

Applying Type Refinements

Joshua Dunfield

10 May 2005

Revised version of a talk given 6 May 2005

Student Seminar Series @ Carnegie Mellon University

Joint work with Frank Pfenning

Conventional static typing

- Used in ML, Haskell, . . .
- “Well typed programs don’t go wrong”
- Types express **properties**:
 - $insert : tree * key \rightarrow tree$
 - $map : \forall \alpha, \beta. (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}$
- Types can be inferred;
type annotations are documentation
(except module interfaces)

Conventional static types

express **coarse** properties:

- $insert : tree * key \rightarrow tree$

fun *insert* (t, k) = Empty

- Intended property: $insert(t, k)$ inserts k into t
- Checked property: $insert$ returns a tree

Conventional static types

express **coarse** properties:

- $insert : tree * key \rightarrow tree$

fun *insert* (t, k) = Empty

- Intended property: *insert*(t, k) inserts k into t
- Checked property: *insert* returns a tree
- All trees look the same
- How can we **refine** the type *tree*?
- In general: How can we refine datatypes so more exact properties can be checked?

Refined static typing

- Static typing is useful, but properties checked are too coarse
- How can we **extend** static typing to check more interesting properties of (functional) programs?
 - **Datasort** refinements
 - **Index** refinements
 - **Intersections** and **unions** to combine properties
- Type inference becomes impractical
- Type annotations are specifications, checked at compile time

The rest of the talk

- Centers on **examples**
- Focuses on refinements from a user's perspective
- Describes typechecking informally through examples
- A tutorial—**not** a proto-job talk

Outline

☞ **Datasort Refinements**

- Index Refinements
- Implementation
- (Demo...)
- (Related) Work
- Summary
- Future Work

Starting point: Datatypes

- Inductively defined datatypes:

datatype list = Nil
 | Cons **of** int * list

datatype bits = Empty
 | Zero **of** bits
 | One **of** bits

datatype tree = Empty
 | Red **of** key * tree * tree
 | Black **of** key * tree * tree

Datasort refinement [Freeman & Pf., Davies & Pf.]

```
datatype list = Nil  
              | Cons of int * list
```

Consider properties expressible in terms of constructors:

- a list is **empty** iff it is Nil
- a list is **nonempty** iff it is not Nil, i.e. is Cons(_, _)

Call “**empty**” and “**nonempty**” **datasorts**.

Viewing types as sets, **empty** \subset **list** and **nonempty** \subset **list**.

Datasort refinement [Freeman & Pf., Davies & Pf.]

```
datatype list = Nil  
              | Cons of int * list
```

Consider properties expressible in terms of constructors:

- a list is **empty** iff it is Nil
- a list is **nonempty** iff it is not Nil, i.e. is Cons(_, _)

Call “**empty**” and “**nonempty**” **datasorts**.

Viewing types as sets, **empty** \subset **list** and **nonempty** \subset **list**.

tail : **list** \rightarrow **list** —Raises **Match** exception if applied to Nil

Datasort refinement [Freeman & Pf., Davies & Pf.]

```
datatype list = Nil  
              | Cons of int * list
```

Consider properties expressible in terms of constructors:

- a list is **empty** iff it is Nil
- a list is **nonempty** iff it is not Nil, i.e. is Cons(_, _)

Call “**empty**” and “**nonempty**” **datasorts**.

Viewing types as sets, **empty** \subset **list** and **nonempty** \subset **list**.

```
tail : list  $\rightarrow$  list      —Raises Match exception if applied to Nil  
tail : nonempty  $\rightarrow$  list   —Cannot raise Match
```

A little more interesting

```
datatype list = Nil
              | Cons of int * list
```

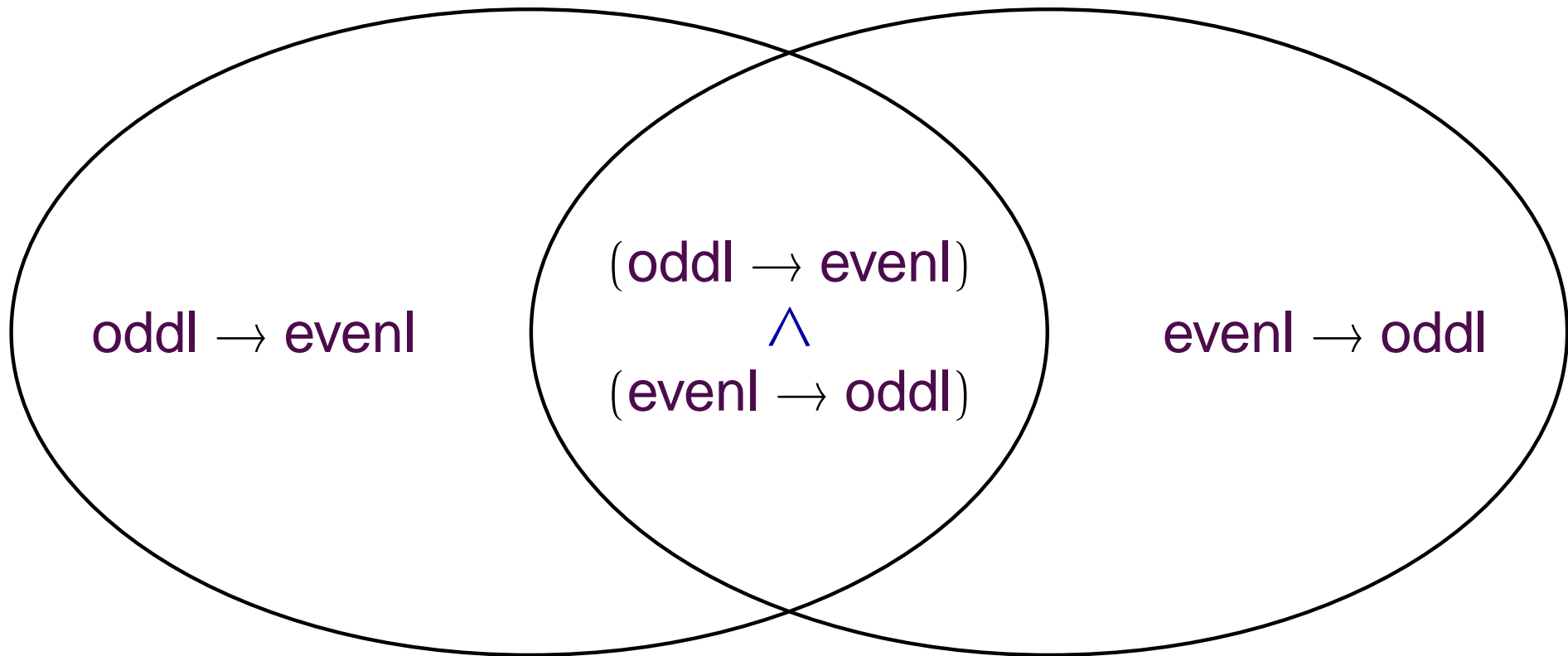
Datasorts that depend on the arguments:

