Feedback oriented static analysis
Type and effect systems

Jurriaan Hage
e-mail: jur@cs.uu.nl
homepage: http://www.cs.uu.nl/people/jur/

Center for Software Technology, Department of Information and Computing Sciences,
Universiteit Utrecht

September 13, 2006

Overview

1. Introduction
2. The type system
3. Adding effects
4. Using constraints instead

Goals of these slides

- Refreshing your knowledge of type systems
- Introducing effects.
- Describe algorithms that enforce a type system.
- How this differs from the `Helium` and TOP way of working.

Types and effects

- At their most general
  - A *type* is a property of a value.
  - An *effect* is a property of the way the value was obtained.
- Consider the result of the expression $1 + 2 * 3$:
  - The parity of 7 is odd, the 'type' of 7 is integer.
  - The computation of (this) 7 involved executing two operators.
- Due to the distinction they are often treated separately.
- The worlds of type and effect do intermix:
  - Function types $\tau \xrightarrow{\phi} \tau'$ often carry an effect ϕ: the effect of executing the body.
  - Such types are then called *annotated types*

Why use type and effect systems?

- Type systems are specified to make sure that programs do not go wrong.
- A type system sometimes forbids programs that would go right.
  - Indeed, it always will
- Effects are used to analyze the behaviour of programs.
  - Memory usage, dead code analysis
- In FOSA we shall usually only consider ordinary type systems.

Setting things up

- Traditionally, type systems are specified as a logical deduction system
- Implemented by an abstract syntax tree traversing algorithm (algorithm $\mathcal{W}$)
- We use a small non-polymorphic, functional language, `Fun`.
- First specify an underlying type system for typing `Fun`.
  - Based on a collection of type rules, one for each syntax construct.
- Then we add effects to perform an additional Control Flow analysis.
- Then we turn this into an algorithm.

Types and judgments

- Types for Fun: $\tau ::= \textbf{int} \mid \textbf{bool} \mid \tau_1 \rightarrow \tau_2$
  - ints, boolean and function types, but no type variables!
- Types of all constants, e.g. 8 and *true* are given by τ_8 and τ_{true}.
- Type environments, Γ, bind identifiers to types $[v \mapsto int, x \mapsto int \rightarrow bool]$
- Type systems are formulated in terms of judgments
- Judgements may have different forms for different type systems.
- In this case: $\Gamma \vdash_{\mathrm{UL}} e : \tau$
  - $\vdash_{\mathrm{UL}}$ is a predicate
  - Read as: the expression e has type τ under type environment Γ.
- The judgment is the basic predicate for specifying well-typedness, but others can be used at will.

The type rules (1)

$$\frac{}{\Gamma \vdash_{\mathrm{UL}} c : \tau_c} \text{[con]}$$

$$\frac{\tau = \Gamma(x)}{\Gamma \vdash_{\mathrm{UL}} x : \tau} \text{[var]}$$

$$\frac{\Gamma[x \mapsto \tau_x] \vdash_{\mathrm{UL}} e_0 : \tau_0}{\Gamma \vdash_{\mathrm{UL}} \mathbf{fn}_\pi \ x \ => \ e_0 : \tau_x \to \tau_0} \text{[fn]}$$

$$\frac{\Gamma[f \mapsto \tau_x \to \tau_0][x \mapsto \tau_x] \vdash_{\mathrm{UL}} e_0 : \tau_0}{\Gamma \vdash_{\mathrm{UL}} \mathbf{fun}_\pi \ f \ x \ => \ e_0 : \tau_x \to \tau_0} \text{[fun]}$$

$$\frac{\Gamma \vdash_{\mathrm{UL}} e_1 : \tau_2 \to \tau_0 \qquad \Gamma \vdash_{\mathrm{UL}} e_2 : \tau_2}{\Gamma \vdash_{\mathrm{UL}} e_1 \ e_2 : \tau_0} \text{[app]}$$

