[School of Mathematical and Computer Sciences (MACS)]

Type Based Static Analysis

Jurriaan Hage

School of Mathematical and Computer Sciences
E-mail: J.Hage@hw.ac.uk
With contributions by Stefan Holdermans

Topics

- Terminology, context, motivation
- Some really basic functional programming
- Typing the polymorphic lambda calculus
- Type based static analyis
  - control-flow analysis
  - adding effects (if time permits)
- Contents taken from a master course largely based on Chapter 5 of Nielson, Nielson and Hankin.

A bit about me

- Professor at Heriot-Watt University, Edinburgh
- Research focus:
  - static analysis of functional languages
  - type error diagnosis
  - maintainer of the Helium Haskell compiler
    - But all we do today is in a strict setting!

Static Analysis and Types

- Static program analysis: compile-time techniques for approximating the set of values or behaviours that arise at run-time when a program is executed.
- Applications: verification, optimization.
- Different approaches: data-flow analysis, constraint-based analysis, abstract interpretation, type-based analysis.
- Type-based analysis: equipping a programming language with a nonstandard type system that keeps track of some properties of interest.
- Advantages: reuse of tools, techniques, and infrastructure (polymorphism, subtyping, type inference, . . .).
- Focus: accuracy vs. modularity.

Examples

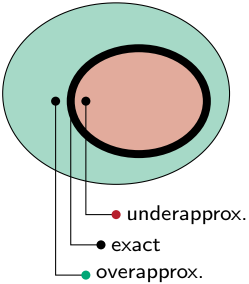
HERIOT WATT UNIVERSITY

. . .
Side-effect analysis.
Callability analysis.
Reachability analysis.
Uniqueness analysis.
Totality analysis.
Security analysis.
Strictness analysis.
Sharing analysis.
Trust analysis.
. . .

. . .
Sign analysis.
Flow analysis.
Control-flow analysis.
Class-hierarchy analysis.
Region analysis.
Binding-time analysis.
Alias analysis.
Communication analysis.
Escape analysis.
. . .

Accuracy

- Establishing nontrivial properties of programs is in general undecidable (halting problem, Rice's theorem).
- In static analysis we have to settle for "useful" approximations of properties.
- "Useful" means: sound ("erring at the safe side") and accurate (as precise as possible).



How do types help us?

- Consider a higher-order setting

$$compose\ f\ g = \lambda x. f\ (g\ x)$$

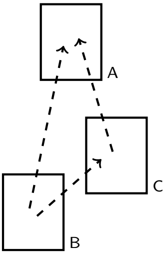
- When we analyse $g\ x$ and $f\ (g\ x)$, we must analyze their bodies
- However, not every combination of functions can arise
  - Only those where the output of g is compatible with the input type of f.
- A type based approach to analyze takes advantage of this implicitly, weeding out combinations that cannot actually occur
- What information we shall compute, also depends on the type

How would this be for, say, Python?

- O.O. style: what `x.foo();` can target depends on what the receiver `x` can be (and vice versa): type and control-flow are mutually dependent
- If you call a function parameter f of a function p in this setting you have even fewer clues, particularly if you export p as part of a library.
- Here, a more natural approach is data-flow analysis (where functions are considered data!)

Modularity

- Breaking up a (large) program in smaller units or modules is generally considered good programming style.
- Separate compilation: compile each module in isolation.
- Advantage: only modules that have been edited need to be recompiled.
- To facilitate seperate compilation, each unit of compilation needs to be analysed in isolation, i.e., without knowledge of how it's used from within the rest of the program.



☞ Tension between accuracy and modularity: whole-program analysis typically yields more precise results.

Hindley-Milner and Algorithm W

A simple functional language

f, x	$\in$	$\mathbf{Var}$	variables
t	$\in$	$\mathbf{Tm}$	terms

$$
\begin{array}{lll}
t & ::= & \mid\ x \mid \lambda x.\, t_1 \\
 & \mid & t_1\ t_2 \\
 & \mid &
\end{array}
$$

[School of Mathematical and Computer Sciences (MACS)]

A simple functional language

f, x	$\in$	$\mathbf{Var}$	variables
t	$\in$	$\mathbf{Tm}$	terms

$$
\begin{array}{lll}
t & ::= & \quad | \; x \; | \; \lambda x.\, t_1 \\
 & | & t_1 \; t_2 \\
 & | &
\end{array}
$$

A simple functional language

$$
\begin{array}{rcll}
f, x & \in & \mathbf{Var} & \text{variables} \\
t & \in & \mathbf{Tm} & \text{terms}
\end{array}
$$

$$
\begin{array}{rll}
t & ::= & \mid x \mid \lambda x.\, t_1 \\
& \mid & t_1\ t_2 \qquad \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \\
& \mid &
\end{array}
$$

[School of Mathematical and Computer Sciences (MACS)]

A simple functional language

f, x	$\in$	$\mathbf{Var}$	variables
t	$\in$	$\mathbf{Tm}$	terms

$$
\begin{array}{lll}
t & ::= & \mid x \mid \lambda x.\, t_1 \mid \mu f.\, \lambda x.\, t_1 \\
& \mid & t_1\ t_2 \qquad \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \\
& \mid &
\end{array}
$$

A simple functional language

$$
\begin{array}{rcll}
n & \in & \mathbf{Num} = \mathbb{N} & \text{numerals} \\
f, x & \in & \mathbf{Var} & \text{variables} \\
 & & & \\
t & \in & \mathbf{Tm} & \text{terms}
\end{array}
$$

$$
\begin{array}{rcl}
t & ::= & n \quad \mid x \mid \lambda x.\, t_1 \mid \mu f.\, \lambda x.\, t_1 \\
 & \mid & t_1\ t_2 \quad \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \\
 & \mid &
\end{array}
$$

A simple functional language

$$
\begin{array}{rcll}
n & \in & \mathbf{Num} = \mathbb{N} & \text{numerals} \\
f, x & \in & \mathbf{Var} & \text{variables} \\
 & & & \\
t & \in & \mathbf{Tm} & \text{terms}
\end{array}
$$

$$
\begin{array}{rcl}
t & ::= & n \mid \texttt{false} \mid \texttt{true} \mid x \mid \lambda x.\, t_1 \mid \mu f.\lambda x.\, t_1 \\
 & \mid & t_1\ t_2 \mid \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \mid \textbf{let } x = t_1 \textbf{ in } t_2 \\
 & \mid &
\end{array}
$$

A simple functional language

n	$\in$	$\mathbf{Num} = \mathbb{N}$	numerals
f, x	$\in$	$\mathbf{Var}$	variables
$\oplus$	$\in$	$\mathbf{Op}$	binary operators
t	$\in$	$\mathbf{Tm}$	terms

$$
\begin{array}{rcl}
t & ::= & n \mid \texttt{false} \mid \texttt{true} \mid x \mid \lambda x.\, t_1 \mid \mu f.\, \lambda x.\, t_1 \\
 & \mid & t_1\ t_2 \mid \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \mid \textbf{let } x = t_1 \textbf{ in } t_2 \\
 & \mid & t_1 \oplus t_2
\end{array}
$$

Some simple terms (aka programs)

if `true` **then** `false` **else** `true`

Some simple terms (aka programs)

if `true` **then** `false` **else** `true`

$\lambda x.\, x$

Some simple terms (aka programs)

$\textbf{if}\ \texttt{true}\ \textbf{then}\ \texttt{false}\ \textbf{else}\ \texttt{true}$

$\lambda x.\, x$

$(\lambda x.\, x + 1)\ 2$

Some simple terms (aka programs)

$\textbf{if}\ \texttt{true}\ \textbf{then}\ \texttt{false}\ \textbf{else}\ \texttt{true}$

$\lambda x.\, x$

$(\lambda x.\, x + 1)\ 2$

$\textbf{let}\ d\ a\ b\ c = b * b - 4 * a * c\ \textbf{in}\ d\ 1\ 3\ 2$

Some less simple terms (aka programs)

$\mathbf{let}\ niet\ b = \mathbf{if}\ b\ \mathbf{then}\ \mathtt{false}\ \mathbf{else}\ \mathtt{true}\ \mathbf{in}\ niet\ \mathtt{true}$

Some less simple terms (aka programs)

$\textbf{let}\ \mathit{niet}\ b = \textbf{if}\ b\ \textbf{then}\ \texttt{false}\ \textbf{else}\ \texttt{true}\ \textbf{in}\ \mathit{niet}\ \texttt{true}$

$\textbf{let}\ \mathit{apply} = \lambda f.\lambda x.f\ x\ \textbf{in}\ \mathit{apply}\ (\lambda x.x + 1)\ 2$

