Shared Subtypes

Subtyping Recursive Parameterized Algebraic Data Types

Ki Yung Ahn
kya@cs.pdx.edu

Tim Sheard
sheard@cs.pdx.edu

Department of Computer Science
Maseeh College of Engineering & Computer Science

Portland State
UNIVERSITY

Background

Subtype Polymorphism in OOPL
(Object-Oriented Programming Language)

In Java, `String readLine(IntputStream s)`

Parametric Polymorphism in FPL

In Haskell, $length :: [a] \rightarrow Int$

Subtype Polymorphism

Subtype Polymorphism in OOPL

- In Java, `String readLine(IntputStream s)`
- Can call this function any subtype of `InputStream`

```
readLine(System.in);   // InputStream type
FileInputStream ifs = new FileInputStream(...);
readLine(ifs);   // FileInputStream is a subtype
```

Parametric Polymorphism

Parametric Polymorphism in FPL

In Haskell, $length :: [a] \rightarrow Int$

We can apply this function on ANY list

$$
\begin{array}{ll}
length\ [1,2,3] & \text{-- } a = Int \\
length\ [True, False] & \text{-- } a = Bool \\
length\ [[],[1],[2],[1,2]] & \text{-- } a = [Int]
\end{array}
$$

Some History

Lack of Parametric Polymorphism in OOPLs was quite a pain. So, most major OOPLs finally decided to extend their language

Templates in C++, Generics in Java / C#

FPLs, especially Haskell, lived happily without Subtypes because of Type Classes (somewhat like abstract interfaces, but more flexible)

Shared Subtypes

In abbr. SSubtypes = Shared Subtypes

- Subtyping by sharing representations
  - Newtypes and now SSubtypes
  - While OO subtyping by consistent interface
- We can use SSubtypes when we want to
  - Restrict number of data constructors
  - Ensure static properties using type indexes
  - reuse library without efficiency loss
- Relatively small changes to type system

Why Subtype in FPL?

Well, not the good old subtyping in OOPLs

- Haskell can mimic dynamic subtyping (sharing interface) using type classes
- See Oleg & Ralf's OOHaskell library

But a different one namely Shared Subtypes

- static subtyping (sharing the structure) can be quite useful in FPLs too!

Example

$$\mathbf{data}\ Nat = Zero \mid Succ\ Nat$$

$$\begin{array}{ll} Zero & \text{-- } 0 \\ Succ\ Zero & \text{-- } 1 \\ Succ\ (Succ\ Zero) & \text{-- } 2 \end{array}$$

Example

$$\mathbf{data}\ Nat = Zero \mid Succ\ Nat$$

$$\begin{aligned}
&nat2int && :: Nat \rightarrow Int \\
&nat2int\ Zero && = 0 \\
&nat2int\ (Succ\ n) && = 1 + nat2int\ n
\end{aligned}$$

Example

$$
\begin{array}{lll}
\textbf{data}\ Nat & = Zero & \mid Succ\ Nat \\
\textbf{data}\ Even & = ZeroE & \mid SuccE\ Odd \\
\textbf{data}\ Odd & = & \quad SuccO\ Even \\
nat2int & & :: Nat \rightarrow Int \\
even2int & & :: Even \rightarrow Int \\
odd2int & & :: Odd \rightarrow Int
\end{array}
$$

Example

$$
\begin{aligned}
&\textbf{data}\ Nat \ = Zero \ \mid Succ\ Nat \\
&\textbf{data}\ Even = ZeroE \mid SuccE\ Odd \\
&\textbf{data}\ Odd \ = \qquad\quad\ \ SuccO\ Even
\end{aligned}
$$

$$
\begin{aligned}
&nat2int && :: Nat \rightarrow Int \\
&nat2int\ Zero && = 0 \\
&nat2int\ (Succ\ n) && = 1 + nat2int\ n
\end{aligned}
$$

$$
\begin{aligned}
&even2int && :: Even \rightarrow Int \\
&even2int\ ZeroE && = 0 \\
&even2int\ (SuccE\ n) && = 1 + odd2int\ n
\end{aligned}
$$

$$
\begin{aligned}
&odd2int && :: Odd \rightarrow Int \\
&odd2int\ (SuccO\ n) && = 1 + even2int\ n
\end{aligned}
$$

Example

$$
\begin{aligned}
&\textbf{data}\ Nat \ = Zero \mid Succ\ Nat \\
&\textbf{data}\ Even = ZeroE \mid SuccE\ Odd \\
&\textbf{data}\ Odd \ = \qquad\quad\ \ SuccO\ Even
\end{aligned}
$$

$$
\begin{aligned}
&nat2int && :: Nat \to Int \\
&nat2int\ Zero && = 0 \\
&nat2int\ (Succ\ n) && = 1 + nat2int\ n \\
&even2int && :: Even \to Int \\
&even2int\ ZeroE && = 0 \\
&even2int\ (SuccE\ n) && = 1 + odd2int\ n \\
&odd2int && :: Odd \to Int \\
&odd2int\ (SuccO\ n) && = 1 + even2int\ n
\end{aligned}
$$

(x3) Code Duplication !!!

What's the Problem?

Both ***Even*** (even natural numbers)
and ***Odd*** (odd natural numbers)
are subsets of ***Nat*** (all natural numbers)

But, Haskell and many other FPLs
do not have language feature that can
express this idea of subtyping directly

Idea (Shared Subtypes)

$$
\begin{array}{lll}
\textbf{data}\ Nat & = Zero & \\
 & \mid\ Succ & Nat \\
\textbf{data}\ Even < Nat & = (ZeroE < Zero) & \\
 & \mid\ (SuccE < Succ) & Odd \\
\textbf{data}\ Odd\ < Nat & = (SuccO < Succ) & Even
\end{array}
$$

