A Relational Framework for Higher-Order Shape Analysis

Gowtham Kaki

Joint work with Suresh Jagannathan

PURDUE UNIVERSITY

Overview

- A verification framework for (higher-order) morphisms algebraic datatypes.
  - ★morphisms : maps between algebras (eg: lists, trees etc).
- Verification framework = specification language + verification procedure.
  - ★Programmers write invariants over data structure morphisms.
  - ★ Verification procedure makes use of first-order provers (off-the-shelf SMT solvers) to automatically discharge assertions.
- Invariants over morphisms are, most often, assertions over shapes of initial and final algebras (datatypes).
- Overarching goal : automatically verify safety properties of program transformations.

Introduction

- Algebraic datatypes reflect the inductive structure of semantic objects they model.
  - ★ A list is either empty or pair of an element and another list

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

  - ★ A tree is a either leaf with an element, or a branch of two trees and an element

```
datatype 'a tree = Leaf of 'a
                 | Branch of 'a tree * 'a * 'a tree
```

  - ★ A lambda expr is either application of one lambda expr over other or ...

```
datatype expr = Var of id
              | App of expr * expr
              | Abs of id * expr
```

Introduction

- Most often, constructors can be perceived as terms defining simple relations that are, nonetheless, semantically relevant.

  - ★ In l = Cons (x0,xs), relations that can be immediately identified:
    - ✦ {(l,x0)} - *head*
    - ✦ {(l,xs)} - *tail*

  - ★ Similarly, in t = Branch (t0,x,t1):
    - ✦ {(t,x)} - *root*
    - ✦ {(t,t0)} - *left-child*
    - ✦ {(t,t1)} - *right-child*
    - ✦ {(t,t1), (t,t0)} - *sub-tree*

Introduction

- Let us lift the *head* relation to an inductive definition (*R*)

  ★ $$\forall l, xs : \text{`}a\ list, x, x' : \text{`}a,$$
  $$head(l, x) \Rightarrow R(l, x)$$
  $$tail(l, xs) \Rightarrow R(xs, x') \Rightarrow R(l, x')$$

- For l = Cons (x0, Cons(x1,(Cons x2,Nil))),

  ★{(l,x0), (l,x1), (l,x2)} $\subseteq$ *R*

- Define *R*(l) = $\pi_{\#2}(\sigma_{\#1=l}(R))$, where π and σ are selection and projection operators from relational algebra.

  ✦ *R*(l) = {x0, x1, x2}

- *R* is actually the *membership* relation over lists.

  ★*R = Rmem*

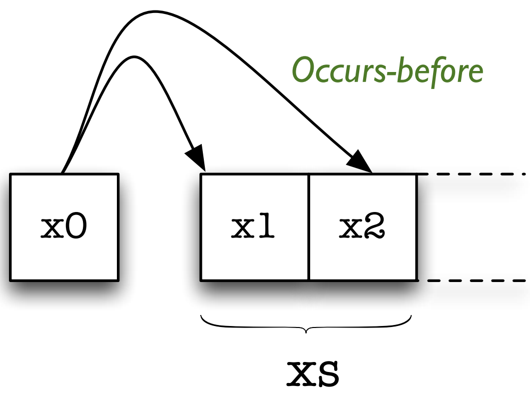
Introduction

- Similarly, define *Rob* relation as

$$\forall l, xs : \text{`}a\ list, x, x' : \text{`}a,$$
$$head(l, x) \Rightarrow tail(l, xs) \Rightarrow R_{mem}(xs, x') \Rightarrow R_{ob}(l, x, x')$$

- and its inductive version, *Rob** as

$$\forall l, xs : \text{`}alist, x, x' : \text{`}a,$$
$$R_{ob}(l, x, x') \Rightarrow R^*_{ob}(l, x, x')$$
$$R^*_{ob}(xs, x, x') \Rightarrow R^*_{ob}(l, x, x')$$



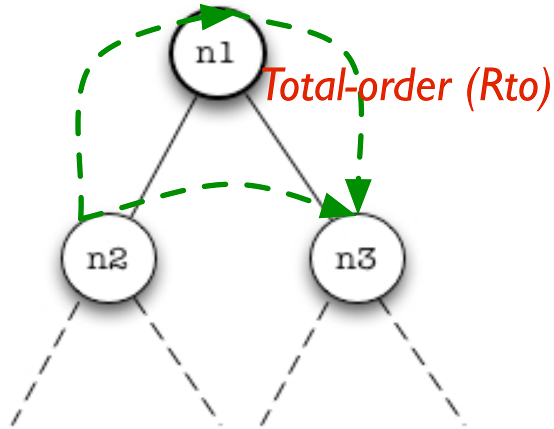
- For l = Cons (x0, Cons(x1,(Cons x2,Nil))),

  ★*Rob**(l) = {(x0,x1), (x0,x2), (x1,x2)}

- *Rob* is *Occurs-before* relation over the list!
- *Rob** succinctly captures the notion of order in the list

Introduction

- Relations of similar flavour can be defined inductively over trees:
  - ★ tree-membership (*Rtm*)
  - ★ pre/post-order (*Rpre*/*Rpost*)
  - ★ total-order (*Rto*)
  - ★ depth-first-order (*Rdfo*)



- Over any inductive structure
  - ★ Control dependence relation between expressions of AST:

    datatype ast = ... | If of expr * expr * expr | ...