$\textbf{let}\ \mathit{revapp} = \lambda x.\lambda f.f\ x\ \textbf{in}\ \mathit{revapp}\ 2\ (\lambda x.x + 1)$

Some less simple terms (aka programs)

$$\textbf{let}\ \mathit{niet}\ b = \textbf{if}\ b\ \textbf{then}\ \texttt{false}\ \textbf{else}\ \texttt{true}\ \textbf{in}\ \mathit{niet}\ \texttt{true}$$

$$\textbf{let}\ \mathit{apply} = \lambda f.\lambda x.f\ x\ \textbf{in}\ \mathit{apply}\ (\lambda x.x+1)\ 2$$

$$\textbf{let}\ \mathit{revapp} = \lambda x.\lambda f.f\ x\ \textbf{in}\ \mathit{revapp}\ 2\ (\lambda x.x+1)$$

$$\textbf{let}\ \mathit{flip} = \lambda f.\lambda x.\lambda y.f\ y\ x$$

Some less simple terms (aka programs)

$\mathbf{let}\ \mathit{niet}\ b = \mathbf{if}\ b\ \mathbf{then}\ \mathtt{false}\ \mathbf{else}\ \mathtt{true}\ \mathbf{in}\ \mathit{niet}\ \mathtt{true}$

$\mathbf{let}\ \mathit{apply} = \lambda f.\lambda x.f\ x\ \mathbf{in}\ \mathit{apply}\ (\lambda x.x + 1)\ 2$

$\mathbf{let}\ \mathit{revapp} = \lambda x.\lambda f.f\ x\ \mathbf{in}\ \mathit{revapp}\ 2\ (\lambda x.x + 1)$

$\mathbf{let}\ \mathit{flip} = \lambda f.\lambda x.\lambda y.f\ y\ x$

$\mathbf{let}\ x = 2\ \mathbf{in}\ \mathbf{let}\ y = x * x \equiv x + x\ \mathbf{in}\ \mathbf{if}\ y\ \mathbf{then}\ x\ \mathbf{else}\ 0$

Some less simple terms (aka programs)

HERIOT WATT UNIVERSITY

$\mathbf{let}\ \mathit{niet}\ b = \mathbf{if}\ b\ \mathbf{then}\ \texttt{false}\ \mathbf{else}\ \texttt{true}\ \mathbf{in}\ \mathit{niet}\ \texttt{true}$

$\mathbf{let}\ \mathit{apply} = \lambda f.\lambda x.f\ x\ \mathbf{in}\ \mathit{apply}\ (\lambda x.x+1)\ 2$

$\mathbf{let}\ \mathit{revapp} = \lambda x.\lambda f.f\ x\ \mathbf{in}\ \mathit{revapp}\ 2\ (\lambda x.x+1)$

$\mathbf{let}\ \mathit{flip} = \lambda f.\lambda x.\lambda y.f\ y\ x$

$\mathbf{let}\ x = 2\ \mathbf{in}\ \mathbf{let}\ y = x * x \equiv x + x\ \mathbf{in}\ \mathbf{if}\ y\ \mathbf{then}\ x\ \mathbf{else}\ 0$

$\mathbf{let}\ \mathit{fac} = \mu f.\lambda_{\mathrm{F}} x.\mathbf{if}\ x \equiv 0\ \mathbf{then}\ 1\ \mathbf{else}\ x * f\ (x-1)$
$\mathbf{in}\ \ \mathit{fac}\ 6$

What's missing?

- Implicit recursion, so we can't simply write

$$fac\ n = \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ x * fac\ (n-1)$$

- Lists and list comprehensions
- Datatypes and pattern matching
- Advanced types (higher-rank, type classes)
- Module system
- Many syntactic niceties
- Think of the language as a strict, desugared functional language without datatypes

What's missing?

- Implicit recursion, so we can't simply write

$$fac\ n = \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ x * fac\ (n-1)$$

- Lists and list comprehensions
- Datatypes and pattern matching
- Advanced types (higher-rank, type classes)
- Module system
- Many syntactic niceties
- Think of the language as a strict, desugared functional language without datatypes
- Something else that's missing: a type system!

A simple functional language (reprise)

f, x	$\in$	$\mathbf{Var}$	variables
t	$\in$	$\mathbf{Tm}$	terms

$$
\begin{array}{lll}
t & ::= & \mid\ x \mid \lambda\ x.\, t_1 \\
 & \mid & t_1\ t_2 \\
 & \mid &
\end{array}
$$

A simple functional language (reprise)

f, x	$\in$	$\mathbf{Var}$	variables
π	$\in$	$\mathbf{Pnt}$	program points
t	$\in$	$\mathbf{Tm}$	terms

$$
\begin{array}{lll}
t & ::= & \quad \mid x \mid \lambda_{\pi} x.\, t_1 \\
 & \mid & t_1 \; t_2 \\
 & \mid &
\end{array}
$$

[School of Mathematical and Computer Sciences (MACS)]

A simple functional language (reprise)

f, x	$\in$	$\mathbf{Var}$	variables
π	$\in$	$\mathbf{Pnt}$	program points
t	$\in$	$\mathbf{Tm}$	terms

$$
\begin{array}{lll}
t & ::= & \mid x \mid \lambda_{\pi} x.\, t_1 \\
 & \mid & t_1\ t_2 \qquad \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \\
 & \mid &
\end{array}
$$

A simple functional language (reprise)

f, x	$\in$	$\mathbf{Var}$	variables
π	$\in$	$\mathbf{Pnt}$	program points
t	$\in$	$\mathbf{Tm}$	terms

$$\begin{array}{lll} t & ::= & \mid x \mid \lambda_{\pi} x.\, t_1 \mid \mu f.\, \lambda_{\pi} x.\, t_1 \\ & \mid & t_1\ t_2 \quad \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \\ & \mid & \end{array}$$

[School of Mathematical and Computer Sciences (MACS)]

A simple functional language (reprise)

n	$\in$	$\mathbf{Num} = \mathbb{N}$	numerals
f, x	$\in$	$\mathbf{Var}$	variables
π	$\in$	$\mathbf{Pnt}$	program points
t	$\in$	$\mathbf{Tm}$	terms

$$
\begin{array}{lll}
t & ::= & n \quad \mid x \mid \lambda_\pi x.\, t_1 \mid \mu f.\, \lambda_\pi x.\, t_1 \\
 & \mid & t_1\ t_2 \quad \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \\
 & \mid &
\end{array}
$$

A simple functional language (reprise)

n	$\in$	$\mathbf{Num} = \mathbb{N}$	numerals
f, x	$\in$	$\mathbf{Var}$	variables
π	$\in$	$\mathbf{Pnt}$	program points
t	$\in$	$\mathbf{Tm}$	terms

$$
\begin{array}{lcl}
t & ::= & n \mid \texttt{false} \mid \texttt{true} \mid x \mid \lambda_\pi x.\, t_1 \mid \mu f.\, \lambda_\pi x.\, t_1 \\
 & \mid & t_1\ t_2 \mid \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \mid \textbf{let}\ x = t_1\ \textbf{in}\ t_2 \\
 & \mid &
\end{array}
$$

[School of Mathematical and Computer Sciences (MACS)]

A simple functional language (reprise)

n	$\in$	$\mathbf{Num} = \mathbb{N}$	numerals
f, x	$\in$	$\mathbf{Var}$	variables
$\oplus$	$\in$	$\mathbf{Op}$	binary operators
π	$\in$	$\mathbf{Pnt}$	program points
t	$\in$	$\mathbf{Tm}$	terms

$$
\begin{array}{lll}
t & ::= & n \mid \texttt{false} \mid \texttt{true} \mid x \mid \lambda_\pi x.\, t_1 \mid \mu f.\, \lambda_\pi x.\, t_1 \\
 & \mid & t_1\ t_2 \mid \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \\
 & \mid & t_1 \oplus t_2
\end{array}
$$

[School of Mathematical and Computer Sciences (MACS)]

Monomorphic types

$\tau \quad \in \quad \mathbf{Ty}$ types

$\tau \quad ::= \quad Nat \mid Bool \mid \tau_1 \to \tau_2$

Monomorphic types

τ	$\in$	**Ty**	types
Γ	$\in$	**TyEnv**	type environments

$$\tau ::= Nat \mid Bool \mid \tau_1 \to \tau_2$$
$$\Gamma ::= [\,] \mid \Gamma_1[x \mapsto \tau]$$

Monomorphic types

τ	$\in$	$\mathbf{Ty}$	types
Γ	$\in$	$\mathbf{TyEnv}$	type environments

$$\tau \quad ::= \quad \mathit{Nat} \mid \mathit{Bool} \mid \tau_1 \rightarrow \tau_2$$
$$\Gamma \quad ::= \quad [\,] \mid \Gamma_1[x \mapsto \tau]$$

Typing judgements:

$\Gamma \vdash_{\mathrm{UL}} t : \tau$	typing

"Term t has type τ assuming that any of its free variables has the type given by Γ."