- a list is **evenl** iff it is Nil or Cons($_$, t) where t is **oddl**
- a list is **oddl** iff it is Cons($_$, t) where t is **evenl**

Again, **evenl** \subset **list** and **oddl** \subset **list**.

```
tail : (oddl  $\rightarrow$  evenl)
       $\wedge$  (evenl  $\rightarrow$  oddl)
```

Intersection types ~ set intersection



If

$\text{tail} : (\text{oddl} \rightarrow \text{evenl}) \wedge (\text{evenl} \rightarrow \text{oddl})$

$\text{od} : \text{oddl}, \quad \text{ev} : \text{evenl}$

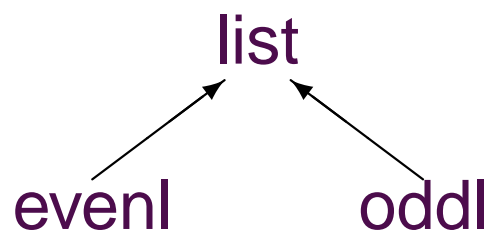
then

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

Specifics of specification

- a list is **evenl** iff it is Nil or Cons($_$, t) where t is **oddl**
- a list is **oddl** iff it is Cons($_$, t) where t is **evenl**

We can turn this into a formal spec by giving a **subsort relation** and types for Nil and Cons:



Nil : evenl

Cons : (int * oddl \rightarrow evenl)

\wedge (int * evenl \rightarrow oddl)

\wedge (int * list \rightarrow list)

- Subtyping based on the subsort relation:
from evenl \leq list, oddl \leq list we have
list \rightarrow (evenl * list) \leq oddl \rightarrow (list * list)

Typechecking

$tail : (oddl \rightarrow evenl) \wedge (evenl \rightarrow oddl)$

fun *tail* $x_s =$

case x_s **of** $Cons(h, t) \Rightarrow t$
| $Nil \Rightarrow$ **raise** *Match*

- To check first conjunct ($oddl \rightarrow evenl$):
 - Assume $x_s : oddl$
 - Show $t : evenl$
- To check second conjunct ($evenl \rightarrow oddl$):
 - Assume $x_s : evenl$
 - Show $t : oddl$ and **raise** *Match* : $oddl$

Example: Formulas

datatype formula = Lit of string
| And of formula * formula
| Or of formula * formula

$x \wedge (y \vee z)$ is represented by `And(Lit "x", Or(Lit "y", Lit "z"))`.

Suppose that **part** of a program works only on formulas in conjunctive normal form:

$x \wedge (y \vee z)$ ✓
 $(x \wedge y) \wedge (y \vee z)$ ✓
 $x \wedge ((y_1 \wedge y_2) \vee z)$ ✗
 $y \vee z$ ✓

How can we check this statically?

You don't need type refinements

We could make up new datatypes:

```
datatype disj = DLit of string  
              | DOr of disj * disj  
and cnf = CnfLit of string  
          | CnfAnd of cnf * cnf  
          | CnfOr of disj * disj
```

- Values of type `disj` cannot include \wedge
- Need to write functions to convert `formula` \Leftrightarrow `cnf`
- Code duplication (e.g. a substitution function), or new abstractions

...but you want type refinements

formula	Lit : string \rightarrow disj
↑	And : (cnf * cnf \rightarrow cnf)
cnf	∧ (formula * formula \rightarrow formula)
↑	Or : (disj * disj \rightarrow disj)
disj	∧ (formula * formula \rightarrow formula)

- Need to annotate functions with their refined types
- But there's no need for coercions or code duplication

Datasort refinements are useful

- Datasorts express **properties of datatypes**
- In function types, they express **properties of functions**
- **Intersection types** express conjunction of properties
- Extra datatype definitions can achieve similar results
 - But code must be duplicated
 - With datasorts, just annotate existing code

Limitations of datasorts

- Set of datasorts must be **finite**
- Datasorts equivalent to finite tree automata
- Datasort refinements can only count as high as the number of datasorts


✓ Datasort Refinements

☞ **Index Refinements**

- Implementation
- (Demo...)
- (Related) Work
- Summary
- Future Work

Index Refinements

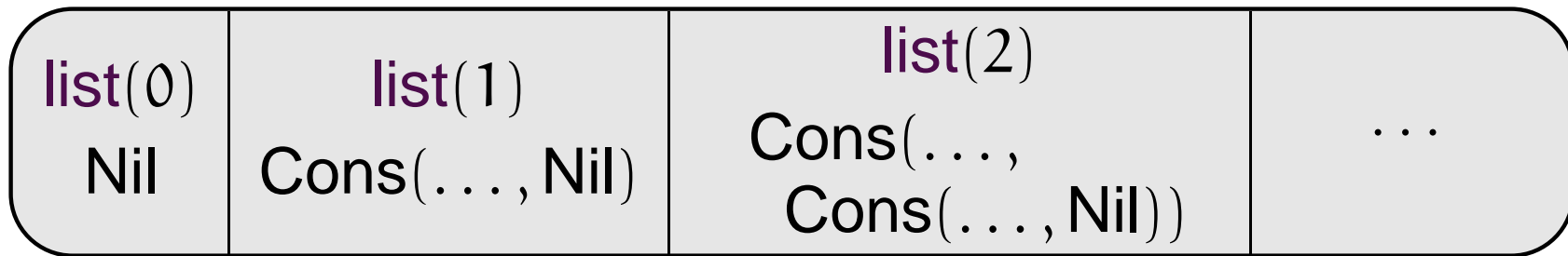
- Dependent types restricted to a decidable constraint domain [Xi & Pfenning '99]
- **Example:** Index lists by their length:



list
all lists

Index Refinements

- Dependent types restricted to a decidable constraint domain [Xi & Pfenning '99]
- **Example:** Index lists by their length:



Nil : list

Nil : list(0)

Cons : int * list → list

Cons : $\forall a:\mathcal{N}. \text{int} * \text{list}(a) \rightarrow \text{list}(a+1)$

append : $\forall a, b:\mathcal{N}. \text{list}(a) * \text{list}(b) \rightarrow \text{list}(a+b)$

filter : $\forall a:\mathcal{N}. (\text{int} \rightarrow \text{bool}) \rightarrow \text{list}(a) \rightarrow (\exists b:\mathcal{N}. (b \leq a) \wedge \text{list}(b))$

Static array bounds checking

Index integers 'by themselves': $2 : \text{int}(2)$, $33 : \text{int}(33)$

Index arrays by size: $\text{array}(0)$, $\text{array}(1)$, ...

get : $\text{array} \ * \ \text{int} \ \rightarrow \ \text{int}$

set : $\text{array} \ * \ \text{int} \ * \ \text{int} \ \rightarrow \ \text{unit}$

size : $\text{array} \ \rightarrow \ \text{int}$

Static array bounds checking

Index integers 'by themselves': $2 : \text{int}(2)$, $33 : \text{int}(33)$

Index arrays by size: $\text{array}(0)$, $\text{array}(1)$, ...

get : $\forall s, i:\mathcal{N}. (i < s) \Rightarrow \text{array}(s) * \text{int}(i) \rightarrow \text{int}$

set : $\forall s, i:\mathcal{N}. (i < s) \Rightarrow \text{array}(s) * \text{int}(i) * \text{int} \rightarrow \text{unit}$

size : $\forall s:\mathcal{N}. \text{array}(s) \rightarrow \text{int}(s)$

Static array bounds checking

Index integers 'by themselves': $2 : \text{int}(2)$, $33 : \text{int}(33)$

Index arrays by size: $\text{array}(0)$, $\text{array}(1)$, ...

