Universiteit Utrecht

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

Usage Analysis

Stolen from Stefan Holdermans

Dept. of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: jur@cs.uu.nl
Web pages: http://people.cs.uu.nl/jur/

May 23, 2016

1. Introduction to usage analysis

Usage analysis

- Usage analysis: determining which objects in a (functional) program are guaranteed to be used at most once and—dually— which objects may be used more than once.
- Two flavours: uniqueness analysis (a.k.a. uniquenss typing) and sharing analysis.
- Hage et al. (ICFP 2007): A generic usage analysis with subeffect qualifiers.

Hage et al. (ICFP 2007): abstract

"Sharing analysis and uniqueness typing are static analyses that aim at determining which of a program's objects are to be used at most once. There are many commonalities between these two forms of usage analysis. We make their connection precise by developing an expressive generic analysis that can be instantiated to both sharing analysis and uniqueness typing. The resulting system, which combines parametric polymorphism with effect subsumption, is specifed within the general framework of qualified types, so that readily available tools and techniques can be used for the development of implementations and metatheory."

Destructive updates

- An important property of pure functional languages is referential transparency: a given expression will yield one and the same value each time it is evaluated.
- Referential transparency enables equational reasoning.
- But some operations are destructive by nature: for example, altering the contents of a file.
- Such destructive operations break referential transparency.

Problems with destructive updates

Simple I/O interface:

$$\begin{aligned} &readFile :: String \to File \\ &fPutChar :: Char \to File \to File \end{aligned}$$

For example:

$$\begin{aligned} &\mathbf{let}\ f = readFile\ \texttt{"DATA"} \\ &\mathbf{in}\ \ (fPutChar\ \texttt{'O'}\ f, fPutChar\ \texttt{'K'}\ f) \end{aligned}$$

☞ What is the meaning of this program? (Assume lazy evaluation.)

“Safe” destructive updates

Idea: referential transparency can be recovered if we restrict destructive updates to operations that hold the only reference to the object that is to be destructed.

Example:

$$\textbf{let}\ f = \mathit{readFile}\ \texttt{"DATA"}$$
$$\textbf{in}\ \ (\mathit{fPutChar}\ \texttt{'K'} \circ \mathit{fPutChar}\ \texttt{'O'})\ f$$

☞ Each file handle is used at most once.

Self-updating closures

- Lazy evaluation is typically implemented by means of self-updating closures.
- For example:

$$(\lambda x \rightarrow x + x)\ (2 + 3)$$

- A closure is created for the expression $(2 + 3)$ and associated with x.
- When x is first accessed, the closure evaluates its expression and updates itself with the result (5).
- For the second access of x, the closure can immediately produce the value 5.
- The update avoids re-evaluation of $(2 + 3)$.

Unnecessary updates

Another example:

$$(\lambda x \rightarrow 2 * x)\ (2 + 3)$$

☞ Now, the update of the closure is unneccesary, because x is accessed only once.

Two flavours of usage analysis

Uniqueness analysis:

- Determines which objects have at most one reference.
- Application: destructive updates that are "safe" w.r.t. referential transparency.
- Used in Clean as an alternative to monads.

Sharing analysis:

- Determines which function arguments are accessed at most once.
- Application: avoiding unneccesary closure updates.
- For other applications, see Turner et al. (FPCA 1995), Wansbrough and Peyton Jones (POPL 1999), and Gustavsson and Sands (ENTCS 26).

Generic usage analysis

- Both uniqueness analysis and sharing analysis aim at keeping track of objects that are used at most once.
- If we forget about modularity and settle for little accuracy, we can use a single nonstandard type system for both analyses.
- For more realistic requirements, we can still define a single parameterized type system that can be instantiated to uniqueness analysis as well as sharing analysis.

2. The underlying type system

Term language

- It would be impractical to define the analysis for a full-fledged language like Haskell or Clean.
- Instead, we use a small toy language.

n	$\in$	$\mathbf{Num}$	numerals
x	$\in$	$\mathbf{Var}$	variables
t	$\in$	$\mathbf{Tm}$	terms
v	$\in$	$\mathbf{Val} \subset \mathbf{Tm}$	values

$$
\begin{array}{lcl}
t & ::= & n \mid x \mid \lambda x.\, t_1 \mid t_1\ t_2 \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} \\
 & \mid & t_1 + t_2 \\
v & ::= & n \mid \lambda x.\, t_1
\end{array}
$$

Natural semantics

- The meaning of programs is defined by means of a so-called big-step or natural semantics.
- Evaluation relation: judgements of the form $t \longrightarrow v$.
- Rules are given in natural deduction style:

$$\frac{\mathit{hyp}_1 \quad \cdots \quad \mathit{hyp}_n}{\mathit{concl}}$$

Natural semantics: numerals and abstractions

Numerals and abstractions are already values:

$$\frac{}{n \longrightarrow n}$$

$$\frac{}{\lambda x.\, t_1 \longrightarrow \lambda x.\, t_1}$$

Natural semantics: applications

Beta-reduction:

$$\frac{t_1 \longrightarrow \lambda x.\, t_{11} \qquad [x \mapsto t_2]t_{11} \longrightarrow v}{t_1\ t_2 \longrightarrow v}$$

☞ $[x \mapsto t_2]t_{11}$ means
"replace each free occurrence of x in t_{11} by t_2".