Monomorphic type system: constants

$$\frac{}{\Gamma \vdash_{\mathrm{UL}} n : Nat} \; [\textit{t-num}]$$

Monomorphic type system: constants

$$\frac{}{\Gamma \vdash_{\mathrm{UL}} n : \mathit{Nat}} \; [\textit{t-num}]$$

$$\frac{}{\Gamma \vdash_{\mathrm{UL}} \texttt{false} : \mathit{Bool}} \; [\textit{t-false}]$$

$$\frac{}{\Gamma \vdash_{\mathrm{UL}} \texttt{true} : \mathit{Bool}} \; [\textit{t-true}]$$

Monomorphic type system: variables

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_{\mathrm{UL}} x : \tau} \; [\textit{t-var}]$$

Monomorphic type system: functions

$$\frac{\Gamma[x \mapsto \tau_1] \vdash_{\text{UL}} t_1 : \tau_2}{\Gamma \vdash_{\text{UL}} \lambda_\pi x.\, t_1 : \tau_1 \to \tau_2} \; [\textit{t-lam}]$$

Monomorphic type system: functions

$$\frac{\Gamma[x \mapsto \tau_1] \vdash_{\mathrm{UL}} t_1 : \tau_2}{\Gamma \vdash_{\mathrm{UL}} \lambda_{\pi} x.\, t_1 : \tau_1 \to \tau_2} \; [\textit{t-lam}]$$

$$\frac{\Gamma[f \mapsto (\tau_1 \to \tau_2)][x \mapsto \tau_1] \vdash_{\mathrm{UL}} t_1 : \tau_2}{\Gamma \vdash_{\mathrm{UL}} \mu f.\, \lambda_{\pi} x.\, t_1 : \tau_1 \to \tau_2} \; [\textit{t-mu}]$$

Monomorphic type system: functions

$$\frac{\Gamma[x \mapsto \tau_1] \vdash_{\mathrm{UL}} t_1 : \tau_2}{\Gamma \vdash_{\mathrm{UL}} \lambda_\pi x.\, t_1 : \tau_1 \to \tau_2} \; [\textit{t-lam}]$$

$$\frac{\Gamma[f \mapsto (\tau_1 \to \tau_2)][x \mapsto \tau_1] \vdash_{\mathrm{UL}} t_1 : \tau_2}{\Gamma \vdash_{\mathrm{UL}} \mu f.\, \lambda_\pi x.\, t_1 : \tau_1 \to \tau_2} \; [\textit{t-mu}]$$

$$\frac{\Gamma \vdash_{\mathrm{UL}} t_1 : \tau_2 \to \tau \qquad \Gamma \vdash_{\mathrm{UL}} t_2 : \tau_2}{\Gamma \vdash_{\mathrm{UL}} t_1\ t_2 : \tau} \; [\textit{t-app}]$$

Monomorphic type system: conditionals

$$\frac{\Gamma \vdash_{\text{UL}} t_1 : \mathit{Bool} \quad \Gamma \vdash_{\text{UL}} t_2 : \tau \quad \Gamma \vdash_{\text{UL}} t_3 : \tau}{\Gamma \vdash_{\text{UL}} \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 : \tau} \; [\textit{t-if}]$$

Monomorphic type system: local definitions

$$\frac{\Gamma \vdash_{\text{UL}} t_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash_{\text{UL}} t_2 : \tau}{\Gamma \vdash_{\text{UL}} \textbf{let } x = t_1 \textbf{ in } t_2 : \tau} \; [t\text{-}let]$$

Monomorphic type system: binary operators

$$\frac{\Gamma \vdash_{\mathrm{UL}} t_1 : \tau_{\oplus}^{1} \quad \Gamma \vdash_{\mathrm{UL}} t_2 : \tau_{\oplus}^{2}}{\Gamma \vdash_{\mathrm{UL}} t_1 \oplus t_2 : \tau_{\oplus}} \ [\textit{t-op}]$$

Monomorphic type system: example

$$\frac{}{\Gamma \vdash_{\mathrm{UL}} \mu f\,.\,\lambda_{\mathrm{F}} x.\, \mathbf{if}\ x \equiv 0\ \mathbf{then}\ 1\ \mathbf{else}\ x * f\ (x-1) : \mathit{Nat} \rightarrow \mathit{Nat}}$$

Monomorphic type system: example

$$\dfrac{\dfrac{\vdots}{\Gamma_F \vdash_{UL} x \equiv 0 : \mathit{Bool}} \quad \dfrac{}{\Gamma_F \vdash_{UL} 1 : \mathit{Nat}} \quad \dfrac{\vdots}{\Gamma_F \vdash_{UL} x * f\ (x-1) : \mathit{Nat}}}{\dfrac{\Gamma_F \vdash_{UL} \textbf{if } x \equiv 0 \textbf{ then } 1 \textbf{ else } x * f\ (x-1) : \mathit{Nat}}{\Gamma \vdash_{UL} \mu f.\lambda_F x.\textbf{if } x \equiv 0 \textbf{ then } 1 \textbf{ else } x * f\ (x-1) : \mathit{Nat} \to \mathit{Nat}}}$$

$$\Gamma_F = \Gamma[f \mapsto (\mathit{Nat} \to \mathit{Nat})][x \mapsto \mathit{Nat}]$$

Polymorphic functions

Polymorphic functions

$\lambda_{\mathrm{F}} x . x$

Polymorphic functions

$$\lambda_{\mathrm{F}} x.\, x$$

$$\lambda_{\mathrm{F}} x.\, \lambda_{\mathrm{G}} y.\, x$$

Polymorphic functions

$\lambda_{\mathrm{F}} x.\, x$

$\lambda_{\mathrm{F}} x.\, \lambda_{\mathrm{G}} y.\, x$

$\lambda_{\mathrm{F}} f.\, \lambda_{\mathrm{G}} x.\, f\ x$

Polymorphic functions

$$\lambda_{\mathrm{F}} x.\, x$$

$$\lambda_{\mathrm{F}} x.\, \lambda_{\mathrm{G}} y.\, x$$

$$\lambda_{\mathrm{F}} f.\, \lambda_{\mathrm{G}} x.\, f\ x$$

$$\mu f.\, \lambda_{\mathrm{F}} g.\, \lambda_{\mathrm{G}} x.\, \lambda_{\mathrm{H}} y.\, \textbf{if}\ x \equiv 0\ \textbf{then}\ y\ \textbf{else}\ f\ g\ (x-1)\ (g\ y)$$

Polymorphic types

$$\tau \in \mathbf{Ty} \qquad \text{types}$$

$$\Gamma \in \mathbf{TyEnv} \qquad \text{type environments}$$

$$\tau ::= \quad \mid \; Nat \mid Bool \mid \tau_1 \to \tau_2$$

$$\Gamma ::= [\,] \mid \Gamma_1[x \mapsto \tau]$$

$$\Gamma \vdash_{\mathrm{UL}} t : \tau \qquad \text{typing}$$

Polymorphic types

α	$\in$	**TyVar**	type variables
τ	$\in$	**Ty**	types
Γ	$\in$	**TyEnv**	type environments

$$\tau ::= \alpha \mid Nat \mid Bool \mid \tau_1 \to \tau_2$$

$$\Gamma ::= [\,] \mid \Gamma_1[x \mapsto \tau]$$

$\Gamma \vdash_{\mathrm{UL}} t : \tau$ typing

Polymorphic types

α	$\in$	**TyVar**	type variables
τ	$\in$	**Ty**	types
σ	$\in$	**TyScheme**	type schemes
Γ	$\in$	**TyEnv**	type environments

$$\tau ::= \alpha \mid Nat \mid Bool \mid \tau_1 \to \tau_2$$
$$\sigma ::= \tau \mid \forall \alpha.\, \sigma_1$$
$$\Gamma ::= [\,] \mid \Gamma_1[x \mapsto \tau]$$

$\Gamma \vdash_{\mathrm{UL}} t : \tau$ typing

Polymorphic types

α	$\in$	**TyVar**	type variables
τ	$\in$	**Ty**	types
σ	$\in$	**TyScheme**	type schemes
Γ	$\in$	**TyEnv**	type environments

$$\tau ::= \alpha \mid \mathit{Nat} \mid \mathit{Bool} \mid \tau_1 \to \tau_2$$
$$\sigma ::= \tau \mid \forall\alpha.\,\sigma_1$$
$$\Gamma ::= [\,] \mid \Gamma_1[x \mapsto \sigma]$$

