A Relational Framework for Higher-Order Shape Analysis

Suresh Jagannathan

Joint work with Gowtham Kaki

Introduction

- Inductively defined algebraic datatypes are a key feature in modern functional languages
  - ★ Enable the expression of rich data structures - lists, trees, graphs, etc.
- But, they also pose challenges for verification
  - ★ Recursive types
    - ✦ Quantification
  - ★ Important attributes are often not manifest in a constructor's signature
    - ✦ E.g., length, sorted-ness, height, balance, membership, dominance, etc.
  - ★ Polymorphism and higher-order functions
- Tension
  - ★ desire expressive specifications over the *shape* of data
  - ★ require specifications to be efficiently checkable using existing (first-order) tools

Goals

- Validating structural transforms within a compiler
  - ★ MLton's SSA intermediate representation expressed as a graph datatype whose nodes represent instruction blocks.
  - ★ Statically type-check that SSA structure is preserved across different optimization passes

- Establishing equivalence of heap-sensitive program transformations
  - ★ Example: Deterministic parallelism in the presence of interference

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

Example

```
fun len l = case l of [] => 0 | x :: xs => 1 + len xs

insert: { x : int } -> { l : int } -> { l' : int | len l' = 1 + len l }

fun sort: { l : int list } -> { ν : int list | len(ν) = len(l) } =
  case l of
     [] => []
  |  x::xs => let val l' = sort xs
              in insert x l'
              end
```

Simple inductive argument sufficient to show specification holds:

$$\frac{\texttt{len(l)} = 1 + \texttt{len(xs)}, \qquad \texttt{l}' : \{\nu : \texttt{int list} \mid \texttt{len}(\nu) = \texttt{len(xs)}\}}{\{\nu : \texttt{int list} \mid \texttt{len}(\nu) = 1 + \texttt{len}(\texttt{l}')\} <: \{\nu : \texttt{int list} \mid \texttt{len}(\nu) = \texttt{len(l)}\}}$$

Example

```
fun rev : { l : int list } -> { ν : int list | ν = rev(l) } =
  case l of
    [] => []
  | x::xs => let val xs' = rev(xs)
                 val x' = [x]
             in concat(xs',x')
             end
```

Consider the self-referential specification:

```
rev : { l : int list } -> { ν : int list | ν = rev(l) }
```

Reasoning about "rev" in the specification requires about reasoning about concat

Inject a full functional definition of concat into a theory of lists as a first-order definition

Directly using such a methodology complicates automatic verification

Issues

- The specification of rev describes a catamorphism in terms of another catamorphism
  - ★ For rev, abstract domain is as complex as concrete one
- What we need:
  - ★ Simple abstract domains (e.g., sets) that can help us reason about shape properties of algebraic datatypes
    - ✦ Not defined in terms of complex catamorphisms
- For reverse:
  - ★ Want to reason *structurally* about the order of elements in the list
  - ★ Without appealing to an *operational* definition of how that ordering is realized

Idea

- Reason *relationally* about the correspondence between the input and output lists of rev
- Key property: list order
  - ★ An ordering relation on lists relates every element with every other element based on relative position
  - ★ Transitive closure is a faithful characterization of a list representation
- Example:

$$\texttt{l} = \texttt{[x1,x2,x3]}$$

$$R^*_{ob}(\texttt{l}) = \{\langle \texttt{x1}, \texttt{x2} \rangle, \langle \texttt{x1}, \texttt{x3} \rangle, \langle \texttt{x2}, \texttt{x3} \rangle\}$$

$$R^*_{oa}(\texttt{l}) = \{\langle \texttt{x2}, \texttt{x1} \rangle, \langle \texttt{x3}, \texttt{x1} \rangle, \langle \texttt{x3}, \texttt{x2} \rangle\}$$

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

Abstract Relations and Higher-Order Functions

```
datatype 'a ordpair = Pair of 'a*'a
fun toList p f  = case p of Pair(x1,x2) => f(x2,f(x1,Nil))
```

Consider a specification that asserts that the last application of f is on the second element of p

Slightly weaker variant of an ordering specification of f to p

Assume a hypothetical (abstract) relation that relates f's second (list) argument with its first (element from p)

$$\texttt{toList} : \{\texttt{p} : {'\texttt{a ordpair}}\} \rightarrow \{\texttt{f} : \{\texttt{x} : {'\texttt{a}}\}\} \rightarrow \{\texttt{acc} : {'\texttt{a list}}\} \rightarrow$$

$$\{\texttt{z} : {'\texttt{a list}} \mid \hat{R}(\texttt{z}) = \{(\texttt{x})\}\} \rightarrow \{\nu : {'\texttt{a list}} \mid \hat{R}(\nu) = R_{snd}(\texttt{p})\}$$

At a call-site instantiate $\hat{R}$ with a concrete relation

Example: if f were "Cons" then concrete relation could be R_{hd}

Structural Relations

- Describe structural properties of a data structure
- Suitable as abstract domains used by specifications
  - ★ Natural translation to decidable theories (e.g., sets) supported by existing first-order solvers
    - ✦ Common relational abstractions can be built using cross-product, union, etc.
- Framework
  - ★ Specification language
    - ✦ Dependent types equipped with expressive relational algebra to specify shape invariants
    - ✦ *Superimpose* relational definitions on the datatype definition
  - ★ Type checking verifies specifications
    - ✦ Interpret constructor applications as set operations within an abstract relational domain
    - ✦ Natural encoding of relations into (decidable) set theory

