Refined typechecking with Stardust

Joshua Dunfield

Carnegie Mellon University
(current affiliation: McGill University)

PLPV 2007
Freiburg, Germany

5 October 2007

Conventional Static Typing

- ML, Haskell, ...
- Types express **properties**:
  - *append* : $\mathsf{list} * \mathsf{list} \rightarrow \mathsf{list}$
- Types can be inferred;
  type annotations are documentation
  (except module interfaces)
- "Well typed programs don't go wrong"

Conventional Static Typing

- ML, Haskell, ...
- Types express **properties**:
  - *append* : $\mathsf{list} * \mathsf{list} \rightarrow \mathsf{list}$
- Types can be inferred;
  type annotations are documentation
  (except module interfaces)
- "Well typed programs don't go wrong"
- We extend conventional static typing
  to verify more precise invariants:
  - *append* : $\mathsf{-all}\ a, b : \mathsf{nat}\text{-}\ \underbrace{\mathsf{list}(a)}_{\text{length } a} * \mathsf{list}(b) \rightarrow \mathsf{list}(a{+}b)$

Background

- Part of a larger project...
  - Type assignment system*
  - Tridirectional type system*
  - Let-normal transform + type system
  - Implementation: Stardust ⟵ this paper

* joint work with Frank Pfenning

Outline

- ☞ **Introduction**
  - Integer Domain—Red-Black Trees
  - Dimension Domain
  - Implementation
  - Related Work
  - Summary and Future Work

Setting

- Subset of core Standard ML:
  - First class functions
  - Primitive types **int**, **real**, **string**, ...
  - Algebraic datatypes

$$\textbf{datatype list} = \text{Nil} \mid \text{Cons } \textbf{of int} * \textbf{list}$$

$$\textbf{datatype tree} = \text{Empty} \mid \text{Tree } \textbf{of key} * \textbf{tree} * \textbf{tree}$$

  - ✗ syntactic sugar
  - ✓ patterns, e.g.

```
Black(e, Red lt, Red (rt as (_,_,Red _)))
```

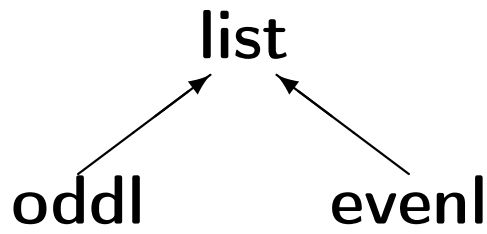
Datasort Refinements

- a.k.a. refinement types [Freeman+Pf. '91, Davies '97,'05]
- Refine an algebraic datatype by a datasort
- Example: Lists of integers

$$\mathsf{Nil} : \mathbf{list} \qquad \mathsf{Nil} : \mathbf{evenl}$$

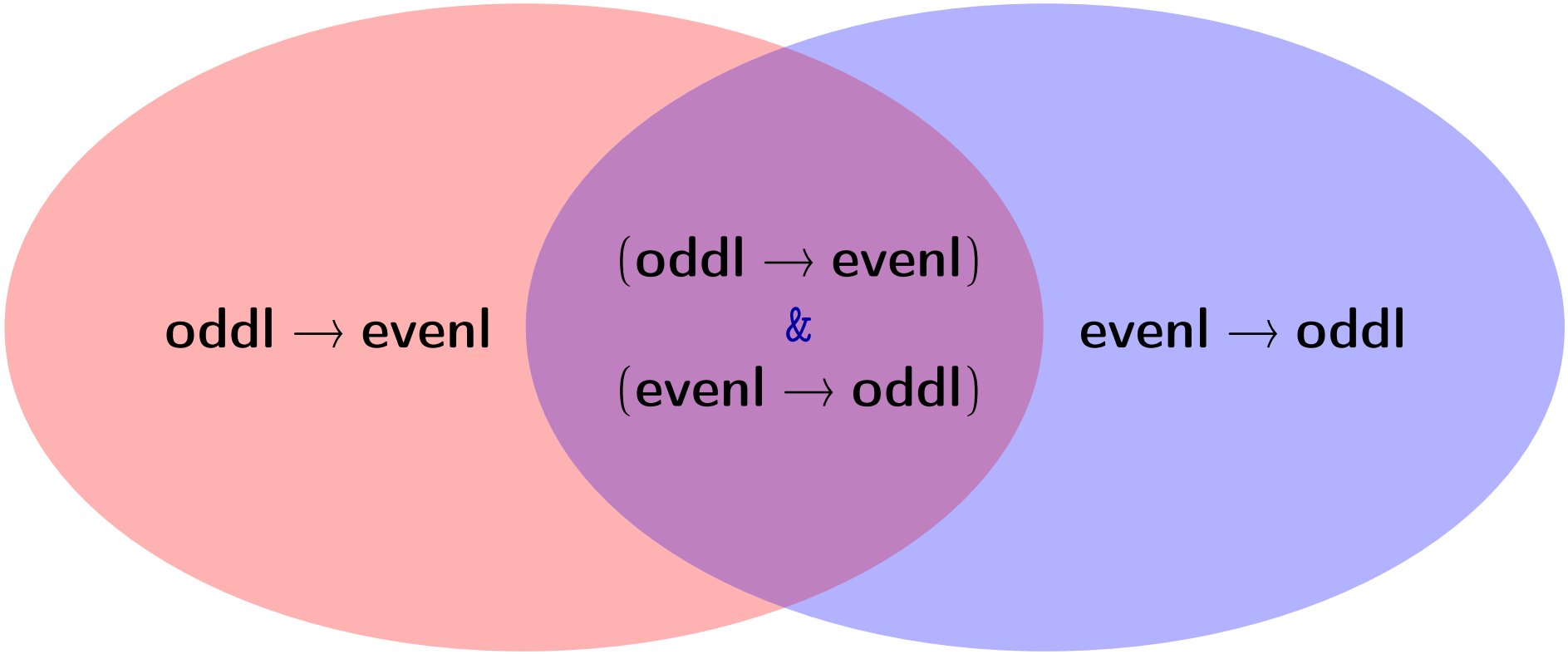
$$\mathsf{Cons} : \mathbf{int} * \mathbf{list} \to \mathbf{list}$$

$$\mathsf{Cons} : (\mathbf{int} * \mathbf{oddl} \to \mathbf{evenl}) \;\&\; (\mathbf{int} * \mathbf{evenl} \to \mathbf{oddl}) \;\&\; (\mathbf{int} * \mathbf{list} \to \mathbf{list})$$