$\Gamma \vdash_{\mathrm{UL}} t : \tau$ typing

Polymorphic types

α	$\in$	**TyVar**	type variables
τ	$\in$	**Ty**	types
σ	$\in$	**TyScheme**	type schemes
Γ	$\in$	**TyEnv**	type environments

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \mathit{Nat} \mid \mathit{Bool} \mid \tau_1 \to \tau_2 \\ \sigma & ::= & \tau \mid \forall \alpha.\, \sigma_1 \\ \Gamma & ::= & [\,] \mid \Gamma_1[x \mapsto \sigma] \end{array}$$

$\Gamma \vdash_{\mathrm{UL}} t : \sigma$ typing

[School of Mathematical and Computer Sciences (MACS)]

Polymorphic types

α	$\in$	**TyVar**	type variables
τ	$\in$	**Ty**	types
σ	$\in$	**TyScheme**	type schemes
Γ	$\in$	**TyEnv**	type environments

$$\tau ::= \alpha \mid \mathit{Nat} \mid \mathit{Bool} \mid \tau_1 \rightarrow \tau_2$$
$$\sigma ::= \tau \mid \forall \alpha.\, \sigma_1$$
$$\Gamma ::= [\,] \mid \Gamma_1[x \mapsto \sigma]$$

$\Gamma \vdash_{\mathrm{UL}} t : \sigma$ typing

☞ **Ty** $\subseteq$ **TyScheme**

Introduction:

$$\frac{\Gamma \vdash_{\mathrm{UL}} t : \sigma_1 \quad \alpha \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash_{\mathrm{UL}} t : \forall \alpha.\, \sigma_1}\ [\mathit{t\text{-}gen}]$$

generalisation and instantiation

Introduction:

$$\frac{\Gamma \vdash_{\text{UL}} t : \sigma_1 \quad \alpha \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash_{\text{UL}} t : \forall \alpha.\, \sigma_1} \; [\mathit{t\text{-}gen}]$$

Elimination:

$$\frac{\Gamma \vdash_{\text{UL}} t : \forall \alpha.\, \sigma_1}{\Gamma \vdash_{\text{UL}} t : [\alpha \mapsto \tau_0]\sigma_1} \; [\mathit{t\text{-}inst}]$$

variables and local definitions

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash_{\mathrm{UL}} x : \sigma} \; [\textit{t-var}]$$

variables and local definitions

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash_{\mathrm{UL}} x : \sigma} \ [\textit{t-var}]$$

$$\frac{\Gamma \vdash_{\mathrm{UL}} t_1 : \sigma_1 \quad \Gamma[x \mapsto \sigma_1] \vdash_{\mathrm{UL}} t_2 : \tau}{\Gamma \vdash_{\mathrm{UL}} \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 : \tau} \ [\textit{t-let}]$$

Polymorphic types: example

$$\lambda_F x. x : \forall\alpha. \alpha \to \alpha$$

$$\lambda_F x. \lambda_G y. x : \forall\alpha_1. \forall\alpha_2. \alpha_1 \to \alpha_2 \to \alpha_1$$

$$\lambda_F f. \lambda_G x. f\ x : \forall\alpha_1. \forall\alpha_2. (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_2$$

$$\mu f. \lambda_F g. \lambda_G x. \lambda_H y. \textbf{if } x \equiv 0 \textbf{ then } y \textbf{ else } f\ g\ (x - 1)\ (g\ y)$$
$$: \forall\alpha. (\alpha \to \alpha) \to Nat \to \alpha \to \alpha$$

Inference algorithm

$$\theta \in \mathbf{TySubst} = \mathbf{TyVar} \rightarrow_{\mathsf{fin}} \mathbf{Ty} \qquad \text{type substitution}$$

$$\begin{array}{lcll}
\mathit{generalise}_{\mathrm{UL}} & : & \mathbf{TyEnv} \times \mathbf{Ty} & \rightarrow \mathbf{TyScheme} \\
\mathit{instantiate}_{\mathrm{UL}} & : & \mathbf{TyScheme} & \rightarrow \mathbf{Ty} \\
\mathcal{U}_{\mathrm{UL}} & : & \mathbf{Ty} \times \mathbf{Ty} & \rightarrow \mathbf{TySubst} \\
\mathcal{W}_{\mathrm{UL}} & : & \mathbf{TyEnv} \times \mathbf{Tm} & \rightarrow \mathbf{Ty} \times \mathbf{TySubst}
\end{array}$$

Inference algorithm: constants

$$\mathcal{W}_{\mathrm{UL}}(\Gamma, n) = (Nat, \quad \mathit{id})$$

Inference algorithm: constants

$$\mathcal{W}_{\mathrm{UL}}(\Gamma, n) = (Nat, \quad \mathit{id})$$

$$\mathcal{W}_{\mathrm{UL}}(\Gamma, \texttt{false}) = (Bool, \quad \mathit{id})$$

$$\mathcal{W}_{\mathrm{UL}}(\Gamma, \texttt{true}) = (Bool, \quad \mathit{id})$$

Inference algorithm: variables

$$\mathcal{W}_{\mathrm{UL}}\ (\Gamma, x) = (\mathit{instantiate}_{\mathrm{UL}}(\Gamma(x)), \quad \mathit{id})$$

- The instantiation rule is built into the case for variables.
- By choosing fresh type variables, we commit to nothing,
- and let the actual types be determined by future unifications.

Inference algorithm: functions

$$\begin{aligned}\mathcal{W}_{\text{UL}}\ (\Gamma, \lambda_{\pi} x.\, t_1) = \text{let } & \alpha_1 \text{ be fresh} \\ & (\tau_2, \theta) = \mathcal{W}_{\text{UL}}(\Gamma[x \mapsto \alpha_1], t_1) \\ \text{in } & ((\theta\ \alpha_1) \to \tau_2, \quad \theta)\end{aligned}$$

Inference algorithm: functions

$$\mathcal{W}_{\text{UL}}\ (\Gamma, \lambda_{\pi} x.\, t_1) = \text{let } \alpha_1 \text{ be fresh}$$
$$(\tau_2, \theta) = \mathcal{W}_{\text{UL}}(\Gamma[x \mapsto \alpha_1], t_1)$$
$$\text{in } ((\theta\ \alpha_1) \to \tau_2, \quad \theta)$$

$$\mathcal{W}_{\text{UL}}\ (\Gamma, \mu f.\, \lambda_{\pi} x.\, t_1) =$$
$$\text{let } \alpha_1, \alpha_2 \text{ be fresh}$$
$$(\tau_2, \theta_1) = \mathcal{W}_{\text{UL}}(\Gamma[f \mapsto (\alpha_1 \to \alpha_2)][x \mapsto \alpha_1], t_1)$$
$$\theta_2 = \mathcal{U}_{\text{UL}}(\tau_2, \theta_1\ \alpha_2)$$
$$\text{in } (\theta_2\ (\theta_1\ \alpha_1) \to \theta_2\ \tau_2, \quad \theta_2 \circ \theta_1)$$

Inference algorithm: functions

$$\begin{aligned}\mathcal{W}_{\mathrm{UL}}\ (\Gamma, \lambda_\pi x.\, t_1) = \text{let } & \alpha_1 \text{ be fresh}\\ & (\tau_2, \theta) = \mathcal{W}_{\mathrm{UL}}(\Gamma[x \mapsto \alpha_1], t_1)\\ \text{in } & ((\theta\ \alpha_1) \to \tau_2, \quad \theta)\end{aligned}$$

$$\begin{aligned}&\mathcal{W}_{\mathrm{UL}}\ (\Gamma, \mu f.\, \lambda_\pi x.\, t_1) =\\ &\quad \text{let } \alpha_1, \alpha_2 \text{ be fresh}\\ &\qquad (\tau_2, \theta_1) = \mathcal{W}_{\mathrm{UL}}(\Gamma[f \mapsto (\alpha_1 \to \alpha_2)][x \mapsto \alpha_1], t_1)\\ &\qquad \theta_2 = \mathcal{U}_{\mathrm{UL}}(\tau_2, \theta_1\ \alpha_2)\\ &\quad \text{in } (\theta_2\ (\theta_1\ \alpha_1) \to \theta_2\ \tau_2, \quad \theta_2 \circ \theta_1)\end{aligned}$$

$$\begin{aligned}\mathcal{W}_{\mathrm{UL}}\ (\Gamma, t_1\ t_2) = \text{let } & (\tau_1, \theta_1) = \mathcal{W}_{\mathrm{UL}}(\Gamma, t_1)\\ & (\tau_2, \theta_2) = \mathcal{W}_{\mathrm{UL}}(\theta_1\ \Gamma, t_2)\\ & \alpha \text{ be fresh}\\ & \theta_3 = \mathcal{U}_{\mathrm{UL}}(\theta_2\ \tau_1, \tau_2 \to \alpha)\\ \text{in } & (\theta_3\ \alpha, \quad \theta_3 \circ \theta_2 \circ \theta_1)\end{aligned}$$