    Control Dependence

    `If (e0,e1,e2)`

  - ★ Scope/Visibility of lambda-bound variables

    Scope/Visibility

    $\lambda x.\ \underbrace{(\lambda z.\ z\ x)\ x}$

Structural Relations

- We call such relations as *structural relations*
  - ★ Defined inductively over the structure of algebraic datatypes (data structures).
- Pleasant properties:
  - ★ Succintly capture shape properties of algebraic datatypes
  - ★ Can be encoded as sets of tuples, which is a decidable theory in SMT
  - ★ Inductive structure of definitions match inductive structure of morphisms over algebraic datatypes making them highly amenable for automatic verification.
- Useful tool to reason about correctness of morphisms over algebraic datatypes
  - ★ Data structure operations
  - ★ Program transformations over abstract syntax trees

Example - rev

- list reverse function:

```
fun rev l = case l of
      [] => []
    | x::xs => concat (rev(xs),[x])
```

- What is its specification ?
  - Dependent type with under-specification :

$$rev : \{l : int\ list\} \rightarrow \{l' : int\ list \mid len(l') = len(l)\}$$

  - Specification makes use of structurally recursive *len* function that maps lists to integer domain.
  - Type checking is decidable as logic of algebraic datatypes with abstraction functions to decidable domains is decidable (Suter et. al., POPL'10)
  - Implemented as type checker in Kawaguchi et. al., PLDI'09

Example - rev

- list reverse function:

```
fun rev l = case l of
      [] => []
    | x::xs => concat (rev(xs),[x])
```

- What is its specification ?
  - Dependent type with full-functional specification :

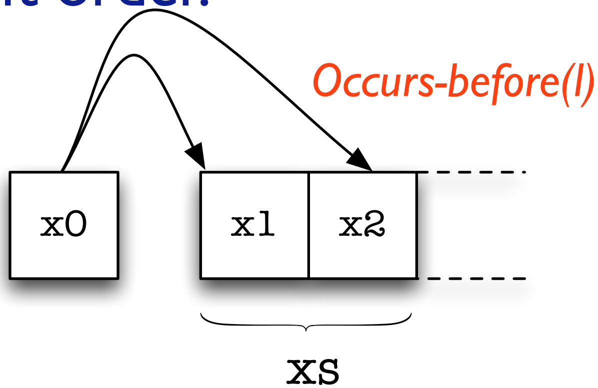
$$rev : \{l : int\ list\} \rightarrow \{l' : int\ list \mid l' = rev(l)\}$$

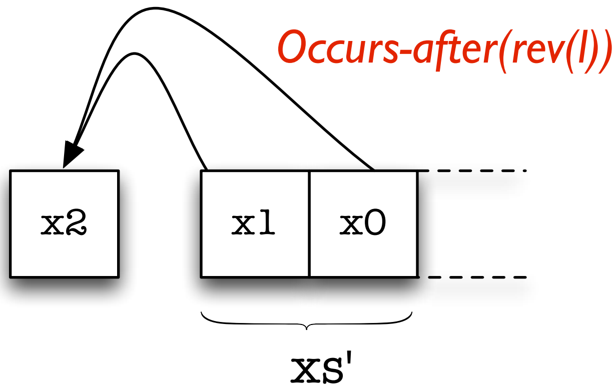
★ *rev* in type refinement is an abstraction function from concrete lists to abstract lists

★ Self-referential. Tautology.

A Relational Spec for Reverse

- Recall that *Occurs-before* relation captures the notion of left-to-right order in a list. Similarly, an *Occurs-after* relation captures the right-to-left order.





l = [x0, x1, x2] rev(l) = [x2, x1, x0]

$$R_{ob}^{*}(l) = \{(x0,x1), (x0,x2), (x1,x2)\}$$

$$R_{oa}^{*}(rev(l)) = \{(x0,x1), (x0,x2), (x1,x2)\}$$

$$\texttt{rev} : \{\texttt{l} : \texttt{'a}\ \text{list}\} \longrightarrow \{\nu : \texttt{'a}\ \text{list} \mid R^{*}_{ob}(\texttt{l}) = R^{*}_{oa}(\nu)\}$$