Structural Relations

Given a constructor list pattern, relate the head of the list with the tail

$$R_{hd} = \{(x :: xs, x) \,|\, x : \texttt{'a} \ \wedge \ xs : \texttt{'a list}\}$$

Using list projection π_n and selection σ_θ

$$R_{hd}(l) = \pi_{\#2}(\sigma_{\#1=l} R_{hd})$$

Equivalently

$$R_{hd}(x :: xs) = \{\langle x \rangle\} \text{ or } \texttt{relation } R_{hd}(x::xs) = \{(x)\}$$

For any list pattern constructed using "::"

- $(x :: xs, x) \in R_{hd}$
- $\nexists x' \text{s.t. } x' \neq x, \text{but } (x :: xs, x') \in R_{hd}$

Composition

- Can build complex relations from simple structural ones
- Example:
  - ★ `datatype expr = ... | ITE of expr * expr * expr option | ...`
  - ★ Specify a control dependence that relates the Boolean guard with the true and (optional) false branches

```
relation Rcnt(Some(a)) = {(a)}
relation Rcd(ITE(e1,e2,e3op)) = {(e1,e2)} ∪ {{(e1)}} × Rcnt(e3op)
```

Composition

- Consider a relation that defines list membership

$$R_{mem}(\, x :: xs \,) \;=\; \{(x)\} \cup R_{mem}(\, xs \,)$$

- Note that this is a simple inductive extension of the head relation
  - ★ If R is a relation that relates elements of a list of type 'a list to elements of 'a then its inductive extension (written R^*) is the least relation that satisfies:
    - ✦ $R(l, v) \Rightarrow R^*(l, v)$
    - ✦ if $l = x :: xs$, then $R(xs, v) \Rightarrow R^*(l, v)$

- $R^*(\, x :: xs \,) \;=\; R(\, x :: xs \,) \cup R^*(xs)$
- `relation Rmem = Rhd*`
  - ★ Can automatically generate inductive definitions from simple structural relations

Composition

$$\texttt{relation}\ \mathtt{R_{cnt}}(\mathtt{Some}(\mathtt{a})) = \{(\mathtt{a})\}$$
$$\texttt{relation}\ \mathtt{R_{root}}(\mathtt{Node}(\mathtt{l},\mathtt{k},\mathtt{v},\mathtt{r})) = \{(\mathtt{k},\mathtt{v})\}$$
$$\texttt{relation}\ \mathtt{R_{ch}}(\mathtt{Node}(\mathtt{l},\mathtt{k},\mathtt{v},\mathtt{r})) = \{(\mathtt{l},\mathtt{r})\} \cup \{(\mathtt{r},\mathtt{l})\}$$
$$\texttt{relation}\ \mathtt{R_{mem}} = \mathtt{R}^*_{\mathtt{hd}}$$
$$\texttt{relation}\ \mathtt{R_{ob}}(\mathtt{x} :: \mathtt{xs}) = \{(\mathtt{x})\} \times \mathtt{R_{mem}}(\mathtt{xs})$$
$$\texttt{relation}\ \mathtt{R_{sib}} = \mathtt{R}^*_{\mathtt{ch}}$$
$$\texttt{relation}\ \mathtt{R_{dict}} = \mathtt{R}^*_{\mathtt{root}}$$

Dependent Types

- Specifications expressed within a dependent type system

$$\{\nu : T \mid \hat{e}\}$$

- Two kinds of type refinements
  - ★ Base predicates : equality predicates over linear arithmetic

$$\{\nu : int \mid \nu = x + 1\}$$

  - ★ Relational predicates : equality or subset-inclusion predicates over relational abstractions

$$\{\nu : \texttt{'a list} \mid \bar{R}_{mem}(\nu) = \{\}\}$$

Datatypes

- Expressed using a sum-of-products formulation

$$\mu t.\ \texttt{Nil} \ + \ \texttt{Cons}\,\langle\, \texttt{x} :' \texttt{a}, \texttt{xs} : \texttt{t} \,\rangle$$

- Generic form:

$$\mu t.\ \sum_i \ \texttt{C}\,_i \langle x_i : T_i, y_i : t \rangle$$

Types and Well-formedness

$$\frac{R \notin Domain(\Gamma) \qquad \forall i,\ \Gamma;\ \langle x_i : T_i \rangle;\ \langle y_i : \sum_i \mathtt{C}_i \langle \overline{T_i} \rangle \rangle \vdash \rho_i : \{\overline{T}\}}{\Gamma \vdash_{wf} (R, \{\sum_i \mathtt{C}_i \langle \overline{T_i} \rangle * \overline{T}\}, \langle \mathtt{C}_i \langle x_i, y_i \rangle \Rightarrow \rho_i \rangle)}$$

relation name

relation type

relation definition

$$r \in Relational\ Expressions ::= \{\langle x \rangle\} \mid R(x) \mid r \times r \mid r \cup r$$
$$\rho \in Relational\ Terms ::= r \mid R^*(x)$$