Unification

- To combine (join) two given types we apply unification
- I.e., in case rule for applications, $\mathcal{U}_{\mathrm{UL}}(\theta_2\ \tau_1, \tau_2 \to \alpha)$
- Unification computes a substitution from two types:
  $\mathcal{U}_{\mathrm{UL}} : \mathbf{Ty} \times \mathbf{Ty} \to \mathbf{TySubst}$
- If $\mathcal{U}_{\mathrm{UL}}(\tau_1, \tau_2) = \theta$ then $\theta\ \tau_1 = \theta\ \tau_2$
  - And θ is the least such substitution
- Ex. $\mathcal{U}_{\mathrm{UL}}(\alpha_1 \to \mathit{Nat} \to \mathit{Bool}, \mathit{Nat} \to \mathit{Nat} \to \alpha_2)$ equals θ with $\theta(\alpha_1) = \mathit{Nat}$ and $\theta(\alpha_2) = \mathit{Bool}$
- Note: unification is basically the $\sqcup$ in the lattice of monotypes

Unification Algorithm

$$
\begin{array}{l}
\mathcal{U}_{\mathrm{UL}}\ (Nat,\ Nat)\ = \mathit{id} \\
\mathcal{U}_{\mathrm{UL}}\ (Bool, Bool) = \mathit{id} \\
\mathcal{U}_{\mathrm{UL}}\ (\tau_1 \to \tau_2, \tau_3 \to \tau_4) = \theta_2 \circ \theta_1 \\
\quad \textbf{where} \\
\qquad \theta_1 = \mathcal{U}_{\mathrm{UL}}\ (\tau_1, \tau_3) \\
\qquad \theta_2 = \mathcal{U}_{\mathrm{UL}}\ (\theta_1\ \tau_2, \theta_1\ \tau_4) \\
\mathcal{U}_{\mathrm{UL}}\ (\alpha, \tau) = [\alpha \mapsto \tau]\ \textbf{if}\ \mathit{chk}\ (\alpha, \tau) \\
\mathcal{U}_{\mathrm{UL}}\ (\tau, \alpha) = [\alpha \mapsto \tau]\ \textbf{if}\ \mathit{chk}\ (\alpha, \tau) \\
\mathcal{U}_{\mathrm{UL}}\ (\_, \_)\ \ = \mathsf{fail}
\end{array}
$$

Here, $\mathit{chk}\ (\alpha, \tau)$ returns true if $\tau = \alpha$ or α is not a free variable in τ.

Inference algorithm: conditionals

$$
\begin{aligned}
&\mathcal{W}_{\mathrm{UL}}(\Gamma, \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3) = \\
&\quad \text{let } (\tau_1, \theta_1) = \mathcal{W}_{\mathrm{UL}}(\Gamma, t_1) \\
&\qquad\quad (\tau_2, \theta_2) = \mathcal{W}_{\mathrm{UL}}(\theta_1\ \Gamma, t_2) \\
&\qquad\quad (\tau_3, \theta_3) = \mathcal{W}_{\mathrm{UL}}(\theta_2\ (\theta_1\ \Gamma), t_3) \\
&\qquad\quad \theta_4 = \mathcal{U}_{\mathrm{UL}}(\theta_3\ (\theta_2\ \tau_1), Bool) \\
&\qquad\quad \theta_5 = \mathcal{U}_{\mathrm{UL}}(\theta_4\ (\theta_3\ \tau_2), \theta_4\ \tau_3) \\
&\quad \text{in } (\theta_5\ (\theta_4\ \tau_3), \quad \theta_5 \circ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1)
\end{aligned}
$$

- Subsitutions are applied as soon as possible.
- Error prone process of putting the right composition of substitutions everywhere.
- Substitutions are idempotent: blindly applying all of them all the time can only influence efficiency.

Inference algorithm: local definitions

$$
\begin{aligned}
&\mathcal{W}_{\mathrm{UL}}(\Gamma, \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2) = \\
&\quad \text{let } (\tau_1, \theta_1) = \mathcal{W}_{\mathrm{UL}}(\Gamma, t_1) \\
&\qquad\ \ (\tau, \theta_2) \ = \mathcal{W}_{\mathrm{UL}}((\theta_1\ \Gamma)[x \mapsto \textit{generalise}_{\mathrm{UL}}(\theta_1\ \Gamma, \tau_1)], t_2) \\
&\quad \text{in } \ (\tau, \quad \theta_2 \circ \theta_1)
\end{aligned}
$$

$\textit{generalise}_{\mathrm{UL}}$ generalizes all variables free in $\theta_1\ \Gamma$ at once.

Inference algorithm: binary operators

$$
\begin{array}{l}
\mathcal{W}_{\mathrm{UL}}(\Gamma, t_1 \oplus t_2) = \\
\quad \text{let } (\tau_1, \theta_1) = \mathcal{W}_{\mathrm{UL}}(\Gamma, t_1) \\
\qquad\quad (\tau_2, \theta_2) = \mathcal{W}_{\mathrm{UL}}(\theta_1\ \Gamma, t_2) \\
\qquad\quad \theta_3 = \mathcal{U}_{\mathrm{UL}}(\theta_2\ \tau_1, \tau_{\oplus}^{1}) \\
\qquad\quad \theta_4 = \mathcal{U}_{\mathrm{UL}}(\theta_3\ \tau_2, \tau_{\oplus}^{2}) \\
\quad \text{in } (\tau_{\oplus}, \quad \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1)
\end{array}
$$

Control-flow Analysis with Annotated Types

Control-flow analysis

Control-flow analysis (or closure analysis) determines:

For each function application, which functions may be applied.

Annotated types

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

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

Annotated types

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

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

$$\widehat{\tau} \quad ::= \quad \alpha \mid \mathit{Nat} \mid \mathit{Bool} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$$

Annotated types

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

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

Annotated 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

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

Annotated 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

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

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

Control-flow analysis: constants

$$\frac{}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} n : Nat} \; [\textit{cfa-num}]$$

Control-flow analysis: constants

$$\frac{}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} n : \mathit{Nat}}\ [\mathit{cfa\text{-}num}]$$

$$\frac{}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} \mathtt{false} : \mathit{Bool}}\ [\mathit{cfa\text{-}false}]$$

$$\frac{}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} \mathtt{true} : \mathit{Bool}}\ [\mathit{cfa\text{-}true}]$$

Control-flow analysis: variables

$$\frac{\widehat{\Gamma}(x) = \widehat{\sigma}}{\widehat{\Gamma} \vdash_{\text{CFA}} x : \widehat{\sigma}} \; [\textit{cfa-var}]$$

Control-flow analysis: functions

$$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_1] \vdash_{\text{CFA}} t_1 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\text{CFA}} \lambda_\pi x.\, t_1 : \widehat{\tau}_1 \xrightarrow{\{\pi\}} \widehat{\tau}_2} \; [\textit{cfa-lam}]$$

Control-flow analysis: functions

$$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_1] \vdash_{\mathrm{CFA}} t_1 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} \lambda_\pi x.\, t_1 : \widehat{\tau}_1 \xrightarrow{\{\pi\}} \widehat{\tau}_2} \; [\textit{cfa-lam}]$$

$$\frac{\widehat{\Gamma}[f \mapsto (\widehat{\tau}_1 \xrightarrow{\{\pi\}} \widehat{\tau}_2)][x \mapsto \widehat{\tau}_1] \vdash_{\mathrm{CFA}} t_1 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} \mu f.\, \lambda_\pi x.\, t_1 : \widehat{\tau}_1 \xrightarrow{\{\pi\}} \widehat{\tau}_2} \; [\textit{cfa-mu}]$$

Control-flow analysis: functions

$$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_1] \vdash_{\text{CFA}} t_1 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\text{CFA}} \lambda_\pi x.\, t_1 : \widehat{\tau}_1 \xrightarrow{\{\pi\}} \widehat{\tau}_2} \ [\textit{cfa-lam}]$$

$$\frac{\widehat{\Gamma}[f \mapsto (\widehat{\tau}_1 \xrightarrow{\{\pi\}} \widehat{\tau}_2)][x \mapsto \widehat{\tau}_1] \vdash_{\text{CFA}} t_1 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\text{CFA}} \mu f.\, \lambda_\pi x.\, t_1 : \widehat{\tau}_1 \xrightarrow{\{\pi\}} \widehat{\tau}_2} \ [\textit{cfa-mu}]$$

$$\frac{\widehat{\Gamma} \vdash_{\text{CFA}} t_1 : \widehat{\tau}_2 \xrightarrow{\varphi} \widehat{\tau} \qquad \widehat{\Gamma} \vdash_{\text{CFA}} t_2 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\text{CFA}} t_1\ t_2 : \widehat{\tau}} \ [\textit{cfa-app}]$$

- φ describes what may be applied!