Verifying rev - Annotations

- Define Structural Relations:

  relation Rhd (x::xs) = {(x)}

  relation Rmem = Rhd*

  relation Rob (x::xs) = {(x)} X Rmem(xs)

  relation Roa (x::xs) = Rmem(xs) X {(x)}

- Write assertions

$$\texttt{rev} : \{\texttt{l} : \texttt{'a list}\} \longrightarrow \{\nu : \texttt{'a list} \mid R_{mem}(\nu) = R_{mem}(\texttt{l}) \;\wedge\; R^*_{oa}(\nu) = R^*_{ob}(\texttt{l})\ \}$$

$$\texttt{concat} : \{\texttt{l1} : \texttt{'a list}\} \longrightarrow \{\texttt{l2} : \texttt{'a list}\} \longrightarrow \{\nu : \texttt{'a list} \mid (R_{mem}(\nu) = R_{mem}(\texttt{l1}) \cup R_{mem}(\texttt{l2})) \;\wedge\; (R^*_{oa}(\nu) = R^*_{oa}(\texttt{l1}) \cup R^*_{oa}(\texttt{l2}) \cup R_{mem}(\texttt{l2}) \times R_{mem}(\texttt{l1}))\}$$

Verifying rev - Elaboration

- Elaborate program - A-Normalization

```
fun rev l = case l of
              Nil => Nil
            | Cons(x,xs) => let val xs' = rev(xs)
                                  val x' = [x]
                            in concat(xs',x')
                            end
```

- Elaborate Specifications and populate Env.

$$Nil : \{\nu :' a\ list \mid R_{mem}(\nu) = \{\} \wedge R^*_{oa}(\nu) = \{\} \wedge R^*_{ob}(\nu) = \{\}\}$$

$$\begin{aligned} Cons : \{x :' a\} * \{xs :' a\ list\} \rightarrow \{\nu :' a\ list \mid\ & R_{mem}(\nu) = \{(x)\} \cup R_{mem}(xs) \wedge \\ & R^*_{oa}(\nu) = \{R_{mem}(xs) \times \{(x)\}\} \cup R^*_{oa}(xs) \quad \wedge \\ & R^*_{ob}(\nu) = \{\{(x)\} \times R_{mem}(xs)\} \cup R^*_{ob}(xs)\} \end{aligned}$$

Verifying rev - Type Checking

- Type check specification - Generate VC

```
fun rev l = case l of
              Nil => Nil
            | Cons(x,xs) => let val xs' = rev(xs)
                                val x' = [x]
                            in concat(xs',x')
                            end
```

Strengthens type refinement of l with type refinement of cons(x,xs)

Formal args in the type of concat instantiated with xs' and x'

Recursive invocation. Provides inductive hypothesis.

- Type check specification - Encode VC in SMT Language. Check SAT/UNSAT.

Is this formula satisfiable?

```
(assert (forall ((n T)) (= (Rmemx1 n)(= n x))))
;; Robsx1 = {}
(assert (forall ((n (Pair T))) (= (Roasx1 n) false)))
;; Rmemv = Rmemxs1 U Rmemx1
(assert (forall ((n T)) (= (Rmemv n)(or (Rmemxs1 n) (Rmemx1 n)))))
;; Roasv = Roasxs1 U Roasx1 U (Rmemx1 X Rmemxs1)
(declare-fun A ((Pair T)) Bool)
(assert (forall ((n (Pair T))) (= (A n)(or (Roasxs1 n) (Roasx1 n)))))
(declare-fun B ((Pair T)) Bool)
(assert (forall ((n1 T)(n2 T)) (= (B (mk-pair n1 n2))(and (Rmemx1 n1) (Rmemxs1
n2)))))
(assert (forall ((n (Pair T))) (= (Roasv n)(or (A n) (B n)))))
;; Goal
(declare-const conj1 Bool)
(declare-const conj2 Bool)
;;(assert (= conj1 (forall ((n T)) (= (Rmemv n) (Rmeml n)))))
(assert (= conj2 (forall ((n (Pair T))) (= (Roasv n) (Robsl n)))))
;;(assert (not (and conj1 conj2)))
(assert (not conj2))
;;
(check-sat)
```

tutorial

home video permalink

'►' shortcut: Alt+B

```
unsat
```

Higher-Order Functions

- Majority of catamorphisms are higher-order : *map*, *fold* etc.
- What is the useful specification for *foldl*?

```
fun foldl l f acc =
  case l of
       [] => acc
     | [x::xs] => foldl(xs,f, f(x,acc))
```

- A useful specification of foldl might require the following:
  - ★ The membership relation of the output defined in terms of membership on the input list and the accumulator
  - ★ The ordering relation of the output preserves ordering properties of the input list and the accumulator
    - ✦ Moreover, every element contained in the accumulator ordered with respect to every element in the input list

Abstract relations

- Useful specification for *foldl* can be written using abstract relations.
- Intuition : Assume a hypothetical relation relating inputs and result of the higher-order argument in a way that is convenient to make useful assertion at post-condition.
- We refer to such relations as Abstract Relations.
- Abstract Relations : Uninterpreted relations over which the relational specification is parametrized
  - ★ Lack an operational manifestation
  - ★ Can be instantiated to a concrete structural relation
  - ★ Consequently, useful to specify higher-order catamorphisms.