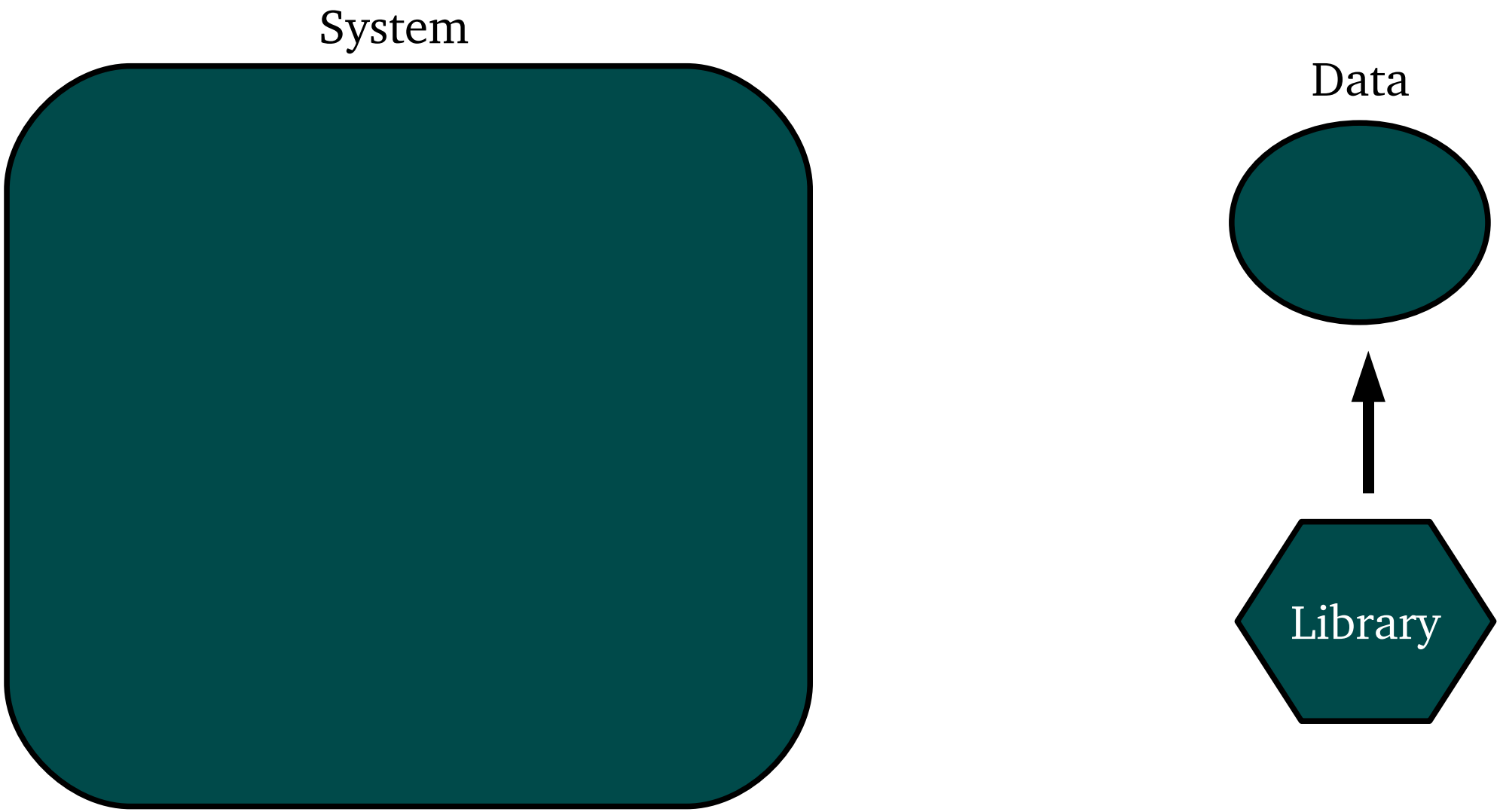
$$
\begin{array}{ll}
nat2int :: Nat \rightarrow Int & \\
nat2int\ Zero & = 0 \\
nat2int\ (Succ\ n) & = 1 + nat2int\ n
\end{array}
$$

Is this really important?

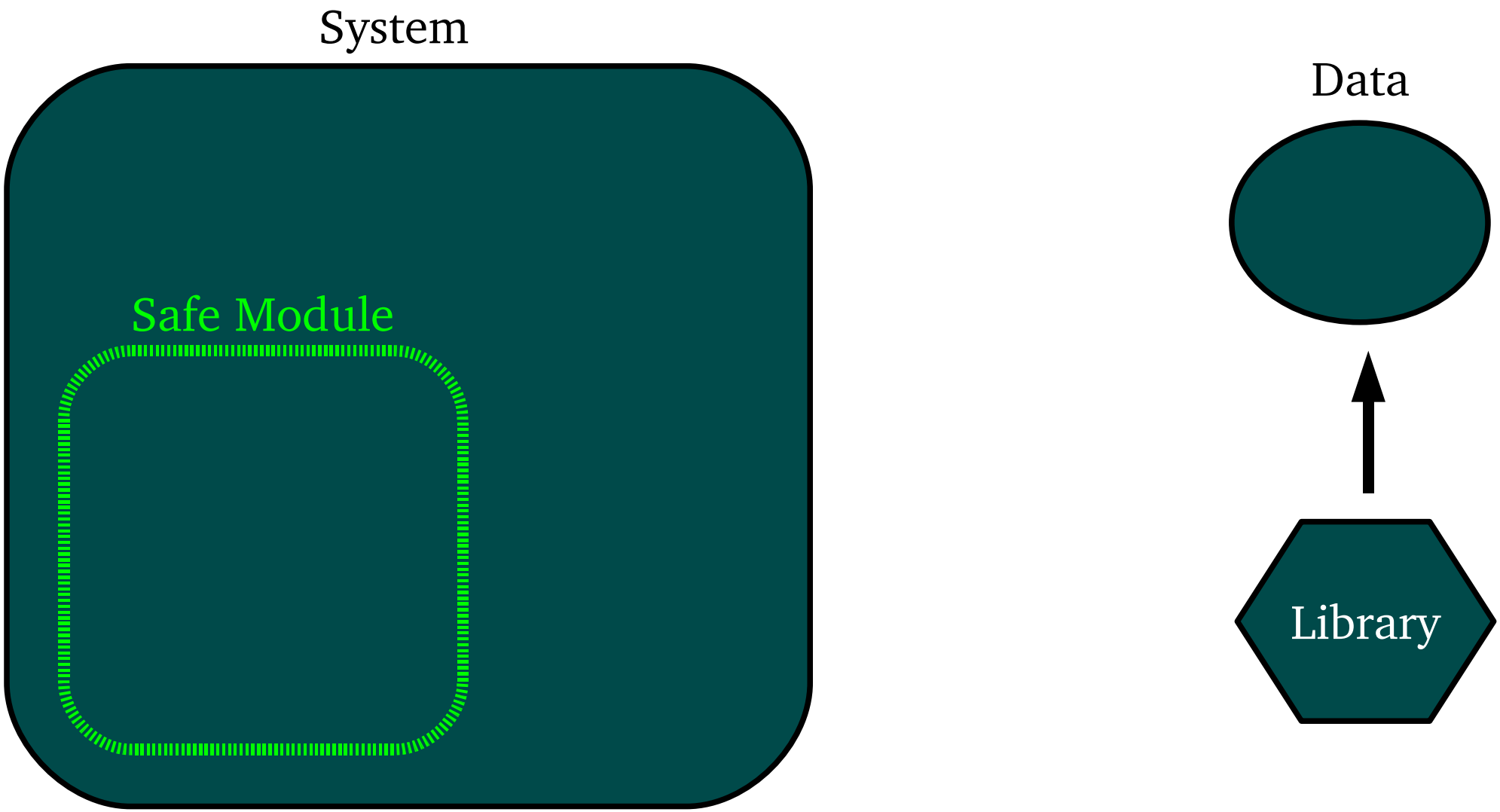
Even and Odd unary numbers
may not sound very interesting to you