[School of Mathematical and Computer Sciences (MACS)]

Control-flow analysis: conditionals

$$\frac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t_1 : \mathit{Bool} \quad \widehat{\Gamma} \vdash_{\mathrm{CFA}} t_2 : \widehat{\tau} \quad \widehat{\Gamma} \vdash_{\mathrm{CFA}} t_3 : \widehat{\tau}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 : \widehat{\tau}}\ [\mathit{cfa\text{-}if}]$$

Control-flow analysis: local definitions

$$\frac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t_1 : \widehat{\sigma}_1 \quad \widehat{\Gamma}[x \mapsto \widehat{\sigma}_1] \vdash_{\mathrm{CFA}} t_2 : \widehat{\tau}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 : \widehat{\tau}}\ [\textit{cfa-let}]$$

Control-flow analysis: binary operators

$$\frac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t_1 : \tau_{\oplus}^{1} \quad \widehat{\Gamma} \vdash_{\mathrm{CFA}} t_2 : \tau_{\oplus}^{2}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} t_1 \oplus t_2 : \tau_{\oplus}} \; [\textit{cfa-op}]$$

Control-flow analysis: example

$(\lambda_{\mathrm{F}} x.\, x)\ (\lambda_{\mathrm{G}} y.\, y)$

Control-flow analysis: example

$(\lambda_{\mathrm{F}} x.\, x)\ (\lambda_{\mathrm{G}} y.\, y)$

$$\frac{}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} (\lambda_{\mathrm{F}} x.\, x)\ (\lambda_{\mathrm{G}} y.\, y) : \forall \alpha.\, \alpha \xrightarrow{\{\mathrm{G}\}} \alpha}$$

Control-flow analysis: example

HERIOT WATT UNIVERSITY

$$(\lambda_{\mathrm{F}} x.\, x)\ (\lambda_{\mathrm{G}} y.\, y)$$

$$\dfrac{\dfrac{\dfrac{\vdots}{\widehat{\Gamma}[x \mapsto \widehat{\tau}_{\mathrm{G}}] \vdash_{\mathrm{CFA}} x : \widehat{\tau}_{\mathrm{G}}}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} \lambda_{\mathrm{F}} x.\, x : \widehat{\tau}_{\mathrm{G}} \xrightarrow{\{\mathrm{F}\}} \widehat{\tau}_{\mathrm{G}}} \qquad \dfrac{\dfrac{\vdots}{\widehat{\Gamma}[y \mapsto \alpha] \vdash_{\mathrm{CFA}} y : \alpha}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} \lambda_{\mathrm{G}} y.\, y : \widehat{\tau}_{\mathrm{G}}}}{\dfrac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} (\lambda_{\mathrm{F}} x.\, x)\ (\lambda_{\mathrm{G}} y.\, y) : \widehat{\tau}_{\mathrm{G}}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} (\lambda_{\mathrm{F}} x.\, x)\ (\lambda_{\mathrm{G}} y.\, y) : \forall \alpha.\, \alpha \xrightarrow{\{\mathrm{G}\}} \alpha}}$$

$$\widehat{\tau}_{\mathrm{G}} = \alpha \xrightarrow{\{\mathrm{G}\}} \alpha$$

Higher-order functions

$$\begin{array}{l}
\textbf{let } f = \lambda_{\mathrm{F}} x.\ x + 1 \textbf{ in} \\
\textbf{let } g = \lambda_{\mathrm{G}} y.\ y * 2 \textbf{ in} \\
\textbf{let } h = \lambda_{\mathrm{H}} z.\ z\ 3 \textbf{ in} \\
h\ g + h\ f
\end{array}$$

Higher-order functions

$$
\begin{array}{l}
\textbf{let } f = \lambda_{F} x.\ x + 1 \textbf{ in} \\
\textbf{let } g = \lambda_{G} y.\ y * 2 \textbf{ in} \\
\textbf{let } h = \lambda_{H} z.\ z\ 3 \textbf{ in} \\
h\ g + h\ f
\end{array}
$$

$$
\begin{array}{lcl}
f & : & Nat \xrightarrow{\{F\}} Nat \\
g & : & Nat \xrightarrow{\{G\}} Nat
\end{array}
$$

Higher-order functions

$$\begin{array}{l}
\textbf{let } f = \lambda_{\mathrm{F}} x.\, x + 1 \textbf{ in} \\
\textbf{let } g = \lambda_{\mathrm{G}} y.\, y * 2 \textbf{ in} \\
\textbf{let } h = \lambda_{\mathrm{H}} z.\, z\ 3 \textbf{ in} \\
h\ g + h\ f
\end{array}$$

$$\begin{array}{lcl}
f & : & Nat \xrightarrow{\{\mathrm{F}\}} Nat \\
g & : & Nat \xrightarrow{\{\mathrm{G}\}} Nat \\
h & : & (Nat \xrightarrow{??} Nat) \xrightarrow{\{\mathrm{H}\}} Nat
\end{array}$$

Higher-order functions

$$
\begin{aligned}
&\textbf{let } f = \lambda_{F} x.\, x + 1 \textbf{ in}\\
&\textbf{let } g = \lambda_{G} y.\, y * 2 \textbf{ in}\\
&\textbf{let } h = \lambda_{H} z.\, z\ 3 \textbf{ in}\\
&h\ g + h\ f
\end{aligned}
$$

$$
\begin{aligned}
f &: Nat \xrightarrow{\{F\}} Nat\\
g &: Nat \xrightarrow{\{G\}} Nat\\
h &: (Nat \xrightarrow{??} Nat) \xrightarrow{\{H\}} Nat
\end{aligned}
$$

Should we have $h : (Nat \xrightarrow{\{F\}} Nat) \xrightarrow{\{H\}} Nat$ or $h : (Nat \xrightarrow{\{G\}} Nat) \xrightarrow{\{H\}} Nat$?

Conditionals

$$\begin{aligned}\lambda_{\mathrm{H}} z.\, \textbf{if} \quad & z \equiv 0 \\ \textbf{then}\ & \lambda_{\mathrm{F}} x.\, x + 1 \\ \textbf{else}\ & \lambda_{\mathrm{G}} y.\, y * 2\end{aligned}$$

$$\begin{array}{ll} \lambda_{\mathrm{H}} z.\, \mathbf{if} & z \equiv 0 \\ \quad \mathbf{then} & \lambda_{\mathrm{F}} x.\, x + 1 \\ \quad \mathbf{else} & \lambda_{\mathrm{G}} y.\, y * 2 \end{array}$$

Should we have $Nat \xrightarrow{\{\mathrm{H}\}} (Nat \xrightarrow{\{\mathrm{F}\}} Nat)$ or $Nat \xrightarrow{\{\mathrm{H}\}} (Nat \xrightarrow{\{\mathrm{G}\}} Nat)$?