- Intersections $v : A \;\&\; B$: v has type A and type B

Intersection Types ~ Set Intersection



If

$tail : (\mathsf{oddl} \to \mathsf{evenl})\ \&\ (\mathsf{evenl} \to \mathsf{oddl})$

$\mathrm{od} : \mathsf{oddl}$, $ev : \mathsf{evenl}$

then

$tail(\mathrm{od}) : \mathsf{evenl}$ and $tail(ev) : \mathsf{oddl}$

Index Refinements

- Dependent types with restrictions [Xi & Pfenning '99]
- Refine an algebraic datatype by an index
- Indices drawn from a decidable constraint domain
- Example: Lists indexed by their length

$$\mathsf{Nil} : \mathbf{list}(0)$$

$$\mathsf{Cons} : \text{-}\mathbf{all}\ a : \mathsf{nat}\text{-}\ \mathbf{int} * \mathbf{list}(a) \to \mathbf{list}(a+1)$$

$$\mathit{append} : \text{-}\mathbf{all}\ a, b : \mathsf{nat}\text{-}\ \mathbf{list}(a) * \mathbf{list}(b) \to \mathbf{list}(a+b)$$

$$\mathit{filter} : (\mathbf{int} \to \mathbf{bool}) \to \text{-}\mathbf{all}\ a : \mathsf{nat}\text{-}\ \mathbf{list}(a) \to (\text{-}\mathbf{exists}\ b : \mathsf{nat}\text{-}\ [b \leq a]\ \mathbf{list}(b))$$

Index Refinements + Union Types

- **Untagged** unions $v : A \vee B$: v has type A or type B
- Choice function:

  $choose$: -**all** a, b : nat- $\mathbf{list}(a) * \mathbf{list}(b) \rightarrow (\mathbf{list}(a) \vee \mathbf{list}(b))$

  fun *choose* $(xs, ys) =$

  case *random_bit* () **of** true $\Rightarrow xs$

  | false $\Rightarrow ys$

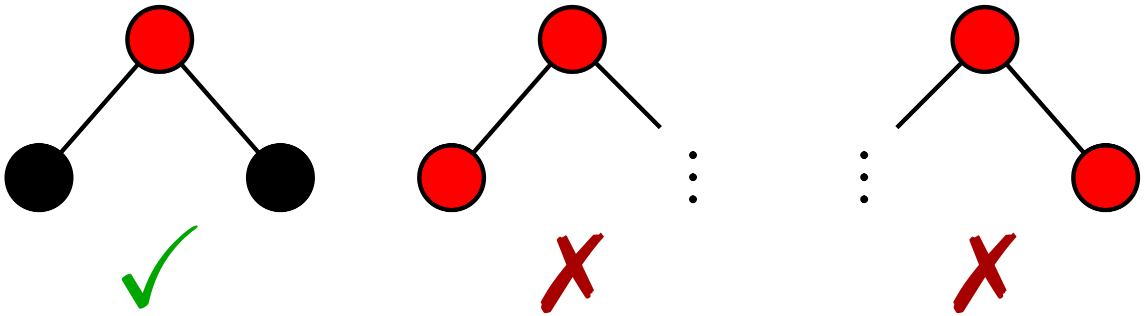
- Assume xs:$\mathbf{list}(a)$, ys:$\mathbf{list}(b)$
- Show body checks against $\mathbf{list}(a) \vee \mathbf{list}(b)$:
  - Show $xs : \mathbf{list}(a) \vee \mathbf{list}(b)$—show $xs : \mathbf{list}(a)$ ✓
  - Show $ys : \mathbf{list}(a) \vee \mathbf{list}(b)$— show $ys : \mathbf{list}(a)$ ✗

    or, show $ys : \mathbf{list}(b)$ ✓

Red-black trees

datatype tree $=$ Empty
$\mid$ Red **of key** $*$ **tree** $*$ **tree**
$\mid$ Black **of key** $*$ **tree** $*$ **tree**

- Invariants:
  (1) Empty nodes are considered black
  (2) The children of a red node must be black

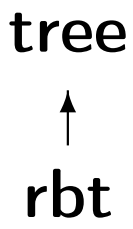


  (3) For every node t, there exists $bh(\mathrm{t})$ s.t. the number of black nodes on **all** paths from t to its leaves is $bh(\mathrm{t})$

Use datasort for invariants (1), (2)

$$\mathsf{tree} \;\Big|\; \begin{array}{l} \mathsf{Empty} : \mathsf{tree} \\ \quad \mathsf{Red} : \mathsf{key} * \mathsf{tree} * \mathsf{tree} \rightarrow \mathsf{tree} \\ \\ \mathsf{Black} : \mathsf{key} * \mathsf{tree} * \mathsf{tree} \rightarrow \mathsf{tree} \end{array}$$