☞ Lazy evaluation: arguments are passed unevaluated.

Natural semantics: local definitions

Local definitions are also evaluated by means of beta-reduction:

$$\frac{[x \mapsto t_1]\,t_2 \longrightarrow v}{\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} \longrightarrow v}\ [\textit{e-let}]$$

☞ Local definitions are evaluated as if

$$\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 \qquad \equiv \qquad (\lambda x.\, t_2)\ t_1.$$

Natural semantics: addition

Addition is strict, i.e., it first evaluates both its operands:

$$\frac{t_1 \longrightarrow n_1 \quad t_2 \longrightarrow n_2 \quad n_1 \oplus n_2 = n}{t_1 + t_2 \longrightarrow n}$$

☞ $\oplus$ denotes "ordinary" addition of natural numbers.

Types and type environments

- Types are built from the type Nat of natural numbers and the function-type constructor $\to$.
- Type environments map variables to types.

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

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

- We write $\Gamma(x) = \tau$ if the rightmost binding for x in Γ associates to τ.

Typing

- We approximate the set of "well-behaved" programs by means of a type system.
- Typing relation: judgements of the form $\Gamma \vdash_{\mathsf{UL}} t : \tau$.
- "In type environment Γ, the term t can be assigned the type τ."
- Γ is supposed to contain types for the free variables of t.
- The subscript UL is used to distinguish the judgements of this underlying type system from the (nonstandard) type systems we will consider later on.

3. The analysis

Examples

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

2 is used at most once.

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

2 is used more than once.

$$(\lambda x.\, \lambda y.\, x)\ 2\ 3$$

2 is used at most once; 3 is used at most once.

$$(\lambda f.\, \lambda x.\, f\ x)\ (\lambda y.\, y + y)\ 2$$

2 is used more than once.

Annotated type system

- Our usage analysis will be specified as an annotated type system.
- We extend the Damas-Milner type system by annotating types, type environments, and typing judgements with information on how often a term is used.
- Two annotations: 1 and ω.
- 1: the term is guaranteed to be used at most once.
- ω: the term may be used more than once.
- Judgements have the form $\widehat{\Gamma} \vdash_{\mathsf{UA}} t :^{\varphi} \widehat{\sigma}$.
- φ ranges over annotations.
- $\widehat{\Gamma}$ ranges over annotated type environments.
- $\widehat{\sigma}$ ranges over annotated type schemes.

Usage analysis: syntax

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

$$\begin{array}{lcl}
\varphi & ::= & 1 \mid \omega \\
\widehat{\tau} & ::= & \alpha \mid \mathit{Nat} \mid \widehat{\tau}_1{}^{\varphi_1} \to \widehat{\tau}_2{}^{\varphi_2} \\
\widehat{\sigma} & ::= & \widehat{\tau} \mid \forall\alpha.\,\widehat{\sigma}_1 \\
\widehat{\Gamma} & ::= & [\,] \mid \widehat{\Gamma}_1[x \mapsto^{\varphi} \widehat{\sigma}]
\end{array}$$

- We write $\widehat{\Gamma}(x) =^{\varphi} \widehat{\sigma}$ if the rightmost binding for x in $\widehat{\Gamma}$ associates to φ and $\widehat{\sigma}$.
- We write $\widehat{\Gamma} \setminus x$ for the environment obtained by removing all bindings for x from $\widehat{\Gamma}$.

Usage analysis: numerals

It depends on the context of a numeral whether it used at most once:

$$\frac{}{\widehat{\Gamma} \vdash_{\mathsf{UA}} n :^{1} \mathit{Nat}}$$

—or possibly more than once:

$$\frac{}{\widehat{\Gamma} \vdash_{\mathsf{UA}} n :^{\omega} \mathit{Nat}}$$

Merging the two rules:

$$\frac{}{\widehat{\Gamma} \vdash_{\mathsf{UA}} n :^{\varphi} \mathit{Nat}}$$

Usage analysis: variables

To analyse a variable, we look it up in the environment:

$$\frac{\widehat{\Gamma}(x) =^{\varphi} \widehat{\sigma}}{\widehat{\Gamma} \vdash_{\mathsf{UA}} x :^{\varphi} \widehat{\sigma}}$$

The rôle of environments

An annotated type environment should reflect how often the free variables of a term are used:

$$[x \mapsto^{1} Nat] \vdash_{\mathsf{UA}} x + 1 :^{\varphi} Nat$$

should be valid.

$$[x \mapsto^{1} Nat] \vdash_{\mathsf{UA}} x + x :^{\varphi} Nat$$

should not be valid.

$$[x \mapsto^{\omega} Nat] \vdash_{\mathsf{UA}} x + 1 :^{\varphi} Nat$$

should be valid.

$$[x \mapsto^{\omega} Nat] \vdash_{\mathsf{UA}} x + x :^{\varphi} Nat$$

should be valid.

Context splitting

- **Idea:** for every possible branch in a term's control-flow graph (for example a function application or an addition), we split the type environment in a left and a right part: $\widehat{\Gamma} \sim_{\mathsf{UA}} \widehat{\Gamma}_1 \bowtie \widehat{\Gamma}_2$.
- Bindings for 1-annotated variables go either left or right.
- Bindings for ω-annotated variables may go both ways.