But, this same problem can happen for more complex and realistic data structures such as lists and trees, as we shall see soon

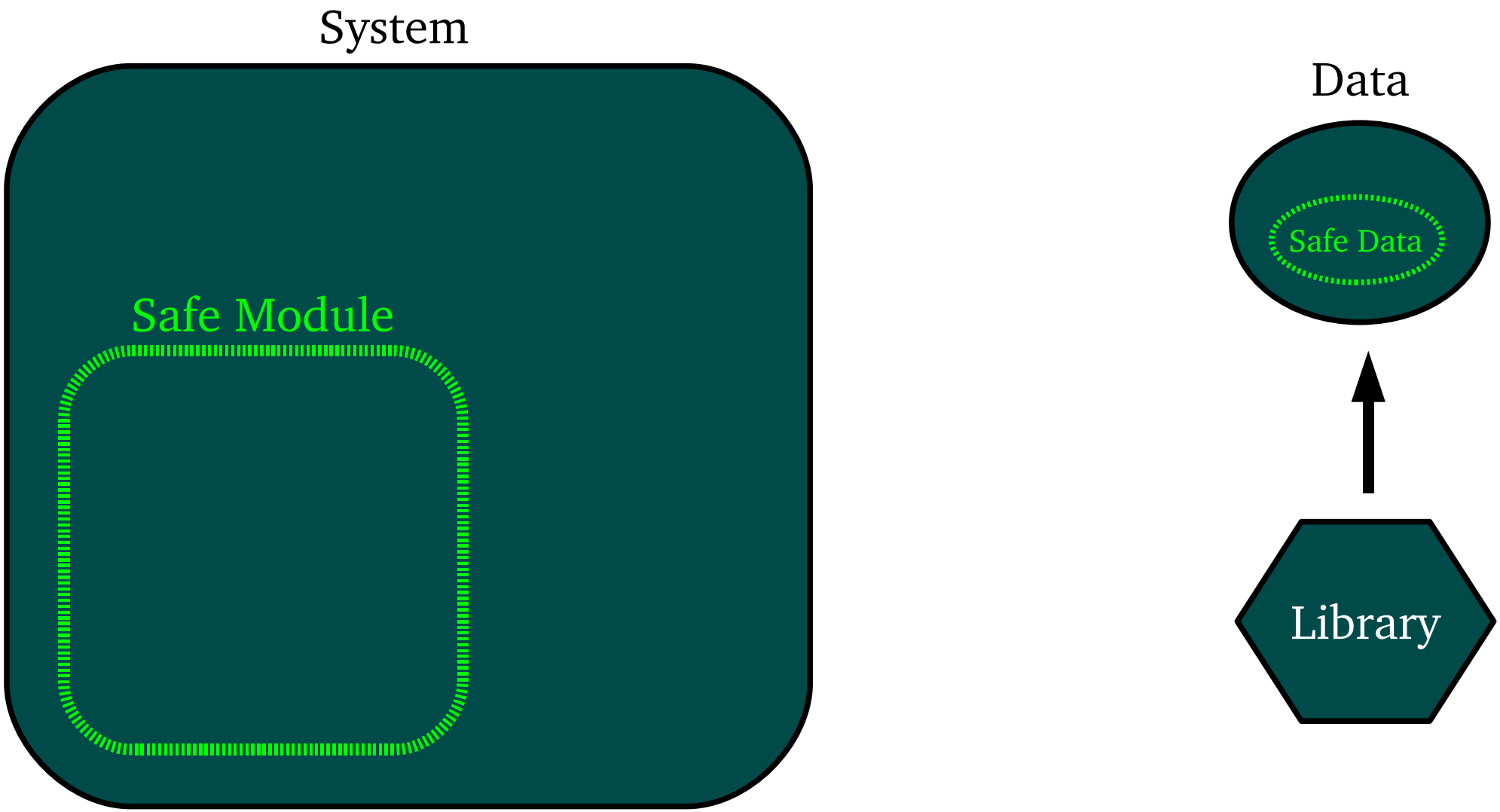
Is this really important?



