Subeffecting and subtyping

- We have now seen subeffecting at work.
- The main ideas of all of these are:
  - compute types and annotations independent of context,
  - allow to weaken the outcomes whenever convenient.
- Weakening provides a form of context-sensitiveness.
- In (shape conformant) subtyping we may also weaken annotation deeper in the type.

Example: parity analysis

- The natural number 1 can be analysed to have type $Nat^{\{O\}}$.
- A function like $double$ on naturals should work for all naturals: $Nat^{\{O,E\}} \rightarrow Nat^{\{E\}}$.
- The type of 1 can then be weakened to $Nat^{\{O,E\}}$ as it is passed into $double$, without influencing the type and other uses of 1.

$$\begin{array}{l} \textbf{let}\ one = \quad 1\ \textbf{in} \\ \textbf{let}\ double = \lambda_{G} y.\, y * 2\ \textbf{in} \\ one * double\ one \end{array}$$

Limitations to subeffecting and subtyping

- Weakening prevents certain forms of poisoning,
- but it does not help propagate analysis information.
- For *id* on naturals we expect the type $Nat^{\{O,E\}} \rightarrow Nat^{\{O,E\}}$.
- However, we also know that O inputs leads to O outputs, and similar for E.
- Our annotated types cannot represent this information.
- Is it realistic that *id* 1 and 1 give different analyses?

Polyvariance

- We consider only let-polyvariance.
- Exactly analogous to let-polymorphism, but for annotations.
- For *id* we can instead derive the type $\forall \beta. \, Nat^{\beta} \to Nat^{\beta}$.
- For *id* 1 we can choose $\beta = \{ O \}$ so that *id* 1 has annotation $\{ O \}$.
- Allows us to propagate properties through functions that are property-agnostic.
- Polyvariant analyses with subtyping are current state of the art.
- But it depends somewhat on the analysis.

Annotated polyvariant types

$\varphi \quad \in \quad \mathbf{Ann}$ annotations

$$\varphi \quad ::= \quad \beta \mid \emptyset \mid \{\pi\} \mid \varphi_1 \cup \varphi_2$$

Annotated polyvariant types

φ	$\in$	$\mathbf{Ann}$	annotations
$\widehat{\tau}$	$\in$	$\widehat{\mathbf{Ty}}$	annotated types

$$\varphi \quad ::= \quad \beta \mid \emptyset \mid \{\pi\} \mid \varphi_1 \cup \varphi_2$$
$$\widehat{\tau} \quad ::= \quad \alpha \mid \mathit{Nat} \mid \mathit{Bool} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$$

Annotated polyvariant types

φ	$\in$	$\mathbf{Ann}$	annotations
$\widehat{\tau}$	$\in$	$\widehat{\mathbf{Ty}}$	annotated types
$\widehat{\sigma}$	$\in$	$\widehat{\mathbf{TyScheme}}$	annotated type schemes

$$
\begin{array}{lcl}
\varphi & ::= & \beta \mid \emptyset \mid \{\pi\} \mid \varphi_1 \cup \varphi_2 \\
\widehat{\tau} & ::= & \alpha \mid \mathit{Nat} \mid \mathit{Bool} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \\
\widehat{\sigma} & ::= & \widehat{\tau} \mid \forall \alpha.\, \widehat{\sigma}_1 \mid \forall \beta.\, \widehat{\sigma}_1
\end{array}
$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Annotated polyvariant types

φ	$\in$	$\mathbf{Ann}$	annotations
$\widehat{\tau}$	$\in$	$\widehat{\mathbf{Ty}}$	annotated types
$\widehat{\sigma}$	$\in$	$\widehat{\mathbf{TyScheme}}$	annotated type schemes
$\widehat{\Gamma}$	$\in$	$\widehat{\mathbf{TyEnv}}$	annotated type environments

$$
\begin{array}{lcl}
\varphi & ::= & \beta \mid \emptyset \mid \{\pi\} \mid \varphi_1 \cup \varphi_2 \\
\widehat{\tau} & ::= & \alpha \mid \mathit{Nat} \mid \mathit{Bool} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \\
\widehat{\sigma} & ::= & \widehat{\tau} \mid \forall \alpha . \widehat{\sigma}_1 \mid \forall \beta . \widehat{\sigma}_1 \\
\widehat{\Gamma} & ::= & [\,] \mid \widehat{\Gamma}_1[x \mapsto \widehat{\sigma}]
\end{array}
$$