Context splitting: rules

$$\frac{}{[\,] \sim_{\mathsf{UA}} [\,] \bowtie [\,]}$$

$$\frac{\widehat{\Gamma}_1 \sim_{\mathsf{UA}} \widehat{\Gamma}_{11} \bowtie \widehat{\Gamma}_{12}}{\widehat{\Gamma}_1[x \mapsto^{\varphi} \widehat{\sigma}] \sim_{\mathsf{UA}} \widehat{\Gamma}_{11}[x \mapsto^{\varphi} \widehat{\sigma}] \bowtie \widehat{\Gamma}_{12} \setminus x}$$

$$\frac{\widehat{\Gamma}_1 \sim_{\mathsf{UA}} \widehat{\Gamma}_{11} \bowtie \widehat{\Gamma}_{12}}{\widehat{\Gamma}_1[x \mapsto^{\varphi} \widehat{\sigma}] \sim_{\mathsf{UA}} \widehat{\Gamma}_{11} \setminus x \bowtie \widehat{\Gamma}_{12}[x \mapsto^{\varphi} \widehat{\sigma}]}$$

$$\frac{\widehat{\Gamma}_1 \sim_{\mathsf{UA}} \widehat{\Gamma}_{11} \bowtie \widehat{\Gamma}_{12}}{\widehat{\Gamma}_1[x \mapsto^{\omega} \widehat{\sigma}] \sim_{\mathsf{UA}} \widehat{\Gamma}_{11}[x \mapsto^{\omega} \widehat{\sigma}] \bowtie \widehat{\Gamma}_{12}[x \mapsto^{\omega} \widehat{\sigma}]}$$

Usage analysis: addition

$$\frac{\widehat{\Gamma} \sim_{\mathsf{UA}} \widehat{\Gamma}_1 \bowtie \widehat{\Gamma}_2 \quad \widehat{\Gamma}_1 \vdash_{\mathsf{UA}} t_1 :^{\varphi_1} \mathit{Nat} \quad \widehat{\Gamma}_2 \vdash_{\mathsf{UA}} t_2 :^{\varphi_2} \mathit{Nat}}{\widehat{\Gamma} \vdash_{\mathsf{UA}} t_1 + t_2 :^{\varphi} \mathit{Nat}}$$

☞ If a variable is used in both t_1 and t_2, context splitting guarantees that it is ω-annotated in $\widehat{\Gamma}$.

Usage analysis: example

$$\dfrac{\dfrac{\dfrac{\widehat{\Gamma}_{11}(x) =^{\omega} Nat}{\widehat{\Gamma}_{11} \vdash_{\mathsf{UA}} x :^{\omega} Nat} \quad \dfrac{\widehat{\Gamma}_{12}(y) =^{1} Nat}{\widehat{\Gamma}_{12} \vdash_{\mathsf{UA}} y :^{1} Nat}}{\widehat{\Gamma}_{1} \vdash_{\mathsf{UA}} x + y :^{1} Nat} \quad \dfrac{\widehat{\Gamma}_{2}(x) =^{\omega} Nat}{\widehat{\Gamma}_{2} \vdash_{\mathsf{UA}} x :^{\omega} Nat}}{[x \mapsto^{\omega} Nat, y \mapsto^{1} Nat, z \mapsto^{1} Nat] \vdash_{\mathsf{UA}} (x + y) + x :^{1} Nat}$$

(context splits omitted)

$$\begin{aligned}
\widehat{\Gamma}_{1} &= [x \mapsto^{\omega} Nat, y \mapsto^{1} Nat, z \mapsto^{1} Nat] \\
\widehat{\Gamma}_{11} &= [x \mapsto^{\omega} Nat, \phantom{y \mapsto^{1} Nat,} z \mapsto^{1} Nat] \\
\widehat{\Gamma}_{12} &= [x \mapsto^{\omega} Nat, y \mapsto^{1} Nat \phantom{, z \mapsto^{1} Nat}] \\
\\
\widehat{\Gamma}_{2} &= [x \mapsto^{\omega} Nat \phantom{, y \mapsto^{1} Nat, z \mapsto^{1} Nat}]
\end{aligned}$$

Usage analysis: local definitions

$$\frac{\widehat{\Gamma} \sim_{\mathsf{UA}} \widehat{\Gamma}_1 \bowtie \widehat{\Gamma}_2 \quad \widehat{\Gamma}_1 \vdash_{\mathsf{UA}} t_1 :^{\varphi_1} \widehat{\sigma}_1 \quad \widehat{\Gamma}_2[x \mapsto^{\varphi_1} \widehat{\sigma}_1] \vdash_{\mathsf{UA}} t_2 :^{\varphi} \widehat{\tau}}{\widehat{\Gamma} \vdash_{\mathsf{UA}} \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} :^{\varphi} \widehat{\tau}}$$

Usage analysis: applications

$$\frac{\widehat{\Gamma} \sim_{\mathsf{UA}} \widehat{\Gamma}_1 \bowtie \widehat{\Gamma}_2 \quad \widehat{\Gamma}_1 \vdash_{\mathsf{UA}} t_1 :^{\varphi_1} \widehat{\tau}_2{}^{\varphi_2} \rightarrow \widehat{\tau}^{\varphi} \quad \widehat{\Gamma}_2 \vdash_{\mathsf{UA}} t_2 :^{\varphi_2} \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathsf{UA}} t_1\ t_2 :^{\varphi} \widehat{\tau}}$$

- Domain and domain annotation should match type and usage of argument.
- Result type and usage of application are retrieved from codomain and codomain annotation.

