mixed-variance:

$$\lambda X.\, X \to X \;:\; \circ\mathsf{Set} \to \mathsf{Set}$$

- Twice contra = covariant:

$$\lambda X.\, (X \to \mathsf{Nat}) \to \mathsf{Nat} \;:\; +\mathsf{Set} \to \mathsf{Set}$$

- Constant:

$$\lambda X.\, \mathsf{Nat} \;:\; \bullet\mathsf{Set} \to \mathsf{Set}$$

Variance Composition

If $g : pB \to C$ and $f : qA \to B$ then $g \circ f : (pq)A \to C$.

pq	$\circ$	$+$	$-$	$\bullet$
$\circ$	$\circ$	$\circ$	$\circ$	$\bullet$
$+$	$\circ$	$+$	$-$	$\bullet$
$-$	$\circ$	$-$	$+$	$\bullet$
$\bullet$	$\bullet$	$\bullet$	$\bullet$	$\bullet$

Typing for Variant Function Types

- Variances are stored in the context Γ:

$$\frac{\Gamma, x : pU \vdash t : T}{\Gamma \vdash \lambda xt : pU \to T}$$

- Interpret $\Gamma \vdash t : T$ as $t : (\Gamma \to T)$.
- Contra- and invariant variables are not usable

$$x : pU \not\vdash x : U \text{ if } p \in \{-, \bullet\}$$

because $\lambda x.x$ is not a antitone nor constant function.

$$\frac{x : pU \in \Gamma \text{ with } p \leq +}{\Gamma \vdash x : U}$$

Function Application

- How a apply variant functions?

$$\frac{\Gamma \vdash t : pU \to T \qquad \Gamma \vdash u : U}{\Gamma \vdash t\, u : T}\,??$$

- Consider $\Gamma = (x : +V)$.

$$\lambda x.\ t\ x\ (u\ x) : +U \to T$$

If t is p-variant in its second arg., what is the condition on u in dependence on p?

$p = \circ$	u constant	$\bullet$
$p = +$	u monotone	$+$
$p = -$	u antitone	$-$
$p = \bullet$	u arbitrary	$\circ$

- We write this as $p^{-1}+$ (inverse composition).

Typing of Function Application

- General rule for one-variable contexts:

$$\frac{x : qV \vdash t : pU \to T \qquad x : p^{-1}qV \vdash u : U}{x : qV \vdash t\,u : T}$$

- Law for inverse composition: $p^{-1}x \leq y \iff x \leq py$.

$p^{-1}q$	$\circ$	$+$	$-$	$\bullet$
$\circ$	$\circ$	$\bullet$	$\bullet$	$\bullet$
$+$	$\circ$	$+$	$-$	$\bullet$
$-$	$\circ$	$-$	$+$	$\bullet$
$\bullet$	$\circ$	$\circ$	$\circ$	$\circ$

- Application rule:

$$\frac{\Gamma \vdash t : pU \to T \qquad p^{-1}\Gamma \vdash u : U}{\Gamma \vdash t\,u : T}$$

Summary: Variances for Simple Types

- Typing:

$$\frac{x : pU \in \Gamma}{\Gamma \vdash x : U}\ p \leq +
\qquad
\frac{\Gamma, x : pU \vdash t : T}{\Gamma \vdash \lambda x t : pU \to T}$$

$$\frac{\Gamma \vdash t : pU \to T \qquad p^{-1}\Gamma \vdash u : U}{\Gamma \vdash t\, u : T}$$

- Subtyping:

$$\frac{x : pU \in \Gamma}{\Gamma \vdash x \leq x : U}\ p \leq +
\qquad
\frac{\Gamma, x : pU \vdash t \leq t' : T}{\Gamma \vdash \lambda x t \leq \lambda x t' : pU \to T}$$

$$\frac{\Gamma \vdash t \leq t' : pU \to T \qquad p^{-1}\Gamma \vdash u \leq^{p} u' : U}{\Gamma \vdash t\, u \leq t'\, u' : T}$$

where $\leq^{+}$ is $\leq$ and $\leq^{-}$ is $\geq$ and $\leq^{\circ}$ is $=$ and $\leq^{\bullet}$ is always true.