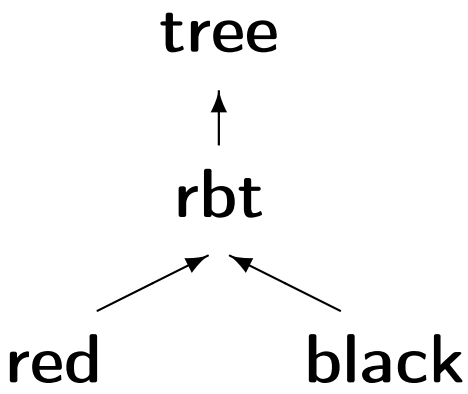
Use datasort for invariants (1), (2)



$$\begin{array}{l} \mathsf{Empty} : \mathbf{tree} \\ \quad \mathsf{Red} : \mathbf{key} * \mathbf{tree} * \mathbf{tree} \rightarrow \mathbf{tree} \\ \\ \mathsf{Black} : \mathbf{key} * \mathbf{tree} * \mathbf{tree} \rightarrow \mathbf{tree} \end{array}$$

- Add **rbt** to represent valid red-black trees

Use datasort for invariants (1), (2)



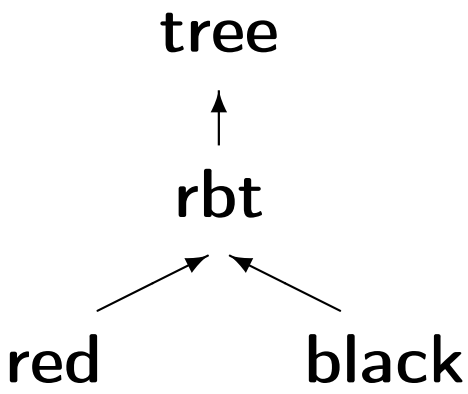
$$\mathsf{Empty} : \mathbf{tree}$$

$$\mathsf{Red} : \mathbf{key} * \mathbf{tree} * \mathbf{tree} \longrightarrow \mathbf{tree}$$

$$\mathsf{Black} : \mathbf{key} * \mathbf{tree} * \mathbf{tree} \longrightarrow \mathbf{tree}$$

- Add **rbt** to represent valid red-black trees
- Add **red**, **black** to distinguish colors

Use datasort for invariants (1), (2)



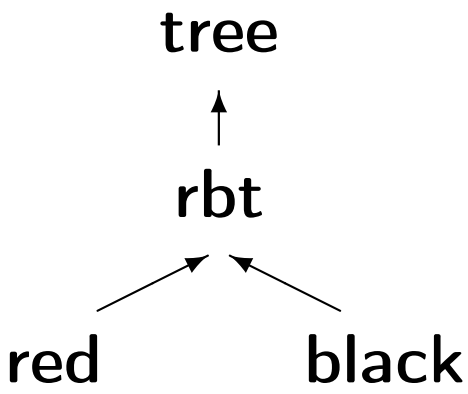
Empty : **black**

Red : **key** $*$ **tree** $*$ **tree** $\longrightarrow$ **tree**

Black : **key** $*$ **tree** $*$ **tree** $\longrightarrow$ **tree**

- Add **rbt** to represent valid red-black trees
- Add **red**, **black** to distinguish colors
- Invariant (1): Empty nodes are considered black

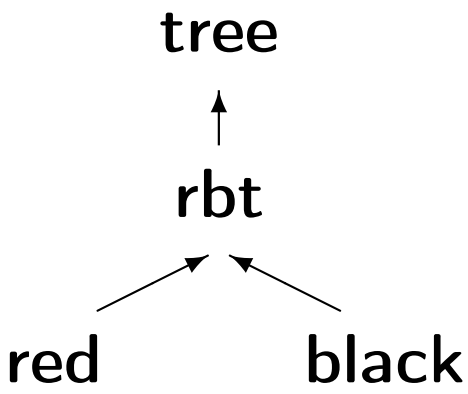
Use datasort for invariants (1), (2)



$$\begin{aligned}
\mathsf{Empty} &: \mathbf{black} \\
\mathsf{Red} &: \mathbf{key} * \mathbf{tree} * \mathbf{tree} \longrightarrow \mathbf{tree} \\
&\quad \& \ \mathbf{key} * \mathbf{black} * \mathbf{black} \longrightarrow \mathbf{red} \\
\mathsf{Black} &: \mathbf{key} * \mathbf{tree} * \mathbf{tree} \longrightarrow \mathbf{tree} \\
&\quad \& \ \mathbf{key} * \mathbf{rbt} * \mathbf{rbt} \longrightarrow \mathbf{black}
\end{aligned}$$

- Add **rbt** to represent valid red-black trees
- Add **red**, **black** to distinguish colors
- Invariant (1): Empty nodes are considered black
- Invariant (2): The children of a red node must be black

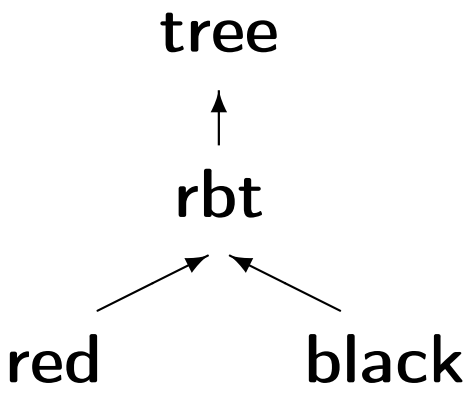
Use index for invariant (3)



$$\begin{array}{l}
\mathsf{Empty}\ :\ \mathbf{black} \\
\mathsf{Red}\ : \\
\qquad \mathbf{key} * \mathbf{tree} \quad * \mathbf{tree} \quad \longrightarrow \mathbf{tree} \\
\qquad \& \ \mathbf{key} * \mathbf{black} \quad * \mathbf{black} \quad \longrightarrow \mathbf{red} \\
\mathsf{Black}\ : \\
\qquad \mathbf{key} * \mathbf{tree} \quad * \mathbf{tree} \quad \longrightarrow \mathbf{tree} \\
\qquad \& \ \mathbf{key} * \mathbf{rbt} \quad * \mathbf{rbt} \quad \longrightarrow \mathbf{black}
\end{array}$$

- Invariant (3): For every node t, there exists $bh(\mathrm{t})$ s.t. the number of black nodes on **all** paths from t to its leaves is $bh(\mathrm{t})$