Usage analysis: abstractions (first attempt)

$$\frac{\widehat{\Gamma}[x \mapsto^{\varphi_1} \widehat{\tau}_1] \vdash_{\mathsf{UA}} t_1 :^{\varphi_2} \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathsf{UA}} \lambda x.\, t_1 :^{\varphi} \widehat{\tau}_1{}^{\varphi_1} \rightarrow \widehat{\tau}_2{}^{\varphi_2}}$$

For example:

$$[\,] \vdash_{\mathsf{UA}} \lambda x.\, x + 1 :^{1} Nat^{1} \rightarrow Nat^{1}$$

$$[\,] \vdash_{\mathsf{UA}} \lambda x.\, x + x :^{1} Nat^{\omega} \rightarrow Nat^{1}$$

Partial applications: problem

$$
\begin{array}{l}
\textbf{let } f = \lambda x.\,\lambda y.\,x + y \\
\textbf{in } \;\textbf{let } g = f\ (2+3) \\
\qquad \textbf{in } \; g\ 7 + g\ 11 \\
\qquad \textbf{ni} \\
\textbf{ni}
\end{array}
$$

- How often is g used?
- How often is $(2+3)$ used?
- $Nat^1 \rightarrow (Nat^1 \rightarrow Nat^1)^\omega$ is a valid type for f. Should it be?

Containment

- **Containment:** an object is potentially used as least as often as an object it is contained in.

$$
\begin{array}{l}
\textbf{let } f = \lambda x.\, \lambda y.\, x + y \\
\textbf{in } \ \textbf{let } g = f\ (2 + 3) \\
\qquad \textbf{in } \ g\ 7 + g\ 11 \\
\qquad \textbf{ni} \\
\textbf{ni}
\end{array}
$$

- The binding of x to $(2 + 3)$ is contained in the partial application g.
- The partial application is used more than once: hence, so is $(2 + 3)$.

$$\frac{\widehat{\Gamma}[x \mapsto^{\varphi_1} \widehat{\tau}_1] \vdash_{\mathsf{UA}} t_1 :^{\varphi_2} \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathsf{UA}} \lambda x.\, t_1 :^{\varphi} \widehat{\tau}_1^{\,\varphi_1} \to \widehat{\tau}_2^{\,\varphi_2}}$$

- **Problem:** the free variables of the abstraction could be used as least as often as the abstraction itself.
- The usage of the free variables is reflected by $\widehat{\Gamma}$.
- The usage of the abstraction is reflected by φ.
- **Solution:** If $\varphi \equiv \omega$, then all bindings in $\widehat{\Gamma}$ that are used in the typing of t_1 should also be ω.

Usage analysis: abstractions (refined)

$$\frac{\widehat{\Gamma} \rhd^{\varphi} \widehat{\Gamma}_{11} \quad \widehat{\Gamma}_{11}[x \mapsto^{\varphi_1} \widehat{\tau}_1] \vdash_{\mathsf{UA}} t_1 :^{\varphi_2} \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathsf{UA}} \lambda x.\, t_1 :^{\varphi} \widehat{\tau}_1{}^{\varphi_1} \to \widehat{\tau}_2{}^{\varphi_2}}$$

$\widehat{\Gamma} \rhd^{\varphi} \widehat{\Gamma}_{11}$:

- $\widehat{\Gamma}_{11}$ is a subenvironment of $\widehat{\Gamma}$;
- if $\varphi \equiv \omega$, then all bindings in $\widehat{\Gamma}_{11}$ are annotated with ω.

Containment: rules

$$\frac{}{[\,] \rhd^{\varphi} [\,]}$$

$$\frac{\widehat{\Gamma}_{11} \rhd^{\varphi} \widehat{\Gamma}_{2}}{\widehat{\Gamma}_{11}[x \mapsto^{\varphi_0} \widehat{\sigma}] \rhd^{\varphi} \widehat{\Gamma}_{2}}$$

$$\frac{\widehat{\Gamma}_{11} \rhd^{1} \widehat{\Gamma}_{2}}{\widehat{\Gamma}_{11}[x \mapsto^{\varphi_0} \widehat{\sigma}] \rhd^{1} \widehat{\Gamma}_{2}[x \mapsto^{\varphi_0} \widehat{\sigma}]}$$

$$\frac{\widehat{\Gamma}_{11} \rhd^{\omega} \widehat{\Gamma}_{2}}{\widehat{\Gamma}_{11}[x \mapsto^{\omega} \widehat{\sigma}] \rhd^{\omega} \widehat{\Gamma}_{2}[x \mapsto^{\omega} \widehat{\sigma}]}$$

Containment: examples

$$\begin{array}{l}
\textbf{let}\ f = \lambda x.\,\lambda y.\,x + y \\
\textbf{in}\ \ \textbf{let}\ g = f\ (2+3) \\
\qquad \textbf{in}\ \ g\ 7 + g\ 11 \\
\qquad \textbf{ni} \\
\textbf{ni}
\end{array}$$

$$\dfrac{\dfrac{\dfrac{\vdots}{[x \mapsto^{\omega} Nat] \rhd^{\omega} [x \mapsto^{\omega} Nat]} \quad \dfrac{\vdots}{[x \mapsto^{\omega} Nat, y \mapsto^{1} Nat] \vdash_{\mathsf{UA}} x + y :^{1} Nat}}{[x \mapsto^{\omega} Nat] \vdash_{\mathsf{UA}} \lambda y.\,x + y :^{\omega} Nat^{1} \to Nat^{1}}}{\vdots}$$

Where are we?