Variant Dependent Function Type

- What's the deal? Just do it!

$$\frac{\Gamma, x : pU \vdash t : T}{\Gamma \vdash \lambda xt : (x : pU) \to T}$$

- Example: conceiving Fin as a subset type:

```
record Bounded (n : Nat) : Set where
  constructor bounded
  field
    num    : Nat
    .proof : num < n
```

we have (in fantasy Agda)

```
emb : (n : +Nat) -> Bounded (suc n)
emb n = bounded n trivial
```

Monotone Dependent Function Type

- Well-typedness

$$n : +\mathsf{Nat} \vdash \mathsf{bounded}\ n\ \mathsf{trivial} : \mathsf{Bounded}\,(\mathsf{suc}\ n)$$

  implies well-formedness of type

$$n : +\mathsf{Nat} \vdash \mathsf{Bounded}\,(\mathsf{suc}\ n) : \mathsf{Set}$$

- Ok for this example, because Bounded is monotone.

Antitone Codomain

Example: lower bounded natural numbers $\{num \mid num > n\}$

```
record Above (n : Nat) : Set where
  constructor above
  field
    num    : Nat
    .proof : num > n
```

then what about?

```
emb : (n : +Nat) -> Above n
emb n = above (suc n) trivial
```

Crumbling Paradigm

- Now

$$n : +\mathsf{Nat} \vdash \mathsf{above}\ (\mathsf{suc}\ n)\ \mathsf{trivial} : \mathsf{Above}\ n$$

  but not (!)

$$n : +\mathsf{Nat} \vdash \mathsf{Above}\ n : \mathsf{Set}$$

- We have to give up on the principle

$$\Gamma \vdash t : T \implies \Gamma \vdash T : \mathsf{Set}$$

New Paradigm

- We still have

$$n : \circ\mathsf{Nat} \vdash \mathsf{Above}\, n : \mathsf{Set}$$

- Idea: lose some information when going to the type side.

$$\Gamma \vdash t : T \implies \widehat{\Gamma} \vdash T : \mathsf{Set}$$

- First attempt: $\widehat{p} = \circ$.

Function Type Formation

- Well-formedness of function types:

$$\frac{\Gamma \vdash U : \mathsf{Set} \qquad \Gamma, x : \widehat{p} U \vdash T : \mathsf{Set}}{\Gamma \vdash (x : pU) \to T : \mathsf{Set}}$$

- Example:

$$\frac{\vdash \mathsf{Nat} : \mathsf{Set} \qquad \dfrac{\mathsf{Above} : -\mathsf{Nat} \to \mathit{Set} \qquad \circ n : \mathsf{Nat} \vdash n : \mathsf{Nat}}{\circ n : \mathsf{Nat} \vdash \mathsf{Above}\ n : \mathsf{Set}}}{\vdash (n : +\mathsf{Nat}) \to \mathsf{Above}\ n : \mathsf{Set}}$$

Heterogeneous Subtyping

- Consider a derivation of $\mathsf{emb}\,0 \leq \mathsf{emb}\,n$.

$$\frac{\mathsf{emb} : (n : +\mathsf{Nat}) \to \mathsf{Above}\,n \qquad 0 \leq n : \mathsf{Nat}}{\mathsf{emb}\ 0 \leq \mathsf{emb}\ n : ?}$$

- Need to compare things at different type!
- Subtyping becomes $(\Gamma \vdash t : T) \leq (\Gamma' \vdash t' : T')$.

$$\frac{\mathsf{emb} : (n : +\mathsf{Nat}) \to \mathsf{Above}\,n \qquad 0 : \mathsf{Nat} \ \leq\ n : \mathsf{Nat}}{\mathsf{emb}\ 0 : \mathsf{Above}\ 0 \ \leq\ \mathsf{emb}\ n : \mathsf{Above}\ n}$$