Type Checking

$$\frac{\Gamma \vdash \hat{T}_1 <: \hat{T}_2 \quad [\![\Gamma]\!] \wedge [\![\hat{e}_1]\!] \Rightarrow [\![\hat{e}_2]\!]}{\Gamma \vdash \{\nu : \hat{T}_1 \mid \hat{e}_1\} <: \{\nu : \hat{T}_1 \mid \hat{e}_1\}}$$

Subtype rule propagates dependent type constraints

Reduces to checking the validity of

$$[\![\Gamma]\!] \wedge [\![\hat{e}_1]\!] \Rightarrow [\![\hat{e}_2]\!]$$

Encoding within the decidable theories of underlying SMT engine

Example

```
let x = (l=l') in case (x) of true => l' | false => l
```

Want to typecheck the true branch against the type

$$\{\nu : \texttt{int list} \mid R_{mem}(\nu) = R_{mem}(\texttt{l})\}$$

$$[\![(\texttt{l}, \texttt{l}' : \texttt{int list}, \texttt{x} : \texttt{bool}) \wedge (\texttt{l} = \texttt{l}' \Leftrightarrow \texttt{x} = \texttt{true}) \wedge \texttt{x} = \texttt{true}]\!] \Rightarrow$$

$$[\![R_{mem}(\texttt{l}') = R_{mem}(\texttt{l})]\!]$$

Abstract lists into uninterpreted sorts, relations as sets

$$(\texttt{l}, \texttt{l}' : \texttt{T}, \texttt{x} : \texttt{bool}, \texttt{S}, \texttt{S}' : \texttt{int set}) \wedge$$
$$(\texttt{l} = \texttt{l}' \Leftrightarrow \texttt{x} = \texttt{true}) \wedge (\texttt{l} = \texttt{l}' \Rightarrow \texttt{S} = \texttt{S}') \wedge (\texttt{x} = \texttt{true})$$
$$\Rightarrow (\texttt{S}' = \texttt{S})$$

derived from equality
x = y => R(x) = R(y)

Static Semantics

Static Semantics

$$\frac{\Gamma \vdash R : \{\sum_i \mathtt{C}_i\langle\overline{T_i}\rangle * \overline{T}\} \quad \Gamma \vdash x : \sum_i \mathtt{C}_i\langle x_i : T_i, y_i : t\rangle}{R^*(x) \hookrightarrow R(x) \cup \bigcup_{y_i} R^*(y_i)}$$

[R-Induct]: rewrite rule that unfolds inductive definition of the relation

Static Semantics

$$\frac{\Gamma \vdash R : \{\sum_i \mathtt{C}_i\langle\overline{T_i}\rangle * \overline{T}\} \quad \Gamma \vdash x : \sum_i \mathtt{C}_i\langle x_i : T_i, y_i : t\rangle}{R^*(x) \hookrightarrow R(x) \cup \bigcup_{y_i} R^*(y_i)}$$

[R-Induct]: rewrite rule that unfolds inductive definition of the relation

[T-Cons]:instantiate relation for each constructor application

$$\frac{\Gamma \vdash \langle x_i : T_i\rangle \quad \Gamma \vdash \langle y_i : \sum_i \mathtt{C}_i\langle\overline{T_i}\rangle\rangle \quad \Gamma \vdash_{wf} (R, \{\sum_i \mathtt{C}_i\langle\overline{T_i}\rangle * \overline{T}\}, \langle \mathtt{C}_i\langle x_i, y_i\rangle \Rightarrow \rho_i\rangle)}{\Gamma \vdash \mathtt{C}_i\langle x_i, y_i\rangle : \{\nu : \sum_i \mathtt{C}_i(\overline{T_i}) \mid R(\nu) = \rho_i(\langle x_i, y_i\rangle)\}}$$

Static Semantics

$$\frac{\Gamma \vdash R : \{\sum_i \mathtt{C}_i\langle\overline{T_i}\rangle * \overline{T}\} \quad \Gamma \vdash x : \sum_i \mathtt{C}_i\langle x_i : T_i, y_i : t\rangle}{R^*(x) \hookrightarrow R(x) \cup \bigcup_{y_i} R^*(y_i)}$$

[R-Induct]: rewrite rule that unfolds inductive definition of the relation

[T-Cons]:instantiate relation for each constructor application

$$\frac{\Gamma \vdash \langle x_i : T_i\rangle \quad \Gamma \vdash \langle y_i : \sum_i \mathtt{C}_i\langle\overline{T_i}\rangle\rangle \quad \Gamma \vdash_{wf} (R, \{\sum_i \mathtt{C}_i\langle\overline{T_i}\rangle * \overline{T}\}, \langle \mathtt{C}_i\langle x_i, y_i\rangle \Rightarrow \rho_i\rangle)}{\Gamma \vdash \mathtt{C}_i\langle x_i, y_i\rangle : \{\nu : \sum_i \mathtt{C}_i(\overline{T_i}) \mid R(\nu) = \rho_i(\langle x_i, y_i\rangle)\}}$$