$\text{get} : \forall s, i:\mathcal{N}. (i < s) \Rightarrow \text{array}(s) * \text{int}(i) \rightarrow \text{int}$

$\text{set} : \forall s, i:\mathcal{N}. (i < s) \Rightarrow \text{array}(s) * \text{int}(i) * \text{int} \rightarrow \text{unit}$

$\text{size} : \forall s:\mathcal{N}. \text{array}(s) \rightarrow \text{int}(s)$

- get , set no longer need runtime bounds checks!
- Operations with runtime checks can be derived:
- $\text{slow_get} : \text{array} * \text{int} \rightarrow \text{int}$

```
fun slow_get (arr, i) =  
  if (i ≥ 0) andalso (i < size(arr))  
  then get(arr, i)  
  else raise Subscript
```

Example: Bitstrings

- datatype `bits` = E
 - | Zero of bits
 - | One of bits
- 0 (empty bitstring ϵ) represented by E
- Zero, One accumulate on the right:
 $\text{Zero}(n) = n0$, $\text{One}(n) = n1$
- 4 represented by $\text{Zero}(\text{Zero}(\text{One}(E))) = \epsilon100$
- Desired invariant: in **standard form**—no leading zeros:

$$\text{One}(\text{Zero}(E)) = \epsilon01 \quad \times$$

$$\text{Zero}(E) = \epsilon0 \quad \times$$

$$\text{Zero}(\text{One}(E)) = \epsilon10 \quad \checkmark$$

Datasorts express “no leading zeros”

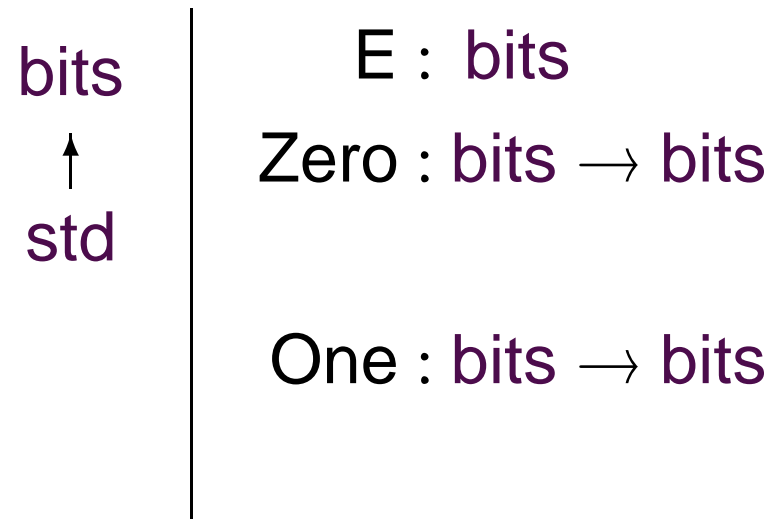
bits

E : bits

Zero : bits \rightarrow bits

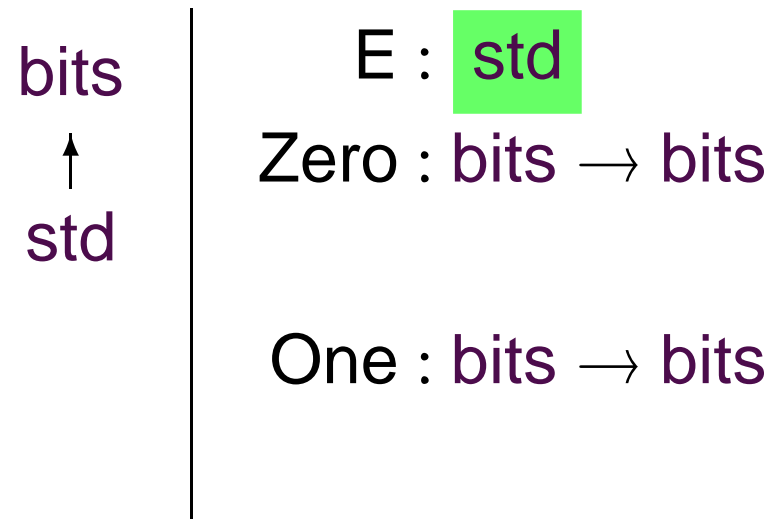
One : bits \rightarrow bits

Datasorts express “no leading zeros”



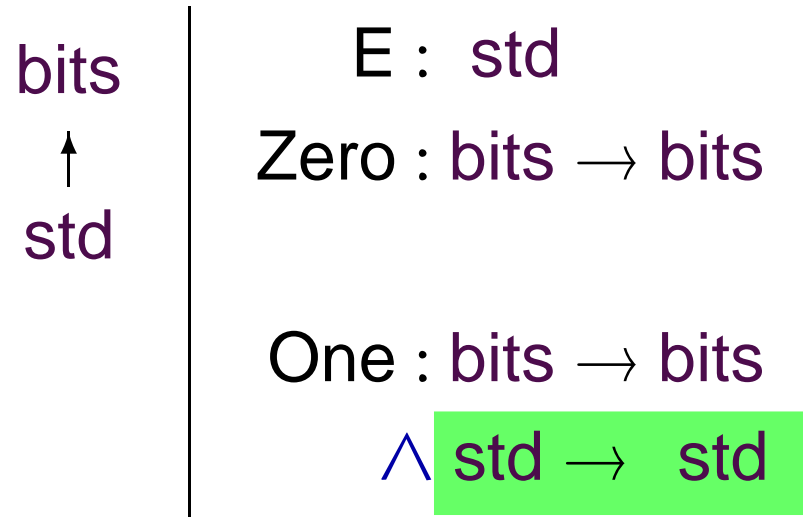
- Add `std` to represent standard form (no leading zeros)

Datasorts express “no leading zeros”