- An annotated type system for usage analysis.
- Judgements of the form $\widehat{\Gamma} \vdash_{\mathsf{UA}} t :^{\varphi} \widehat{\sigma}$.
- Auxiliary judgement for context splitting: $\widehat{\Gamma} \sim_{\mathsf{UA}} \widehat{\Gamma}_1 \bowtie \widehat{\Gamma}_2$.
- Auxiliary judgement for containment: $\widehat{\Gamma} \triangleright^{\varphi} \widehat{\Gamma}_{11}$.

Applications

- Verification: type checking destructive updates (uniqueness typing).
- Optimization: avoiding unnecessary closure updates (sharing analysis).

4. Type checking destructive updates

Construct for destructive updates

- To demonstrate how the analysis can be used to perform uniqueness typing, we extend the language with a simple construct for destructive updates.

$$t \quad ::= \quad \cdots \mid x@t$$

- Meaning: update x with t.
- Can be formalized with a semantics that explicitly models memory usage.
- See Hage and Holdermans (PEPM 2008).

Typing rule for updates

- Require that updated object is unique.

$$\frac{\widehat{\Gamma}(x) =^{1} \widehat{\sigma}_0 \quad \widehat{\Gamma} \vdash_{\mathsf{UA}} t :^{\varphi} \widehat{\sigma}}{\widehat{\Gamma} \vdash_{\mathsf{UA}} x@t :^{\varphi} \widehat{\sigma}}$$

- Then: show that a program with updates has the same meaning as the same program with all updates removed.

5. Avoiding unnecessary closure updates

Generating use-once closures

- To avoid unnecessary closure updates, we compile to a target language that distinguishes between closures that can be used at most once and closures that can be used more than once.
- For each let-binding we indicate what kind of closure needs to be constructed.
- We make sure that closures are only created at let-bindings.

$\widehat{t}$	$\in$	$\widehat{\mathbf{Tm}}$	annotated terms
$\widehat{t}$	$::=$	$\cdots \mid \widehat{t}_1\ x \mid \mathbf{let}\ x =^{\varphi} \widehat{t}_1\ \mathbf{in}\ \widehat{t}_2\ \mathbf{ni} \mid \cdots$	

- We equip the target language with a semantics that makes memory usage explicit and renders use-once closures inaccessible after their first use.

Target language: examples

$$\textbf{let } z =^{1} 2 + 3$$
$$\textbf{in } (\lambda x.\, x + 1)\ z$$
$$\textbf{ni}$$

$$\textbf{let } z =^{\omega} 2 + 3$$
$$\textbf{in } (\lambda x.\, x + x)\ z$$
$$\textbf{ni}$$

Translation

- We write $\mathcal{T} :: \widehat{\Gamma} \vdash_{\mathsf{UA}} t :^{\varphi} \widehat{\sigma}$ to indicate that $\mathcal{T}$ is a proof tree for $\widehat{\Gamma} \vdash_{\mathsf{UA}} t :^{\varphi} \widehat{\sigma}$.
- Next, we define a translation $[\![-]\!]$ from proof trees to target terms.
- For example:

$$\left[\!\!\left[\frac{\begin{array}{c}\mathcal{T}_0 :: \widehat{\Gamma} \sim_{\mathsf{UA}} \widehat{\Gamma}_1 \bowtie \widehat{\Gamma}_2 \\ \mathcal{T}_1 :: \widehat{\Gamma}_1 \vdash_{\mathsf{UA}} t_1 :^{\varphi_1} \widehat{\sigma}_1 \\ \mathcal{T}_2 :: \widehat{\Gamma}_2[x \mapsto^{\varphi_1} \widehat{\sigma}_1] \vdash_{\mathsf{UA}} t_2 :^{\varphi} \widehat{\tau}\end{array}}{\widehat{\Gamma} \vdash_{\mathsf{UA}} \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} :^{\varphi} \widehat{\tau}}\right]\!\!\right] = \mathbf{let}\ x =^{\varphi_1} [\![\mathcal{T}_1]\!]\ \mathbf{in}\ [\![\mathcal{T}_2]\!]\ \mathbf{ni}$$

- Then, show that each translated program evaluates to the value of the original program.

6. Subeffecting

Lack of modularity

$$\textbf{let } x = 2 + 3 \\ \textbf{in } (\lambda x.\, x + 1)\ x \\ \textbf{ni}$$

$$x :^{1} Nat$$

$$\textbf{let } x = 2 + 3 \\ \textbf{in } (\lambda x.\, x + x)\ x \\ \textbf{ni}$$

$$x :^{\omega} Nat$$

☞ Use of x in body determines its usage annotation.

