Universiteit Utrecht

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

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.
- $\texttt{bool}^{\{\ell_1,\ell_2\}}$ describes
  - a boolean value
  - produced at either program location ℓ_1 or ℓ_2.
- $(\texttt{bool}^{\{\ell_1\}} \to \texttt{bool}^{\{\ell_1,\ell_3\}})^{\{\ell_2\}}$ 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

$$
\begin{aligned}
h\ f &= \mathbf{if}\, f\ \texttt{false}^{\ell_1}\ \mathbf{then}\, f\ \texttt{true}^{\ell_2}\ \mathbf{else}\ \texttt{false}^{\ell_3} \\
id\ x &= x \\
main &= h\ id
\end{aligned}
$$

- h can have type $(\texttt{bool}^{\{\ell_1,\ell_2\}} \to \texttt{bool}^{\{\ell_1,\ell_2\}}) \to \texttt{bool}^{\{\ell_1,\ell_2,\ell_3\}}$

An imprecise control-flow analysis

$$\begin{aligned} h\ f &= \mathbf{if}\ f\ \texttt{false}^{\ell_1}\ \mathbf{then}\ f\ \texttt{true}^{\ell_2}\ \mathbf{else}\ \texttt{false}^{\ell_3} \\ id\ x &= x \\ main &= h\ id \end{aligned}$$

- h can have type $(\texttt{bool}^{\{\ell_1,\ell_2\}} \to \texttt{bool}^{\{\ell_1,\ell_2\}}) \to \texttt{bool}^{\{\ell_1,\ell_2,\ell_3\}}$
- id can have type $\texttt{bool}^{\{\ell_1,\ell_2\}} \to \texttt{bool}^{\{\ell_1,\ell_2\}}$
- 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

$$
\begin{aligned}
id\;x &= x \\
h\;\;f &= \textbf{if}\, f\; \texttt{false}^{\ell_1}\;\textbf{then}\, f\; \texttt{true}^{\ell_2}\;\textbf{else}\;\texttt{false}^{\ell_3}, \\
main &= h\; id
\end{aligned}
$$

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

Let-polyvariance to the rescue

$$\begin{aligned} id\,x &= x \\ h\ f &= \textbf{if}\, f\ \texttt{false}^{\ell_1}\ \textbf{then}\, f\ \texttt{true}^{\ell_2}\ \textbf{else}\ \texttt{false}^{\ell_3}, \\ main &= h\ id \end{aligned}$$

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

Higher-ranked polyvariance to finish the job

$$
\begin{aligned}
id\,x &= x\\
h\ \ f &= \mathbf{if}\,f\,\mathtt{false}^{\ell_1}\ \mathbf{then}\,f\,\mathtt{true}^{\ell_2}\ \mathbf{else}\,\mathtt{false}^{\ell_3},\\
main &= h\,id
\end{aligned}
$$

- Type of *main* is $\mathtt{bool}^{\{\ell_1,\ell_2,\ell_3\}}$
- But: the value of ℓ_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

$$\begin{aligned} id\, x &= x \\ h\ f &= \mathbf{if}\, f\ \texttt{false}^{\ell_1}\ \mathbf{then}\, f\ \texttt{true}^{\ell_2}\ \mathbf{else}\ \texttt{false}^{\ell_3}, \\ main &= h\ id \end{aligned}$$

- Type of *main* is $\texttt{bool}^{\{\ell_1,\ell_2,\ell_3\}}$
- But: the value of ℓ_1 never flows to result of h.
- Poisoning still applies to different uses of f in h.
- Because f has to be assigned a monovariant type.
- If f could have type $\forall\beta.\texttt{bool}^{\beta} \to \texttt{bool}^{\beta}$, then
  - $\beta = \{\ell_1\}$ for condition: does not propagate to result $h\ id$
  - $\beta = \{\ell_2\}$ for then-part: propagates to result $h\ id$

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.

$$
\begin{array}{rcl}
t & ::= & x \mid p^{\ell} \mid c^{\ell} \\
p & ::= & \texttt{false} \mid \texttt{true} \mid \lambda x : \tau . t_1 \\
c & ::= & \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \mid t_1\ t_2 \mid \textbf{fix } t_1 .
\end{array}
$$