Example: foldl

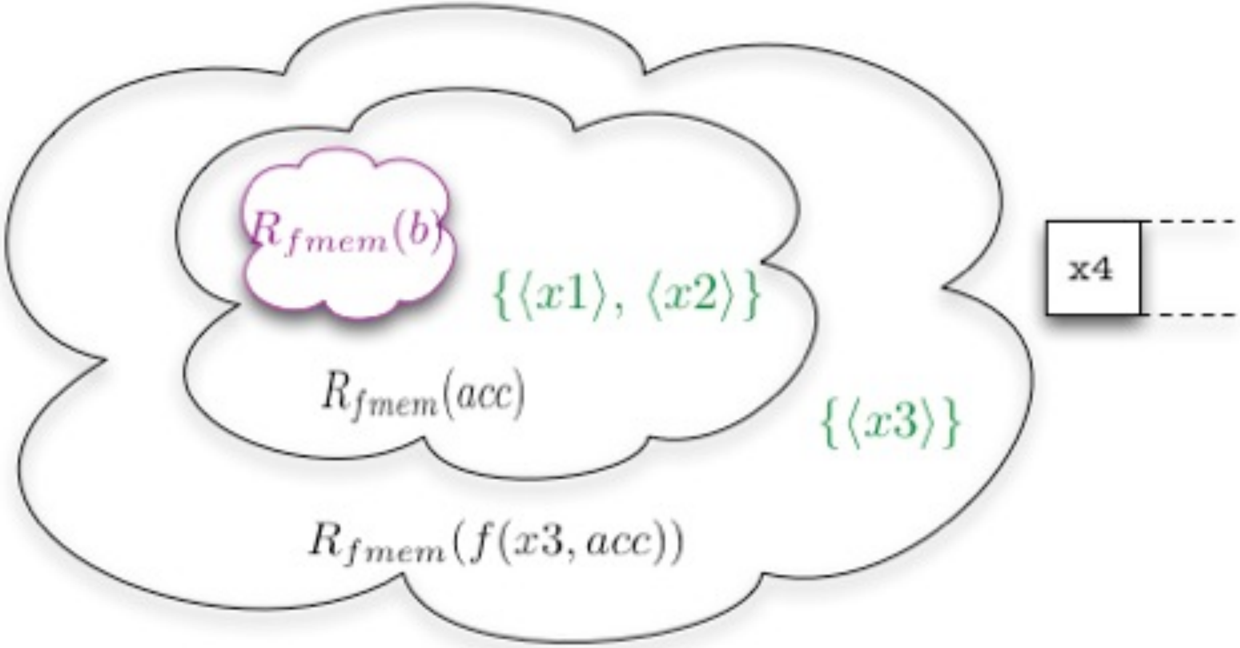
```
fun foldl l f acc =
  case l of
       [] => acc
     | [x::xs] => foldl(xs,f, f(x,acc))
```

- A specification for fold parametrized over abstract relation *Rfmem* :

```
(Rfmem) fold : {l} -> {f : ({x},{acc}) -> {z | Rfmem(z) =
                  {x} U Rfmem(acc)}} -> {b} ->
             {v | Rfmem(v) = Rmem(l) U Rfmem(b)}
```

- Observe that post-condition on higher-order argument *f* is written in terms of *Rfmem* and provides necessary premise to write useful specification for fold.