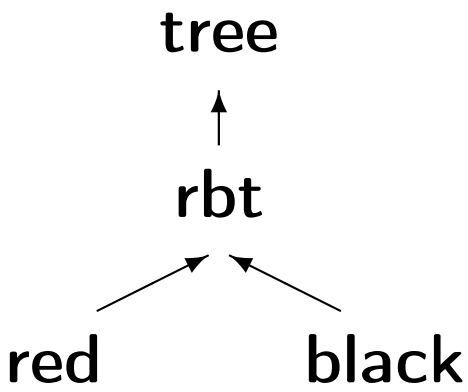
Use index for invariant (3)



$$
\begin{array}{l}
\mathsf{Empty} \ : \ \mathsf{black} \\
\mathsf{Red} : \\
\qquad \mathsf{key} * \mathsf{tree} \quad * \mathsf{tree} \quad \longrightarrow \mathsf{tree} \\
\qquad \& \ \mathsf{key} * \mathsf{black} \quad * \mathsf{black} \quad \longrightarrow \mathsf{red} \\
\mathsf{Black} : \\
\qquad \mathsf{key} * \mathsf{tree} \quad * \mathsf{tree} \quad \longrightarrow \mathsf{tree} \\
\qquad \& \ \mathsf{key} * \mathsf{rbt} \quad * \mathsf{rbt} \quad \longrightarrow \mathsf{black}
\end{array}
$$

- Invariant (3): For every node t, there exists $bh(\mathrm{t})$ s.t. the number of black nodes on **all** paths from t to its leaves is $bh(\mathrm{t})$
- Index by the black height $bh(\mathrm{t})$, a natural number

Use index for invariant (3)



Empty : **black**(0)

Red : -**all** h:nat-

key ∗ **tree**(h) ∗ **tree**(h) → **tree**(h)

& **key** ∗ **black**(h) ∗ **black**(h) → **red**(h)

Black : -**all** h:nat-

key ∗ **tree**(h) ∗ **tree**(h) → **tree**(h + 1)

& **key** ∗ **rbt**(h) ∗ **rbt**(h) → **black**(h + 1)

- Invariant (3): For every node t, there exists $bh(\mathrm{t})$ s.t. the number of black nodes on **all** paths from t to its leaves is $bh(\mathrm{t})$
- Index by the black height $bh(\mathrm{t})$, a natural number

What the user writes

```
(*[
 datasort     rbt < tree;
              red < rbt; black < rbt

 datacon Empty : black(0)
 datacon Red   : -all h:nat-
                    key * tree(h) * tree(h) -> tree(h)
                  & key * black(h) * black(h) -> red(h)
 datacon Black : -all h:nat-
                    key * tree(h) * tree(h) -> tree(h+1)
                  & key * rbt(h) * rbt(h) -> black(h+1)
]*)
```

Index Domains

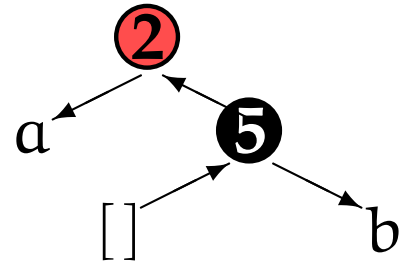
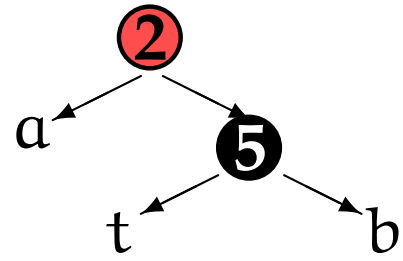
- Integers int
  - linear inequalities
- Booleans bool
- Dimensions dim

- ✓ Introduction
- ☞ **Integer Domain—Red-Black Trees**
  - Dimension Domain
  - Implementation
  - Related Work
  - Summary and Future Work

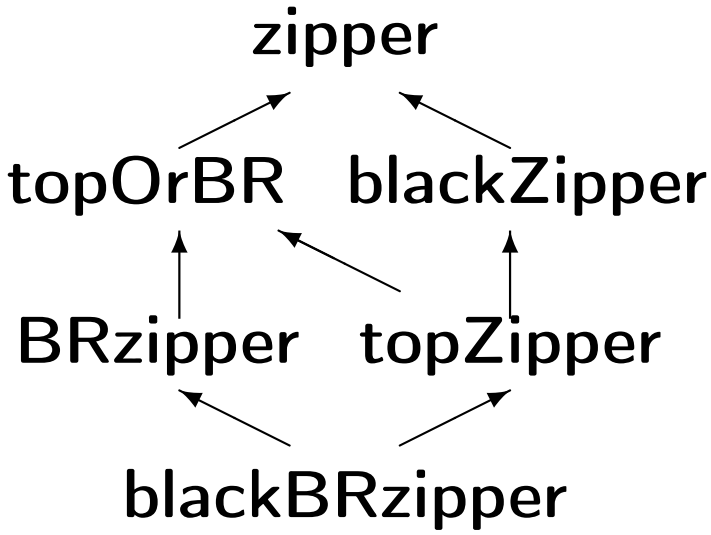
Integers: Red-Black Trees

- Insertion (with slightly more complex datasorts)
- Deletion: SML/NJ library
  - Found two bugs,
    each violating the color invariant
  - Buggy module looks correct to clients
  - After bugs fixed, typechecks