Types and type environments

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

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

Type environments are given by

$$\Gamma \quad \in \quad \mathbf{TyEnv} \quad = \quad \mathbf{Var} \to_{\mathsf{fin}} \mathbf{Ty} \quad .$$

Control-flow annotations

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

$$\widehat{\tau} \quad ::= \quad \texttt{bool} \mid \widehat{\tau}_1^{\,\psi_1} \xrightarrow{\varphi} \widehat{\tau}_2^{\,\psi_2} \mid \ldots$$

- We extend to annotated type environments:

$$\widehat{\Gamma} \quad \in \quad \widehat{\mathbf{TyEnv}} \quad = \quad \mathbf{Var} \to_{\mathsf{fin}} (\widehat{\mathbf{Ty}} \times \mathbf{Ann}) \quad .$$

Your first fully flexible (annotated) type

$$(\lambda x : \texttt{bool}. (\textbf{if}\ x\ \textbf{then}\ \texttt{false}^{\ell_1}\ \textbf{else}\ \texttt{true}^{\ell_2})^{\ell_3})^{\ell_4}.$$

which may result in

$$(\forall \beta . \texttt{bool}^{\beta} \xrightarrow{\{(\ell_3,\beta)\}} \texttt{bool}^{\{\ell_1,\ell_2\}})^{\{\ell_4\}} \,\&\, \{\,\},$$

- Produces a result constructed at ℓ_1 or ℓ_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 β.
- Whatever is passed in is consumed by the conditional, ℓ_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

$$(\lambda f : \mathtt{bool} \to \mathtt{bool}.\,(f\ \mathtt{true}^{\ell_5})^{\ell_6})^{\ell_7},$$

- To be fully flexible f has annotation β_f.
- All functions passed into f are fully flexible: give f type $\forall\beta.\,\mathtt{bool}^{\beta} \xrightarrow{\varphi} \mathtt{bool}^{\psi}$.
- In general, the latent effect of f and the flow of the result of f depend on β.
- Let's make that explicit: $\forall\beta.\,\mathtt{bool}^{\beta} \xrightarrow{\varphi_0\ \beta} \mathtt{bool}^{\psi_0\ \beta}$
- Now, φ_0 and ψ_0 have become effect operators.

Delivery time for the motivating example

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

has fully flexible annotated type

$$\forall \beta_f.\forall \delta_0.\forall \beta_0.(\forall \beta.\texttt{bool}^{\beta} \xrightarrow{\delta_0\ \beta} \texttt{bool}^{(\beta_0\ \beta)})^{\beta_f} \xrightarrow{\{(\ell_2,\beta_f)\}\cup\{(\ell_4,\beta_f)\}\cup\delta_0\ \{\ell_1\}\cup\delta_0\ \{\ell_3\}\cup\{(\ell_6,\beta_0\ \{\ell_1\})\}} \texttt{bool}^{(\beta_0\ \{\ell_3\}\cup\{\ell_5\})},$$

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

$$(\forall \beta.\texttt{bool}^{\beta} \xrightarrow{\{\}} \texttt{bool}^{\beta}) \xrightarrow{\{(\ell_2,\ell_8),(\ell_4,\ell_8),(\ell_6,\ell_1)\}} \texttt{bool}^{\{\ell_3,\ell_5\}}.$$

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

- Algorithm W style constraint based algorithm.
  - $\mathbf{R}(\widehat{\Gamma}, t)$ returns $(\widehat{\tau}, \beta, \delta, C)$.
  - $\widehat{\tau}$ is the annotated type.
  - β is an annotation variable representing the top-level annotation of $\widehat{\tau}$.
  - δ is an effect variable.
  - Constraint set C to constrain these.