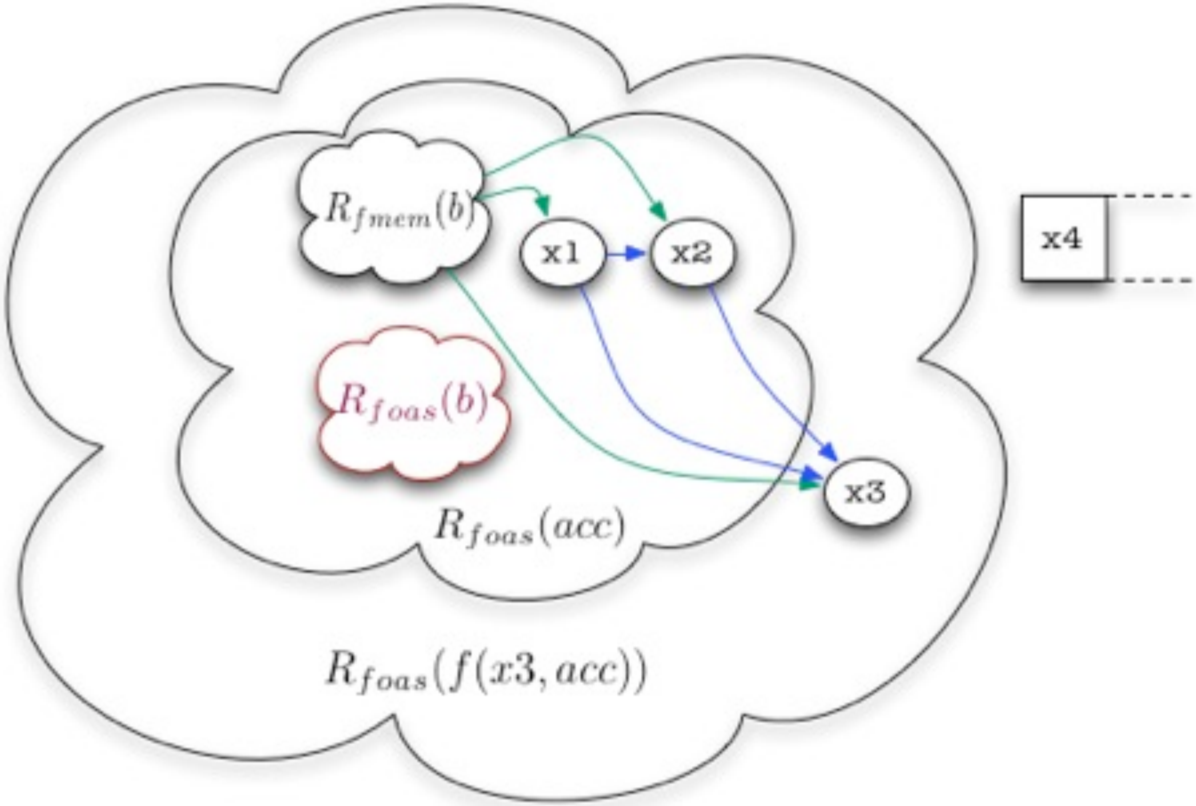
Example - foldl



$$f : (\{x3\} * \{acc\}) \rightarrow \{z \mid Rfmem(z) = \{(x3)\} \cup Rfmem(acc)\}$$

$$v = f(xn, ..f(x3, f(x2, f(x1, b)))) \Rightarrow$$
$$R_{fmem}(v) = R_{mem}(l) \cup R_{fmem}(b)$$

Example - foldl



$$f : (\{x3\}*\{acc\}) \to \{z \mid Rfoas(z) = \{Rfmem(acc) \times \{(x3)\}\} \cup Rfoas(acc)\}$$

$$v = f(xn, ..f(x3, f(x2, f(x1, b)))) \Rightarrow$$
$$R_{foas}(v) = R^*_{ob}(l) \cup R_{foas}(b) \cup \{R_{fmem}(b) \times R_{mem}(l)\}$$

Example - foldl

- Abstract relations can be instantiated with concrete relations at the call-sites.
- Abstract relation instantiation is superficially similar to type variable instantiation.
- In case of foldl, *Rfmem* and *Rfoas* can be instantiated with *Rmem* and *Roa** respectively in the following definition of to assert that result list is reversal of original list.

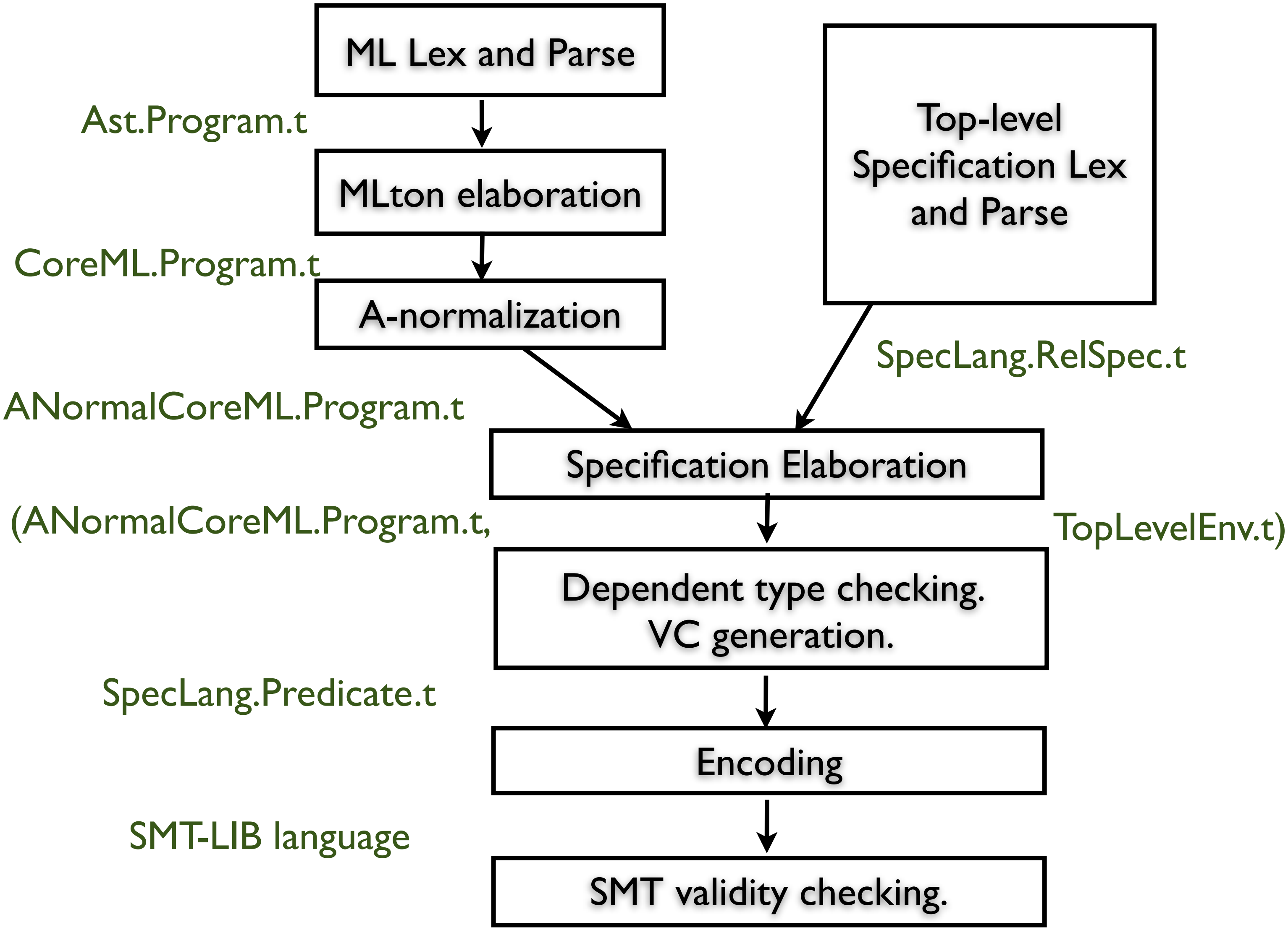
```
fun 'a2 rev = fold ('a2, 'a2 list, Rmem, Roa*) l
                  (Cons ('a2)) (Nil ('a2));
```