$$\frac{\Gamma \vdash x : \sum_i \mathtt{C}_i(\overline{T_i}) \quad \Gamma \vdash_{wf} (R, \{\sum_i \mathtt{C}_i\langle\overline{T_i}\rangle * \overline{T}\}, \langle \mathtt{C}_i\langle x_i, y_i\rangle \Rightarrow \rho_i\rangle) \quad \forall i,\ \Gamma;\ \langle x_i : T_i\rangle;\ \langle y_i : \sum_i \mathtt{C}_i(\overline{T_i})\rangle;\ R(x) = \rho_i(\langle x_i, y_i\rangle) \vdash e_i : \hat{T}}{\Gamma \vdash \mathtt{case}\ x\ \mathtt{of}\ |_i\ \mathtt{C}_i\langle x_i, y_i\rangle \Rightarrow e_i : \hat{T}}$$

[T-Case]: case-match

Example - rev

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

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

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

Example - rev

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

Use T-Case rule and given relations R_{hd}, R_{mem}, R_{ob} , typecheck body in environment:

*By [R-Induct] on Rhd**

$$\Gamma' = \Gamma;\ \texttt{x:'a};\ \texttt{xs:'a list};\ R_{mem}(\texttt{l}) = \{(\texttt{x})\} \cup R_{mem}(\texttt{xs}) \quad \wedge$$
$$R^*_{ob}(\texttt{l}) = \{\{(\texttt{x})\} \times R_{mem}(\texttt{xs})\} \cup R^*_{ob}(\texttt{xs})$$

*By [R-Induct] on Rob**

Example - rev

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

Typecheck in environment:

post-condition of **rev(xs)**

Γ'' = Γ'; xs':'a list; R_{mem}(xs') = R_{mem}(xs) $\wedge$ R^*_{oa}(xs') = R^*_{ob}(xs); x':'a list; R_{mem}(x') = {(x)} $\wedge$ R^*_{oa}(x') = {}

properties of Cons(x,Nil)

Example - rev

Γ'' = Γ'; xs':'a list; R_{mem}(xs') = R_{mem}(xs) $\wedge$ R^*_{oa}(xs') = R^*_{ob}(xs); x':'a list; R_{mem}(x') = {(x)} $\wedge$ R^*_{oa}(x') = {}

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

Instantiating formals with actuals in concat's type gives:

$$\{\nu :' a\ \texttt{list} \mid (R_{mem}(\nu) = R_{mem}(\texttt{xs}') \cup R_{mem}(\texttt{x}')) \wedge (R^*_{oa}(\nu) = R^*_{oa}(\texttt{xs}') \cup R^*_{oa}(\texttt{x}') \cup \{R_{mem}(\texttt{x}') \times R_{mem}(\texttt{xs}')\})\}$$

Example - rev

Γ'' = Γ'; xs' : 'a list; R_{mem}(xs') = R_{mem}(xs) $\wedge$ R^*_{oa}(xs') = R^*_{ob}(xs); x' : 'a list; R_{mem}(x') = {(x)} $\wedge$ R^*_{oa}(x') = {}

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

Instantiating formals with actuals in concat's type gives:

$$\{\nu :' a\ \texttt{list} \mid (R_{mem}(\nu) = R_{mem}(\texttt{xs}') \cup R_{mem}(\texttt{x}')) \wedge (R^*_{oa}(\nu) = R^*_{oa}(\texttt{xs}') \cup R^*_{oa}(\texttt{x}') \cup \{R_{mem}(\texttt{x}') \times R_{mem}(\texttt{xs}')\})\}$$

Typechecking against Γ'':

$$R_{mem}(\texttt{xs}') = R_{mem}(\texttt{xs}), R_{mem}(\texttt{x}') = (\{\texttt{x}\}) \implies R_{mem}(\texttt{xs}') \cup R_{mem}(\texttt{x}') = R_{mem}(\texttt{l})$$

matches desired specification for rev

$$R^*_{oa}(\texttt{xs}') \cup R^*_{oa}(\texttt{x}') = R_{ob} * (\texttt{xs})$$
$$\{R_{mem}(\texttt{x}') \times R_{mem}(\texttt{xs}')\} = R_{ob}(\texttt{x} :: \texttt{xs})$$

$$R^*_{ob}(\texttt{l} = \texttt{x} :: \texttt{xs}) = R_{ob}(\texttt{l}) \cup R^*_{ob}(\texttt{xs})$$

Abstract relations

- Relations whose domain is defined over an uninterpreted (polymorphic) type
  - ★ Lack an operational manifestation
  - ★ Can be instantiated to a concrete structural relation
  - ★ Consequently, useful to specify higher-order catamorphisms.

$$e \in \mathit{Expressions} ::= \ldots \mid [\Lambda\alpha\langle R : S\rangle]\, e \mid [T\langle R\rangle]\, e$$

$$R \in \mathit{Relation} ::= \ldots \quad \mathit{Types} \mid \{\alpha * \overline{T}\}$$