The type rules (2)

$$\frac{\Gamma \vdash_{\mathrm{UL}} e_0 : \textbf{bool} \qquad \Gamma \vdash_{\mathrm{UL}} e_1 : \tau \qquad \Gamma \vdash_{\mathrm{UL}} e_2 : \tau}{\Gamma \vdash_{\mathrm{UL}} \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 : \tau} \text{[if]}$$

$$\frac{\Gamma \vdash_{\mathrm{UL}} e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash_{\mathrm{UL}} e_2 : \tau_2}{\Gamma \vdash_{\mathrm{UL}} \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2} \text{[let]}$$

$$\frac{\Gamma \vdash_{\mathrm{UL}} e_1 : \tau_1 \qquad \Gamma \vdash_{\mathrm{UL}} e_2 : \tau_2 \qquad \Gamma(op) = \tau_1 \rightarrow \tau_2 \rightarrow \tau}{\Gamma \vdash_{\mathrm{UL}} e_1 \; op \; e_2 : \tau} \text{[op]}$$

A type derivation

- If you tell the deduction system the right choices for all the τs and for Γ, then we can check that the program is well-typed.
- Example program: $(\mathbf{fn}_X\ x\ \ =>\ \ x)\ (\mathbf{fn}_Y\ y\ \ =>\ \ y)$
- Choose type of x to be $(\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau)$ and y that of $(\tau \rightarrow \tau)$ for e.g. $\tau = \mathbf{int}$.

Annotated types

- Add extra information to (some) types.
  - Example: add function names to function types.
- For Control Flow Analysis: $\varphi ::= \{\pi\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset$
  - π refers to specific functions in the program, like X in ($\mathbf{fn}_{\mathrm{X}}\ x\ \ =>\ \ x$).
- Annotations are combined with type information: $\widehat{\tau} ::= \mathbf{int} \mid \mathbf{bool} \mid \widehat{\tau_1} \xrightarrow{\varphi} \widehat{\tau_2}$
  - Annotations represent the possible function abstractions that the expression with this type may originate from.
- Annotated type of $inc = \mathbf{fn}_{\mathrm{INC}}\ x\ =>\ \ x + 1$ is expected to be $\mathbf{int} \xrightarrow{\{\mathrm{INC}\}} \mathbf{int}$.
- Extends similarly to annotated environments $\widehat{\Gamma}$.
- We now give the annotated type system....

The annotated type rules (1)

$$\frac{\widehat{\tau} = \widehat{\Gamma}(x)}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} x : \widehat{\tau}} \quad [\mathrm{var}]$$

$$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau_x}] \vdash_{\mathrm{CFA}} e_0 : \widehat{\tau_0}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} \mathbf{fn}_\pi \ x \ => \ e_0 : \widehat{\tau_x} \overset{\{\pi\} \cup \varphi}{\rightarrow} \widehat{\tau_0}} \quad [\mathrm{fn}]$$

$$\frac{\widehat{\Gamma}[f \mapsto \widehat{\tau_x} \overset{\{\pi\} \cup \varphi}{\rightarrow} \widehat{\tau_0}][x \mapsto \widehat{\tau_x}] \vdash_{\mathrm{CFA}} e_0 : \widehat{\tau_0}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} \mathbf{fun}_\pi \ f \ x \ => \ e_0 : \widehat{\tau_x} \overset{\{\pi\} \cup \varphi}{\rightarrow} \widehat{\tau_0}} \quad [\mathrm{fun}]$$

$$\frac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} e_1 : \widehat{\tau_2} \overset{\varphi}{\rightarrow} \widehat{\tau_0} \qquad \widehat{\Gamma} \vdash_{\mathrm{CFA}} e_2 : \widehat{\tau_2}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} e_1 \ e_2 : \widehat{\tau_0}} \quad [\mathrm{app}]$$

The annotated type rules (2)

$$\frac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} e_0 : \textbf{bool} \qquad \widehat{\Gamma} \vdash_{\mathrm{CFA}} e_1 : \widehat{\tau} \qquad \widehat{\Gamma} \vdash_{\mathrm{CFA}} e_2 : \widehat{\tau}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 : \widehat{\tau}} \text{[if]}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} e_1 : \widehat{\tau_1} \qquad \widehat{\Gamma}[x \mapsto \widehat{\tau_1}] \vdash_{\mathrm{CFA}} e_2 : \widehat{\tau_2}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} \textbf{let } x = e_1 \textbf{ in } e_2 : \widehat{\tau_2}} \text{[let]}$$

$$\frac{\widehat{\Gamma} \vdash_{\mathrm{CFA}} e_1 : \tau_1 \qquad \widehat{\Gamma} \vdash_{\mathrm{CFA}} e_2 : \tau_2 \qquad \widehat{\Gamma}(op) = \tau_1 \rightarrow \tau_2 \rightarrow \tau}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} e_1 \ op \ e_2 : \tau} \text{[op]}$$