CATALYST

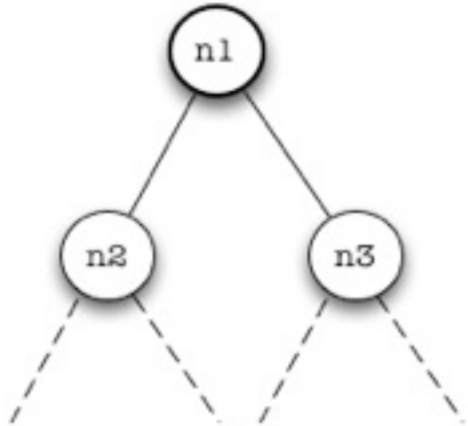
- Catamorphism Analyst.
- Implementation of the verification procedure.
- To Validate structural transforms within a compiler
  - ★ Statically validate that MLton SSA IR structure is preserved across different optimization passes
- To Establish equivalence of heap-sensitive program transformations
  - ★ Example: Deterministic parallelism in the presence of interference

$$\forall H'' \text{s.t. } \{H\}\mathtt{e}_1;\mathtt{e}_2\ \{H'\} \text{ and } \{H\}\ \mathtt{e}_1' \mid\mid \mathtt{e}_2'\ \{H''\}, H' \stackrel{?}{\equiv} H''$$

CATALYST


ML Lex and Parse
Ast.Program.t
MLton elaboration
CoreML.Program.t
A-normalization
ANormalCoreML.Program.t
Top-level Specification Lex and Parse
SpecLang.RelSpec.t
Specification Elaboration
(ANormalCoreML.Program.t,
TopLevelEnv.t)
Dependent type checking. VC generation.
SpecLang.Predicate.t
Encoding
SMT-LIB language
SMT validity checking.

