Polyvariant Flow Analysis with Higher-ranked Polymorphic Types and Higher-order Effect Operators

Jurriaan Hage
Joint work with Stefan Holdermans (Vector Fabrics)

Dept. of Information and Computing Sciences
Utrecht University
The Netherlands
E-mail: jur@cs.uu.nl
September 27, 2010
Type based program analysis

- Compilers for strongly typed functional languages need to implement the intrinsic type system of the language.
- In TBPA:
  - Other analyses take advantage of standardised concepts, vocabulary, and implementation.
  - Moreover, the (underlying) types lend structure to the analysis.
Control-flow analysis

▶ Control-flow analysis:

*Determine for every expression, the locations where its value may have been produced.*

▶ In type and effect systems: annotate types with analysis information.

▶ `bool^{ℓ_1,ℓ_2}` describes
  ▶ a boolean value
  ▶ produced at either program location `ℓ_1` or `ℓ_2`.

▶ `(bool^{ℓ_1} → bool^{ℓ_1,ℓ_3})^{ℓ_2}` describes
  ▶ a boolean-valued function produced at location `ℓ_2`
  ▶ that takes a value produced at `ℓ_1` and
  ▶ returns a value produced at `ℓ_1` or `ℓ_3`. 
An imprecise control-flow analysis

\[ h \ f = \textbf{if } f \text{ false}^{\ell_1} \ \textbf{then } f \text{ true}^{\ell_2} \ \textbf{else } \text{false}^{\ell_3} \]

\[ id \ x = x \]

\[ main = h \ id \]

- \( h \) can have type \( (\text{bool}\{\ell_1,\ell_2\} \rightarrow \text{bool}\{\ell_1,\ell_2\}) \rightarrow \text{bool}\{\ell_1,\ell_2,\ell_3\} \)
An imprecise control-flow analysis

\[
h \ f = \text{if } f \text{ false}^{\ell_1} \text{ then } f \text{ true}^{\ell_2} \text{ else } f \text{ false}^{\ell_3}
\]

\[
id \ x = x
\]

\[
\text{main} = h \ id
\]

- \(h\) can have type \((\text{bool}^{\{\ell_1, \ell_2\}} \rightarrow \text{bool}^{\{\ell_1, \ell_2\}}) \rightarrow \text{bool}^{\{\ell_1, \ell_2, \ell_3\}}\)
- \(id\) can have type \(\text{bool}^{\{\ell_1, \ell_2\}} \rightarrow \text{bool}^{\{\ell_1, \ell_2\}}\)
- Unacceptable:
  - analysis is not modular: all uses of \(id\) must be known.
  - other uses of \(id\) poisoned by effect of passing \(id\) to \(h\)
Let-polyvariance to the rescue

\[ id \ x = x \]
\[ h \ f = \text{if } f \text{ false}^{\ell_1} \text{ then } f \text{ true}^{\ell_2} \text{ else } \text{false}^{\ell_3}, \]
\[ \text{main} = h \ id \]

- Let-defined and top-level identifiers identifiers can obtain a context-sensitive, polyvariant type.
- \( h \) can now have type
  \[ \forall \beta. (\text{bool}\{\ell_1, \ell_2\} \to \text{bool}\beta) \to \text{bool}\beta\cup\{\ell_3\} \]
- For \( h \ id \), instantiate \( \beta \) to \( \{\ell_1, \ell_2\} \) to obtain \( \text{bool}\{\ell_1, \ell_2, \ell_3\} \).
- Improvement visible for \( h \ ctrue \) where \( ctrue \ z = \text{true}^{\ell_4} : \text{bool}\{\ell_3, \ell_4\} \) instead of \( \text{bool}\{\ell_1, \ell_2, \ell_3, \ell_4\} \).
- Moreover, type of \( h \) independent of other calls to \( h \).
Let-polyvariance to the rescue

\[
\begin{align*}
  id\ x & = x \\
  h\ f & = \text{if}\ f\ \text{false}^\ell_1\ \text{then}\ f\ \text{true}^\ell_2\ \text{else}\ \text{false}^\ell_3, \\
  \text{main} & = h\ id
\end{align*}
\]

- Let-defined and top-level identifiers identifiers can obtain a context-sensitive, polyvariant type.
- \( h \) can now have type 
  \[
  \forall \beta. (\text{bool}^{\{\ell_1, \ell_2\}} \rightarrow \text{bool}^\beta) \rightarrow \text{bool}^\beta \cup \{\ell_3\}
  \]
- For \( h\ id \), instantiate \( \beta \) to \( \{\ell_1, \ell_2\} \) to obtain \( \text{bool}^{\{\ell_1, \ell_2, \ell_3\}} \).
- Improvement visible for \( h\ ctrue \) where \( ctrue\ z = \text{true}^\ell_4 \): 
  \[
  \text{bool}^{\{\ell_3, \ell_4\}}
  \]
  instead of \( \text{bool}^{\{\ell_1, \ell_2, \ell_3, \ell_4\}} \).
- Moreover, type of \( h \) independent of other calls to \( h \).
- But there is still some poisoning left.
Higher-ranked polyvariance to finish the job

\[ id \, x = x \]
\[ h \, f = \text{if } f \text{ false}\,^\ell_1 \text{ then } f \text{ true}\,^\ell_2 \text{ else false}\,^\ell_3, \]
\[ \text{main} = h \, \text{id} \]