T-Gen

$$\frac{\Gamma;\ \langle R\ :\ \{\alpha * \overline{T}\}\rangle \vdash e\ :\ \hat{T}}{\Gamma \vdash [\Lambda\alpha\langle R\ :\ \{\alpha * \overline{T}\}\rangle]\, e\ :\ \forall\alpha\langle R\ :\ \{\alpha * \overline{T}\}\rangle,\ \hat{T}}$$

T-Inst

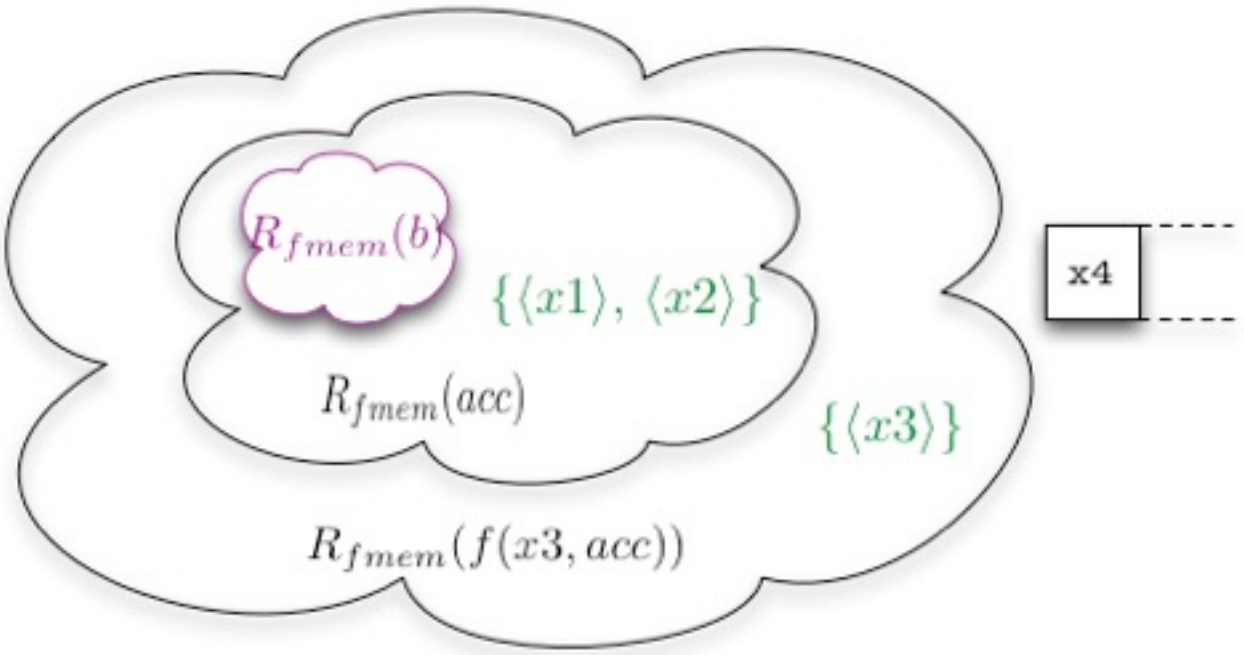
$$\frac{\Gamma \vdash e\ :\ \forall\alpha\langle R\ :\ \{\alpha * \overline{T_R}\}\rangle,\ \hat{T} \qquad \Gamma \vdash \langle R'\ :\ \{T' * \overline{T_R}\}\rangle}{\Gamma \vdash [T'\langle R'\rangle]\, e\ :\ \langle [R'/R]\rangle\, [T'/\alpha]\, \hat{T}}$$