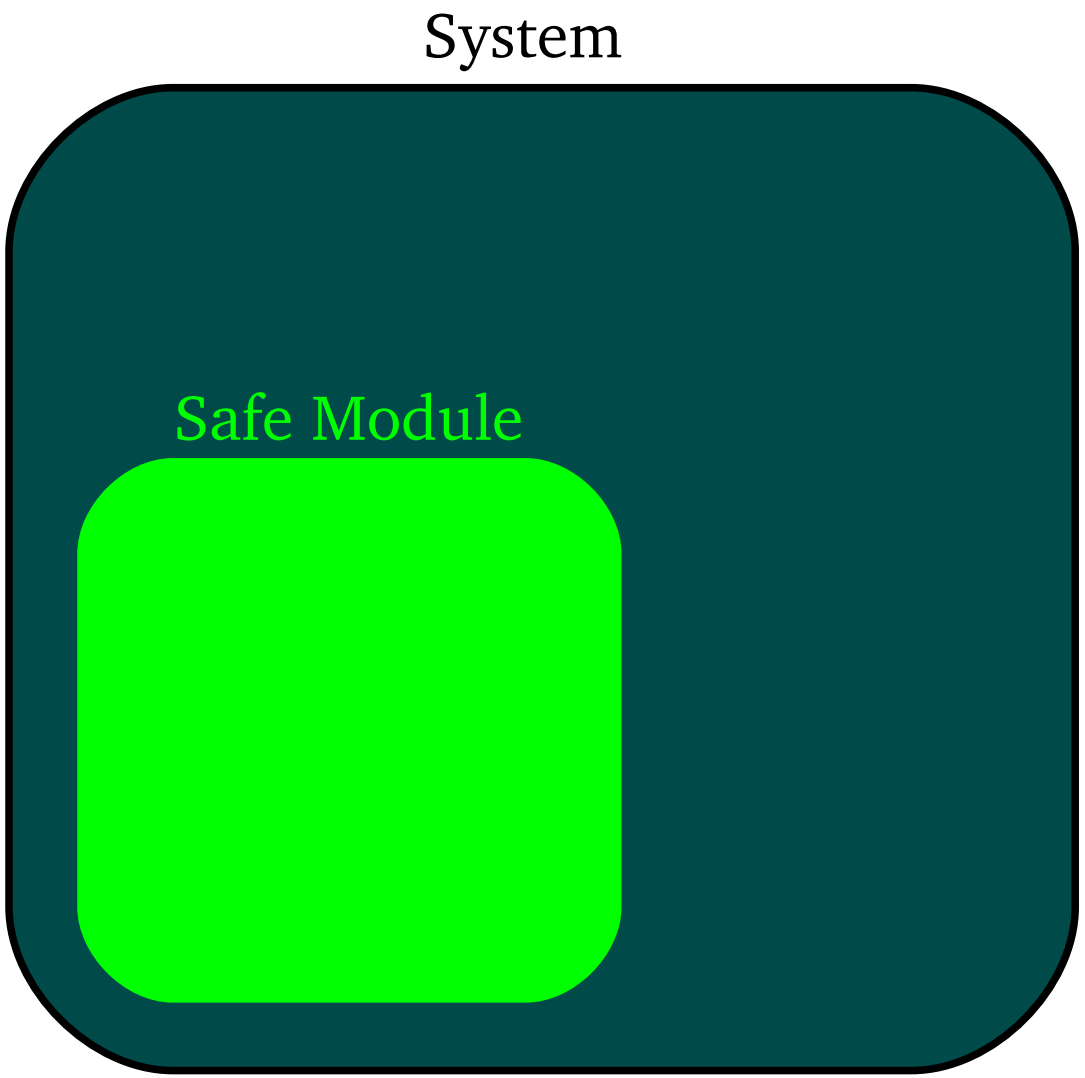
Is this really important?

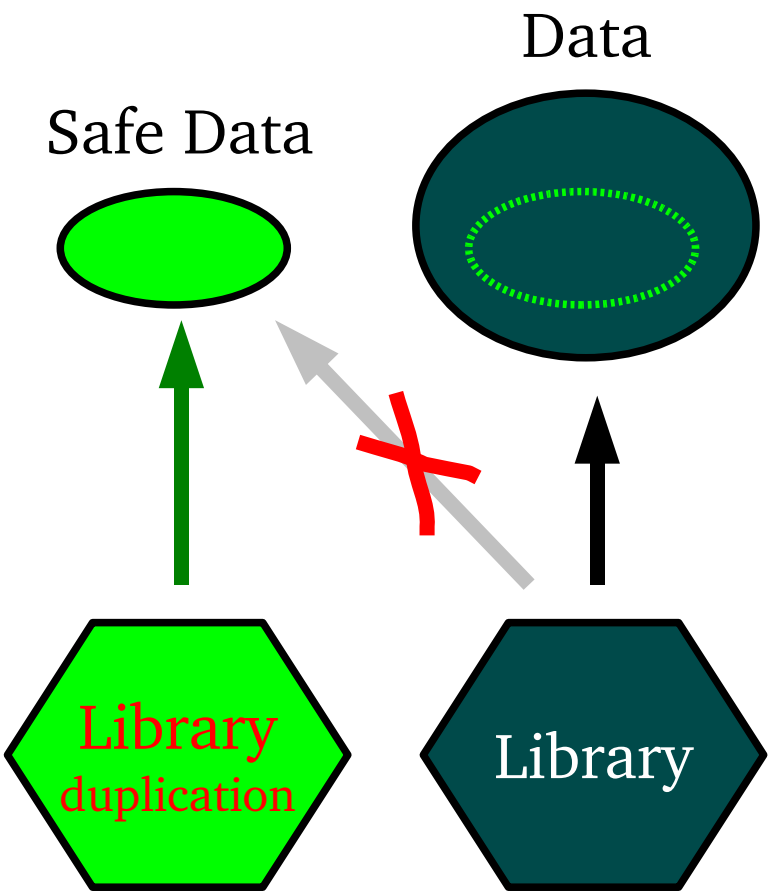


Is this really important?