A Problem with Large Eliminations

- Consider a type defined by cases:

$$\begin{array}{lcl} T & : & \circ\mathsf{Bool} \to \mathsf{Set} \\ T\ \mathsf{true} & = & \mathsf{Nat} \\ T\ \mathsf{false} & = & \mathsf{Nat} \to \mathsf{Nat} \end{array}$$

- Assume a constant function $g : (b : \bullet\mathsf{Bool}) \to T\ b$.
- Since g is constant, $g\ \mathsf{true} = g\ \mathsf{false}$.

$$g\ \mathsf{true} : \mathsf{Nat} \;=\; g\ \mathsf{false} : \mathsf{Nat} \to \mathsf{Nat}$$

- How can a natural number be equal to a function??
- Moral: type $(b : \bullet\mathsf{Bool}) \to T\ b$ is bad!

Shape

- To make sense of $t : T \leq t' : T'$, we require T and T' to have the same shape.

emb 0	:	Above 0	$\leq$	emb n	:	Above n	good
g true	:	Nat	$=$	g false	:	Nat $\to$ Nat	bad
A	:	Set i	$\leq$	B	:	Set j	good
A	:	Set	$\leq$	B	:	Set $\to$ Set	bad

- Two types have the same shape if they erase to the same thing:

$$T \approx T' \iff |T| = |T'|$$

- Choose sensible erasure function $|\cdot|$.

$$\begin{array}{lcl} |\text{Above } n| & = & \text{“Above } (-1)\text{”} \\ |\text{Bounded } n| & = & \text{“Bounded } \omega\text{”} \\ |\text{Set } i| & = & \text{Set } \omega \end{array}$$

Shape Irrelevance

- Instead of erasure, define shape equivalence directly:

$$\begin{array}{lcl} \text{Above } 0 & \approx & \text{Above } n \\ \text{Bounded } n & \approx & \text{Bounded } m \\ \text{Set } i & \approx & \text{Set } j \end{array}$$

- Introduce a new variance $\circledwedge$, called shape irrelevance.

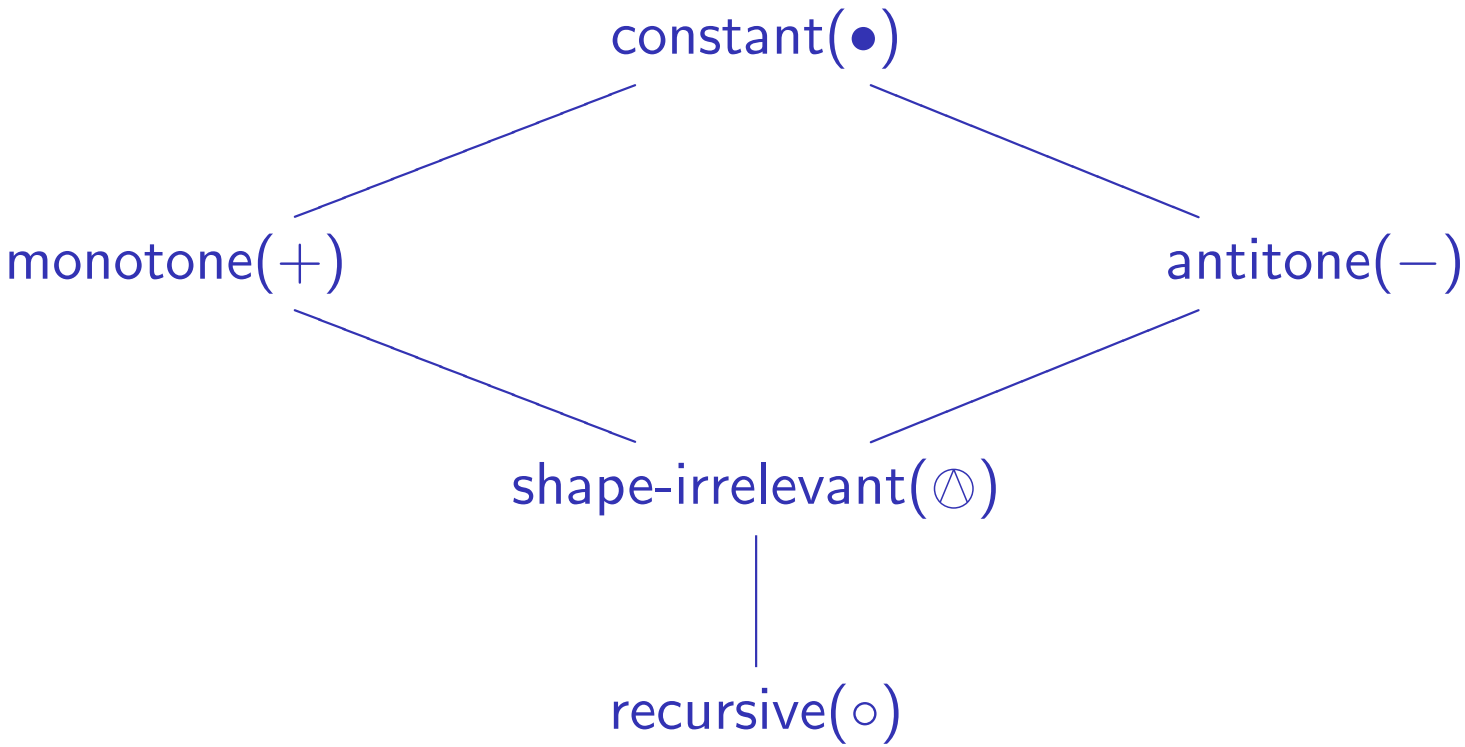
$$\begin{array}{lcl} \text{Above} & : & \circledwedge\text{Nat} \to \text{Set} \\ \text{Set}\ \_ & : & \circledwedge\text{Nat} \to \text{Type} \end{array}$$