- Type of \( \text{main} \) is \( \text{bool}\{\ell_1, \ell_2, \ell_3\} \)
- But: the value of \( \ell_1 \) never flows to result of \( h \).
- Poisoning still applies to different uses of \( f \) in \( h \).
- Why?
Higher-ranked polyvariance to finish the job

\[
\begin{align*}
id x & = x \\
h \ f & = \text{if } f \ \text{false}^{\ell_1} \ \text{then} f \ \text{true}^{\ell_2} \ \text{else} \ \text{false}^{\ell_3}, \\
\text{main} & = h \ id
\end{align*}
\]

- Type of \textit{main} is \texttt{bool}^{\{\ell_1, \ell_2, \ell_3\}}
- But: the value of \ell_1 never flows to result of \textit{h}.
- Poisoning still applies to different uses of \textit{f} in \textit{h}.
- Because \textit{f} has to be assigned a monovariant type.
- If \textit{f} could have type \( \forall \beta . \texttt{bool}^\beta \rightarrow \texttt{bool}^\beta \), then
  - \( \beta = \{ \ell_1 \} \) for condition: does not propagate to result \textit{h id}
  - \( \beta = \{ \ell_2 \} \) for then-part: propagates to result \textit{h id}
Central question

But can such types, annotated with flow-sets, be inferred?
Central question

But can such types, annotated with flow-sets, be inferred?

- Unassisted inference for higher-ranked polymorphism is undecidable.
Central question

But can such types, annotated with flow-sets, be inferred?

- Unassisted inference for higher-ranked polymorphism is undecidable.
- For control-flow analysis we much prefer not to assist.
Central question

But can such types, annotated with flow-sets, be inferred?

- Unassisted inference for higher-ranked polymorphism is undecidable.
- For control-flow analysis we much prefer not to assist.
- But note that our types are not higher-ranked, only the annotations are.
Our contributions

- Undecidability of inference for higher-order polymorphism on types does not imply undecidability of inference for higher-ranked annotations on (ordinary) types.
  - Inspired by Dussart, Henglein and Mossin
- Type inference algorithm is remarkably like Damas and Milner’s algorithm W.
- Enabling technology of fully flexible types
  - Modularity helps.
- The algorithm computes the best analysis for a given fully flexible type derivation.
The source language

- Simple monomorphic language:
  - Producers: lambda-abstractions and boolean literals
  - Consumers: applications, fix and conditional
  - Variables propagate.

- Each expression is labelled to express its location.

\[

t ::= \ x \ | \ p^\ell \ | \ c^\ell \\
p ::= \ \text{false} \ | \ \text{true} \ | \ \lambda x : \tau . t_1 \\
c ::= \text{if} \ t_1 \ \text{then} \ t_2 \ \text{else} \ t_3 \ | \ t_1 \ t_2 \ | \ \text{fix} \ t_1.
\]
Types and type environments

Types, taken from $\mathbf{Ty}$, are given by

$$\tau ::= \text{bool} \mid \tau_1 \to \tau_2.$$ 

Type environments are given by

$$\Gamma \in \mathbf{TyEnv} = \text{Var} \to_{\text{fin}} \mathbf{Ty}.$$
Control-flow annotations

- Associate with each term $t$ a triple $\hat{\tau}^\psi \& \varphi$
  - $\psi$ is an annotation, a set of labels describing the production sites of the values of $t$.
  - $\varphi$ is an effect value that describes the flow $(\ell, \psi)$ that may result from evaluating $t$: values produced at $\ell_1 \in \psi$ may flow to $\ell$.
  - $\hat{\tau}$ is an annotated type that may contain further annotations:

$$\hat{\tau} ::= \text{bool} \mid \hat{\tau}_1^\psi \varphi \rightarrow \hat{\tau}_2^\psi \mid \ldots$$

- We extend to annotated type environments:

$$\hat{\Gamma} \in \widehat{\text{TyEnv}} = \text{Var} \rightarrow_{\text{fin}} (\widehat{\text{Ty}} \times \text{Ann})$$
Your first fully flexible (annotated) type

\[(\lambda x: \text{bool}. (\text{if } x \text{ then } \text{false}^{\ell_1} \text{ else } \text{true}^{\ell_2})^{\ell_3})^{\ell_4} \]

which may result in

\[(\forall \beta. \text{bool}^{\ell_3, \beta} \xrightarrow{\{(\ell_3, \beta)\}} \text{bool}^{\{\ell_1, \ell_2\}}\{\ell_4\} & \{\} ),\]

