Universiteit Utrecht

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

Program transformation

Ivo Gabe de Wolff
Some parts based on slides of Stefan Holdermans

June 20, 2022

Recap

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.

- uniqueness analysis: unique at use site, for in-place updates
- sharing analysis: unique at declaration site, for thunk creation

Today

- How do we transform this program?
  $\textbf{let}\ xs = [1, 2]\ \textbf{in}\ sum\ (map\ (+1)\ xs)\ \textbf{ni}$

Today

- How do we transform this program?
  $\textbf{let}\ xs = [1, 2]\ \textbf{in}\ sum\ (map\ (+1)\ xs)\ \textbf{ni}$

- Desired output:
  $\textbf{let}\ xs =^{1}\ [1, 2]\ \textbf{in}\ sum\ (map\_inplace\ (+1)\ xs)\ \textbf{ni}$

Today

- How do we transform this program?
  $\textbf{let}\ xs = [1,2]\ \textbf{in}\ sum\ (map\ (+1)\ xs)\ \textbf{ni}$
- Desired output:
  $\textbf{let}\ xs =^{1}\ [1,2]\ \textbf{in}\ sum\ (map\_inplace\ (+1)\ xs)\ \textbf{ni}$
- Algorithm W:
  Nat

Today

- How do we transform this program?
  $\mathbf{let}\ xs = [1,2]\ \mathbf{in}\ sum\ (map\ (+1)\ xs)\ \mathbf{ni}$

- Desired output:
  $\mathbf{let}\ xs =^1\ [1,2]\ \mathbf{in}\ sum\ (map\_inplace\ (+1)\ xs)\ \mathbf{ni}$

- Algorithm W:
  Nat

- How can we preserve the required information?

Typed terms

- To simplify things, we consider the underlying type system.
- We annotate each binding with a type.

t	$\in$	$\mathbf{Tm}$	terms
$\widehat{t}$	$\in$	$\mathbf{TypedTm}$	typed terms

$$
\begin{array}{lll}
t & ::= & \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} \\
 & \mid & \lambda x.\, t_1 \mid \cdots \\
\widehat{t} & ::= & \mathbf{let}\ x : \sigma = \widehat{t}_1\ \mathbf{in}\ \widehat{t}_2\ \mathbf{ni} \\
 & \mid & \lambda x : \tau .\, \widehat{t}_1 \mid \cdots
\end{array}
$$

Recap: Algorithm W

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

Later extended with annotation variables and constraints

Universiteit Utrecht

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

Idea 1: Proof trees

- Shows how typing rules are applied.
- Contains types of subterms.

Idea 1: Proof trees

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

$$\left[\!\!\left[\frac{\mathcal{T}_1 :: \Gamma \vdash_{\mathsf{UL}} t_1 : \sigma_1 \qquad \mathcal{T}_2 :: \Gamma[x \mapsto^{\cdot} \sigma_1] \vdash_{\mathsf{UL}} t_2 : \tau}{\Gamma \vdash_{\mathsf{UL}} \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} : \tau}\right]\!\!\right] = \mathbf{let}\ x : \sigma_1 =^{\cdot} [\![\mathcal{T}_1]\!]\ \mathbf{in}\ [\![\mathcal{T}_2]\!]\ \mathbf{ni}$$

Idea 1: Proof trees

- We can proof that each translated program evaluates to the value of the original program (meta theory).
- But how do we construct a proof tree?
  That is actually a similar problem as constructing the transformed (typed) terms.

Idea 2: Map variable names to types

- Algorithm W gives a type and a *substitution*.
- $\mathcal{W} : \mathbf{TyEnv} \times \mathbf{Tm} \rightarrow \mathbf{Ty} \times \mathbf{TySubst}$
- If we know the type variable (or type) that was assigned to a variable, then we can find its type.
- We can construct a mapping from variable names to type variables in $\mathcal{W}$,
- if we have globally unique variable names.

Variable names

How should we represent identifiers?

- Named variables (String or number)
  Seems easy here, but rewrite rules as beta reduction become harder.
- Debruijn indices
  Number of binders between declaration and use
- Debruijn level
  Number of binders between declaration and root

Always use named variables in a pretty printer!

Debruijn indices

- Debruijn indices can be used for a typed environment.
- Environment becomes a type-level list.
- Parameterize the expression data type over the environment.
- Debruijn indices index into that list.