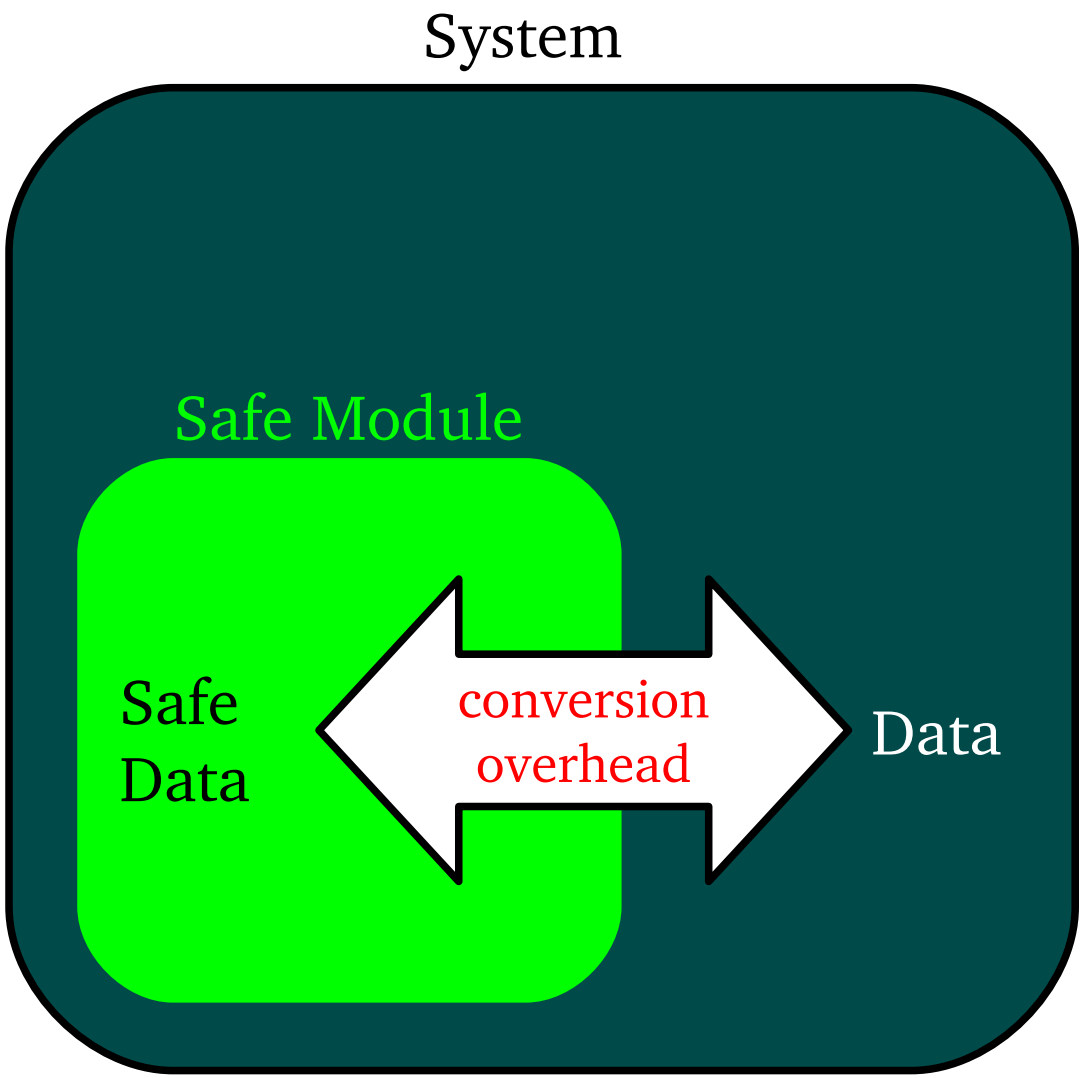


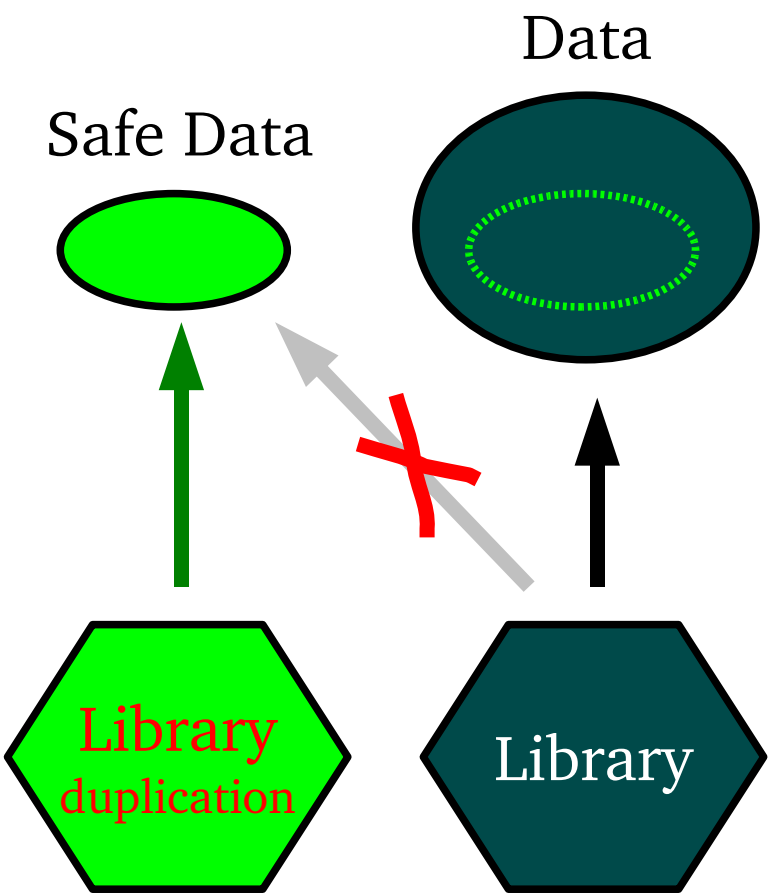
Is this really important?