- Produces a result constructed at \( \ell_1 \) or \( \ell_2 \).
- A lambda has no effect, and produces itself.
- No need to restrict the annotation of the argument \( x \).
  - Always annotate with an annotation variable.
- For every use of the expression we may choose a different instance for \( \beta \).
- Whatever is passed in is consumed by the conditional, \( \ell_3 \).
Fully flexible types

- Types in which all argument positions are labelled with a quantified annotation variable.
- Our algorithm only infers fully flexible types.
From fully flexible types to effect operators

\[(\lambda f : \text{bool} \rightarrow \text{bool}. (f \text{ true}^5 \text{ true}^6) \text{ true}^7),\]

- To be fully flexible \(f\) has annotation \(\beta_f\).
- All functions passed into \(f\) are fully flexible: give \(f\) type \(\forall \beta. \text{bool}^\beta \rightarrow \text{bool}^\psi\).
- In general, the latent effect of \(f\) and the flow of the result of \(f\) depend on \(\beta\).
- Let’s make that explicit: \(\forall \beta. \text{bool}^\beta \xrightarrow{\varphi_0^\beta} \text{bool}^\psi_0^\beta\)
- Now, \(\varphi_0\) and \(\psi_0\) have become effect operators.
Delivery time for the motivating example

\((\lambda f : \text{bool} \to \text{bool}. \quad \text{if} \ (f \ \text{false}^\ell_1) \ell_2 \ \text{then} \ (f \ \text{true}^\ell_3) \ell_4 \ \text{else} \ \text{false}^\ell_5) \ell_6) \ell_7\)

has fully flexible annotated type

\[ \forall \beta_f. \forall \delta_0. \forall \beta_0. (\forall \beta. \text{bool}^\beta \xrightarrow{\delta_0 \beta} \text{bool}^{(\beta_0 \beta)})^{\beta_f} \]

\[ \{(\ell_2,\beta_f)\} \cup \{(\ell_4,\beta_f)\} \cup \delta_0 \{ \ell_1 \} \cup \delta_0 \{ \ell_3 \} \cup \{(\ell_6,\beta_0 \{ \ell_1 \})\} \xrightarrow{\text{bool}^{(\beta_0 \{ \ell_3 \} \cup \{ \ell_5 \})}} \text{bool}^{(\beta_0 \{ \ell_3 \} \cup \{ \ell_5 \})} \]

Instantiating it to prepare it for receiving \((\lambda x : \text{bool}. \ x)^{\ell_8}\) gives

\[ (\forall \beta. \text{bool}^\beta \xrightarrow{} \text{bool}^\beta) \xrightarrow{\{(\ell_2,\ell_8),(\ell_4,\ell_8),(\ell_6,\ell_1)\}} \text{bool}^{\{ \ell_3,\ell_5 \}} \]

Finally commit to particular choices: \(\beta_f = \{ \ell_8 \}, \delta_0 = \lambda \beta. \{ \} \) and \(\beta_0 = \lambda \beta. \beta.\)
Further remarks

- Analysis of a function is parameterised over the analysis of its argument.
- The relation between those is captured by the annotation/effect operators.
Further remarks

▶ Analysis of a function is parameterised over the analysis of its argument.
▶ The relation between those is captured by the annotation/effect operators.
▶ Changes are not without consequences.
  ▶ Unification of types now needs beta-reduction of expressions over annotations and effects.
  ▶ And a notion of well-typedness (sorting) for such expressions.
The ubiquitous deduction rules

- See the paper.
- Includes
  - definitions for sorting the annotations and effects,
  - definitional equivalence for annotations and effects,
  - definition of type well-formedness,
  - and metatheoretic properties.
The algorithm

- Remarkably like Algorithm W.
- Traverse $t$ to perform “unifications”, and generates constraints that describe the actual flow.
- Solving is a bit more complicated due to beta-reduction for annotations and effects.
- Compared to Algorithm W:
  - Solve occurs for each lambda-abstraction (vs. let-definition)
  - Instantiation performed in the application rule (vs. identifier).
Summary

- Full annotated-type inference in the presence of higher-ranked polymorphism for annotations.
- Allows to parameterise functions over the analysis of their arguments,
- which provides context-sensitivity for lambda-bound identifiers.
Future work, lots of it