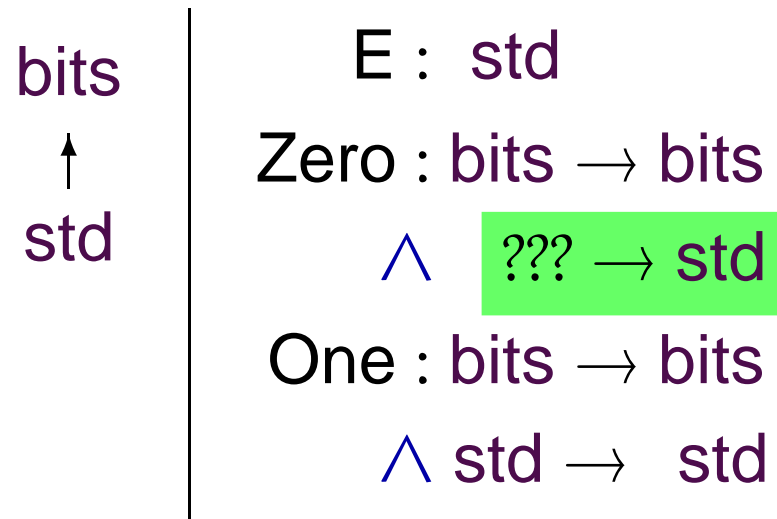
- Add `std` to represent standard form (no leading zeros)
- The empty bitstring has no leading zeros

Datasorts express “no leading zeros”



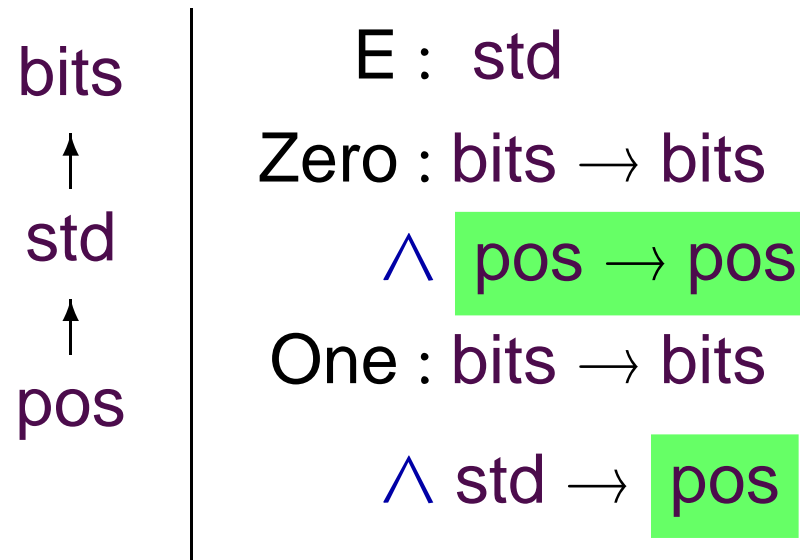
- Add `std` to represent standard form (no leading zeros)
- The empty bitstring has no leading zeros
- $n1$ has no leading zeros if n has no leading zeros

Datasorts express “no leading zeros”



- Add **std** to represent standard form (no leading zeros)
- The empty bitstring has no leading zeros
- $n1$ has no leading zeros if n has no leading zeros
- $n0$ has no leading zeros if n has no leading zeros
and $n \neq \epsilon$ ($\text{Zero}(E) = \epsilon 0$ **X**)

Datasorts express “no leading zeros”



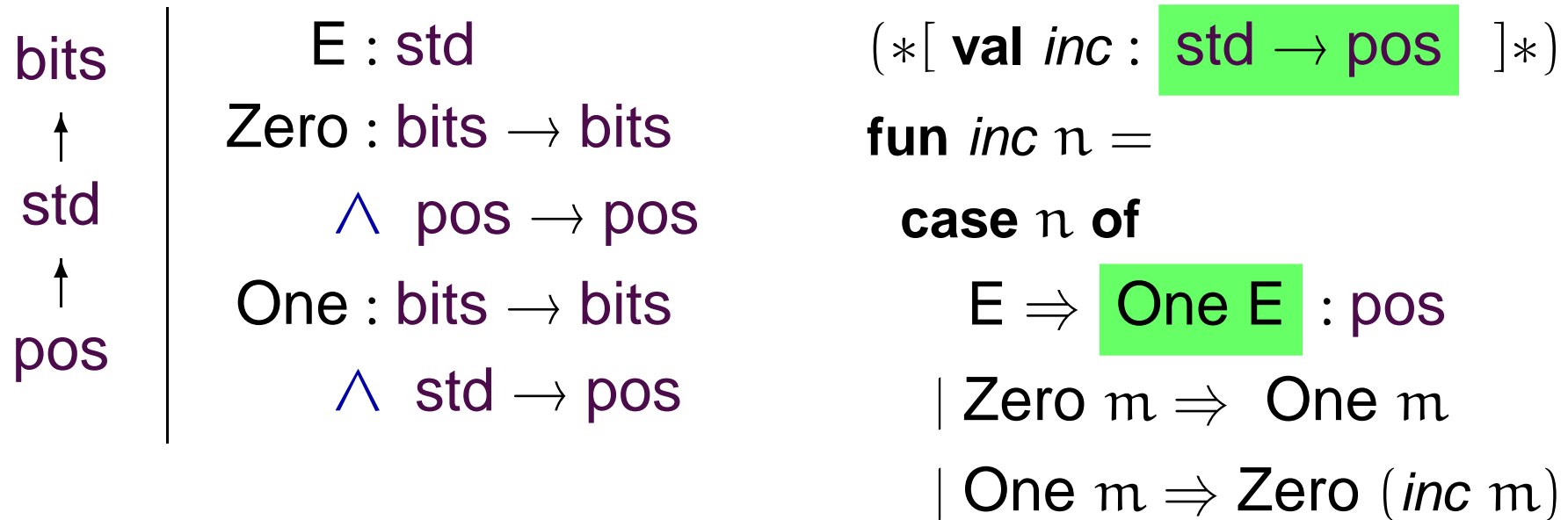
- Add **std** to represent standard form (no leading zeros)
- The empty bitstring has no leading zeros
- $n1$ has no leading zeros **(and pos.)** if n has no leading zeros
- $n0$ has no leading zeros if n has no leading zeros
and $n0 \neq \epsilon$ **and** $n \neq \epsilon$ (Zero(E) = $\epsilon0$ **X**)
- Add **pos** to represent (standard form **and** strictly positive)

Typechecking

bits	E : std	(*[val <i>inc</i> : std → pos]*)
↑	Zero : bits → bits	fun <i>inc</i> n =
std	∧ pos → pos	case n of
↑	One : bits → bits	E ⇒ One E
pos	∧ std → pos	Zero m ⇒ One m
		One m ⇒ Zero (<i>inc</i> m)

