Term Graph Representations for Cyclic Lambda-Terms

Clemens Grabmayer  Jan Rochel

Department of Philosophy
Utrecht University
The Netherlands
clemens@phil.uu.nl

Department of Computing Sciences
Utrecht University
The Netherlands
jan@rochel.info

23rd March 2013
Area of Research

- functional programming languages
- untyped $\lambda$-calculus with \texttt{letrec} ($\lambda_{\text{letrec}}$)
- sharing in $\lambda_{\text{letrec}}$
Motivation

Example
\[
\lambda x. \text{letrec } f = x f \text{ in } f \quad \rightarrow \nabla \quad \lambda x. x (x (x \ldots))
\]

Example
\[
\lambda x. \text{letrec } f = x (x f) \text{ in } f \quad \rightarrow \nabla \quad \lambda x. x (x (x \ldots))
\]

Efficient methods for determining
- whether two \(\lambda_{\text{letrec}}\)-terms have the same unfolding
- the maximally shared form of a \(\lambda_{\text{letrec}}\)-term

On the theoretical side:
- a notion of maximal sharing
- a sharing preorder
To reason about unfolding equivalence and sharing we want to abstract over:

- order of \texttt{letrec}-bindings
- position of binding groups
- names of recursion- and \(\lambda\)-variables

\(\implies\) work with graph representations of \(\lambda_{\text{letrec}}\)-terms that faithfully represent the sharing that occurs in a \(\lambda_{\text{letrec}}\)-term.

Bisimulation \(\sim\) unfolding equivalence.
Functional bisimulation \(\sim\) compactification.
$G$ computes the graph representation of a term. $R$ is a ‘readback’. $G$ is a left inverse of $R$: $G \circ R = id$
Computing the maximally shared form of a term

\[ \lambda x. \text{letrec } f = x \ (x \ f) \ \text{in } f \leq \lambda x. \text{letrec } f = x \ f \ \text{in } f \]
We study various graph formalisms and show how they relate:

- $\lambda$-higher-order-term-graphs: first-order term graphs + a scope function (based on ‘higher-order term graphs’ [Blom, 2001])
- Abstraction-prefix based $\lambda$-higher-order-term-graphs first-order term graphs + an abstraction prefix function (motivated by [G&R, 2012])
- $\lambda$-term-graphs with scope delimiters: plain first-order term graphs with scope delimiter vertices

We want to establish correspondences between the formalisms to show that one can be implemented in terms of the other.
\[
\text{letrec} \quad f = \lambda x. (\lambda y. (y (\times g))) (\lambda z. g f) \quad \text{in } f \\
g = \lambda i. i
\]

The scope function assigns to each abstraction node a set of nodes.
letrec

\[
\begin{align*}
  f &= \lambda x. (\lambda y. (y (x g))) (\lambda z. g f) \\
  g &= \lambda i. i
\end{align*}
\]

in f
An isomorphic correspondence

$A$ preserves the sharing order:

- $G_1 \rightarrow G_2 \quad \Longrightarrow \quad A(G_1) \rightarrow A(G_2)$
An isomorphic correspondence

A preserves and reflects the sharing order:

\[ G_1 \Rightarrow G_2 \iff A(G_1) \Rightarrow A(G_2) \]
An isomorphic correspondence

\[ \lambda \]

\[ A \]

\[ A \] preserves and reflects the sharing order:

- \( G_1 \Rightarrow G_2 \iff A(G_1) \Rightarrow A(G_2) \)
- \( A^{-1}(G_1) \Rightarrow A^{-1}(G_2) \iff G_1 \Rightarrow G_2 \)
Scopes are important

\[
\text{letrec } f = \lambda x. x (\lambda y. x f) \text{ in } f
\]

If scoping information is omitted, bisimulation would relate terms with different unfoldings.
\( \lambda \text{-term-graphs with scope delimiters} \)

\[
\text{letrec } \quad f = \lambda x. (\lambda y. (y (x g))) (\lambda z. g f) \\
g = \lambda i. i
\]

in \(f\)
A correspondence

\[
\begin{align*}
\text{G} & \quad \text{and} \quad \text{G} \\
\text{also preserve and reflect the sharing order}
\end{align*}
\]
The correspondence is not an isomorphism:

\[ \lambda \]
\[ @ \]
\[ \lambda \]
\[ S \]
\[ 0 \]

\[ \mathcal{G} \]
\[ \lambda \]
\[ @ \]
\[ \lambda \]
\[ S \]
\[ 0 \]

\[ \lambda^{(v_1)} \]
\[ @^{(v_1)} \]
\[ \lambda^{(v_1)} \]
\[ S \]
\[ S \]
\[ 0 \]

\[ \lambda \]
\[ @ \]
\[ \lambda \]
\[ S \]
\[ 0 \]

\[ \mathcal{G} : G \]
\[ \rightarrow \]
\[ \mathcal{G} : G \]
\[ \leftarrow \]

\[ \mathcal{G} \] is not injective because of S-sharing  \[ \implies \] \[ \mathcal{G} \neq G^{-1} \]

But:

- \[ \mathcal{G} \circ G = id \]  
  (\[ \mathcal{G} \] is a left-inverse of \[ G \])
- \[ G \circ \mathcal{G}(g) \xrightarrow{S} g \]  
  (\[ G \] is a left-inverse of \[ \mathcal{G} \] up to S-sharing)
Correspondences yield implementations

\[ \lambda \text{-higher-order-term-graphs} \]

\[ \lambda \text{-term-graphs with scope delimiters} \]

\[ \text{abstraction-prefix based graphs} \]
Closedness-Issues: variable backlinks

\[ \lambda \rightarrow \lambda \]

\[ \lambda \rightarrow \lambda \]

\[ \lambda \rightarrow \lambda \]

\[ \lambda \rightarrow \lambda \]

\[ \lambda \rightarrow \lambda \]
Closedness-Issues: eager scope-closure
Closedness-Issues: eager scope-closure
Closedness-Issues:  $S$ – backlinks
$\lambda$-term-graphs are closed under unrestricted functional bisimulation if they have:

- scope delimiters
- delimiter backlinks
- variable backlinks
- eager placement of delimiters
What have we gained?

- **Practical:** Implementation of maximal sharing through bisimulation collapse

- **Theoretical:** Transfer of properties known for first-order term graphs to the higher-order term graphs
  - E.g. for all graphs $g$ from the classes:
    - $\lambda$-higher-order-term-graphs
    - abstraction-prefix based $\lambda$-higher-order-term-graphs
  it holds: $\langle [g]_\leftrightarrow, \rightarrow \rangle$ is a complete lattice.

- **Easy generalisation:** e.g. to higher-order term graphs representing iCRS-terms (instead of infinite $\lambda$-terms).