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

★ In $l = \text{Cons}(x_{0},xs)$, relations that can be immediately identified:
  ✦ $\{(l,x_{0})\}$ - head
  ✦ $\{(l,xs)\}$ - tail

★ Similarly, in $t = \text{Branch}(t_{0},x,t_{1})$:
  ✦ $\{(t,x)\}$ - root
  ✦ $\{(t,t_{0})\}$ - left-child
  ✦ $\{(t,t_{1})\}$ - right-child
  ✦ $\{(t,t_{1}), (t,t_{0})\}$ - sub-tree
Let us lift the head relation to an inductive definition ($R$)

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

For $l = \text{Cons} (x0, \text{Cons}(x1,(\text{Cons} x2,Nil)))$,

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

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

$$R(l) = \{x0, x1, x2\}$$

$R$ is actually the membership relation over lists.

$$R = R_{\text{mem}}$$
Similarly, define \textit{Rob} relation as
\[
\forall l, xs : 'a list, x, x' : 'a,
\text{head}(l, x) \Rightarrow \text{tail}(l, xs) \Rightarrow R_{\text{mem}}(xs, x') \Rightarrow R_{\text{ob}}(l, x, x')
\]
and its inductive version, \textit{Rob} as
\[
\forall l, xs : 'alist, x, x' : 'a,
R_{\text{ob}}(l, x, x') \Rightarrow R_{\text{ob}^*}(l, x, x')
R_{\text{ob}^*}(xs, x, x') \Rightarrow R_{\text{ob}^*}(l, x, x')
\]

For \( l = \text{Cons} \ (x0, \text{Cons}(x1,(\text{Cons} \ x2,\text{Nil}))) \),
\[
\textbf{★} R_{\text{ob}^*}(l) = \{(x0,x1), (x0,x2), (x1,x2)\}
\]
\textit{Rob} is \textit{Occurs-before} relation over the list!
\textit{Rob} as succinctly captures the notion of order in the list.
Introduction

• Relations of similar flavour can be defined inductively over trees:
  - tree-membership ($R_{tm}$)
  - pre/post-order ($R_{pre}$/post)
  - total-order ($R_{to}$)
  - depth-first-order ($R_{dfo}$)

• Over any inductive structure
  - Control dependence relation between expressions of AST:
    ```
    datatype ast = ... | If of expr * expr * expr | ...
    ```

  - Scope/Visibility of lambda-bound variables

Monday, December 9, 13
Structural Relations

- We call such relations as *structural relations*
  - Defined inductively over the structure of algebraic datatypes (data structures).
- Pleasant properties:
  - Succinctly 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:

```plaintext
def rev l = case l of
  [] => []
  _ x::xs => concat (rev xs , [x])
```

- What is its specification?

- Dependent type with under-specification:

```plaintext
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:

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

• What is its specification?

• Dependent type with full-functional specification:

\[
rev : \{ l : \text{int list} \} \rightarrow \{ l' : \text{int list} \mid l' = rev(l) \}
\]

★ \textit{rev} in type refinement is an abstraction function from concrete lists to abstract lists

★ Self-referential. Tautology.
A Relational Spec for Reverse

• Recall that \textit{Occurs-before} relation captures the notion of left-to-right order in a list. Similarly, an \textit{Occurs-after} relation captures the right-to-left order.

\begin{itemize}
\item \( \text{Occurs-before}(l) \)
\item \( \text{Occurs-after}(\text{rev}(l)) \)
\end{itemize}

\begin{align*}
l &= [x_0, x_1, x_2] \\
\text{rev}(l) &= [x_2, x_1, x_0] \\
R_{ob}^*(l) &= \{(x_0,x_1), (x_0,x_2), (x_1,x_2)\} \\
R_{oa}^*(\text{rev}(l)) &= \{(x_0,x_1), (x_0,x_2), (x_1,x_2)\}
\end{align*}

\[ \text{rev} : \{l : 'a \text{ list}\} \longrightarrow \{v : 'a \text{ list} \mid R_{ob}^*(l) = R_{oa}^*(v)\} \]
Verify the rev function - Annotations

- Define Structural Relations:

  relation \( \text{Rhd} \ (x::xs) = \{(x)\} \)
  
  relation \( \text{Rmem} = \text{Rhd}^* \)
  
  relation \( \text{Rob} \ (x::xs) = \{(x)\} \times \text{Rmem}(xs) \)
  
  relation \( \text{Roa} \ (x::xs) = \text{Rmem}(xs) \times \{(x)\} \)

- Write assertions

\[
\begin{align*}
\text{rev} : \{l : \text{'a list}\} & \longrightarrow \{\nu : \text{'a list} \mid R_{\text{mem}}(\nu) = R_{\text{mem}}(l) \land \nonumber \\
& \quad R^*_{\text{oa}}(\nu) = R^*_{\text{ob}}(l) \}
\end{align*}
\]

\[
\begin{align*}
\text{concat} : \{l1 : \text{'a list}\} & \longrightarrow \{l2 : \text{'a list}\} \longrightarrow \{\nu : \text{'a list} \mid \nonumber \\
& \quad (R_{\text{mem}}(\nu) = R_{\text{mem}}(l1) \cup R_{\text{mem}}(l2)) \land \nonumber \\
& \quad (R^*_{\text{oa}}(\nu) = R^*_{\text{oa}}(l1) \cup R^*_{\text{oa}}(l2) \cup R_{\text{mem}}(l2) \times R_{\text{mem}}(l1)) \}
\end{align*}
\]
Verifying rev - Elaboration

• Elaborate program - A-Normalization