Annotated polyvariant types

φ	$\in$	$\mathbf{Ann}$	annotations
$\widehat{\tau}$	$\in$	$\widehat{\mathbf{Ty}}$	annotated types
$\widehat{\sigma}$	$\in$	$\widehat{\mathbf{TyScheme}}$	annotated type schemes
$\widehat{\Gamma}$	$\in$	$\widehat{\mathbf{TyEnv}}$	annotated type environments

$$
\begin{array}{lcl}
\varphi & ::= & \beta \mid \emptyset \mid \{\pi\} \mid \varphi_1 \cup \varphi_2 \\
\widehat{\tau} & ::= & \alpha \mid \mathit{Nat} \mid \mathit{Bool} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \\
\widehat{\sigma} & ::= & \widehat{\tau} \mid \forall\alpha.\,\widehat{\sigma}_1 \mid \forall\beta.\,\widehat{\sigma}_1 \\
\widehat{\Gamma} & ::= & [\,] \mid \widehat{\Gamma}_1[x \mapsto \widehat{\sigma}]
\end{array}
$$

$\widehat{\Gamma} \vdash_{\mathrm{CFA}} t : \widehat{\sigma}$ control-flow analysis

Is this enough?

$$\begin{array}{l}\textbf{let } f = \lambda_{\mathrm{F}} x.\ \mathit{True} \textbf{ in}\\ \textbf{let } g = \lambda_{\mathrm{G}} k.\ \textbf{if } f\ 0 \textbf{ then } k \textbf{ else } (\lambda_{\mathrm{H}} y.\ \mathit{False}) \textbf{ in}\\ g\ f\end{array}$$

A (mono)type for $g\ f$ is $v1 \xrightarrow{\{\mathrm{F}\}\cup\{\mathrm{H}\}} \mathit{Bool}$.

$\{\mathrm{H}\}$ is contributed by the else-part, $\{\mathrm{F}\}$ comes from the parameter passed to g.

But what is the type of g that can lead to such type?

Is this enough?

$$\begin{array}{l}\textbf{let}\ f = \lambda_{\mathrm{F}} x.\ \mathit{True}\ \textbf{in}\\ \textbf{let}\ g = \lambda_{\mathrm{G}} k.\ \textbf{if}\ f\ 0\ \textbf{then}\ k\ \textbf{else}\ (\lambda_{\mathrm{H}} y.\ \mathit{False})\ \textbf{in}\\ g\ f\end{array}$$

A (mono)type for $g\ f$ is $v1 \xrightarrow{\{\mathrm{F}\}\cup\{\mathrm{H}\}} \mathit{Bool}$.

$\{\mathrm{H}\}$ is contributed by the else-part, $\{\mathrm{F}\}$ comes from the parameter passed to g.

But what is the type of g that can lead to such type?

$g : \forall a. \forall \beta. (a \xrightarrow{\beta} \mathit{Bool}) \xrightarrow{\mathrm{G}} (a \xrightarrow{\beta\cup\{\mathrm{H}\}} \mathit{Bool})$

But how can we manipulate such annotations correctly?

☞ Add a few rules

Polyvariant type system: generalisation

Introduction for type variables:

$$\frac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t : \widehat{\sigma} \qquad \alpha \notin \mathit{ftv}(\Gamma)}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t : \forall \alpha . \widehat{\sigma}} \ [\mathit{cfa\text{-}gen}]$$

Introduction for annotation variables:

$$\frac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t : \widehat{\sigma} \qquad \beta \notin \mathit{fav}(\Gamma)}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t : \forall \beta . \widehat{\sigma}} \ [\mathit{cfa\text{-}ann\text{-}gen}]$$

Here $fav(\Gamma)$ computes the free annotation variables in Γ.

Polyvariant type system: instantiation

Elimination for type variables:

$$\frac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t : \forall \alpha.\, \widehat{\sigma}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t : [\alpha \mapsto \widehat{\tau}]\widehat{\sigma}} \; [\textit{cfa-inst}]$$

Elimination for annotation variables:

$$\frac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t : \forall \beta.\, \widehat{\sigma}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t : [\beta \mapsto \varphi]\widehat{\sigma}} \; [\textit{cfa-ann-inst}]$$

Polyvariant type system: subeffecting again

To align the types of the then-part and else-part, and to match arguments to function types, we still need subeffecting.

Recap:

$$\frac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t : \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t : \widehat{\tau}_1 \xrightarrow{\varphi \cup \varphi'} \widehat{\tau}_2}\ [\textit{cfa-sub}]$$

then-part: β can be weakened to $\beta \cup \{\mathrm{H}\}$.

else-part: $\{\mathrm{H}\}$ can be weakened to $\{\mathrm{H}\} \cup \beta$.

But these are not the same!

When are two annotations equal?

The type system can never guess, so we have to tell it when.

$$\frac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t : \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \quad \varphi \equiv \varphi'}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t : \widehat{\tau}_1 \xrightarrow{\varphi'} \widehat{\tau}_1} \ [\textit{cfa-eq}]$$

In other words: you may replace equals by equals.