Example: foldl

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

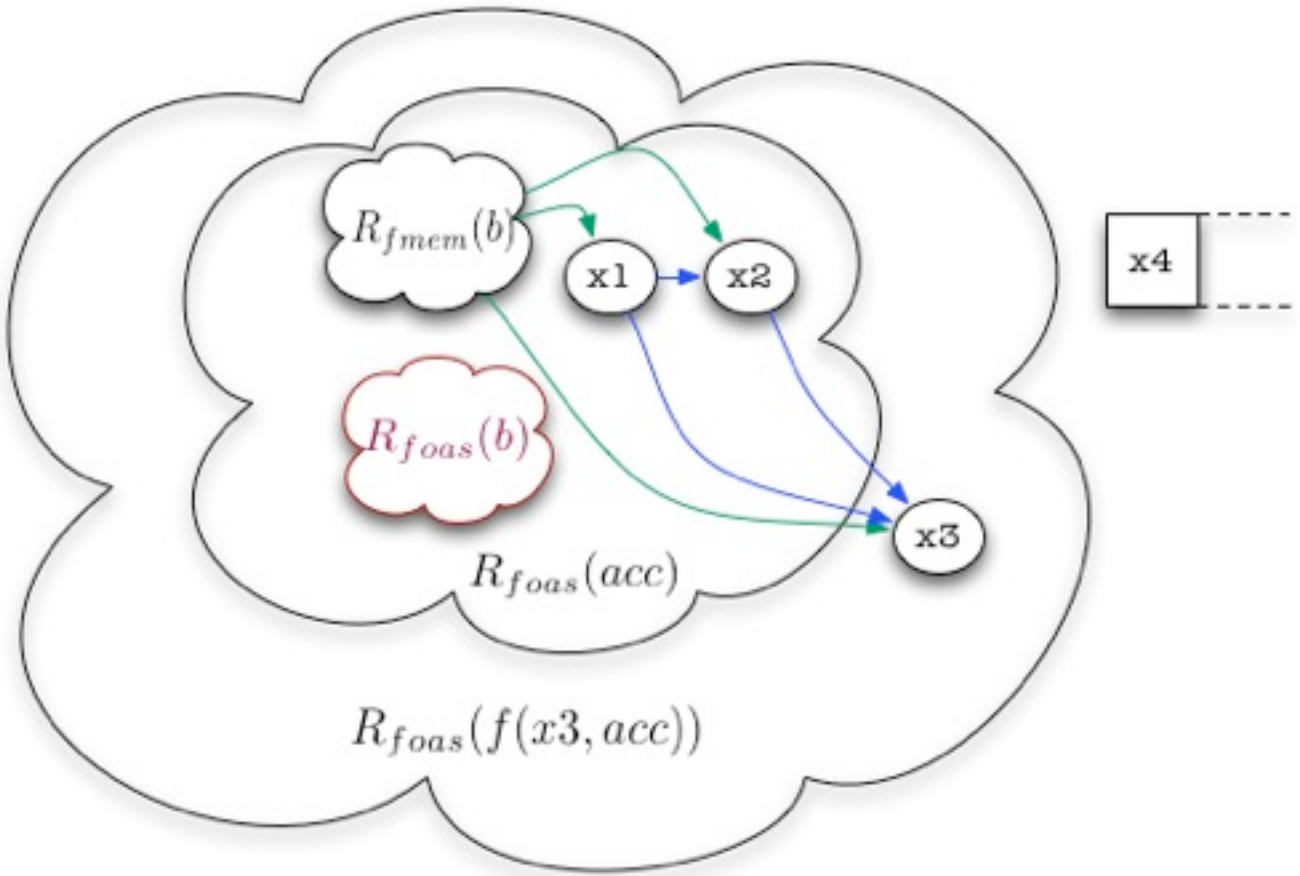
- A useful specification of foldl might require the following:
  - ★ The membership relation of the output defined in terms of membership on the input list and the accumulator
  - ★ The ordering relation of the output preserves ordering properties of the input list and the accumulator
    - ✦ Moreover,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

Example - foldl



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

Example - foldl



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

Example - foldl

Abstract relations R_{fmem} and R_{foas} can be instantiated with appropriate concrete relations at call-site

Example: Instantiating $\texttt{'b}\langle R_{fmem} : \{\texttt{'b} *' \texttt{a}\}, R_{foas} : \{\texttt{'b} *' \texttt{a} *' \texttt{a}\}\rangle$ with $\texttt{'a list}\langle R_{mem}, R^*_{oa}\rangle$ at the following call-site:

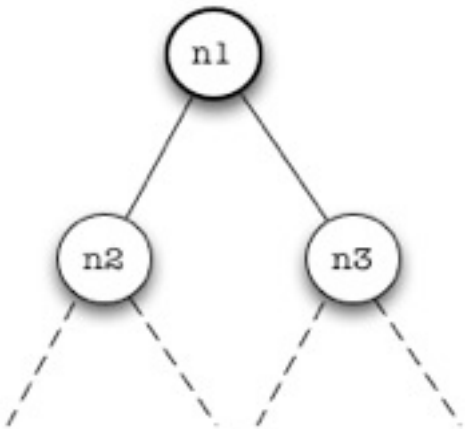
```
foldl l Cons Nil
```

assigns following type to the call:

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

asserting that the result list is a reversal of the original list

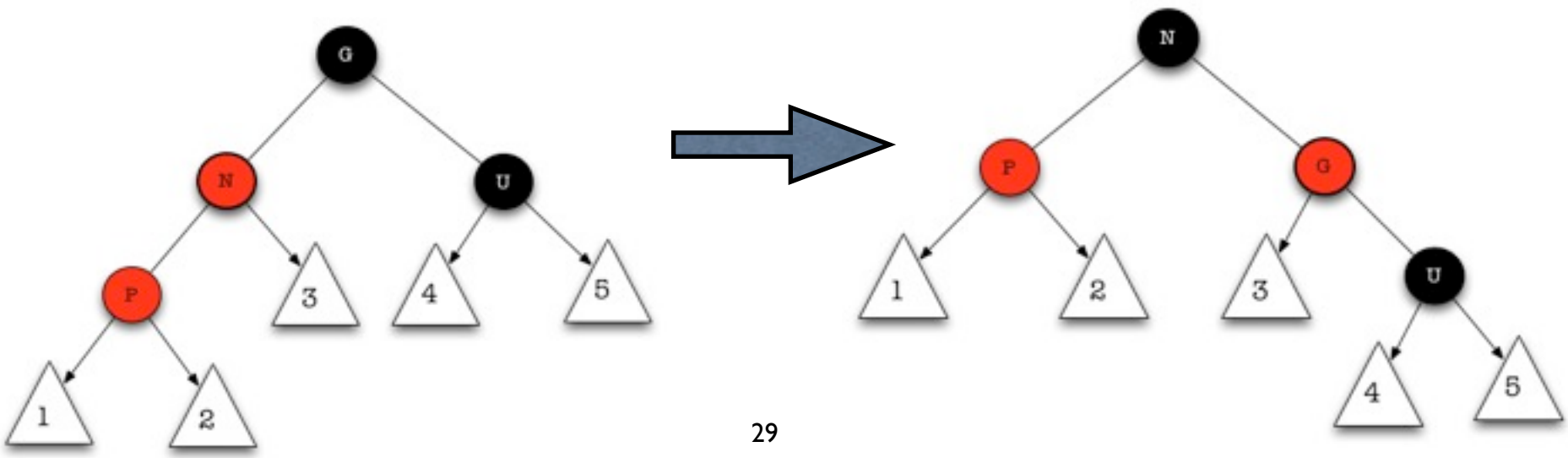
Case study - Red-black tree



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

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

Case study - SSA

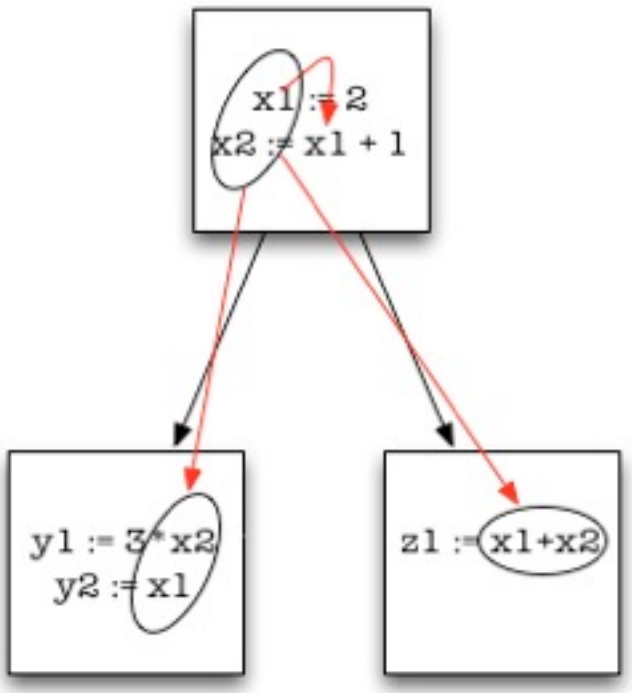
- Def-use domination can be modeled as a structural relation over dominator tree (R_{du}).
- Checking that every use has a corresponding dominating def is equivalent to proving that reflexive closure of a use relation is a subset of the def-use dominator relation.

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

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

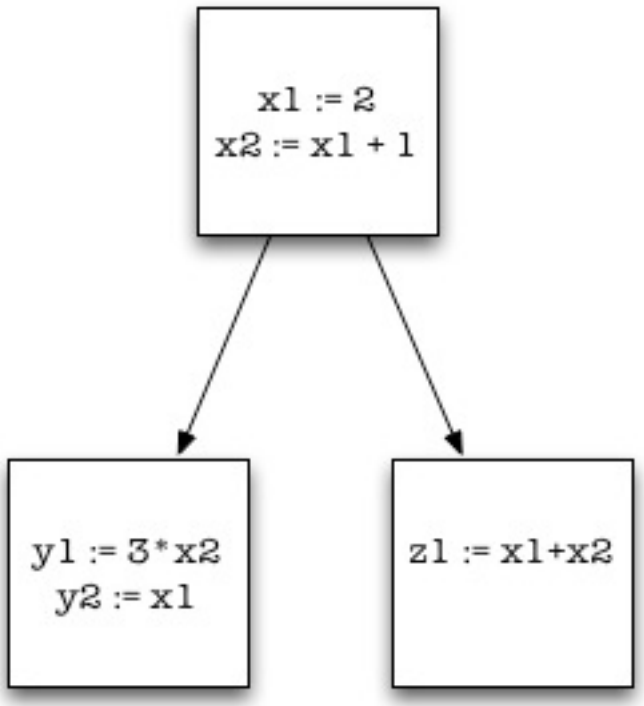
$$R_{use-refl} = \{\langle x1, x1\rangle, \langle x2, x2\rangle\}$$

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



Case study - SSA

- MLton uses SSA as an intermediate language
- Represented internally using dominator tree
- SSASimplify performs multiple optimizations on SSA tree.
  - ★ Example: *remove-unused* pass removes definitions of unused variables from dominator tree.