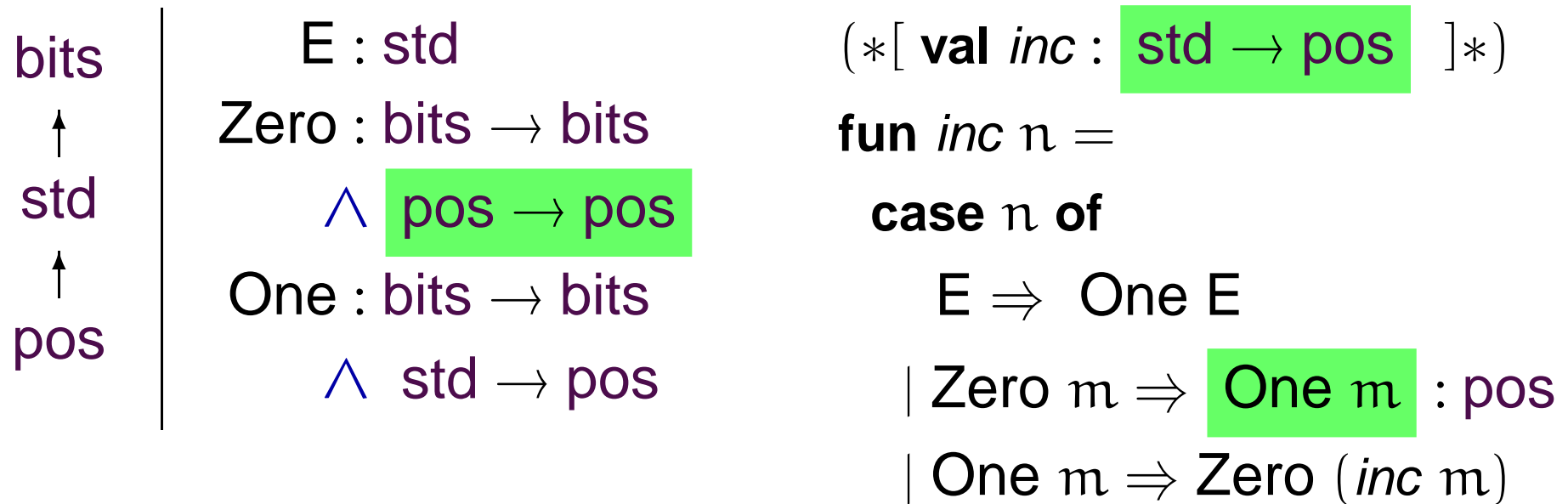
- Assume $n : \text{std}$, check body against pos

Typechecking



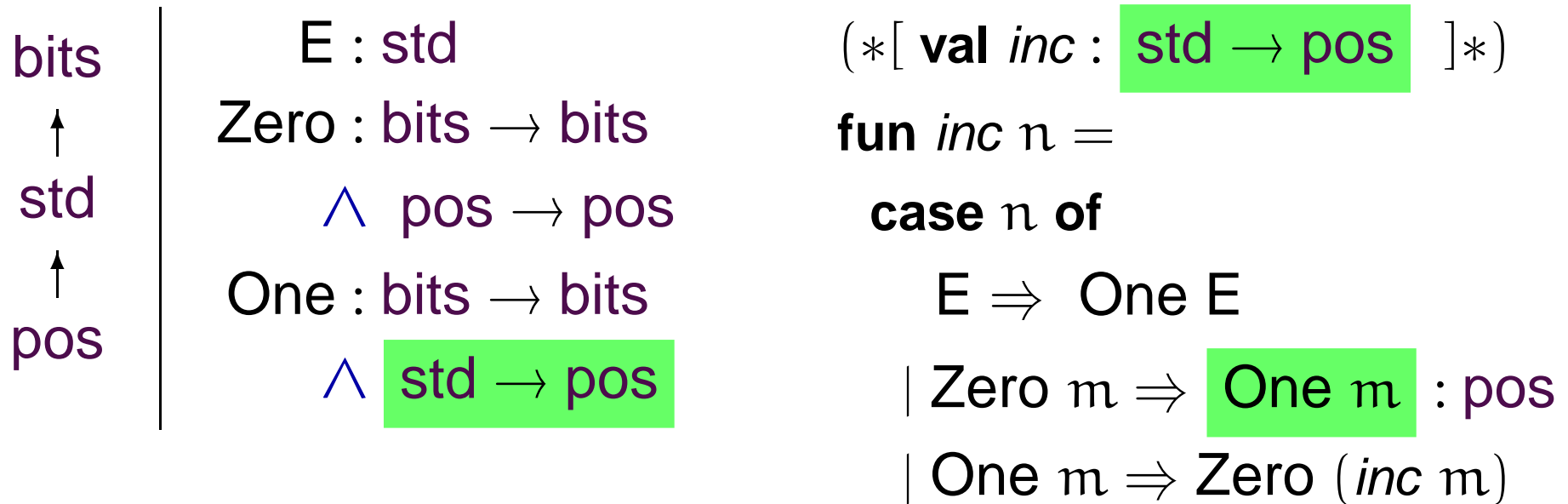
- Assume $n : \text{std}$, check body against pos
 - $E : \text{std}$ so $\text{One}(E) : \text{pos}$

Typechecking



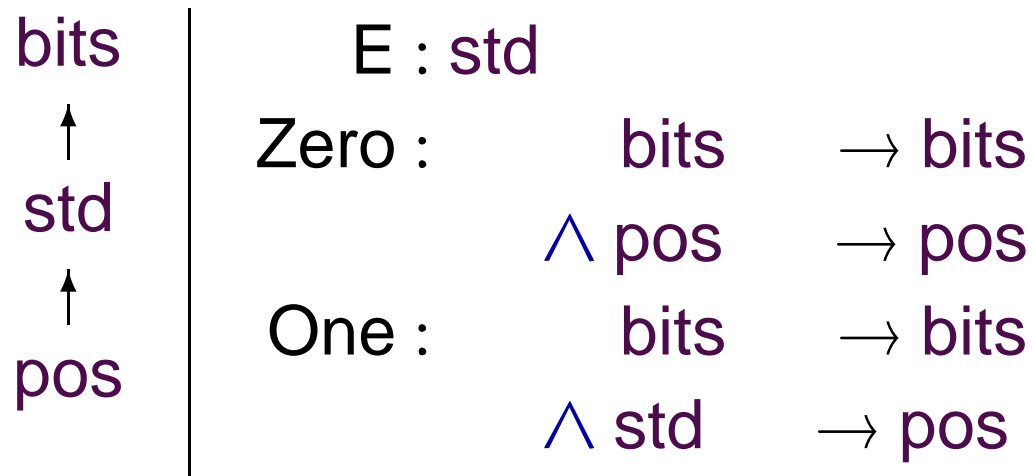
- Assume $n : \text{std}$, check body against pos
 - $E : \text{std}$ so $\text{One}(E) : \text{pos}$
 - If $n = \text{Zero } m$ and $n : \text{std}$ then $n : \text{pos}$ and $m : \text{pos}$