Poisoning

$$
\begin{array}{l}
\textbf{let } id = \lambda x.\, x \\
\textbf{in } \ \textbf{let } y = 2 + 3 \\
\quad\quad \textbf{in } \ \textbf{let } z = 5 \\
\quad\quad\quad\quad \textbf{in } \ id\ y + id\ z + z \\
\quad\quad\quad\quad \textbf{ni} \\
\quad\quad \textbf{ni} \\
\textbf{ni}
\end{array}
$$

- z is used more than once: hence, $z :^{\omega} Nat$.
- id is applied to z: hence, $id :^{\omega} Nat^{\omega} \rightarrow Nat^{\omega}$.
  (Or $id :^{\omega} \forall \alpha .\, \alpha^{\omega} \rightarrow \alpha^{\omega}$.)
- id is applied to y: hence, $y :^{\omega} Nat$.
- But y is used **only once!!**

Who's to blame?

- Recall the rule for function application:

$$\frac{\widehat{\Gamma} \sim_{\mathsf{UA}} \widehat{\Gamma}_1 \bowtie \widehat{\Gamma}_2 \quad \widehat{\Gamma}_1 \vdash_{\mathsf{UA}} t_1 :^{\varphi_1} \widehat{\tau}_2{}^{\varphi_2} \to \widehat{\tau}^{\varphi} \quad \widehat{\Gamma}_2 \vdash_{\mathsf{UA}} t_2 :^{\varphi_2} \widehat{\tau}_2}{\widehat{\Gamma} \vdash_{\mathsf{UA}} t_1\ t_2 :^{\varphi} \widehat{\tau}}$$

- Argument annotation φ_2 should match the annotation on the function domain.
- But in uniqueness typing it's safe to bind a 1-annotated argument to an ω-annotated function parameter.
- But in sharing analysis, it's safe to bind an ω-annotated argument to a 1-annotated function parameter.

Turning the reasoning around...

- In uniqueness analysis, a 1-annotation on a formal parameter may not receive ω-annotated values.
  - The latter may have been duplicated, while the 1-annotation implies that destructive updates may take place on the value.
- In sharing analysis, an ω-annotated formal parameter (that may then use its arguments twice), should not be passed a 1-annotated argument.
  - As a rule, you garbage collect 1-annotated values after their use.
- The difference is then that for uniqueness typing the 1 on the argument matters, and for sharing analysis the 1 on the values.
- The latter decides what kind of thunk must be created, the former what applications of the function are correct.

Ordering on annotations

Partial order on $\mathbf{Ann}$ with $1 \sqsubset \omega$:

$$\frac{}{1 \sqsubseteq \varphi}$$

$$\frac{}{\varphi \sqsubseteq \omega}$$

Subeffecting: uniqueness typing

- From our generic usage analysis we can derive a system that is specific for uniqueness typing.
- Judgements of the form $\widehat{\Gamma} \vdash_{\mathsf{UT}} t :^{\varphi} \widehat{\sigma}$.
- Same rules as before.
- New rule for subeffecting:

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{UT}} t :^{\varphi_0} \widehat{\sigma} \qquad \varphi_0 \sqsubseteq \varphi}{\widehat{\Gamma} \vdash_{\mathsf{UT}} t :^{\varphi} \widehat{\sigma}}$$

Example

- Let $\widehat{\Gamma} = [f \mapsto^{1} (Nat^{\omega} \to Nat^{1}), x \mapsto^{1} Nat]$.
- For example: $f = \lambda x.\, x + x$ and $x = 2 + 3$.

$$\dfrac{\widehat{\Gamma} \sim_{\mathsf{UA}} \widehat{\Gamma}_1 \bowtie \widehat{\Gamma}_2 \qquad \dfrac{\widehat{\Gamma}_1(f) =^{1} Nat^{\omega} \to Nat^{1}}{\widehat{\Gamma}_1 \vdash_{\mathsf{UT}} f :^{1} Nat^{\omega} \to Nat^{1}} \qquad \dfrac{\dfrac{\widehat{\Gamma}_2(x) =^{1} Nat}{\widehat{\Gamma}_2 \vdash_{\mathsf{UT}} x :^{1} Nat} \quad 1 \sqsubseteq \omega}{\widehat{\Gamma}_2 \vdash_{\mathsf{UT}} x :^{\omega} Nat}}{\widehat{\Gamma} \vdash_{\mathsf{UT}} f\ x :^{1} Nat}$$

$\widehat{\Gamma}_1 = [f \mapsto^{1} (Nat^{\omega} \to Nat^{1})]$ and $\widehat{\Gamma}_2 = [x \mapsto^{1} Nat]$

Subeffecting: sharing analysis

- We can also derive a system that is specific for sharing analysis.
- Judgements of the form $\widehat{\Gamma} \vdash_{\mathsf{SA}} t :^{\varphi} \widehat{\sigma}$.
- Same rules as in the generic analysis.
- Again, a new rule for subeffecting:

$$\frac{\widehat{\Gamma} \vdash_{\mathsf{UT}} t :^{\varphi_0} \widehat{\sigma} \qquad \varphi \sqsubseteq \varphi_0}{\widehat{\Gamma} \vdash_{\mathsf{UT}} t :^{\varphi} \widehat{\sigma}}$$

Example

- In sharing analysis, doing a self-update that is not necessary is not unsound.
  - A value created with annotation ω can be used in a 1-annotated setting.
- Let $\widehat{\Gamma} = [f \mapsto^1 (Nat^1 \to Nat^1), x \mapsto^\omega Nat]$.
- For example: $f = \lambda x.\, x + 1$ and $x = 2 + 3$.