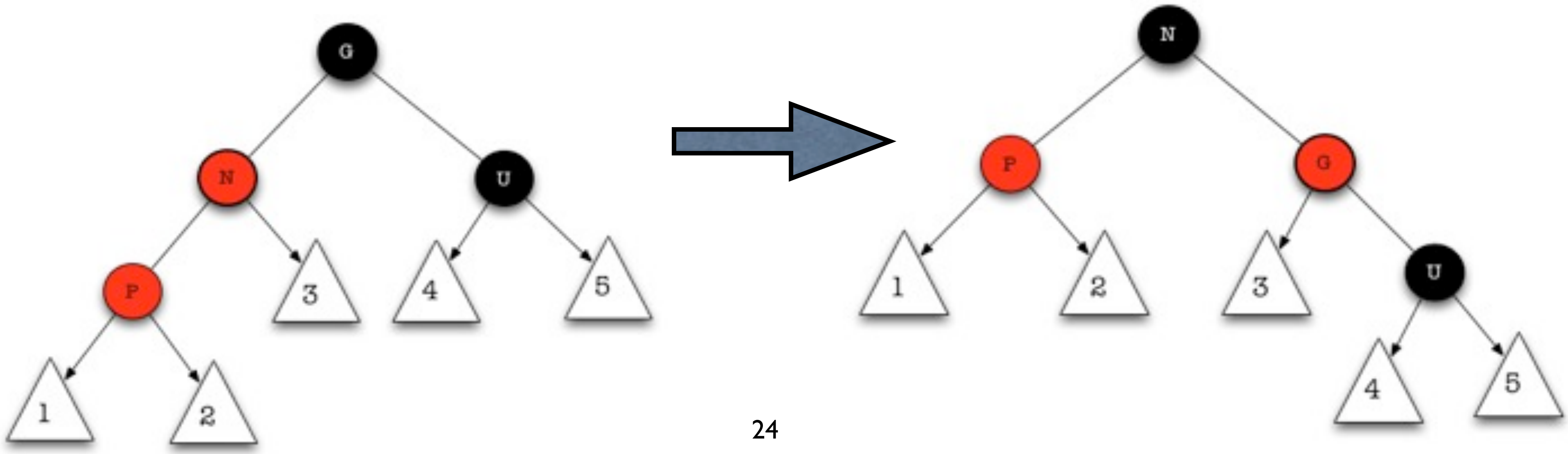
Case study - Red-black tree



BST order = $\{\langle n_2, n_1\rangle, \langle n_1, n_3\rangle, \langle n_2, n_3\rangle, \ldots\}$

Any tree rearrangement should be order-preserving

Red-black tree rotations should preserve BST order



Case study - Red-black tree

Okasaki's Red-black tree balance function

```
datatype Color = R | B
datatype Tree = E | T of Color * Tree * Elem * Tree
fun balance (B,T (R,T (R,a,x,b),y,c),z,d) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance (B,T (R,a,x,T (R,b,y,c)),z,d) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance (B,a,x,T (R,T (R,b,y,c),z,d)) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance (B,a,x,T (R,b,y,T (R,c,z,d))) = T (R,T (B,a,x,b),y,T (B,c,z,d))
  | balance body = T body
```

Structural relations over Tree type

$$\texttt{relation } R_{root}\,\texttt{(T(c,l,n,r))} = \{\texttt{(n)}\}$$
$$\texttt{relation } R_{elem} = R^{*}_{root}$$
$$\texttt{relation } R_{to}\,\texttt{(T(c,l,n,r))} = \{R_{elem}\texttt{(l)} \times \{\texttt{(n)}\}\} \cup \{\{\texttt{(n)}\} \times R_{elem}\texttt{(r)}\} \cup \{R_{elem}\texttt{(l)} \times R_{elem}\texttt{(r)}\}$$

Specification for balance function using *tree-order* relation

$$\texttt{balance : } \{\texttt{t:Tree}\} \longrightarrow \{\texttt{t':Tree} \mid R^{*}_{to}\texttt{(t')} = R^{*}_{to}\texttt{(t)}\}$$

Case study - SSA

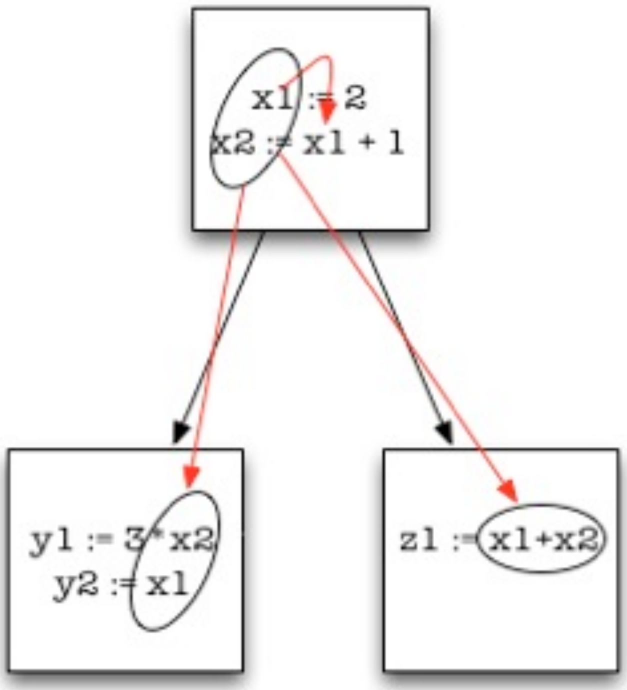
- Def-use domination can be modeled as a structural relation over dominator tree (R_{du}).
- Checking that every use has a corresponding dominating def is equivalent to proving that reflexive closure of a use relation is a subset of the def-use dominator relation.

$$R_{du} = \{\langle x1, x1\rangle,\ \langle x1, x2\rangle,\ \langle x2, x2\rangle,\ \langle x2, x1\rangle\}$$

$$R_{use} = \{\langle x1\rangle,\ \langle x2\rangle\}$$

$$R_{use-refl} = \{\langle x1, x1\rangle,\ \langle x2, x2\rangle\}$$

$$\hat{r} \Leftrightarrow R_{use-refl} \subseteq R_{du}$$



Case study - SSA

- Verifying def-use domination property for MLton SSA requires new relational abstractions and corresponding encodings

```
datatype Stmt.t = Stmt.T of {var: Var.t option,
                             ty: Type.t,
                             exp: Exp.t}
datatype Exp.t = Const of Const.t
        | Var of Var.t
        ....
datatype Block.t =
         Block.T of { statements: Stmt.t list,
                ....  }
datatype Func.t = Func.T of { dominatorTree:Block.t Tree.t
                   .... }

fun removeUnused (t as Func.T{dominatorTree, ...}) =
  let
    ...
    val s = visitVars t
    val f = simplifyBlock s
    val t' = Tree.map dominatorTree f
  in
    t'
  end
```