Typechecking



- Assume $n : \text{std}$, check body against pos
 - $E : \text{std}$ so $\text{One}(E) : \text{pos}$
 - If $n = \text{Zero } m$ and $n : \text{std}$ then $n : \text{pos}$ and $m : \text{pos}$
 - If $m : \text{pos}$ then $\text{One}(m) : \text{pos}$

Indices express size



- Index by the size, a natural number: the size of $\epsilon 01$ is 2

Indices express size

bits		E : std(0)
↑		Zero : $\forall a:\mathcal{N}. \text{bits}(a) \rightarrow \text{bits}(a + 1)$
std		$\wedge \text{pos}(a) \rightarrow \text{pos}(a + 1)$
↑		One : $\forall a:\mathcal{N}. \text{bits}(a) \rightarrow \text{bits}(a + 1)$
pos		$\wedge \text{std}(a) \rightarrow \text{pos}(a + 1)$

- Index by the size, a natural number: the size of $\epsilon 01$ is 2

Indices express size

bits		E : std(0)
↑		Zero : $\forall a:\mathcal{N}. \text{bits}(a) \rightarrow \text{bits}(a + 1)$
std		$\wedge \text{pos}(a) \rightarrow \text{pos}(a + 1)$
↑		One : $\forall a:\mathcal{N}. \text{bits}(a) \rightarrow \text{bits}(a + 1)$
pos		$\wedge \text{std}(a) \rightarrow \text{pos}(a + 1)$

- Index by the size, a natural number: the size of $\epsilon 01$ is 2
- What type can we give *inc*?

```
(*[ val inc : std → pos  
  ]*)
```

```
fun inc n =
```

```
  case n of
```

```
    ⋮
```

Indices express size

bits		E : std(0)
↑		Zero : $\forall a:\mathcal{N}. \text{bits}(a) \rightarrow \text{bits}(a + 1)$
std		$\wedge \text{pos}(a) \rightarrow \text{pos}(a + 1)$
↑		One : $\forall a:\mathcal{N}. \text{bits}(a) \rightarrow \text{bits}(a + 1)$
pos		$\wedge \text{std}(a) \rightarrow \text{pos}(a + 1)$

- Index by the size, a natural number: the size of $\epsilon 01$ is 2
- What type can we give *inc*?

```
(* [ val inc :  $\forall a:\mathcal{N}. \text{std}(a) \rightarrow \text{pos}(\dots)$  ] *)
```

```
fun inc n =
```

```
  case n of
```

```
    ⋮
```

either a or $a + 1$

Indices express size

bits	E : std(0)
↑	Zero : $\forall a:\mathcal{N}. \text{bits}(a) \rightarrow \text{bits}(a + 1)$
std	$\wedge \text{pos}(a) \rightarrow \text{pos}(a + 1)$
↑	One : $\forall a:\mathcal{N}. \text{bits}(a) \rightarrow \text{bits}(a + 1)$
pos	$\wedge \text{std}(a) \rightarrow \text{pos}(a + 1)$

- Index by the size, a natural number: the size of $\epsilon 01$ is 2
- What type can we give *inc*?

(*[**val** *inc* : $\forall a:\mathcal{N}. \text{std}(a) \rightarrow \text{pos}(a) \vee \text{pos}(a + 1)$
]*)