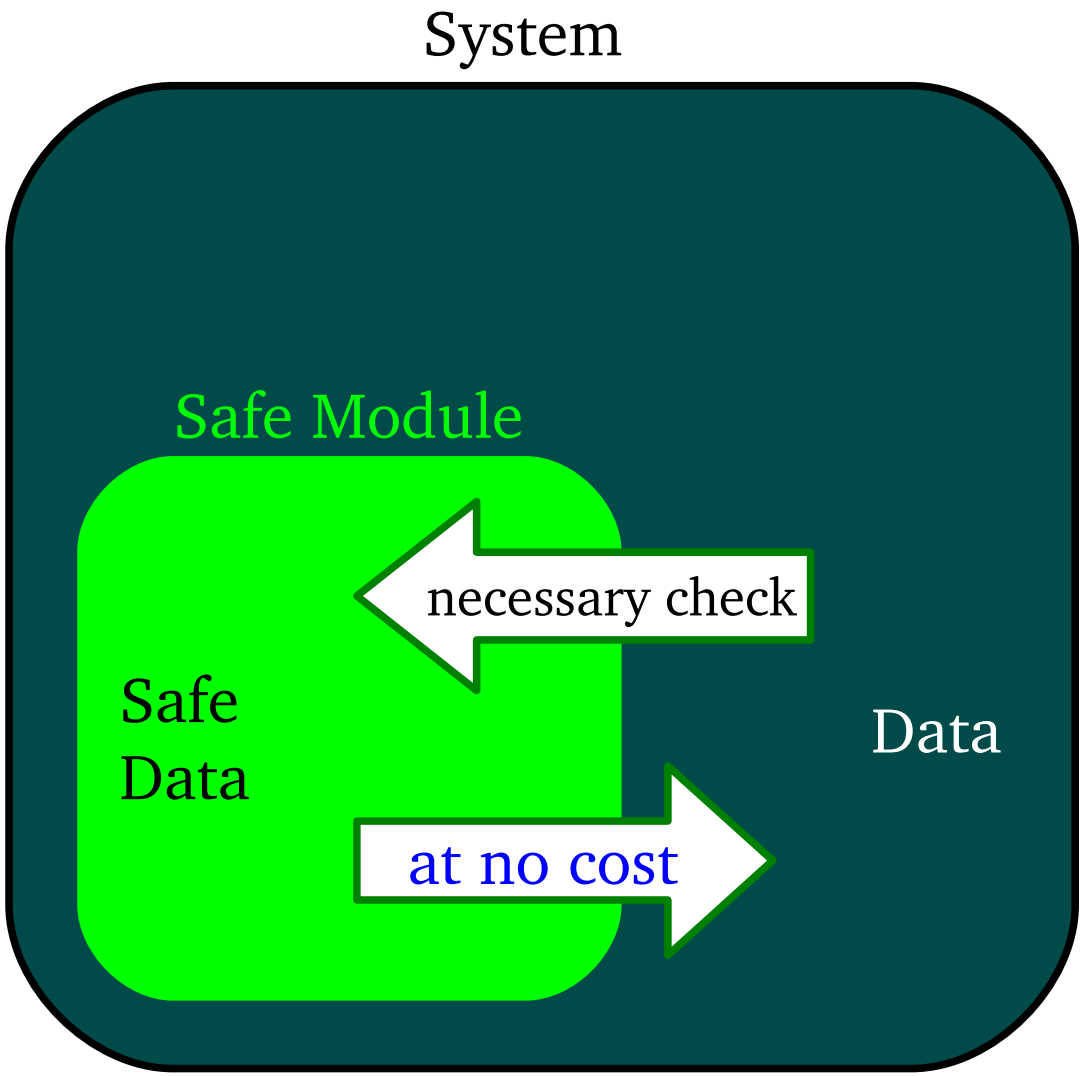
Is this really important?

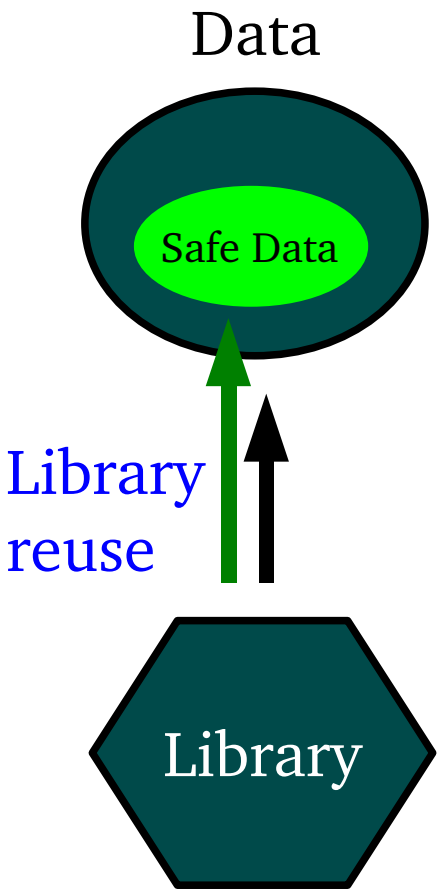




Yes, it is really important!

If we had Shared Subtypes ...





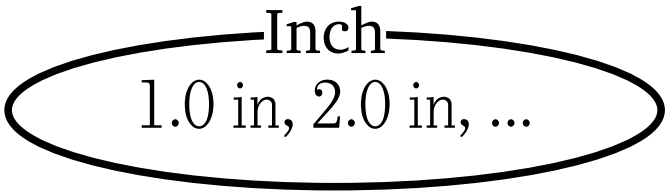
Shared Subtypes

In abbr. SSubtypes = Shared Subtypes

- Subtyping by sharing representations
  - Newtypes and now SSubtypes
  - While OO subtyping by consistent interface
- We can use SSubtypes when we want to
  - Restrict number of data constructors
  - Ensure static properties using type indexes
  - reuse library without efficiency loss
- Relatively small changes to type system

What do we do when two types share a common representation

Isomorphic



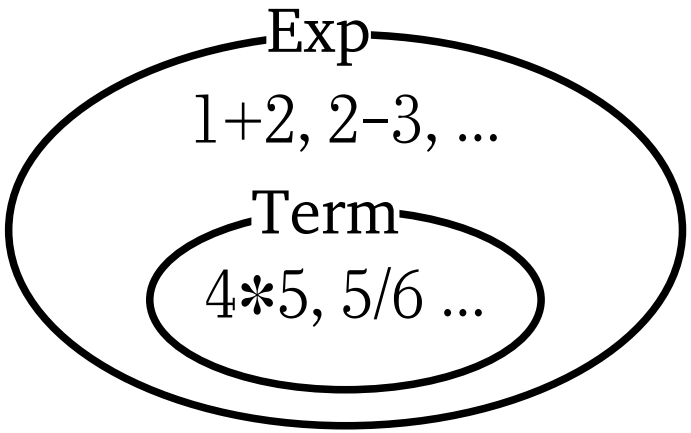
common functions

+, −, ==, ...

In Haskell:

newtype Inch = Inch Double

Subset



common functions

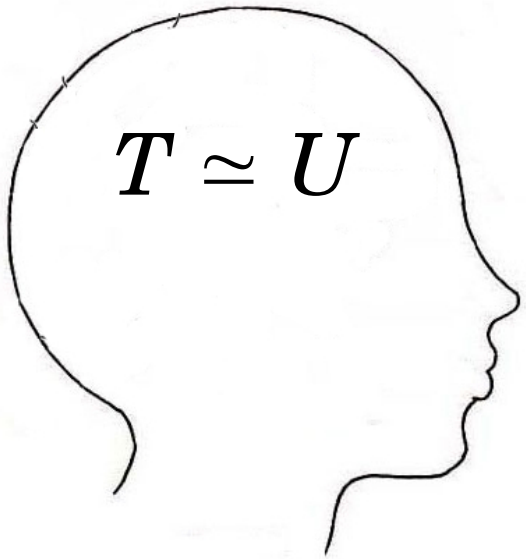
eval, size, hight, ...