Algorithm - the case of lambda

$$
\begin{array}{l}
\mathbf{R}(\widehat{\Gamma}, (\lambda x : \tau_1 . t_1)^{\ell}) = \\
\quad \text{let } (\widehat{\tau}_1, \overline{\chi_i :: s_i}) = \mathbf{C}(\tau_1, \varepsilon) \\
\qquad \beta_1, \beta, \delta \text{ be fresh} \\
\qquad (\widehat{\tau}_2, \beta_2, \delta_0, C_1) = \mathbf{R}(\widehat{\Gamma}[x \mapsto (\widehat{\tau}_1, \beta_1)], t_1) \\
\qquad X = \{\beta_1\} \overline{\cup \{\chi_i\}} \cup \mathit{ffv}(\widehat{\Gamma}) \\
\qquad (\psi_2, \varphi_0) = \mathbf{S}(C_1, X, \beta_2, \delta_0) \\
\qquad \widehat{\tau} = \forall \beta_1 :: \mathrm{ann}. \overline{\forall \chi_i :: s_i .} \, \widehat{\tau}_1{}^{\beta_1} \xrightarrow{\varphi_0} \widehat{\tau}_2{}^{\psi_2} \\
\quad \text{in } (\widehat{\tau}, \beta, \delta, \{\{\ell\} \subseteq \beta\})
\end{array}
$$

- Completion function **C** annotates type τ_1 freshly.
- Solve to obtain actual flows before generalisation.
- Solver **S** treats active variables as annotation constants.
- Active = free in $\widehat{\Gamma}$ or exposed via $\widehat{\tau}$.

Algorithm - the case of application

$$
\begin{aligned}
&\mathbf{R}(\widehat{\Gamma}, (t_1\ t_2)^{\ell}) = \\
&\quad \text{let } (\widehat{\tau}_1, \beta_1, \delta_1, C_1) = \mathbf{R}(\widehat{\Gamma}, t_1) \\
&\qquad\quad (\widehat{\tau}_2, \beta_2, \delta_2, C_2) = \mathbf{R}(\widehat{\Gamma}, t_2) \\
&\qquad\quad \widehat{\tau}_2'^{\beta_2'} \xrightarrow{\varphi_0'} \widehat{\tau}'^{\psi'} = \mathbf{I}(\widehat{\tau}_1) \\
&\qquad\quad \theta = [\beta_2' \mapsto \beta_2] \circ \mathbf{M}([\,], \widehat{\tau}_2, \widehat{\tau}_2') \\
&\qquad\quad \beta, \delta \text{ be fresh} \\
&\qquad\quad C = \{\delta_1 \subseteq \delta\} \cup \{\delta_2 \subseteq \delta\} \cup \{\{(\ell, \beta_1)\} \subseteq \delta\} \cup \{\theta\ \varphi_0' \subseteq \delta\} \cup \\
&\qquad\qquad\quad \{\theta\ \psi' \subseteq \beta\} \cup C_1 \cup C_2 \\
&\quad \text{in } (\theta\ \widehat{\tau}', \beta, \delta, C)
\end{aligned}
$$

- **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

$$
\begin{aligned}
&\mathbf{R}(\widehat{\Gamma}, (t_1\ t_2)^{\ell}) = \\
&\quad \text{let } (\widehat{\tau}_1, \beta_1, \delta_1, C_1) = \mathbf{R}(\widehat{\Gamma}, t_1) \\
&\quad\quad\ \ (\widehat{\tau}_2, \beta_2, \delta_2, C_2) = \mathbf{R}(\widehat{\Gamma}, t_2) \\
&\quad\quad\ \ \widehat{\tau}_2'^{\beta_2'} \xrightarrow{\varphi_0'} \widehat{\tau}'^{\psi'} = \mathbf{I}(\widehat{\tau}_1) \\
&\quad\quad\ \ \theta = [\beta_2' \mapsto \beta_2] \circ \mathbf{M}([\,], \widehat{\tau}_2, \widehat{\tau}_2') \\
&\quad\quad\ \ \beta, \delta \text{ be fresh} \\
&\quad\quad\ \ C = \{\delta_1 \subseteq \delta\} \cup \{\delta_2 \subseteq \delta\} \cup \{\{(\ell, \beta_1)\} \subseteq \delta\} \cup \{\theta\ \varphi_0' \subseteq \delta\} \cup \\
&\quad\quad\quad\quad \{\theta\ \psi' \subseteq \beta\} \cup C_1 \cup C_2 \\
&\quad \text{in } (\theta\ \widehat{\tau}', \beta, \delta, C)
\end{aligned}
$$

- $\delta_1 \subseteq \delta, \delta_2 \subseteq \delta$: 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.