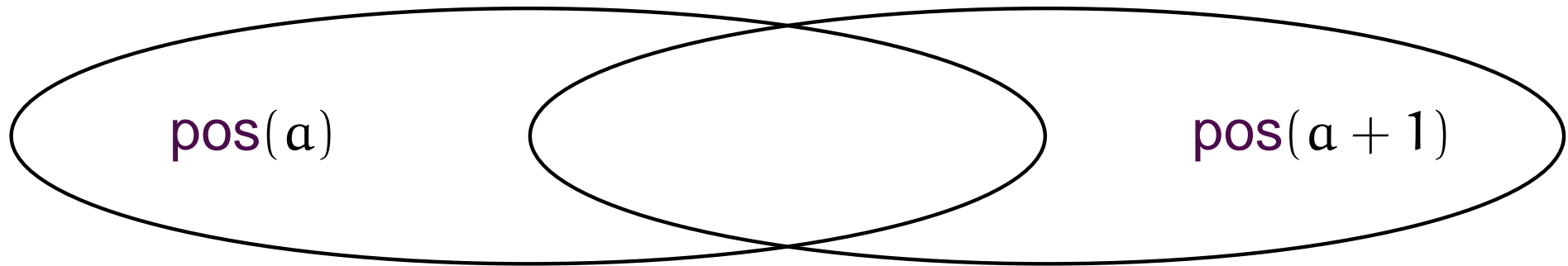
fun *inc* n =

case n **of**

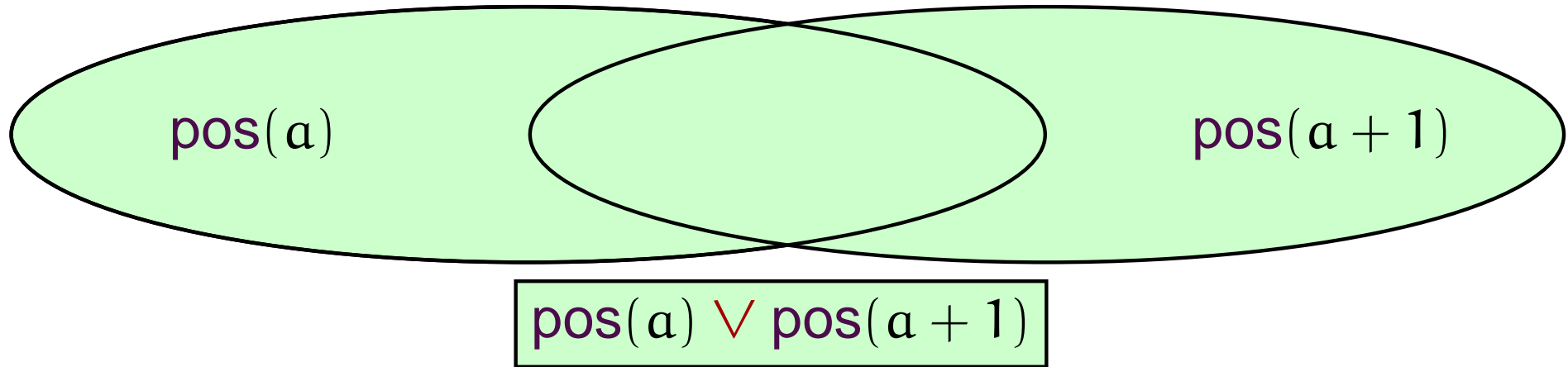
⋮

either a or a + 1

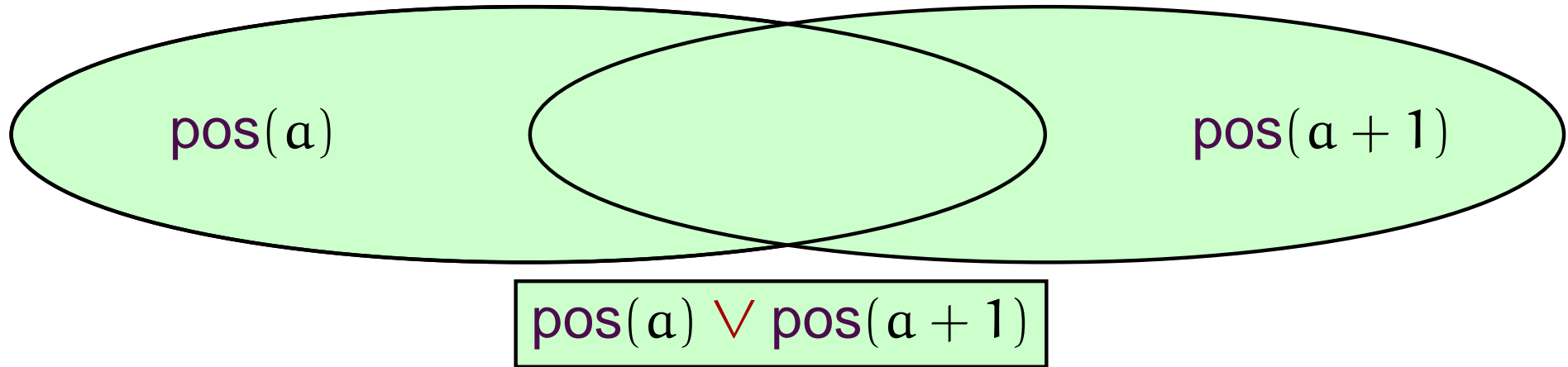
Union types ~ set union



Union types ~ set union



Union types ~ set union



$(*[\text{val } inc : \forall a:\mathcal{N}. \text{std}(a) \rightarrow (\text{pos}(a) \vee \text{pos}(a + 1))]*)$

fun *inc* n =

case n of

E \Rightarrow One E

| Zero m \Rightarrow One m

| One m \Rightarrow Zero (*inc* m)

Known: n : **std**(a), m : **std**(b), a = b + 1

Then: (*inc* m) : (**pos**(b) \vee **pos**(b + 1))

(1) Assume (*inc* m) : **pos**(b),

check against **pos**(a) or **pos**(a + 1)

(2) Assume (*inc* m) : **pos**(b + 1),

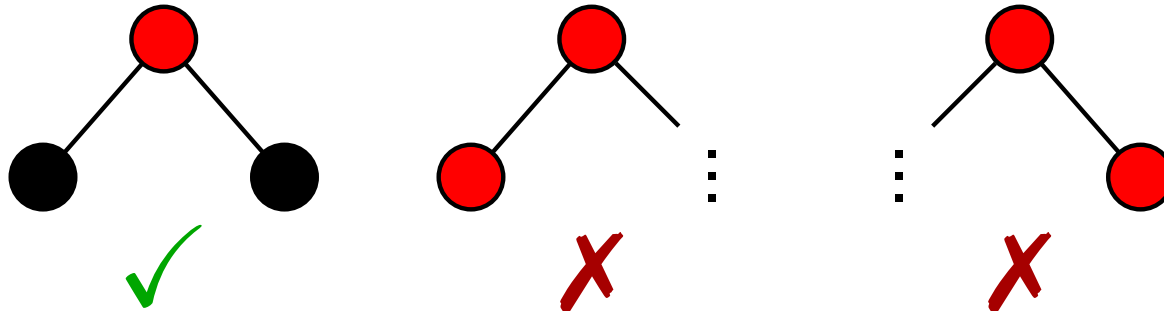
check against **pos**(a) or **pos**(a + 1)

Red-black trees

- **datatype tree** = Empty
| Red of key * tree * tree
| 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 $bh(t)$ s.t. the number of black nodes on **every** path from t to a descendant leaf is $bh(t)$

Use datasorts for invariants (1) and (2)

tree

Empty : tree

Red : key * tree * tree → tree

Black : key * tree * tree → tree

Use datasorts for invariants (1) and (2)

tree
↑
rbt

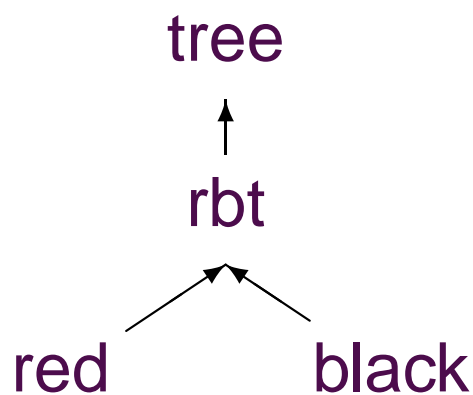
Empty : tree

Red : key * tree * tree → tree

Black : key * tree * tree → tree

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

Use datasorts for invariants (1) and (2)



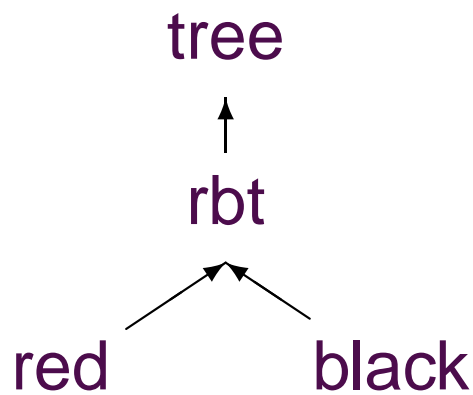
Empty : tree

Red : key * tree * tree \rightarrow tree

Black : key * tree * tree \rightarrow tree

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

Use datasorts for invariants (1) and (2)



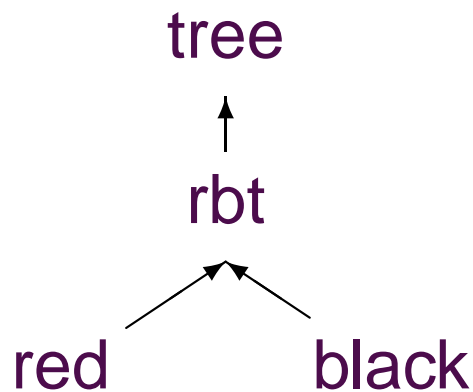
Empty : $\text{tree} \wedge \text{black}$

Red : $\text{key} * \text{tree} * \text{tree} \rightarrow \text{tree}$

Black : $\text{key} * \text{tree} * \text{tree} \rightarrow \text{tree}$

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

Use datasorts for invariants (1) and (2)



