Normalizing Structured Graphs (ongoing work)

Eijiro Sumii
Joint work with Kazuma Kikuchi

Tohoku University
Sendai, Japan

What are Structured Graphs?

- Proposed by [Oliveira-Cook ICFP'12]
- Uses **recursive bindings** and **PHOAS** (parameterized higher-order abstract syntax) to **represent trees with sharing and cycles**

E.g. (in OCaml-like syntax - sorry!)

- let rec x = a[x] in x
- let rec y = c[] in b[y,y]

(What is PHOAS?)

- **Actually, it doesn't quite matter for this talk**
- **Anyway, it is a way of representing bindings of the object language by that of the meta language**

E.g.

```
 type 'a lam =
    Var of 'a
  | Lam of 'a -> 'a lam
  | App of 'a lam * 'a lam
```

General Definition

```
type α sgraph =
  Node of label * α sgraph list
| LetRec of (α list -> α sgraph list) * α sgraph
      (* PHOAS to represent
            (mutually) recursive bindings *)
| Var of α
```

- For readability, we use the ordinary syntax **let rec … in …** instead of LetRec, and also write $l[g_1,\ldots,g_n]$ for Node(l,[g1,…,gn])

What is the Problem?

- The structured graph representations are **not unique** (in fact, some are **redundant**)

E.g.

- let rec x = d[x] in a[]
  → a[]
- let rec x = d[] in a[x]
  → a[d[]]
- let rec x = b[x] in a[x]
  → a[let rec x = b[x] in x]

Our Work:

Normalizing Structured Graphs

- A set of rewriting rules that are confluent and terminating
- Trickier than you might think!

Our settings

- Nodes are identified by labels
- Graphs are rooted

  let rec x = a[y] and y = b[x] in x

  ≠ let rec x = a[y] and y = b[x] in y
- Children of nodes are ordered

  a[b[], c[]]

  ≠ a[c[], b[]]
- Graphs are identified up to bisimilarity

Graph Bisimilarity

- $l[g_1,\ldots,g_n]$ and $l'[g'_1,\ldots,g'_{n'}]$ are bisimilar if $l=l'$, $n=n'$, and each g_i and g'_i are bisimilar
- let rec $x_1,\ldots,x_n = g_1,\ldots,g_n$ in g and g' are bisimilar if $[h_1,\ldots,h_n/x_1,\ldots,x_n]g$ (where each h_i = let rec $x_1,\ldots,x_n = g_1,\ldots,g_n$ in g_i) and g' are bisimilar
  - Ditto for the inverse
  - N.B. Taking h_i = let rec $x_1,\ldots,x_n = g_1,\ldots,g_n$ in x_i is <u>unsound</u>!

Reduction 1/3: Removing

- REMOVE-REC:

  let rec ~x=~s in t → let rec ~y=~u in t

  if {~x=~s} = {~y=~u} ⊎ {~z=~v}

  and {~z=~v} ≠ ∅ and ~z ∉ FV(~u, t)

  - ~x=~s stands for sequence like x_1=s_1,...,x_n=s_n
  - ~x stands for x_1,...,x_n and ~s for s_1,...,s_n etc.
- ERASE-REC: let rec in s → s
- FUSE-REC: let rec ~x=~u in (let rec ~y=~v in s)

  → let rec ~x,~y=~u,~v in s

  - Tricky for termination proof!

Reduction 2/3: Dropping

- DROP-REC-CHILD:

  let rec ~x=~s in l[t_1,…,t_i,…,t_n] →

  let rec ~y=~u in l[t_1,…,(let rec ~z=~v in t_i),…,t_n]

  if {~x=~s} = {~y=~u} ⊎ {~z=~v}

  and {~z=~v} ≠ ∅ and ~z ∉ FV(~u, ~t\\t_i)

- DROP-REC-DEF:

  let rec ~x=~s in t →

  let rec y_1=u_1,…,(let rec ~z=~v in u_i),…,y_n=u_n in t

  if {~x=~s} = {~y=~u} ⊎ {~z=~v}

  and {~z=~v} ≠ ∅ and ~z ∉ FV(~u\\u_i, t)

Reduction 3/3: Inlining

- INILNE-REC-BODY:
  let rec ~x=~s in t →
  let rec $\sim x = \sim s \setminus x_i = s_i$ in $[s_i/x_i]t$
  if $x_i \notin FV(\sim s)$
  and x_i appears at most once in t
- INLINE-REC-DEF:
  let rec ~x=~s in t →
  let rec $(\sim x = \sim s \setminus x_i = s_i \setminus x_j = s_j)$, $x_j = [s_i/x_i]s_j$ in t
  if $x_i \notin FV(\sim s \setminus s_j, t)$
  and x_i appears at most once in s_j

Reduction 4/3: Precongruence

- **Usual precongruence rules CONG-REC-BODY, CONG-REC-DEF, and CONG-CHILD**

Termination (tricky!)

"Inductive lexical order" on Counts(t) = (Bind(t), TopBind(t), TopRec(t), Sub(t)) where

- Bind(t) counts the number of all = in t
  - for REMOVE-REC and INLINE-REC-*
- TopBind(t) counts only "top-level" = (neither under l[] nor on the rhs of =)
  - for DROP-REC-*
- TopRec(t) counts top-level "let rec" (not =)
  - for ERASE-REC and FUSE-REC
- Sub(t) recurses: Sub(x) = ()

  Sub(l[t_1,...,t_n]) = (Counts(t_1),...,Counts(t_n))

  Sub(let rec x_1=s_1,...,x_n=s_n in t) = (Counts(s_1),...,Counts(s_n),Counts(t))
  - for CONG-*

Why the quadruple?

	Bind	TopBind	TopRec	Sub
REMOVE-REC	$<$	$<$	$=$	
ERASE-REC	$=$	$=$	$<$	
FUSE-REC	$=$	$=$	$<$	
DROP-REC-CHILD	$=$	$<$	$=$	
DROP-REC-DEF	$=$	$<$	$=$	
INLINE-REC-BODY	$<$	?	?	
INLINE-REC-DEF	$<$	$<$	$=$	
CONG-*	$\leqq$	$\leqq$	$\leqq$	$<$

Confluence (relatively easy)

Lemma: All critical pairs are locally confluent

Proof: "Just" careful case analyses on pairs of the reduction rules, with set calculations of the side conditions on free variables

Preservation of Bisimilarity

Conjecture: if $s \rightarrow t$, then s and t are bisimilar

Proof: To do

An Open Question

- **Any of the reduction rules are not specific to (structured) graphs**
- **Are they applicable to (or, better, useful for) (mutually) recursive programs in general?**