Subeffecting

$$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_1] \vdash_{\text{CFA}} t_1 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\text{CFA}} \lambda_\pi x.\, t_1 : \widehat{\tau}_1 \xrightarrow{\{\pi\} \cup \varphi} \widehat{\tau}_2} \; [\textit{cfa-lam}]$$

Subeffecting

$$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_1] \vdash_{\text{CFA}} t_1 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\text{CFA}} \lambda_\pi x.\, t_1 : \widehat{\tau}_1 \xrightarrow{\{\pi\}\cup\varphi} \widehat{\tau}_2} \; [\textit{cfa-lam}]$$

$$\frac{\widehat{\Gamma}[f \mapsto (\widehat{\tau}_1 \xrightarrow{\{\pi\}\cup\varphi} \widehat{\tau}_2)][x \mapsto \widehat{\tau}_1] \vdash_{\text{CFA}} t_1 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\text{CFA}} \mu f.\, \lambda_\pi x.\, t_1 : \widehat{\tau}_1 \xrightarrow{\{\pi\}\cup\varphi} \widehat{\tau}_2} \; [\textit{cfa-mu}]$$

Subeffecting: example

$$
\begin{array}{l}
\textbf{let } f = \lambda_{F} x.\ x + 1 \textbf{ in} \\
\textbf{let } g = \lambda_{G} y.\ y * 2 \textbf{ in} \\
\textbf{let } h = \lambda_{H} z.\ z\ 3 \textbf{ in} \\
h\ g + h\ f
\end{array}
$$

$$
\begin{array}{lcl}
f & : & Nat \xrightarrow{\{F,G\}} Nat \\
g & : & Nat \xrightarrow{\{F,G\}} Nat \\
h & : & (Nat \xrightarrow{\{F,G\}} Nat) \xrightarrow{\{H\}} Nat
\end{array}
$$

Subeffecting: example

$$
\begin{array}{ll}
\lambda_{\mathrm{H}} z.\,\mathbf{if} & z \equiv 0 \\
\quad\mathbf{then} & \lambda_{\mathrm{F}} x.\, x + 1 \\
\quad\mathbf{else} & \lambda_{\mathrm{G}} y.\, y * 2
\end{array}
$$

$$
\mathit{Nat} \xrightarrow{\{\mathrm{H}\}} (\mathit{Nat} \xrightarrow{\{\mathrm{F},\mathrm{G}\}} \mathit{Nat})
$$

Inference algorithm: simple types

β	$\in$	$\mathbf{AnnVar}$	annotation variables
$\widehat{\tau}$	$\in$	$\widehat{\mathbf{SimpleTy}}$	simple types
$\widehat{\sigma}$	$\in$	$\widehat{\mathbf{SimpleTyScheme}}$	simple type schemes
$\widehat{\Gamma}$	$\in$	$\widehat{\mathbf{SimpleTyEnv}}$	simple type environments
$\widehat{\theta}$	$\in$	$\widehat{\mathbf{TySubst}}$	hybrid type substitution
C	$\in$	$\mathbf{Constr}$	constraint

$$
\begin{array}{lcl}
\widehat{\tau} & ::= & \alpha \mid \mathit{Nat} \mid \mathit{Bool} \mid \widehat{\tau}_1 \xrightarrow{\beta} \widehat{\tau}_2 \\
\widehat{\sigma} & ::= & \widehat{\tau} \mid \forall \alpha. \widehat{\sigma}_1 \\
\widehat{\Gamma} & ::= & [\,] \mid \widehat{\Gamma}_1[x \mapsto \widehat{\sigma}] \\
C & ::= & \emptyset \mid \{\beta \supseteq \varphi\} \mid C_1 \cup C_2
\end{array}
$$

Inference algorithm

$$
\begin{aligned}
\mathit{generalise}_{\mathrm{CFA}} &: \widehat{\mathbf{SimpleTyEnv}} \times \widehat{\mathbf{SimpleTy}} \to \\
&\quad \widehat{\mathbf{SimpleTyScheme}} \\
\mathit{instantiate}_{\mathrm{CFA}} &: \widehat{\mathbf{SimpleTyScheme}} \to \widehat{\mathbf{SimpleTy}} \\
\mathcal{U}_{\mathrm{CFA}} &: \widehat{\mathbf{SimpleTy}} \times \widehat{\mathbf{SimpleTy}} \to \\
&\quad \widehat{\mathbf{TySubst}} \\
\mathcal{W}_{\mathrm{CFA}} &: \widehat{\mathbf{SimpleTyEnv}} \times \mathbf{Tm} \to \\
&\quad \widehat{\mathbf{SimpleTy}} \times \widehat{\mathbf{TySubst}} \times \mathbf{Constr}
\end{aligned}
$$

Inference algorithm: constants

$$\mathcal{W}_{\text{CFA}}(\widehat{\Gamma}, n) = (Nat, \quad \mathit{id}, \quad \emptyset)$$

$$\mathcal{W}_{\text{CFA}}(\widehat{\Gamma}, \texttt{false}) = (Bool, \quad \mathit{id}, \quad \emptyset)$$

$$\mathcal{W}_{\text{CFA}}(\widehat{\Gamma}, \texttt{true}) = (Bool, \quad \mathit{id}, \quad \emptyset)$$

Inference algorithm: variables

$$\mathcal{W}_{\text{CFA}}\ (\widehat{\Gamma}, x) = (\textit{instantiate}_{\text{CFA}}(\widehat{\Gamma}(x)),\quad \textit{id},\quad \emptyset)$$

Inference algorithm: functions

$$
\begin{aligned}
\mathcal{W}_{\text{CFA}}\ (\widehat{\Gamma}, \lambda_\pi x.\, t_1) = \text{let } & \alpha_1 \text{ be fresh} \\
& (\widehat{\tau}_2, \widehat{\theta}, C_1) = \mathcal{W}_{\text{CFA}}(\widehat{\Gamma}[x \mapsto \alpha_1], t_1) \\
& \beta \text{ be fresh} \\
\text{in } ((\widehat{\theta}\ \alpha_1) \xrightarrow{\beta} \widehat{\tau}_2, \quad & \widehat{\theta}, C_1 \cup \{\beta \supseteq \{\pi\}\})
\end{aligned}
$$

- Introduce fresh variables for annotations.
- Invariant: only variables as annotations in types (aka simple types).
- Put concrete information about the variables into C.
- Solve constraints later to obtain actual sets.
- Simplifies unification substantially.

Changes to unification

Only the case for function changes:

$$\begin{aligned}
&\ldots \\
&\mathcal{U}_{\mathrm{UL}}\ (\tau_1 \xrightarrow{\beta_1} \tau_2, \tau_3 \xrightarrow{\beta_2} \tau_4) = \theta_2 \circ \theta_1 \circ \theta_0 \\
&\quad \textbf{where} \\
&\qquad \theta_0 = [\beta_1 \mapsto \beta_2] \\
&\qquad \theta_1 = \mathcal{U}_{\mathrm{UL}}\ (\theta_0\ \tau_1, \theta_0\ \tau_3) \\
&\qquad \theta_2 = \mathcal{U}_{\mathrm{UL}}\ (\theta_1\ (\theta_0\ \tau_2), \theta_1\ (\theta_0\ \tau_4)) \\
&\ldots
\end{aligned}$$

No need to recurse on annotations: just map one variable to the other.

Inference algorithm: recursive functions

$$
\begin{array}{l}
\mathcal{W}_{\text{CFA}}\ (\widehat{\Gamma}, \mu f\,.\,\lambda_\pi x\,.\,t_1) = \\
\quad \text{let } \alpha_1, \alpha_2, \beta \text{ be fresh} \\
\qquad (\widehat{\tau}_2, \widehat{\theta}_1, C_1) = \mathcal{W}_{\text{CFA}}(\widehat{\Gamma}[f \mapsto (\alpha_1 \xrightarrow{\beta} \alpha_2)][x \mapsto \alpha_1], t_1) \\
\qquad \widehat{\theta}_2 = \mathcal{U}_{\text{CFA}}(\widehat{\tau}_2, \widehat{\theta}_1\ \alpha_2) \\
\quad \text{in } \ (\widehat{\theta}_2\ (\widehat{\theta}_1\ \alpha_1) \xrightarrow{\widehat{\theta}_2\ (\widehat{\theta}_1\ \beta)} \widehat{\theta}_2\ \widehat{\tau}_2, \quad \widehat{\theta}_2 \circ \widehat{\theta}_1, \\
\qquad\quad (\widehat{\theta}_2\ C_1) \cup \{\widehat{\theta}_2\ (\widehat{\theta}_1\ \beta) \supseteq \{\pi\}\})
\end{array}
$$

Remember: $\widehat{\theta}_1$ and $\widehat{\theta}_2$ can only rename annotation variables.