Empty : $tree \wedge black$

Red : $key * tree * tree \rightarrow tree$

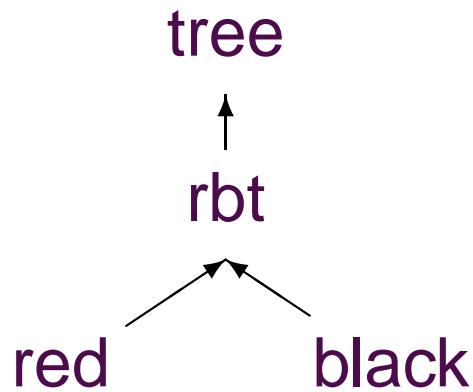
$\wedge key * black * black \rightarrow red$

Black : $key * tree * tree \rightarrow tree$

$\wedge key * rbt * rbt \rightarrow black$

- 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 indices for invariant (3)



Empty : black

Red :

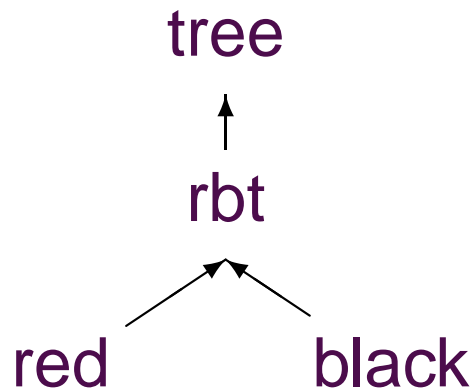
$$\begin{aligned} & \text{key} * \text{tree} \quad * \text{tree} \quad \rightarrow \text{tree} \\ \wedge & \text{key} * \text{black} \quad * \text{black} \quad \rightarrow \text{red} \end{aligned}$$

Black :

$$\begin{aligned} & \text{key} * \text{tree} \quad * \text{tree} \quad \rightarrow \text{tree} \\ \wedge & \text{key} * \text{rbt} \quad * \text{rbt} \quad \rightarrow \text{black} \end{aligned}$$

- Invariant (3): For every node t , there exists $bh(t)$ s.t. the number of black nodes on **every** path from t to a descendant leaf is $bh(t)$

Use indices for invariant (3)



Empty : black

Red :

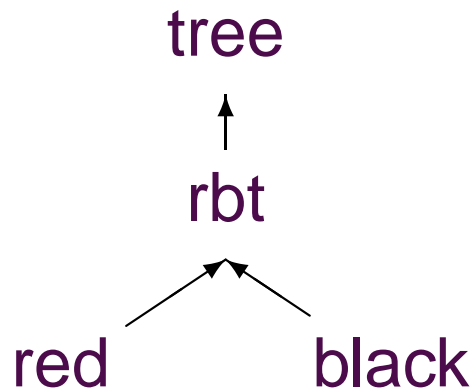
$$\begin{aligned} & \text{key} * \text{tree} \quad * \text{tree} \quad \rightarrow \text{tree} \\ \wedge & \text{key} * \text{black} \quad * \text{black} \quad \rightarrow \text{red} \end{aligned}$$

Black :

$$\begin{aligned} & \text{key} * \text{tree} \quad * \text{tree} \quad \rightarrow \text{tree} \\ \wedge & \text{key} * \text{rbt} \quad * \text{rbt} \quad \rightarrow \text{black} \end{aligned}$$

- Invariant (3): For every node t , there exists $bh(t)$ s.t. the number of black nodes on **every** path from t to a descendant leaf is $bh(t)$
- Index by the black height $bh(t)$, a natural number

Use indices for invariant (3)



Empty : **black**(0)

Red : $\forall h:\mathcal{N}.$

$\text{key} * \text{tree}(h) * \text{tree}(h) \rightarrow \text{tree}(h)$

$\wedge \text{key} * \text{black}(h) * \text{black}(h) \rightarrow \text{red}(h)$

Black : $\forall h:\mathcal{N}.$

$\text{key} * \text{tree}(h) * \text{tree}(h) \rightarrow \text{tree}(h + 1)$

$\wedge \text{key} * \text{rbt}(h) * \text{rbt}(h) \rightarrow \text{black}(h + 1)$

- Invariant (3): For every node t , there exists $bh(t)$ s.t. the number of black nodes on **every** path from t to a descendant leaf is $bh(t)$
- Index by the black height $bh(t)$, a natural number

Red-black trees

- Refined typechecking ensures color and height invariants
- Color and height invariants not observable to clients
- Straightforward runtime testing takes $O(n)$ (vs. $O(\lg n)$)

✓ Datasort Refinements

✓ Index Refinements

👉 **Implementation**

- (Demo...)

- (Related) Work

- Summary

- Future Work

Stardust

- Checks a subset of Standard ML
 - Pattern matching
 - No modules
- Standalone—not part of a compiler
- External tool (ICS) for index constraints

- ✓ Datasort Refinements
- ✓ Index Refinements
- ✓ Implementation
- ☞ **(Demo...)**
 - (Related) Work
 - Summary
 - Future Work

✓ Datasort Refinements

✓ Index Refinements

✓ Implementation

✓ (Demo...)

👉 **(Related) Work**

- Summary

- Future Work

How did we get here?

- Datasort refinements
 - Idea arose in logic programming
 - c. 1990: Freeman & Pfenning
 - late '90s: Davies & Pfenning (for full SML)
- Intersection types \wedge
 - c. 1980
 - c. 1990: Reynolds
 - late '90s: Davies & Pfenning (effects)
- Index refinements
 - Rooted in dependent type theory
 - late '90s: Xi & Pfenning
- Union types \vee
 - late 1980s

Dunfield & Pfenning

- **Datasort refinements**
 - Small innovation in specifying datasorts
- **Union types**
- **Index refinements**
 - New approach to \exists
 - Combination with **intersection types**

Including, of course...

Dunfield & Pfenning

- **Datasort refinements**
 - Small innovation in specifying datasorts
- **Union types**
- **Index refinements**
 - New approach to \exists
 - Combination with **intersection types**

Including, of course...

- **Type safety**

- ✓ Datasort Refinements
- ✓ Index Refinements
- ✓ Implementation
- ✓ (Demo...)
- ✓ (Related) Work
- ☞ **Summary**
 - Future Work

Summary

- Conventional static typecheckers verify coarse properties
- Refined typecheckers verify refined properties
- To express properties of algebraic datatypes:
 - Datasort refinements
 - Lists, formulas
 - Index refinements
 - Lists, arrays
 - Simultaneous refinement
 - Bitstrings, red-black trees
- To combine properties: \wedge , \vee , \forall , \exists

- ✓ Datasort Refinements
- ✓ Index Refinements
- ✓ Implementation
- ✓ (Demo...)
- ✓ (Related) Work
- ✓ Summary
- 👉 **Future Work**

Future work

- Implementation
 - Faster
 - More index domains
 - Better error reporting
 - More language features
- Parametric polymorphism
- Call-by-name, imperative...
- Applications

<http://type-refinements.org>