- SSA is typechecked before and after optimization passes to ensure soundness of optimizations. Typechecking also checks if variable definitions dominate variable uses.
  - Impact on compilation times
  - Verifying specification on SSA shape preservation would remove this overhead

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

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

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

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

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

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

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

R_{use} : {Func.t * Var.t}

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

R_{du} : {Func.t * Var.t * 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

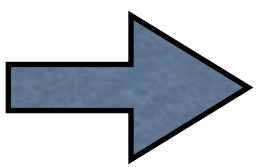
$$\texttt{visitVars} : \{\texttt{t} : \texttt{Func.t}\} \longrightarrow \{\texttt{s} : \texttt{Set.t} \mid R_{set-mem}(\texttt{s}) = R_{use}(\texttt{t})\}$$

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

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

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

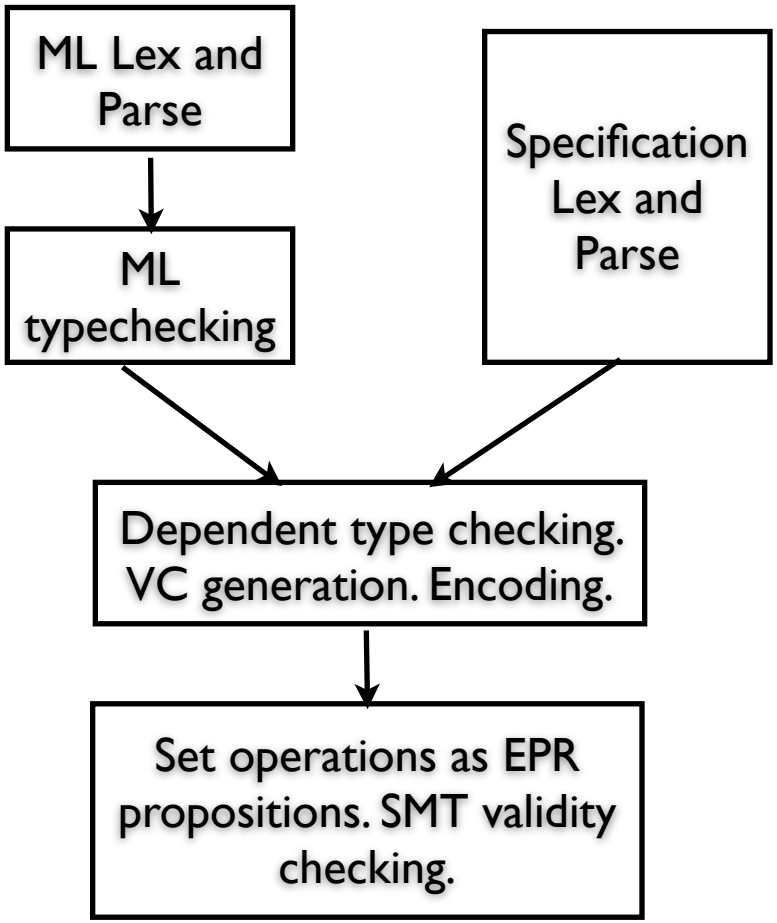
$$\frac{\mathtt{R}_{\mathtt{use-refl}}(\mathtt{t}) \subseteq \mathtt{R}_{\mathtt{du}}(\mathtt{t}) \quad \mathtt{R}_{\mathtt{set-mem}}(\mathtt{s}) = \mathtt{R}_{\mathtt{use}}(\mathtt{t}) \quad \mathtt{R}_{\mathtt{use}}(\mathtt{t}') = \mathtt{R}_{\mathtt{use}}(\mathtt{t})}{\mathtt{R}_{\mathtt{use-refl}}(\mathtt{t}') \subseteq \mathtt{R}_{\mathtt{du}}(\mathtt{t}')}$$



$$\frac{S1.S1 \subseteq S2 \times S1 \quad S3 = S1 \quad S4 = S1}{S4.S4 \subseteq \{S2 \cap S1\} \times S4}$$

Implementation

- Separate lexing and parsing phases for structural relations and relational specifications.
- Verification condition generation (dependent typechecking) is a compiler pass following typechecking phase in the MLton compiler.
- Subtype checking for relational refinements is implication validity checking for formulas in relational algebra.
- Relations translated to sets by encoding each instance of a relation as a unique set.
- Sets and set operations encoded in EPR logic in SMT (Z3).



Extensions

- Reduce annotation burden by deriving lemmas from stated assertions. Eg:

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

$$R^*_{oa}(\nu) = R^*_{ob}(l) \Rightarrow R_{mem}(\nu) = R_{mem}(l)$$ ← Should be deducible

- Exploit inductive properties on sets to improve equational reasoning:

$$R_{ob}(x :: xs) = \{(x)\} \times R_{mem}(xs) \qquad R_{mem}(x :: xs) = \{(x)\} \cup R_{mem}(xs)$$

$$S_1 \times S_2 = S'_1 \times S'_2 \Rightarrow S_1 \cup S_2 = S'_1 \cup S'_2$$

$$\Downarrow$$

$$(S_1 \times S_2)^* = (S'_1 \times S'_2)^* \Rightarrow (S_1 \cup S_2)^* = (S'_1 \cup S'_2)^*$$

Extensions

- Augment solver with direct support for structural relations.
  - ★ Verification procedure can elide translation to sets, and concentrate on typechecking
  - ★ Common substrate of verification strategies that make use of sets as abstract domain can be implemented efficiently once and for all.
- Assertions over inductively defined relations over same datatype can be handled directly by the solver if structural induction scheme can be successfully encoded.
  - ✦ For two well-formed structural relations R_1 and R_2 defined over the same datatype $\mathcal{S}$,

$$R_1 \subseteq R_2^* \Rightarrow R_1 \subseteq (R_2 \cap R_1)^*$$

can be discharged using structural induction over $\mathcal{S}$

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