```haskell
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.

\[
\text{Nil} : \{ \nu : a \text{ list} \mid R_{\text{mem}}(\nu) = \{\} \land R^*_o(\nu) = \{\} \land R^*_b(\nu) = \{\} \}
\]

\[
\text{Cons} : \{ x : a \}^{\ast} \{ xs : a \text{ list} \} \rightarrow \{ \nu : a \text{ list} \mid R_{\text{mem}}(\nu) = \{(x)\} \cup R_{\text{mem}}(xs) \land R^*_o(\nu) = \{ R_{\text{mem}}(xs) \times \{(x)\} \} \cup R^*_o(xs) \land R^*_b(\nu) = \{(x)\} \times R_{\text{mem}}(xs) \} \cup R^*_b(xs) \}
\]
Verifying rev - Type Checking

- Type check specification - Generate VC

```plaintext
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 satisifiable?

```plaintext
35 (assert (forall ((n T)) (= (Rmemx1 n)(= n x))))
36 ;; Robsx1 = {}
37 (assert (forall ((n (Pair T))) (= (Robsx1 n) false)))
38 ;; Rmev = Rmexs1 U Rmemx1
39 (assert (forall ((n T)) (= (Rmev n)(or (Rmexs1 n) (Rmemx1 n)))))
40 ;; Roasv = Roasxs1 U Roasx1 U (Rmex1 X Rmemx1)
41 (declare-fun A ((Pair T)) Bool)
42 (assert (forall ((n (Pair T))) (= (A n)(or (Roasxs1 n) (Robsx1 n)))))
43 (declare-fun B ((Pair T)) Bool)
44 (assert (forall ((n1 T)(n2 T)) (= (B (mk-pair n1 n2))(and (Rmemx1 n1) (Rmexs1 n2)))))
45 (assert (forall ((n (Pair T))) (= (Roasv n)(or (A n) (B n)))))
46 ;; Goal
47 (declare-const conj1 Bool)
48 (declare-const conj2 Bool)
49 ;;(assert (= conj1 (forall ((n T)) (= (Rmev n) (Rmemx1 n)))))
50 (assert (= conj2 (forall ((n (Pair T))) (= (Roasv n) (Robsx1 n)))))
51 ;;(assert (not (and conj1 conj2)))
52 (assert (not conj2))
53 ;;
54 ;;
55 (check-sat)
56
```

unsat
Higher-Order Functions

• Majority of catamorphisms are higher-order: \textit{map, fold} etc.

• What is the useful specification for \textit{foldl}?

\begin{verbatim}
fun foldl l f acc =
    case l of
        [] => acc
    | [x::xs] => foldl(xs,f, f(x,acc))
\end{verbatim}

• A useful specification of \textit{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 \textit{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.
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 \( Rfmem \):

\[
(Rfmem) \text{ fold } : \{l\} \rightarrow \{f : (\{x\},\{acc\}) \rightarrow \{z \mid Rfmem(z) = \{x\} \cup Rfmem(acc))\} \rightarrow \{b\} \rightarrow \\
\{v \mid Rfmem(v) = Rmem(l) \cup Rfmem(b)\}
\]

• 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

\[
f : (\{x3\} \times \{\text{acc}\}) \to \{z \mid R_{f_{\text{mem}}}(z) = \{(x3)\} \cup R_{f_{\text{mem}}}(\text{acc})\}
\]

\[
v = f(xn, \ldots f(x3, f(x2, f(x1, b)))) \Rightarrow \\
R_{f_{\text{mem}}}(v) = R_{\text{mem}}(l) \cup R_{f_{\text{mem}}}(b)
\]
Example - foldl

\[ f : (\{x3\}^*\{acc\}) \rightarrow \{ z \mid R_{foas}(z) = \{ R_{fmem}(acc) \times \{(x3)\} \} \cup R_{foas}(acc) \} \]

\[ v = f(xn, \ldots f(x3, f(x2, f(x1, b)))) \Rightarrow \]

\[ R_{foas}(v) = R^*_ob(l) \cup R_{foas}(b) \cup \{ R_{fmem}(b) \times R_{mem}(l) \} \]
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, $Rf_{mem}$ and $Rf_{oas}$ can be instantiated with $R_{mem}$ and $R_{oa}^*$ 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));```
• 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

\[ \forall H'' \text{s.t. } \{ H \} e_1 ; e_2 \{ H' \} \text{ and } \{ H \} e'_1 \parallel e'_2 \{ H'' \}, H' \equiv H'' \]
ML Lex and Parse

MLton elaboration

A-normalization

Specification Elaboration

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