$$\dfrac{\widehat{\Gamma} \sim_{\mathsf{UA}} \widehat{\Gamma}_1 \bowtie \widehat{\Gamma}_2 \qquad \dfrac{\widehat{\Gamma}_1(f) =^1 Nat^1 \to Nat^1}{\widehat{\Gamma}_1 \vdash_{\mathsf{SA}} f :^1 Nat^1 \to Nat^1} \qquad \dfrac{\dfrac{\widehat{\Gamma}_2(x) =^\omega Nat}{\widehat{\Gamma}_2 \vdash_{\mathsf{SA}} x :^\omega Nat} \quad 1 \sqsubseteq \omega}{\widehat{\Gamma}_2 \vdash_{\mathsf{SA}} x :^1 Nat}}{\widehat{\Gamma} \vdash_{\mathsf{SA}} f\ x :^1 Nat}$$

$\widehat{\Gamma}_1 = [f \mapsto^1 (Nat^1 \to Nat^1), x \mapsto^\omega Nat]$ and $\widehat{\Gamma}_2 = [x \mapsto^\omega Nat]$

Keeping the analysis generic

- Define the inverse partial order $(\mathbf{Ann}, \sqsupseteq)$ with $\omega \sqsupset 1$.
- Let $\diamond$ range over the two partial orders:

$\diamond$	$\in$	$\mathbf{Ord} = \{\sqsubseteq, \sqsupseteq\}$	partial orders

- Parameterize the judgements of the generic analysis with a partial order $\diamond$:

$$\frac{\widehat{\Gamma} \vdash^{\diamond}_{\mathsf{UA}} t :^{\varphi_0} \widehat{\sigma} \qquad \varphi_0 \diamond \varphi}{\widehat{\Gamma} \vdash^{\diamond}_{\mathsf{UA}} t :^{\varphi} \widehat{\sigma}}$$

Instantiation

Uniqueness typing:

$$\frac{\widehat{\Gamma} \vdash^{\sqsubseteq}_{\mathsf{UA}} t :^{\varphi} \widehat{\sigma}}{\widehat{\Gamma} \vdash_{\mathsf{UT}} t :^{\varphi} \widehat{\sigma}}$$

Sharing analysis:

$$\frac{\widehat{\Gamma} \vdash^{\sqsupseteq}_{\mathsf{UA}} t :^{\varphi} \widehat{\sigma}}{\widehat{\Gamma} \vdash_{\mathsf{SA}} t :^{\varphi} \widehat{\sigma}}$$

7. Polyvariance

What about modularity?

- Idea: independent from its use sites, can we assign each function its "most flexible" type:
- For uniqueness analysis:

$$\lambda x.\, x + 1 :^{\omega} Nat^{\omega} \to Nat^{1}$$
$$\lambda x.\, x :^{\omega} Nat^{\omega} \to Nat^{\omega}$$

- For sharing analysis:

$$\lambda x.\, x + 1 :^{\omega} Nat^{1} \to Nat^{\omega}$$
$$\lambda x.\, x :^{\omega} Nat^{??} \to Nat^{??}$$

Polyvariance

- Allow types to be polymorphic in their annotations.
- For uniqueness analysis:

$$\begin{array}{lll} \lambda x.\, x+1 & :^{\omega} \forall\beta_1.\forall\beta_2. & Nat^{\beta_1} \to Nat^{\beta_2} \\ \lambda x.\, x & :^{\omega} \forall\beta. & Nat^{\beta} \to Nat^{\beta} \end{array}$$

- For sharing analysis:

$$\begin{array}{lll} \lambda x.\, x+1 & :^{\omega} \forall\beta_1.\forall\beta_2. & Nat^{\beta_1} \to Nat^{\beta_2} \\ \lambda x.\, x & :^{\omega} \forall\beta. & Nat^{\beta} \to Nat^{\beta} \end{array}$$

8. Subeffect qualifiers

How to capture all valid types?

In uniqueness typing (with subeffecting):

$$\lambda x.\, x :^{\omega} \forall \alpha.\, \alpha^{1} \to \alpha^{1}$$
$$\lambda x.\, x :^{\omega} \forall \alpha.\, \alpha^{1} \to \alpha^{\omega}$$
$$\lambda x.\, x :^{\omega} \forall \alpha.\, \alpha^{\omega} \to \alpha^{\omega}$$

In sharing analysis (with subeffecting):

$$\lambda x.\, x :^{\omega} \forall \alpha.\, \alpha^{1} \to \alpha^{1}$$
$$\lambda x.\, x :^{\omega} \forall \alpha.\, \alpha^{\omega} \to \alpha^{1}$$
$$\lambda x.\, x :^{\omega} \forall \alpha.\, \alpha^{\omega} \to \alpha^{\omega}$$

Which polyvariant type captures all valid types?