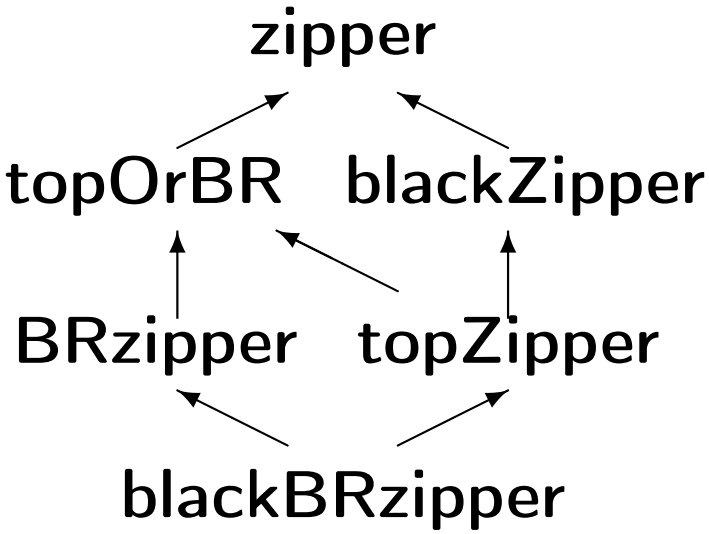
Zipper

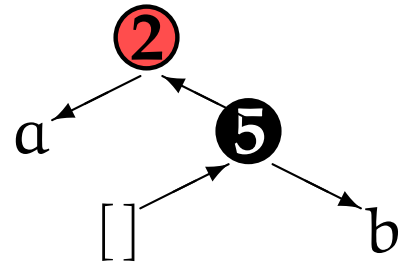
z	$[\,]$	
	TOP	$\mathsf{LEFTB}(\mathsf{RIGHTR}(a, 2, \mathsf{TOP}), 5, b)$
$zip(z, t)$	t	
	t	$\mathsf{Red}(2, a, \mathsf{Black}(5, t, b))$