- Short term: asymptotic complexity estimate
- Scale to realistic language.
- Apply to other optimising analyses.
- Backwards variant
  - For every value, where may it flow to.
- Extend to validating analyses, e.g., dimension analysis.
- Minimal typing derivations.
- Comparison with let-polyvariance:
  - How much does additional precision buy us practically?
- Comparison with intersection types.
  - Currently available implementations of intersection types?
Thank you for your attention
Algorithm W style constraint based algorithm.
  ▶ \( R(\hat{\Gamma}, t) \) returns \((\hat{\tau}, \beta, \delta, C)\).
  ▶ \( \hat{\tau} \) is the annotated type.
  ▶ \( \beta \) is an annotation variable representing the top-level annotation of \( \hat{\tau} \).
  ▶ \( \delta \) is an effect variable.
  ▶ Constraint set \( C \) to constrain these.
Algorithm - the case of lambda

\[ R(\hat{\Gamma}, (\lambda x : \tau_1 . t_1)^\ell) = \]
\[
\text{let } (\hat{\tau}_1, \chi_i :: s_i) = C(\tau_1, \varepsilon) \\
\beta_1, \beta, \delta \text{ be fresh} \\
(\hat{\tau}_2, \beta_2, \delta_0, C_1) = R(\hat{\Gamma}[x \mapsto (\hat{\tau}_1, \beta_1)], t_1) \\
X = \{ \beta_1 \} \cup \{ \chi_i \} \cup \text{ffv}(\hat{\Gamma}) \\
(\psi_2, \varphi_0) = S(C_1, X, \beta_2, \delta_0) \\
\hat{\tau} = \forall \beta_1 :: \text{ann.} \forall \chi_i :: s_i \cdot \hat{\tau}_1 \beta_1 \xrightarrow{\varphi_0} \hat{\tau}_2 \psi_2 \\
\text{in } (\hat{\tau}, \beta, \delta, \{ \{ \ell \} \subseteq \beta \})
\]

- Completion function \( C \) annotates type \( \tau_1 \) freshly.
- Solve to obtain actual flows before generalisation.
- Solver \( S \) treats active variables as annotation constants.
- Active = free in \( \hat{\Gamma} \) or exposed via \( \hat{\tau} \).
Algorithm - the case of application

\[ R(\hat{\Gamma}, (t_1 \ t_2)\ell) = \]

let \((\hat{\tau}_1, \beta_1, \delta_1, C_1) = R(\hat{\Gamma}, t_1)\)

\((\hat{\tau}_2, \beta_2, \delta_2, C_2) = R(\hat{\Gamma}, t_2)\)

\[ \hat{\tau}'_2 \beta'_2 \xrightarrow{\phi'_0} \hat{\tau}' \psi' = I(\hat{\tau}_1) \]

\[ \theta = [\beta'_2 \mapsto \beta_2] \circ M([], \hat{\tau}_2, \hat{\tau}'_2) \]

\(\beta, \delta\) be fresh

\[ C = \{ \delta_1 \subseteq \delta \} \cup \{ \delta_2 \subseteq \delta \} \cup \{ \{ (\ell, \beta_1) \} \subseteq \delta \} \cup \{ \theta \phi'_0 \subseteq \delta \} \cup \{ \theta \psi' \subseteq \beta \} \cup C_1 \cup C_2 \]

in \((\theta \hat{\tau}', \beta, \delta, C)\)

- \(I\) freshes all annotation variables.
- \(M\) performs matching (one-sided unification).
  - Works because the second argument is the result of \(I\).
Algorithm - the case of application

\[ R(\hat{\Gamma}, (t_1 t_2)^\ell) = \]

let \((\hat{\tau}_1, \beta_1, \delta_1, C_1) = R(\hat{\Gamma}, t_1)\)
\((\hat{\tau}_2, \beta_2, \delta_2, C_2) = R(\hat{\Gamma}, t_2)\)

\(\hat{\tau}'_2 \beta'_2 \xrightarrow{\varphi'_0} \hat{\tau}' \psi' = I(\hat{\tau}_1)\)
\(\theta = [\beta'_2 \mapsto \beta_2] \circ M([], \hat{\tau}_2, \hat{\tau}'_2)\)

\(\beta, \delta\) be fresh
\(C = \{ \delta_1 \subseteq \delta \} \cup \{ \delta_2 \subseteq \delta \} \cup \{ \{(\ell, \beta_1)\} \subseteq \delta \} \cup \{ \theta \varphi'_0 \subseteq \delta \} \cup \{ \theta \psi' \subseteq \beta \} \cup C_1 \cup C_2\)

in \((\theta \hat{\tau}', \beta, \delta, C)\)

- \(\delta_1 \subseteq \delta, \delta_2 \subseteq \delta\): flow of evaluating application includes the effects of evaluating the function and argument.
- \(\theta \varphi'_0 \subseteq \delta\): effect of the body is included too.
- \(\{(\ell, \beta_1)\} \subseteq \delta\): the application consumes the function.
- \(\theta \psi' \subseteq \beta\): body result flows to the application result.