- Types in [op] are not annotated: operators do not work on functions

Example $(\mathbf{fn}_X\ x\ \texttt{=>}\ x)\ (\mathbf{fn}_Y\ y\ \texttt{=>}\ y)$

- Abbreviate $\widehat{\tau_Y} = \mathbf{int} \xrightarrow{\{Y\}} \mathbf{int}$.
- We 'guess' the type of each bound variable (i.e., the solution) and then check that all the rules are satisfied.
- Additionally, we mark function definitions with their name.
- End result: the result of the expression is a function application deriving from $(\mathbf{fn}_Y\ y\ \texttt{=>}\ y)$.

$$\dfrac{\dfrac{\dfrac{\widehat{\tau_Y} = [x \mapsto \widehat{\tau_Y}](x)}{[x \mapsto \widehat{\tau_Y}] \vdash_{\mathrm{CFA}} x : \widehat{\tau_Y}}\ [\mathrm{var}]}{[\,] \vdash_{\mathrm{CFA}} (\mathbf{fn}_X\ x\ \texttt{=>}\ x) : \widehat{\tau_Y} \xrightarrow{\{X\}} \widehat{\tau_Y}}\ [\mathrm{fn}] \qquad \dfrac{\dfrac{\mathbf{int} = [y \mapsto \mathbf{int}](y)}{[y \mapsto \mathbf{int}] \vdash_{\mathrm{CFA}} y : \mathbf{int}}\ [\mathrm{var}]}{[\,] \vdash_{\mathrm{CFA}} (\mathbf{fn}_Y\ y\ \texttt{=>}\ y) : \widehat{\tau_Y}}}{[\,] \vdash_{\mathrm{CFA}} (\mathbf{fn}_X\ x\ \texttt{=>}\ x)\ (\mathbf{fn}_Y\ y\ \texttt{=>}\ y) : \widehat{\tau_Y}}$$

Subeffecting

$$\frac{\widehat{\Gamma}[x \mapsto \widehat{\tau_x}] \vdash_{\mathrm{CFA}} e_0 : \widehat{\tau_0}}{\widehat{\Gamma} \vdash_{\mathrm{CFA}} \mathbf{fn}_\pi\ x\ \texttt{=>}\ e_0 : \widehat{\tau_x} \overset{\{\pi\}\cup\varphi}{\rightarrow} \widehat{\tau_0}}\ [\mathrm{fn}]$$

- The rules [fn] and [fun] seem to allow to add garbage to annotations:
  - Subeffecting: $\{\pi\} \cup \varphi$ versus simply $\{\pi\}$.
- Subeffecting is necessary to make all programs typable.
- Consider

$$\mathbf{let}\ g\ =\ (\mathbf{fun}_{\mathsf{F}}\ f\ x\ \texttt{=>}\ f\ (\mathbf{fn}_Y\ y\ \texttt{=>}\ y))\ \mathbf{in}\ g\ (\mathbf{fn}_Z\ z\ \texttt{=>}\ z)$$

- Sometimes x arises from Z, sometimes from Y.
- The [app] rule demands that actual and formal argument are the same.
- Thus, it should be labelled with a set containing all the names.
- Different solution: solve the problem in [app].

Non-determinism

- Type deduction systems are non-deterministic in the following sense:

  $[\,] \vdash_{\mathrm{CFA}} \mathbf{fun}_{\mathsf{F}}\ f\ x\ \ =>\ \ f\ (\mathbf{fn}_{Y}\ y\ \ =>\ \ y) : (\widehat{\tau} \overset{\{\mathsf{Y},\mathsf{Z}\}}{\rightarrow} \widehat{\tau}) \overset{\{\mathsf{F}\}}{\rightarrow} (\widehat{\tau} \overset{\varphi}{\rightarrow} \widehat{\tau})$ for all

- $\varphi = \emptyset$ is the most informative of these, the 'least'
- Compare with Hindley-Milner which also allows all instances of a polymorphic type.