Constraints: example

$$
\begin{array}{l}
\textbf{let } f = \lambda_{F} x.\, x + 1 \textbf{ in} \\
\textbf{let } g = \lambda_{G} y.\, y * 2 \textbf{ in} \\
\textbf{let } h = \lambda_{H} z.\, z\ 3 \textbf{ in} \\
h\ g + h\ f
\end{array}
$$

Constraints: example

$$\begin{array}{l} \textbf{let } f = \lambda_{\text{F}} x.\ x + 1 \textbf{ in} \\ \textbf{let } g = \lambda_{\text{G}} y.\ y * 2 \textbf{ in} \\ \textbf{let } h = \lambda_{\text{H}} z.\ z\ 3 \textbf{ in} \\ h\ g + h\ f \end{array}$$

$$\begin{array}{lll} f & : & Nat \xrightarrow{\beta_1} Nat \\ g & : & Nat \xrightarrow{\beta_2} Nat \\ h & : & (Nat \xrightarrow{\beta_3} Nat) \xrightarrow{\{\text{H}\}} Nat \end{array}$$

Constraints: example

$$\begin{array}{l}
\textbf{let } f = \lambda_{F} x.\, x + 1 \textbf{ in} \\
\textbf{let } g = \lambda_{G} y.\, y * 2 \textbf{ in} \\
\textbf{let } h = \lambda_{H} z.\, z\ 3 \textbf{ in} \\
h\ g + h\ f
\end{array}$$

$$\begin{array}{lll}
f & : & Nat \xrightarrow{\beta_1} Nat \\
g & : & Nat \xrightarrow{\beta_2} Nat \\
h & : & (Nat \xrightarrow{\beta_3} Nat) \xrightarrow{\{H\}} Nat
\end{array}$$

$$\widehat{\theta}(\beta_1) = \beta_3$$
$$\widehat{\theta}(\beta_2) = \beta_3$$

Constraints: example

$$
\begin{array}{l}
\textbf{let } f = \lambda_{F} x.\ x + 1 \textbf{ in} \\
\textbf{let } g = \lambda_{G} y.\ y * 2 \textbf{ in} \\
\textbf{let } h = \lambda_{H} z.\ z\ 3 \textbf{ in} \\
h\ g + h\ f
\end{array}
$$

$$
\begin{array}{lcl}
f & : & Nat \xrightarrow{\beta_1} Nat \\
g & : & Nat \xrightarrow{\beta_2} Nat \\
h & : & (Nat \xrightarrow{\beta_3} Nat) \xrightarrow{\{H\}} Nat
\end{array}
$$

$$\widehat{\theta}(\beta_1) = \beta_3$$
$$\widehat{\theta}(\beta_2) = \beta_3$$

$$C = \{\beta_1 \supseteq \{F\}, \beta_2 \supseteq \{G\}\}$$

Constraints: example

$$\begin{array}{l} \textbf{let } f = \lambda_{\text{F}} x.\ x + 1 \textbf{ in} \\ \textbf{let } g = \lambda_{\text{G}} y.\ y * 2 \textbf{ in} \\ \textbf{let } h = \lambda_{\text{H}} z.\ z\ 3 \textbf{ in} \\ h\ g + h\ f \end{array}$$

$$\begin{array}{lll} f & : & Nat \xrightarrow{\beta_1} Nat \\ g & : & Nat \xrightarrow{\beta_2} Nat \\ h & : & (Nat \xrightarrow{\beta_3} Nat) \xrightarrow{\{\text{H}\}} Nat \end{array}$$

$$\widehat{\theta}(\beta_1) = \beta_3$$
$$\widehat{\theta}(\beta_2) = \beta_3$$

$$C = \{\beta_1 \supseteq \{\text{F}\}, \beta_2 \supseteq \{\text{G}\}\}$$
$$\widehat{\theta}\, C = \{\beta_3 \supseteq \{\text{F}\}, \beta_3 \supseteq \{\text{G}\}\}$$

Constraints: example

$$\begin{array}{l}
\textbf{let } f = \lambda_{\mathrm{F}} x.\ x + 1 \ \textbf{ in} \\
\textbf{let } g = \lambda_{\mathrm{G}} y.\ y * 2 \ \textbf{ in} \\
\textbf{let } h = \lambda_{\mathrm{H}} z.\ z\ 3 \quad \textbf{ in} \\
h\ g + h\ f
\end{array}$$

$$\begin{array}{lcl}
f & : & Nat \xrightarrow{\beta_1} Nat \\
g & : & Nat \xrightarrow{\beta_2} Nat \\
h & : & (Nat \xrightarrow{\beta_3} Nat) \xrightarrow{\{\mathrm{H}\}} Nat
\end{array}$$

$$\widehat{\theta}(\beta_1) = \beta_3$$
$$\widehat{\theta}(\beta_2) = \beta_3$$

$$C = \{\beta_1 \supseteq \{\mathrm{F}\}, \beta_2 \supseteq \{\mathrm{G}\}\}$$
$$\widehat{\theta}\ C = \{\beta_3 \supseteq \{\mathrm{F}\}, \beta_3 \supseteq \{\mathrm{G}\}\}$$

Least solution: $\beta_3 = \{\mathrm{F}, \mathrm{G}\}$.

Poisoning

Naive use of subeffecting is fatal for the precision of your analysis:

$$
\begin{array}{l}
\textbf{let } f = \lambda_{F} x.\, x + 1 \quad \textbf{in} \\
\textbf{let } g = \lambda_{G} y.\, y * 2 \quad \textbf{in} \\
\textbf{let } h = \lambda_{H} z.\, \textbf{if } z \equiv 0 \textbf{ then } f \textbf{ else } g \textbf{ in} \\
f
\end{array}
$$

$$Nat \xrightarrow{\{F,G\}} Nat$$

Separate rule for subeffecting

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

Separate rule for subeffecting

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

We can remove the subeffecting from the lambda rule:

$$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau}_1] \vdash_{\text{CFA}} t_1 : \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\text{CFA}} \lambda_\pi x.\, t_1 : \widehat{\tau}_1 \xrightarrow{\{\pi\}} \widehat{\tau}_2} \; [\textit{cfa-lam}]$$

Separate compilation?

$$\begin{array}{l}\textbf{let } f = \lambda_{F} x.\ x+1 \textbf{ in}\\ \textbf{let } g = \lambda_{G} y.\ y*2 \textbf{ in}\\ \textbf{let } h = \lambda_{H} z.\ z\ 3 \textbf{ in}\\ h\ g + h\ f\end{array}$$

$$\begin{array}{lll} f & : & Nat \xrightarrow{\{F\}} Nat\\ g & : & Nat \xrightarrow{\{G\}} Nat\\ h & : & (Nat \xrightarrow{\{F,G\}} Nat) \xrightarrow{\{H\}} Nat\end{array}$$

Separate compilation?

$$
\begin{array}{l}
\textbf{let } f = \lambda_{F} x.\ x + 1 \textbf{ in} \\
\textbf{let } g = \lambda_{G} y.\ y * 2 \textbf{ in} \\
\textbf{let } h = \lambda_{H} z.\ z\ 3 \textbf{ in} \\
h\ g + h\ f
\end{array}
$$

$$
\begin{array}{lcl}
f & : & Nat \xrightarrow{\{F\}} Nat \\
g & : & Nat \xrightarrow{\{G\}} Nat \\
h & : & (Nat \xrightarrow{\{F,G\}} Nat) \xrightarrow{\{H\}} Nat
\end{array}
$$

☞ We need to analyse the whole program to accurately determine the domain of h.

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 annotations deeper in the type.

Polyvariance

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: $Nat^{\{O,E\}} \to Nat^{\{E\}}$.
- 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 acceptable that *id* 1 and 1 give different analyses?

- We consider only let-polyvariance.
- Exactly analogous to let-polymorphism, but for annotations.
- For *id* we then derive the type $\forall\beta.\ Nat^{\beta} \rightarrow Nat^{\beta}$.
- 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

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

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

Annotated polyvariant types

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

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

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

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

$$\widehat{\tau} ::= \alpha \mid \mathit{Nat} \mid \mathit{Bool} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$$

$$\widehat{\sigma} ::= \widehat{\tau} \mid \forall \alpha.\, \widehat{\sigma}_1 \mid \forall \beta.\, \widehat{\sigma}_1$$

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

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

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

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

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

Is this enough?

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

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?

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

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:

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

Introduction for annotation variables:

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

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

Polyvariant type system: instantiation

Elimination for type variables:

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

Elimination for annotation variables:

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

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:

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

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 has no way of knowing, so we have to tell it when.

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

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)

HERIOT WATT UNIVERSITY

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

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

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

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

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.
  - Some say: simple duplication is not feasible.
- 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 $\mathcal{W}_{\text{CFA}}$ returns type $(v1 \xrightarrow{\beta_1} v1) \xrightarrow{\beta_2} (v1 \xrightarrow{\beta_3} v1)$ and constraint set $\{\beta_2 \supseteq \{\text{G}\}, \beta_3 \supseteq \beta_4, \beta_4 \supseteq \beta_1, \beta_5 \supseteq \{\text{H}\}, \beta_3 \supseteq \beta\}$
- And that β occurs free in $\widehat{\Gamma}$.
- β_5 is not relevant, so it can be omitted (set to $\{\text{H}\}$).
  - It does not occur in the type, or the context
- β_4 is not relevant either, but removing it implies we must add $\beta_3 \supseteq \beta_1$.
- Neither $\beta_2 \supseteq \{\text{G}\}$ 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):

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

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
- Type signature compartmentalizes a local definition: we do not care what happens inside.

If we do get here with time to spare

Hop over to the effect system slides