```haskell
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

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

Specification for balance function using tree-order relation

balance : \{t:Tree\} \rightarrow \{t’:Tree \mid R_{to}^*(t’) = R_{to}^*(t)\}
• 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.

$$R_{du} = \{\langle x_1, x_1 \rangle, \langle x_1, x_2 \rangle, \langle x_2, x_2 \rangle, \langle x_2, x_1 \rangle\}$$

$$R_{use} = \{\langle x_1 \rangle, \langle x_2 \rangle\}$$

$$R_{use-refl} = \{\langle x_1, x_1 \rangle, \langle x_2, x_2 \rangle\}$$

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

- Multiple structural relations defined to compose def-use domination invariant for SSA graph.

- use relation for $\text{Exp.t}$ relates expressions to variables used in the expression. def-use (du) relation for $\text{Stmt.t}$ 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

\[
\text{visitVars} : \{t : \text{Func.t} \} \rightarrow \{s : \text{Set.t} \mid R_{\text{set\text{-}mem}}(s) = R_{\text{use}}(t)\}
\]

\[
\text{simplifyBlock} : \{s : \text{Set.t} \} \rightarrow \{b : \text{Block.t} \rightarrow \{b' : \text{Block.t} \mid R_{\text{block\text{-}def}}(b') = R_{\text{block\text{-}def}}(b) \cap R_{\text{set\text{-}mem}}(s) \land R_{\text{block\text{-}use}}(b') = R_{\text{block\text{-}use}}(b)\}
\]

\[
\text{removeUnused} : \{t : \text{Func.t} \mid R_{\text{use\text{-}refl}}(t) \subseteq R_{\text{du}}(t)\} \rightarrow \{t' : \text{Func.t} \mid R_{\text{use\text{-}refl}}(t') \subseteq R_{\text{du}}(t')\}
\]

\[
t' : \{\nu : \text{Block.t Tree.t} \mid R_{\text{du}}(\nu) = \left[\{R_{\text{block\text{-}def}}(b) \cap R_{\text{set\text{-}mem}}(s)\}/R_{\text{block\text{-}def}}(b)\right] R_{\text{du}}(t) \land R_{\text{use}}(\nu) = R_{\text{use}}(t)\}
\]

\[
R_{\text{use\text{-}refl}}(t) \subseteq R_{\text{du}}(t)
\]

\[
R_{\text{set\text{-}mem}}(s) = R_{\text{use}}(t)
\]

\[
R_{\text{use}}(t') = R_{\text{use}}(t)
\]

\[
R_{\text{use\text{-}refl}}(t') \subseteq R_{\text{du}}(t')
\]

\[
S1.S1 \subseteq S2 \times S1
\]

\[
S3 = S1
\]

\[
S4 = S1
\]

\[
S4.S4 \subseteq \{S2 \cap S1\} \times S4
\]
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