Not in Haskell: (until now)

data Exp = ...

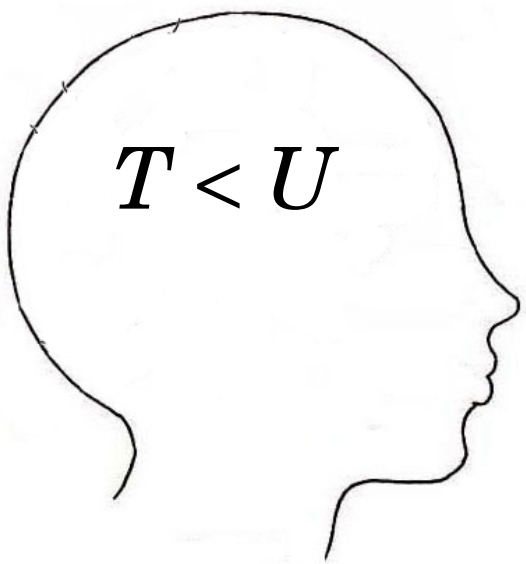
data Term < Exp = ...

Motivation



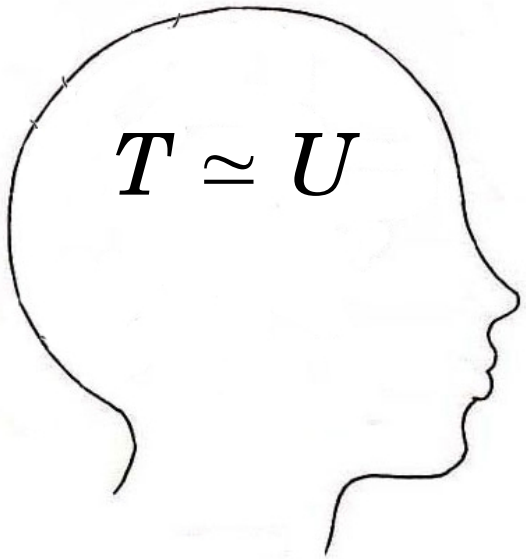
share T and U's representation

declare T as newtype of U



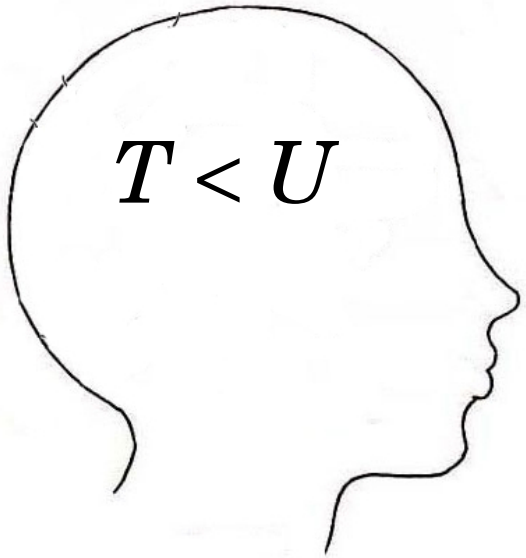
share T and U's representation

Motivation



share T and U's representation

declare T as newtype of U



share T and U's representation

declare T as shared subtype of U

Shared Subtypes

In abbr. SSubtypes = Shared Subtypes

- Subtyping by sharing representations
  - Newtypes and now SSubtypes
  - While OO subtyping by consistent interface
- We can use SSubtypes when we want to
  - Restrict number of data constructors
  - Ensure static properties using type indexes
  - reuse library without efficiency loss
- Relatively small changes to type system

Newtypes

$$\textbf{newtype}\ Inch = Inch\ Double$$

$$\textbf{instance}\ Eq\ Inch\ \textbf{where}$$
$$\quad (Inch\ x) == (Inch\ y) = x == y$$

$$\textbf{class}\ ToMeter\ a\ \textbf{where}$$
$$\quad toMeter :: a \rightarrow Meter$$

$$\textbf{instance}\ ToMeter\ Inch\ \textbf{where}$$
$$\quad toMeter\ (Inch\ x) = \ldots$$

Newtypes

$$\textbf{newtype}\ Inch = \cancel{Inch}\ Double$$

$$\textbf{instance}\ Eq\ Inch\ \textbf{where}$$
$$(\cancel{Inch}\ x) == (\cancel{Inch}\ y) = x == y$$

$$\textbf{class}\ ToMeter\ a\ \textbf{where}$$
$$toMeter :: a \rightarrow Meter$$

$$\textbf{instance}\ ToMeter\ Inch\ \textbf{where}$$
$$toMeter\ (Inch\ x) = \ldots$$