$\forall \alpha. \forall \beta.\ \alpha^{\beta} \to \alpha^{\beta}$	(not general enough)
$\forall \alpha. \forall \beta_1. \forall \beta_2.\ \alpha^{\beta_1} \to \alpha^{\beta_2}$	(too general)

Poisoning (again)

$$
\begin{array}{l}
\textbf{let } h = \lambda f.\lambda x.\lambda y.f\ x + f\ y \\
\textbf{in } \ \textbf{let } g = \lambda z.z + 1 \\
\quad\quad \textbf{in } \ \textbf{let } u = 2 + 3 \\
\quad\quad\quad\quad \textbf{in } \ \textbf{let } v = 5 + 7 \\
\quad\quad\quad\quad\quad\quad \textbf{in } \ h\ g\ u\ v + v \\
\quad\quad\quad\quad\quad\quad \textbf{ni} \\
\quad\quad\quad\quad \textbf{ni} \\
\quad\quad \textbf{ni} \\
\textbf{ni}
\end{array}
$$

- Let $h :^1 \forall\beta. (Nat^\beta \to Nat^1)^\omega \to (Nat^\beta \to (Nat^\beta \to Nat^1)^1)^1$.
- v is used more than once, hence: $v :^\omega Nat$.
- But then, in the call to h, β is instantiated to ω.
- For sharing analysis, this means that $u :^\omega Nat$.
- But u is used **only once!!**

Qualified types

- To gain accuracy, we can store subeffecting conditions in type schemes.
- Qualified types are a generalization of Haskell's type classes that allow constraints to be incorporated in types.
- Elegant and well-established theory: see Jones (ESOP 1992).

$$\lambda x.\, x :^{\omega} \forall \alpha.\, \forall \beta_1.\, \forall \beta_2.\, \beta_1 \diamond \beta_2 \Rightarrow \alpha^{\beta_1} \rightarrow \alpha^{\beta_2}$$

Example revisited

$$
\begin{array}{l}
\textbf{let } h = \lambda f.\lambda x.\lambda y.f\ x + f\ y \\
\textbf{in } \ \textbf{let } g = \lambda z.z + 1 \\
\quad\ \ \textbf{in } \ \textbf{let } u = 2 + 3 \\
\quad\quad\quad \textbf{in } \ \textbf{let } v = 5 + 7 \\
\quad\quad\quad\quad\ \ \textbf{in } \ h\ g\ u\ v + v \\
\quad\quad\quad\quad\ \ \textbf{ni} \\
\quad\quad\quad \textbf{ni} \\
\quad\ \ \textbf{ni} \\
\textbf{ni}
\end{array}
$$

- Sharing analysis.
- Let $h :^1 \forall \beta_1\ \beta_2\ \beta_3.\ \beta_2 \sqsupseteq \beta_1 \Rightarrow \beta_3 \sqsupseteq \beta_1 \Rightarrow (Nat^{\beta_1} \to Nat^1)^\omega \to (Nat^{\beta_2} \to (Nat^{\beta_3} \to Nat^1)^1)^1$.
- v is used more than once, hence: $v :^\omega Nat$.
- So, in the call to h, β_3 is instantiated to ω.
- Still, the constraints are satisfied if $\beta_1 = \beta_2 = 1$.
- Hence, we can have $u :^1 Nat$.

Principal types

Most general types can sometimes be a bit intimidating.

$$\lambda f.\lambda x.\lambda y.f\ x + f\ y :$$
$$\forall\alpha.\forall\beta_1.\forall\beta_2.\forall\beta_3.\forall\beta_4.\forall\beta_5.\forall\beta_6.\forall\beta_7.\forall\beta_8.$$
$$\beta_3 \diamond \beta_1 \Rightarrow \beta_4 \diamond \beta_1 \Rightarrow \beta_7 \sqsubseteq \beta_3 \Rightarrow$$
$$(\alpha^{\beta_1} \to Nat^{\beta_2})^{\omega} \to (\alpha^{\beta_3} \to (\alpha^{\beta_4} \to Nat^{\beta_5})^{\beta_7})^{\beta_8}$$

9. Properties of type systems (Metatheory)

Subject reduction

- If an expression has type τ, then the value it evaluates to also has type τ.
- Type preservation is a bit weaker: every evaluation step keeps the result well-typed.
  - But the types may change

Conservative extension

- If a program can be typed, then it can be analyzed.
- If a program can be analyzed, erasing the annotations from the proof tree gives the proof tree for the type system.

Safety/Soundness

- In the underlying type system: well-typed programs do not go wrong.
- In the annotated type system: acting on the optimisations implied by the annotations does not make evaluation go wrong.
- Usually, the semantics must be changed slightly to observe this.
- In the case of sharing analysis:
  - Distinguish between 1-annotated and w-annotated thunks.
  - Remove the 1-annotated thunks from the heap when they have been used (once).
  - Show that you never need to access something that was removed from the heap.

Progress

- Only with respect to small-step semantics.
- Evaluation of a well-typed term never gets stuck.
- It might loop though.

Completeness

- Usually the analysis is not complete
  - Some never-go-wrong expressions cannot be typed.
  - Static analysis is approximate.
- Still, we do sometimes establish completeness.
- Consider an analysis that generates constraints to capture the analysis.
- And build a solver to find a solution to the constraints.
- We want that solver to be
  - sound: the solution it computes is a solution
  - complete: if a set of constraints has a solution, the solver should find it (or a better solution).

Principality

- We prefer the analysis to provide a best solution,
- from which all other solutions can be derived.
- Depends very much on the expressivity of your types: $\lambda x.\, x$ may have type $Nat \rightarrow Nat$ or $Bool \rightarrow Bool$ if we do not allow type variables in types.
- Neither is better than the other.
- Principality allows to solve constraints, have the result be a principal type, and forget the constraints from then on.
- There is never a need to re-analyze: the principal type says all.
- Not to be confused with principal typings.