☞ $\{\mathrm{H}\} \cup \beta$ by $\beta \cup \{\mathrm{H}\}$

Problem now becomes to define/axiomatize equality for these annotations.

Equality of annotations axiomatized (1)

$$\frac{}{\varphi \equiv \varphi}\ [\textit{q-refl}]$$

$$\frac{\varphi' \equiv \varphi}{\varphi \equiv \varphi'}\ [\textit{q-symm}]$$

$$\frac{\varphi \equiv \varphi'' \quad \varphi'' \equiv \varphi'}{\varphi \equiv \varphi'}\ [\textit{q-trans}]$$

$$\frac{\varphi_1 \equiv \varphi_1' \quad \varphi_2 \equiv \varphi_2'}{\varphi_1 \cup \varphi_2 \equiv \varphi_1' \cup \varphi_2'}\ [\textit{q-join}]$$

Equality of annotations axiomatized (2)

$$\frac{}{\{\,\} \cup \varphi \equiv \varphi}\;[\textit{q-unit}]$$

$$\frac{}{\varphi \cup \varphi \equiv \varphi}\;[\textit{q-idem}]$$

$$\frac{}{\varphi_1 \cup \varphi_2 \equiv \varphi_2 \cup \varphi_1}\;[\textit{q-comm}]$$

$$\frac{}{\varphi_1 \cup (\varphi_2 \cup \varphi_3) \equiv (\varphi_1 \cup \varphi_2) \cup \varphi_3}\;[\textit{q-ass}]$$

UCAI

This combination of axioms often occurs:

- Unit
- Commutativity
- Associativity
- Idempotency

☞ Modulo UCAI

What about the algorithm?

- We still perform generalization in the let.
- And instantiation in the variable case.
- Recall:
  - The algorithm unifies types and identifies annotation variables.
  - It collects constraints on the latter.
- After algorithm $\mathcal{W}_{\mathrm{CFA}}$, we solve the constraints to obtain annotation variables.
- In the monovariant setting this was fine: correctness did not depend on the context.
- In a polyvariant setting, the context plays a role

☞ Constraints on annotations must be propagated along.

Some variations

- Idea 1: simply store all constraints in the type.
- During instantation refresh type and annotations variables in the type, and the constraint set (consistently).
- Includes also trivial and irrelevant constraints.
- Idea 2: simplify constraints as much as possible before storing them.
- Simplification can take many forms.
- Takes place as part of generalisation.
- Type schemes store constraints sets: rather like qualified types.

Simplification

- Simplification = intermediate constraint solving.
- In both cases, annotations left unconstrained can be defaulted to the best possible.
- However, annotation variables that occur in the type to be generalized must be left unharmed.
- Why? Annotation variables provide flexibility for propagation.
  ☞ Defaulting throws that flexibility away.

Example (to illustrate)

- Assume $\mathcal{W}_{\mathrm{CFA}}$ returns type $(v1 \xrightarrow{\beta_1} v1) \xrightarrow{\beta_2} (v1 \xrightarrow{\beta_3} v1)$ and constraint set $\{\beta_2 \supseteq \{\mathrm{G}\}, \beta_3 \supseteq \beta_4, \beta_4 \supseteq \beta_1, \beta_5 \supseteq \{\mathrm{H}\}, \beta_3 \supseteq \beta\}$
- And that β occurs free in $\widehat{\Gamma}$.
- β_5 is not relevant, so it can be omitted (set to $\{\mathrm{H}\}$).
- β_4 is not relevant either, but removing it implies we must add $\beta_3 \supseteq \beta_1$.
- Neither $\beta_2 \supseteq \{\mathrm{G}\}$ and $\beta_3 \supseteq \beta$ may be touched.
- Remember the invariant to keep unification simple: only annotation variables in types.

Constrained types and type schemes

Introduce an additional layer of types (a la qualified types):

$$\begin{array}{lcl}\widehat{\tau} & ::= & \alpha \mid Nat \mid Bool \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2 \\ \widehat{\rho} & ::= & \widehat{\tau} \mid c \Rightarrow \widehat{\rho} \\ \widehat{\sigma} & ::= & \widehat{\rho} \mid \forall \alpha.\, \widehat{\sigma}_1 \mid \forall \beta.\, \widehat{\sigma}_1\end{array}$$

Generalisation and instantiation

- Instantiation provides fresh variables for universally quantified variables.
- Generalisation invokes the simplifier.
- Simplification can be performed by a worklist algorithm, that leaves certain (which?) variables untouched.
  ☞ Considers them to be constants
- Some say: simple duplication (no simplification) is not feasible.
- Let-definition is like a compartment: we only care for its interface to the world, not what happens inside.