- Types T defined by cases are not shape irrelevant!
- Correct information loss: $\widehat{\circ} = \circ$ otherwise $\widehat{p} = \circledwedge$.

$$\frac{\Gamma \vdash U : \text{Type} \qquad \Gamma, x : \widehat{p}U \vdash T : \text{Type}}{\Gamma \vdash (x : pU) \to T : \text{Type}}$$

Information Order

Lattice:



Model

- Types are modeled by partial orders $\mathcal{A}$ on terms.
- Variances (except $\circledwedge$) are operations on types:

$$\begin{array}{lcl} +\mathcal{A} & = & \mathcal{A} \\ -\mathcal{A} & = & \{(a,b) \mid (b,a) \in \mathcal{A}\} \\ \circ\mathcal{A} & = & \mathcal{A} \cap -\mathcal{A} \\ \bullet\mathcal{A} & = & \{(a,b) \mid (a,b'),(b',a) \in \mathcal{A}\} \end{array}$$

- Subtyping $T \leq T'$ is inclusion $\mathcal{A} \subseteq \mathcal{A}'$.
- Erasure $|T|$ is interpreted as a limit.
- Heterogeneous subtyping $t : T \leq t' : T'$ is interpreted via casts:

$$\uparrow_T^{|T|} t \leq \uparrow_{T'}^{|T'|} \in \mathcal{A} \text{ where } \mathcal{A} \text{ interprets } |T|$$

Status of Work

- Integrates my previous work on irrelevance.
- Prototype MiniAgda (some known bugs!).
- Started to work out model.
- Open:
  1. Infer variances automatically.
  2. Hidden arguments and subtyping $\Longrightarrow$ solve subtyping constraints.

Related Work

1. Subtyping Dependent Types (Aspinall, Compagnoni, ... 1990s)
2. Polarized Polymorphic Higher-Order Subtyping (Cardelli, Pierce, M. Steffen 1990s)
3. Proof Irrelevance in LF (Pfenning 2001)
4. Implicit Calculus of Constructions (Miquel, Barras, Bernardo 2008)
5. Erasure Pure Type Systems (Mishra-Linger, Sheard 2008)
6. Pure Subtyping Systems (Hutchins 2010)