Datasort Refinement

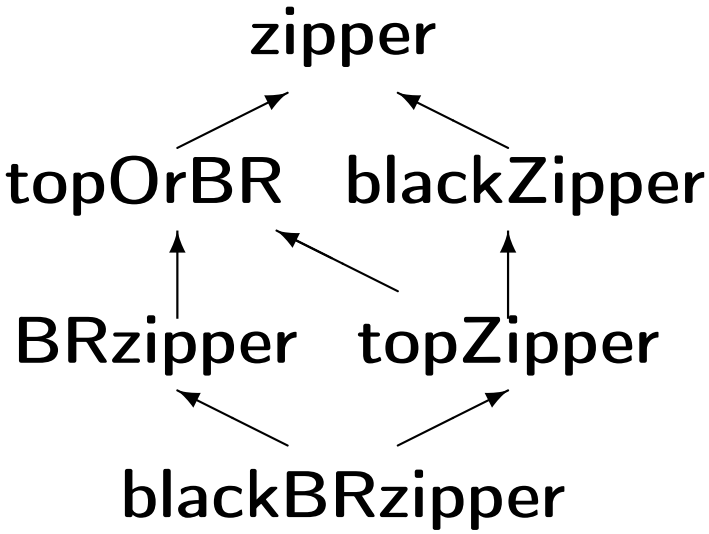


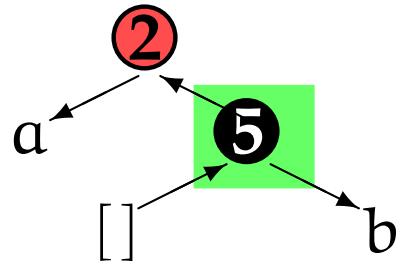
Datasort Refinement





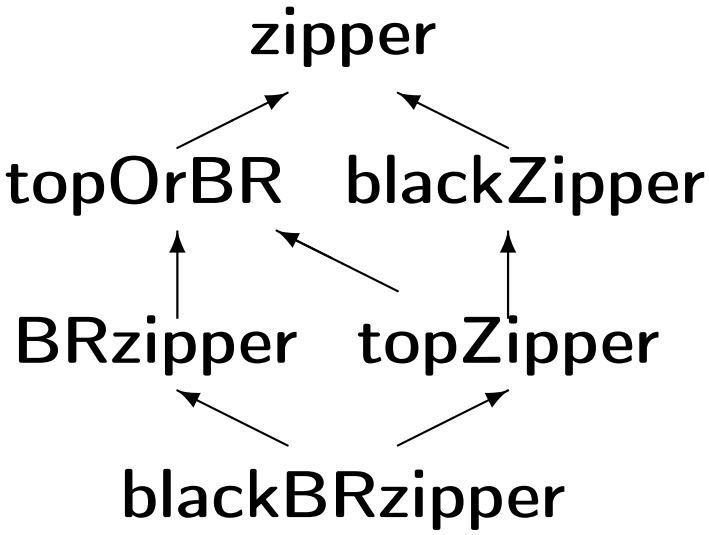
Datasort Refinement

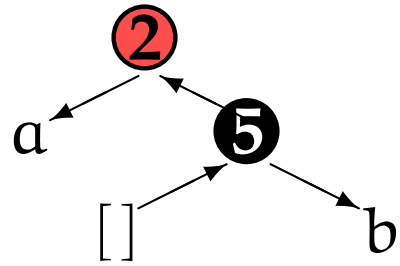




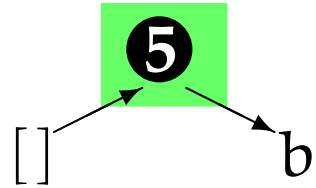
: blackZipper

Datasort Refinement

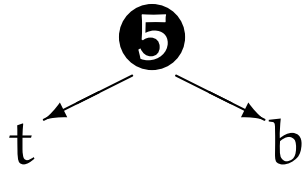




: blackZipper



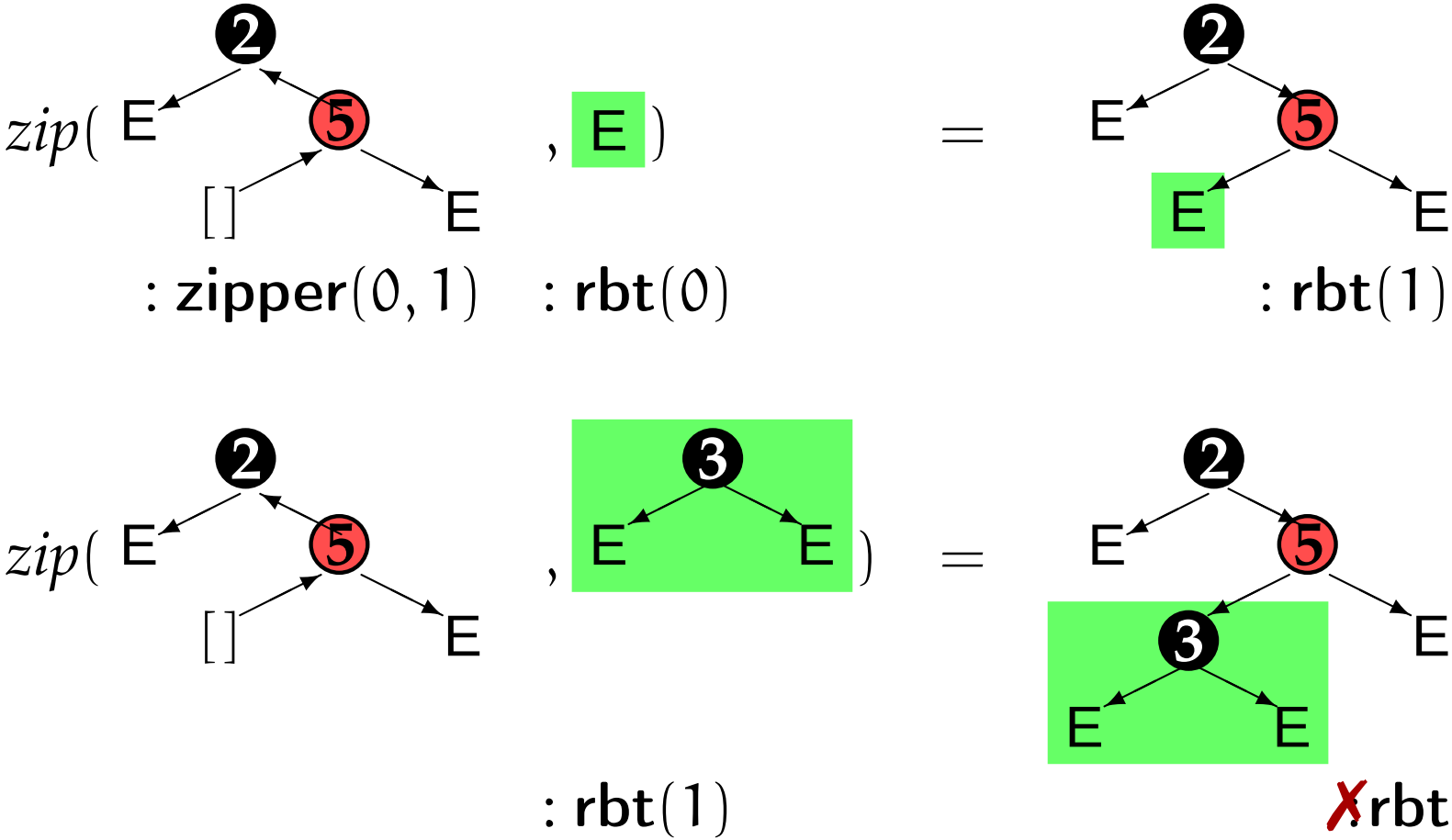
: BRzipper



: black

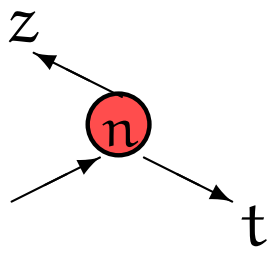
Index Refinement: Height When Zipped

z : **zipper**(h, hz) when $zip(z, t)$ yields **rbt**(hz),
where t : **rbt**(h)



Zipper Constructor Types

$$\begin{aligned}
&\textbf{datacon}\ \mathsf{TOP} : \text{-}\textbf{all}\ \mathrm{h}\!:\!\mathsf{nat}\text{-}\ \mathsf{topZipper}(\mathrm{h},\mathrm{h})\\
&\qquad\qquad\vdots\\
&\textbf{datacon}\ \mathsf{LEFTR} : \text{-}\textbf{all}\ \mathrm{h},\mathrm{hz}\!:\!\mathsf{nat}\text{-}\\
&\qquad\qquad\quad \mathsf{int} * \mathsf{black}(\mathrm{h}) * \mathsf{blackZipper}(\mathrm{h},\mathrm{hz})\\
&\qquad\qquad \rightarrow \mathsf{zipper}(\mathrm{h},\mathrm{hz})\\
&\qquad\quad \&\quad \mathsf{int} * \mathsf{black}(\mathrm{h}) * \mathsf{blackBRzipper}(\mathrm{h},\mathrm{hz})\\
&\qquad\qquad \rightarrow \mathsf{BRzipper}(\mathrm{h},\mathrm{hz})
\end{aligned}$$



$\mathsf{LEFTR}(\mathrm{n},\mathrm{t},z)$ where $\mathrm{t} : \mathsf{black}(\mathrm{h})$

- ✓ Introduction
- ✓ Integer Domain—Red-Black Trees
- ☞ **Dimension Domain**
  - Implementation
  - Related Work
  - Summary and Future Work

Dimension Index Sort dim

- Unit confusion a significant factor
  in the loss of the Mars Climate Orbiter

- Units `M`, `S`, `KG`
- Multiplication and powers:

```
M*KG    M^2    S^a
```

- Multiplicative identity *1*
- **real** indexed by dim

Dimensions Example

```
(*[ val power : -all  d:dim- -all  n:int-
                  int(n) → real(d) → real(d^n)
]*)
fun power n x =
  if  n = 0 then
    1.0
  else  if  n < 0 then
    1.0 / power (~n) x
  else
    x * power (n-1) x
```

Dimensions: Invaluable Refinement

Unlike e.g. red-black trees,
our dimension refinements are **not** tied to values:

- For every d, each type **real**(d) is inhabited by
  exactly the same set of floating-point values:

$$\begin{aligned} \texttt{3.1} &: \mathbf{real}(1) \\ \texttt{3.1} * M &: \mathbf{real}(\texttt{M}) \\ \texttt{3.1} * S &: \mathbf{real}(\texttt{S}) \\ \texttt{3.1} * M * M &: \mathbf{real}(\texttt{M} \hat{} 2) \end{aligned}$$

  All these expressions yield identical values at runtime!