Case study - SSA

$$R_{exp-use} : \{\texttt{Exp.t} * \texttt{Var.t}\}$$

$$R_{stmt-use} : \{\texttt{Stmt.t} * \texttt{Var.t}\}$$

$$R_{stmt-du} : \{\texttt{Stmt.t} * \texttt{Var.t} * \texttt{Var.t}\}$$

$$R_{block-use} : \{\texttt{Block.t} * \texttt{Var.t}\}$$

$$R_{block-def} : \{\texttt{Block.t} * \texttt{Var.t}\}$$

$$R_{block-du} : \{\texttt{Block.t} * \texttt{Var.t} * \texttt{Var.t}\}$$

$$R_{use} : \{\texttt{Func.t} * \texttt{Var.t}\}$$

$$R_{use-refl} : \{\texttt{Func.t} * \texttt{Var.t} * \texttt{Var.t}\}$$

$$R_{du} : \{\texttt{Func.t} * \texttt{Var.t} * \texttt{Var.t}\}$$

- Multiple structural relations defined to compose def-use domination invariant for SSA graph.
- use relation for Exp.t relates expressions to variables used in the expression. def-use (du) relation for Stmt.t is a cross-product of variable defined (LHS) and use relation for RHS expression.
- In similar way, composition is extended to the level of dominator tree

Case study - SSA

$$\texttt{visitVars : \{t : Func.t\}} \longrightarrow \texttt{\{s : Set.t} \mid R_{set-mem}\texttt{(s)=}R_{use}\texttt{(t)\}}$$

$$\begin{aligned}&\texttt{simplifyBlock : \{s : Set.t\}} \longrightarrow \texttt{\{b : Block.t\}} \longrightarrow \texttt{\{b' : Block.t} \mid \\ &\quad R_{block-def}\texttt{(b')} = R_{block-def}\texttt{(b)} \cap R_{set-mem}\texttt{(s)} \quad \wedge \\ &\quad R_{block-use}\texttt{(b')} = R_{block-use}\texttt{(b)\}}\end{aligned}$$

$$\begin{aligned}&\texttt{removeUnused : \{t : Func.t} \mid R_{use-refl}\texttt{(t)} \subseteq R_{du}\texttt{(t)\}} \longrightarrow \\ &\quad \texttt{\{t' : Func.t} \mid R_{use-refl}\texttt{(t')} \subseteq R_{du}\texttt{(t')\}}\end{aligned}$$

$$\begin{aligned}&\texttt{t' : \{}\nu \texttt{ : Block.t Tree.t} \mid \\ &R_{du}(\nu) = [\{R_{block-def}(\texttt{b}) \cap R_{set-mem}(\texttt{s})\}/R_{block-def}(\texttt{b})]\ R_{du}\texttt{(t)} \quad \wedge \\ &R_{use}(\nu) = R_{use}\texttt{(t)\}}\end{aligned}$$

$$\frac{\mathrm{R_{use-refl}}(\mathtt{t}) \subseteq \mathrm{R_{du}}(\mathtt{t}) \qquad \mathrm{R_{set-mem}}(\mathtt{s}) = \mathrm{R_{use}}(\mathtt{t}) \qquad \mathrm{R_{use}}(\mathtt{t}') = \mathrm{R_{use}}(\mathtt{t})}{\mathrm{R_{use-refl}}(\mathtt{t}') \subseteq \mathrm{R_{du}}(\mathtt{t}')} \Longrightarrow \frac{S1.S1 \subseteq S2 \times S1 \qquad S3 = S1 \qquad S4 = S1}{S4.S4 \subseteq \{S2 \cap S1\} \times S4}$$

Conclusions

- Invariants over algebraic datatype morphisms can be expressed in terms of simple assertions over inductively defined structural relations.
- Similarity in inductive structure of morphisms and structural relations can be exploited for automatic verification.
- But, what after verifying *rev* and *concat*?
- Types have to be usable at call-sites for further type checking. Compositionality is crucial to scale the method to tricky program transformations.

Related Work

- Dependent type checking :
  - ★ Refinement types for ML (PLDI'91),
  - ★ DML (POPL'99),
  - ★ *Liquid Types* (PLDI'08),
  - ★ *Lightweight Dependent Type Inference for ML* (VMCAI'13).
- Invariant checking over recursive datatypes:
  - ★ *Type-based data structure verification* (PLDI'09),
  - ★ Decision procedures for algebraic data types with abstractions (POPL'10)
  - ★ Abstract refinements (ESOP'13).
- Imperative shape analysis