Inference algorithm

- In Implementation Of Programming Languages you have seen (a variant) of Algorithm $\mathcal{W}$.
- Returns a substitution mapping type variables to augmented types.
- Augmented types are the usual types, or a type variable.
- These represent (partly) unknown types.
  - Remember: our language is not yet polymorphic
- Basically
  - environment has (partial) types of all identifiers defined at higher levels,
  - we perform a left-to-right traversal of the abstract syntax tree
  - continually 'update' a substitution to reflect newly found facts,
  - using unification of types as a basic operation.
  - Post-order: check subtrees first, then the demands for the node itself.

Unification of types (in short)

- $\mathcal{W}$ uses a standard unification algorithm.
  - two equal constants can be unified.
  - two function types can be unified if arguments and results can be unified pairwise.
  - A type variable α unifies with every other type τ unless α occurs as a non-trivial subtype in τ.
    - $\alpha = \textbf{int} \rightarrow \alpha$ gives rise to infinite type.
    - Otherwise, α will map to τ now in the substitution.
  - Otherwise, unification fails.

Unification of types (in short)

- $\mathcal{W}$ uses a standard unification algorithm.
  - two equal constants can be unified.
  - two function types can be unified if arguments and results can be unified pairwise.
  - A type variable α unifies with every other type τ unless α occurs as a non-trivial subtype in τ.
    - $\alpha = \textbf{int} \rightarrow \alpha$ gives rise to infinite type.
    - Otherwise, α will map to τ now in the substitution.
  - Otherwise, unification fails.

Invariant

Every newly found piece of information is immediately applied to parts of the types still to be unified.

Unification of annotated types

- Problem: unification becomes less straightforward, because we need to include unification of annotations.
- Solution: unify annotated types where annotations are restricted to variables $'1, '2, \ldots$, and
- add special constraints to retrieve what we lost.
- Thus we find which sets of annotations are the same, but not (yet) what they contain.
- Only difference with ordinary unification is how to unify two function types: $\mathcal{U}_{\text{CFA}}(\widehat{\tau_1} \xrightarrow{\beta} \widehat{\tau_2}, \widehat{\tau_1'} \xrightarrow{\beta'} \widehat{\tau_2'}) = \theta_2 \circ \theta_1 \circ \theta_0$ where
  $\theta_0 = [\beta' \mapsto \beta]$
  $\theta_1 = \mathcal{U}_{\text{CFA}}(\theta_0\ \widehat{\tau_1}, \theta_0\ \widehat{\tau_1'})$
  $\theta_2 = \mathcal{U}_{\text{CFA}}(\theta_1(\theta_0\ \widehat{\tau_2}), \theta_1(\theta_0\ \widehat{\tau_2'}))$
- $\mathcal{U}_{\text{CFA}}('a \xrightarrow{'1} 'a, ('b \xrightarrow{'2} 'b) \xrightarrow{'3} 'c)$ gives
  $['3 \mapsto' 1,' a \mapsto ('b \xrightarrow{'2} 'b),' c \mapsto ('b \xrightarrow{'2} 'b)]$

Constraints to make up the difference

- Constraints capture that some arrows are labelled by $\xrightarrow{\{\pi\}\cup\varphi}$.
- The corresponding constraint is $\beta \supseteq \{\pi\}$
  - Here, β is an annotation variable, representing a set of annotations.
- Thus, the inference algorithm only unifies annotation variables.
- More "complicated" checks like $'3 \supseteq \{Z\}, '3 \supseteq \{Y\}$ and $'4 \supseteq \{F\}$ are postponed.
- Afterwards, we can collect these together: $['3 \mapsto \{Z, Y\}, '4 \mapsto \{F\}]$.
- This is the annotation (variable) substitution, the companion to the type (variable) substitution.

Disadvantages

- Deriving the algorithm is quite a bit of work and error prone.
- It has to be done over and over again for each new deduction system.
- However, program analyses have many similarities.
- Algorithms tend to be greedy.
  - They update the substitution, but do not remember which unifications led to the substitution
- Makes it difficult to generate good error messages.
- A different approach is called for.

We move to a different collection of slides