- Also not based on values: qualified types [Foster],
  phantom types [Cheney/Hinze, Fluet/Pucella, . . .],
  state refinement [Mandelbaum et al., . . .]

Dimensions: Related Work

- ’70s and ’80s
- Kennedy ’96: Polymorphism, type inference
- Allen ’04: abelian classes in Java

- ✓ Introduction
- ✓ Integer Domain—Red-Black Trees
- ✓ Dimension Domain
- ☞ **Implementation**
  - Related Work
  - Summary and Future Work

Constraint Solving

- Delegate most work to external tools:
  ICS or CVC Lite
  - Accumulate constraints incrementally
- Solve for existential index variables
- Reduce dimension constraints to integer constraints

(Almost) Ideal Constraint Solver Interface

1. Persistent solver contexts Ω
   encapsulating index-level assumptions
2. $\textsc{assert}(\Omega, \mathsf{P})$ returning

$$\begin{array}{rll} \mathsf{Valid} & \text{if} & \Omega \models \mathsf{P} \\ \mathsf{Invalid} & \text{if} & \Omega, \mathsf{P} \models \bot \\ \mathsf{Contingent}(\Omega') & \text{if} & \Omega \not\models \mathsf{P} \text{ and } \Omega, \mathsf{P} \not\models \bot, \\ & & \text{yielding a new context } \Omega' = \Omega, \mathsf{P} \end{array}$$

3. $\textsc{valid}(\Omega, \mathsf{P})$ returning

$$\begin{array}{rll} \mathsf{Valid} & \text{if} & \Omega \models \mathsf{P} \\ \mathsf{Invalid} & \text{otherwise.} & \end{array}$$

4. $\textsc{declare}(\Omega, \mathfrak{a}, \mathit{sort})$

ICS

- Developed at SRI (de Moura, Rueß, Shankar, . . .)
- Integers (almost), rationals, bitvectors (not really), functional arrays, . . .
- Provides the desired interface
- Fast

- **Future:**
  - Closed source
  - Deprecated, replaced by Yices
  - For our purposes:
    ICS a great improvement on its successor

ICS

- Developed at SRI (de Moura, Rueß, Shankar, . . .)
- Integers (almost), rationals, bitvectors (not really), functional arrays, . . .
- Provides the desired interface
- Fast
  except when it hangs
- **Future:**
  - Closed source
  - Deprecated, replaced by Yices
  - For our purposes:
    ICS a great improvement on its successor

CVC Lite

- Barrett, Berezin, [Dill], [Tinelli], . . .
- Integers, rationals, bitvectors, functional arrays, . . .
- Does not provide persistent contexts
- Slower than ICS, but more stable
- **Future:**
  - Deprecated, replaced by CVC 3
  - Open source

A nontraditional application?

- Most satisfiability-modulo-theories systems aimed at large problems
- Stardust needs to solve many small problems
- {Correctness, variety of theories} $\gg$ speed

Typechecking Time

Input program		Wall-clock time in seconds		
		ICS	CVC Lite library	CVC Lite stand-alone
redblack-full	δ(i) &	1.9	8.2	9.2
redblack-full-bug1	δ(i) &	1.6	6.8	8.1
redblack	δ(i) &	< 1	< 1	< 1
rbdelete	δ(i) &∨	*	37.7	31.6
bits	δ(i) &∨	*	9.5	4.0
bits-un	δ(i) &∨	33.5	298.5	241.4
others	δ(i)	< 1	< 1	< 1

Memoization

- Xi's DML:
  index refinements
  No $\&\vee$: typecheck in single pass,
  don't need to memoize
- Davies' SML-CIDRE:
  datasort refinements, $\&$: backtracking
  No index refinements—no Σ:
  memoization easy
- Stardust:
  datasort + index, $\&\vee$
  memoization ineffective
  - might work with sophisticated irrelevance

- ✓ Introduction
- ✓ Integer Domain—Red-Black Trees
- ✓ Dimension Domain
- ✓ Implementation
- ☞ **Related Work**
- Summary and Future Work

Related Work

- [Davies '97/'05, Davies & Pf. '00]: Bidirectional system with δ, $\&$
- [Xi & Pfenning '99]: Bidirectional, with $\mathfrak{i}$, -**all**-, -**exists**-
- [Coppo et al. '81, Amadio '98]: $\&$ can characterize normal forms; hence full inference undecidable
- [Reynolds '96]: FORSYTHE with $\&$ (and type annotations)
- [Pierce '91]: Language with $\&$, $\vee$, syntactic markers
- Dependent types ((less) restricted): Cayenne, Epigram, [Chen & Xi '05], [Licata & Harper '05], ...