Idea 3: Call W on subterms

$$
\begin{aligned}
&\mathit{transform} : \mathbf{TyEnv} \to \mathbf{Tm} \to \mathbf{TypedTm}\\
&\mathit{transform}\ \Gamma\ (\mathbf{let}\ x = \mathit{bnd}\ \mathbf{in}\ \mathit{body}\ \mathbf{ni}) =\\
&\qquad \mathbf{let}\ x : \sigma = (\mathit{transform}\ \Gamma\ \mathit{bnd})\\
&\qquad \mathbf{in}\ (\mathit{transform}\ \Gamma_1\ \mathit{body})\ \mathbf{ni}\\
&\quad \mathbf{where}\\
&\qquad (\tau, \_) = \mathcal{W}\ (\Gamma, \mathit{bnd})\\
&\qquad \sigma = \mathit{generalise}(\Gamma, \tau)\\
&\qquad \Gamma_1 = \Gamma[x \mapsto \sigma]
\end{aligned}
$$

Idea 3: Call W on subterms

$$
\begin{array}{l}
\mathit{transform} : \mathbf{TyEnv} \to \mathbf{Tm} \to \mathbf{TypedTm} \\
\mathit{transform}\ \Gamma\ (\mathbf{let}\ x = \mathit{bnd}\ \mathbf{in}\ \mathit{body}\ \mathbf{ni}) = \\
\qquad \mathbf{let}\ x : \sigma = (\mathit{transform}\ \Gamma\ \mathit{bnd}) \\
\qquad \mathbf{in}\ (\mathit{transform}\ \Gamma_1\ \mathit{body})\ \mathbf{ni} \\
\quad \mathbf{where} \\
\qquad (\tau, \_) = \mathcal{W}\ (\Gamma, \mathit{bnd}) \\
\qquad \sigma = \mathit{generalise}(\Gamma, \tau) \\
\qquad \Gamma_1 = \Gamma[x \mapsto \sigma]
\end{array}
$$

What are the problems?

Tupling

- $\mathit{transform}$ and $\mathcal{W}$ both recurse on $\mathbf{Tm}$.
- $\mathcal{W}$ may be called many times on some subterms.
- Worst case: quadratic instead of linear.

Tupling

Integrate W in $\mathit{transform}$:

$$
\begin{aligned}
&\mathit{transform} : \mathbf{TyEnv} \rightarrow \mathbf{Tm} \rightarrow \mathbf{Ty} \times \mathbf{TySubst} \times \mathbf{TypedTm}\\
&\mathit{transform}\ \Gamma\ (\mathbf{let}\ x = \mathit{bnd}\ \mathbf{in}\ \mathit{body}\ \mathbf{ni}) =\\
&\qquad (\tau_2\\
&\qquad ,\theta_2 \circ \theta_1\\
&\qquad ,\mathbf{let}\ x : \sigma_1 = \mathit{bnd}'\ \mathbf{in}\ \mathit{body}'\ \mathbf{ni}\\
&\qquad)\\
&\quad \mathbf{where}\\
&\qquad (\tau_1, \theta_1, \mathit{bnd}') = \mathit{transform}\ \Gamma\ \mathit{bnd}\\
&\qquad \sigma_1 = \mathit{generalise}(\theta_1\ \Gamma, \tau_1)\\
&\qquad \Gamma_1 = (\theta_1\ \Gamma)[x \mapsto \sigma_1]\\
&\qquad (\tau_2, \theta_2, \mathit{body}') = \mathit{transform}\ \Gamma_1\ \mathit{body}
\end{aligned}
$$

Substitutions

- When analyzing $body$, we may find substitutions on type variables used in bnd.
- Can we apply the substitution on a term?
- For simple analysis that might be possible, but still undecirable for performance.

Tupling

Return term as a function taking a substitution:

$$
\begin{array}{l}
\mathit{transform} : \mathbf{TyEnv} \to \mathbf{Tm} \\
\quad \to \mathbf{Ty} \times \mathbf{TySubst} \times (\mathbf{TySubst} \to \mathbf{TypedTm}) \\
\mathit{transform}\ \Gamma\ (\mathbf{let}\ x = \mathit{bnd}\ \mathbf{in}\ \mathit{body}\ \mathbf{ni}) = \\
\qquad (\tau_2 \\
\qquad , \theta_2 \circ \theta_1 \\
\qquad , \lambda\theta \to \mathbf{let}\ x : \theta\ \tau_1 = (\mathit{bnd}'\ (\theta . \theta_2)\ \mathbf{in}\ (\mathit{body}'\ \theta)\ \mathbf{ni}) \\
\qquad) \\
\quad \mathbf{where} \\
\qquad (\tau_1, \theta_1, \mathit{bnd}') = \mathit{transform}\ \Gamma\ \mathit{bnd} \\
\qquad (\tau_2, \theta_2, \mathit{body}') = \mathit{transform}\ (\theta_1\ \Gamma)\ \mathit{body}
\end{array}
$$

Type variables

$$
\begin{aligned}
&\mathit{transform} : \mathbf{TyEnv} \to \mathbf{Tm} \\
&\quad \to \mathbf{Ty} \times \mathbf{TySubst} \times (\mathbf{TySubst} \to \mathbf{TypedTm})
\end{aligned}
$$

- This signature doens't allow you to create fresh type variables.
- You could use the State monad to keep track of the next fresh index.

Comparison with Attribute Grammars

- We're now manually doing a multi-pass.
- The first pass returns a function to perform the second pass.
- An Attribute Grammar system would do that for us, though it is not always possible/preferred to integrate that in a project.

Next time

- We now know how to convert a **Tm** to a **TypedTm**.
- Do we need to duplicate the data type **Tm** to define **TypedTm**?
- Do we need to reimplement all utility functions on **Tm** for **TypedTm**?