- ✓ Introduction
- ✓ Integer Domain—Red-Black Trees
- ✓ Dimension Domain
- ✓ Implementation
- ✓ Related Work
- ☞ **Summary and Future Work**

Summary

- Type system
  - Combination of datasort and index refinements, marker-free intersections, unions, quantifiers
- Data structure refinements, e.g. bitstrings, red-black trees
- Invaluable refinements, e.g. dimensions
- Implementation
  - Delegated constraint solving
  - Speed leaves. . . room for improvement

Future Work

- Parametric polymorphism
- Full SML (modules!)
- Index domains: bit vectors, inductive families, finite sets, regular languages, ...
- Memoization and parallelization
- Refinement-based compilation
- Call-by-name (and -need)
- Proof (derivation) / Counterexample generation
- Suggestion tools

http://type-refinements.info

Extra slides

Datasort Refinements

$tail$: $(\mathsf{evenl} \rightarrow \mathsf{oddl})$ & $(\mathsf{oddl} \rightarrow \mathsf{evenl})$ & $(\mathsf{list} \rightarrow \mathsf{list})$

fun *tail* xs =

case xs **of** Nil $\Rightarrow$ **raise** *Error*

| Cons(h, t) $\Rightarrow$ t

- First conjunct $(\mathsf{evenl} \rightarrow \mathsf{oddl})$:
  - Assume xs : **evenl**
  - Show **raise** *Error* : **oddl** and t : **oddl**
- Second conjunct $(\mathsf{oddl} \rightarrow \mathsf{evenl})$:
  - Assume xs : **oddl**
  - Show t : **evenl**

Parametric Polymorphism: Declarative

- As property type

$$\frac{\Gamma \vdash e \uparrow \Pi a{:}\gamma.\, A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e \uparrow [i/a]\, A}\ \Pi E$$

$$\frac{\Gamma \vdash e \uparrow \forall \alpha.\, B \qquad \Gamma \vdash A\ [\text{mono}]\text{type}}{\Gamma \vdash e \uparrow [A/\alpha]\, B}$$

Greed and Failure

- Generate existential variable
- The greedy method [Cardelli '93]
- $\mathit{choose} : \forall\alpha.\ \alpha \to \alpha \to \alpha$

$$\begin{array}{l}\textbf{fun}\ \mathit{choose}\ x\ y = \\ \quad \textbf{case}\ \mathit{random\_bit}\,()\ \textbf{of}\ \mathsf{true} \Rightarrow x \\ \qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{false} \Rightarrow y\end{array}$$

$$x : A, y : B \vdash \mathit{choose}\ x\ y \uparrow \cdots$$

Greed and Failure

- Generate existential variable
- The greedy method [Cardelli '93]
- $\mathit{choose} : \forall\alpha.\ \alpha \to \alpha \to \alpha$

$$\begin{array}{l}\textbf{fun}\ \mathit{choose}\ x\ y = \\ \quad \textbf{case}\ \mathit{random\_bit}\,()\ \textbf{of}\ \mathsf{true} \Rightarrow x \\ \qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{false} \Rightarrow y\end{array}$$

$x : A, y : B \vdash \mathit{choose}\ x\ y \uparrow \cdots$

- $A \leq B$ and $B \not\leq A$: $y \downarrow A$ ✗

Greed and Failure

- Generate existential variable
- The greedy method [Cardelli '93]
- $\mathit{choose} : \forall\alpha.\ \alpha \to \alpha \to \alpha$

$$\begin{array}{l}\textbf{fun}\ \mathit{choose}\ x\ y = \\ \quad \textbf{case}\ \mathit{random\_bit}\,()\ \textbf{of}\ \mathsf{true} \Rightarrow x \\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{false} \Rightarrow y\end{array}$$

$x : A, y : B \vdash \mathit{choose}\ x\ y \uparrow \cdots$

- $A \leq B$ and $B \not\leq A$: $\quad y \downarrow A$ ✗
- $A \not\leq B$ and $B \leq A$: $\quad y \downarrow A$ ✓

Greed and Failure

- Generate existential variable
- The greedy method [Cardelli '93]
- $\mathit{choose} : \forall\alpha.\ \alpha \to \alpha \to \alpha$

$$\begin{array}{l}\textbf{fun}\ \mathit{choose}\ x\ y = \\ \quad \textbf{case}\ \mathit{random\_bit}\,()\ \textbf{of}\ \mathsf{true} \Rightarrow x \\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{false} \Rightarrow y\end{array}$$

$x : A, y : B \vdash \mathit{choose}\ x\ y \uparrow \cdots$

- $A \leq B$ and $B \not\leq A$: $y \downarrow A$ ✗
- $A \not\leq B$ and $B \leq A$: $y \downarrow A$ ✓
- $A \not\leq B$ and $B \not\leq A$: $y \downarrow A$ ✗

Unionism

- The greedy method seems to fail…

Unionism

- The greedy method seems to fail…
- If only we could get an upper bound of A and B!

Unionism

- The greedy method seems to fail…
- If only we could get an upper bound of A and B!

  $$A \vee B$$

  Instantiating $\alpha = A \vee B$:

  $x : A, y : B \vdash \mathit{choose}\ x\ y \uparrow \cdots$ ✓

Typechecker, Phone Home

- Let's not break the subformula property

Typechecker, Phone Home

- Let's not break the subformula property
- Instead, let the user write the union:

  `(*[` $\quad \mathit{choose} : \forall\alpha.\, \forall\beta.\, \alpha \to \beta \to (\alpha \vee \beta) \quad$ `]*)`

Typechecker, Phone Home

- Let's not break the subformula property
- Instead, let the user write the union:

  $(*[\quad \mathit{choose} : \forall\alpha.\,\forall\beta.\,\alpha \to \beta \to (\alpha \vee \beta) \quad]*)$

  $x : A, y : B \vdash \mathit{choose}\ x\ y \uparrow \cdots$
  $\alpha = A$
  $\beta = B$

Typechecker, Phone Home

- Let's not break the subformula property
- Instead, let the user write the union:

  $(*[\quad \mathit{choose} : \forall\alpha.\,\forall\beta.\,\alpha \to \beta \to (\alpha \vee \beta) \quad]*)$

  $x : A, y : B \vdash \mathit{choose}\ x\ y \uparrow \quad A \vee B \quad \checkmark$
  $\alpha = A$
  $\beta = B$