Code Reuse without conversion overhead

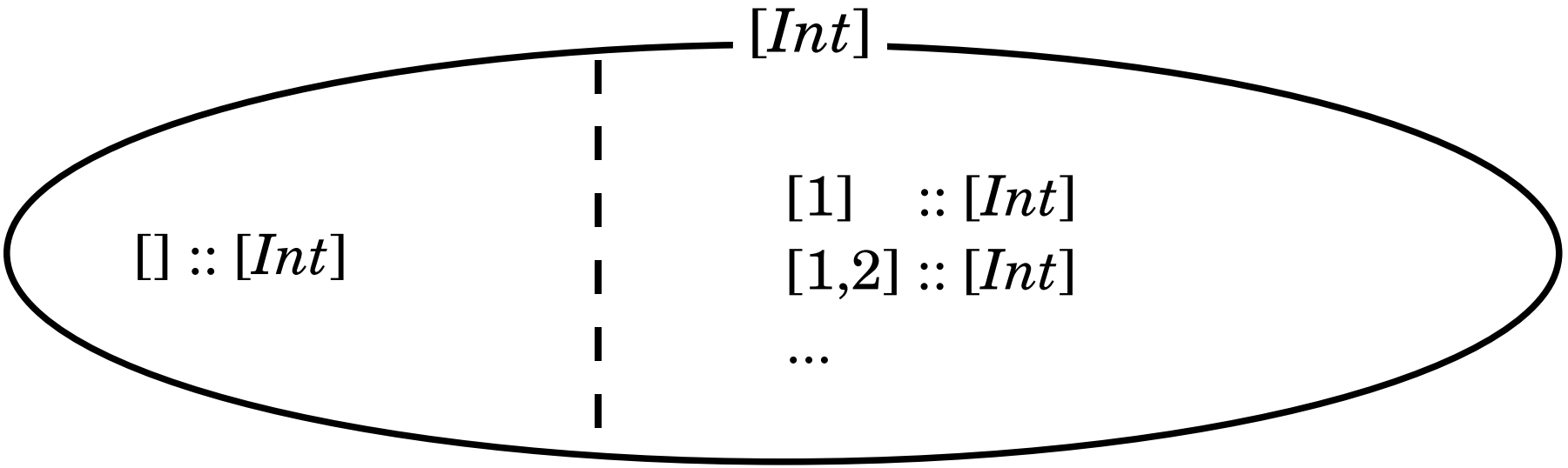
Newtype Summary

$$
\begin{tikzcd}
Inch \arrow[r, leftrightarrow, "\mathcal{N}_{\mathrm{T}}"] & Double \\
Inch\ 2.5 \arrow[u, "::"'] \arrow[r, leftrightarrow, "\mathcal{N}_{\mathrm{V}}"] & 2.5 \arrow[u, "::"']
\end{tikzcd}
$$

$$
\begin{aligned}
\mathcal{N}_{\mathrm{T}} &= \{Inch \ \simeq \ Double\} \\
\mathcal{N}_{\mathrm{V}} &= \{Inch\ x \ \simeq \ x\}
\end{aligned}
$$

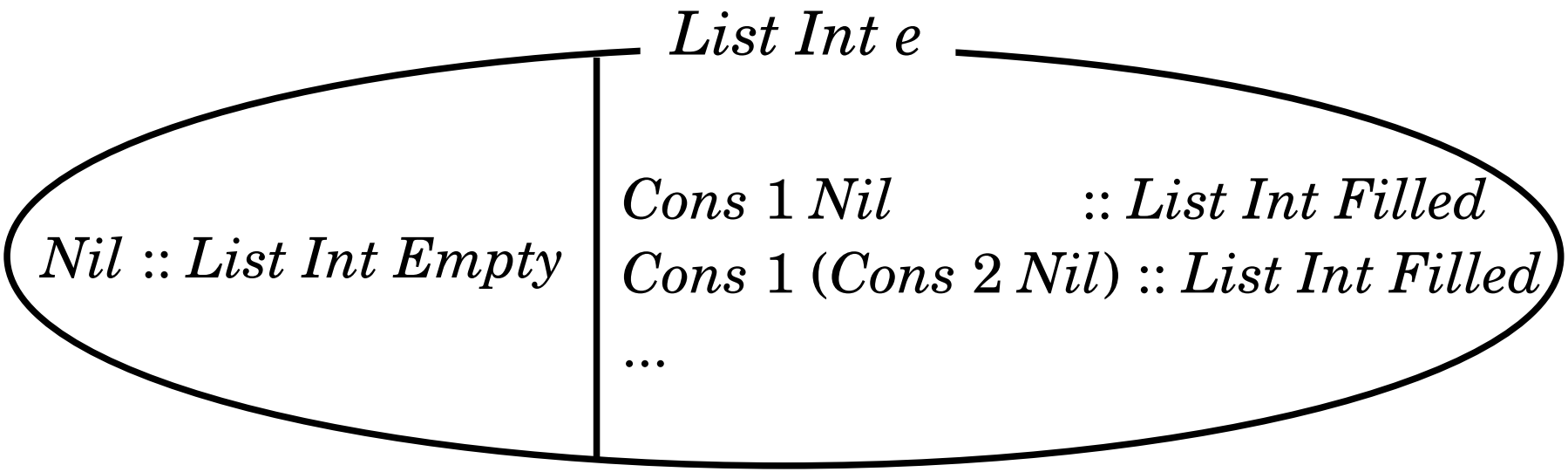
Subtyping by Refinement

- We can Partition a type (e.g. lists) into subsets (e.g. empty lists and filled lists)
- Each subset is a subtype
- Possible with GADTs and type indexes



Subtyping by Refinement

- We can Partition a type (e.g. lists) into subsets (e.g. empty lists and filled lists)
- Each subset is a subtype
- Possible with GADTs and type indexes



GADT = Generalized ADT

We can express any ADT as GADT

```
data [a]    -- ADT
  = []
  | (:) a [a]

data [a] where    -- GADT
  [] :: [a]
  (:) :: a → [a] → [a]
```

The GADT $(List\ a\ e)$

With GADTs, we can define richer types

by using different type indexes
in the result types of data constructors

$$\mathbf{data}\ Empty \quad \text{-- empty index type}$$
$$\mathbf{data}\ Filled \quad \text{-- empty index type}$$

$$\mathbf{data}\ List\ a\ e\ \mathbf{where}$$
$$\quad Nil \quad :: List\ a\ Empty$$
$$\quad Cons :: a \rightarrow List\ a\ e \rightarrow List\ a\ Filled$$

The GADT $(List\ a\ e)$

With GADTs, we can define richer types

by using different type indexes
in the result types of data constructors

```
data Empty    -- empty index type
data Filled   -- empty index type

data List a e where
  Nil  :: List a Empty
  Cons :: a → List a e → List a Filled
```

Values of $(List\ a\ e)$

$$
\begin{aligned}
&\textbf{data}\ List\ a\ e\ \textbf{where}\\
&\quad Nil \quad :: List\ a\ Empty\\
&\quad Cons :: a \rightarrow List\ a\ e \rightarrow List\ a\ Filled
\end{aligned}
$$

$$
\begin{aligned}
Nil \qquad\qquad\qquad &:: List\ a\ Empty\\
Cons\ True\ Nil &:: List\ Bool\ Filled
\end{aligned}
$$

Relating $(List\ a\ e)$ and $[a]$

$$
\begin{aligned}
&\textbf{data}\ List\ a\ e\ \textbf{where}\\
&\quad Nil \quad :: List\ a\ Empty\\
&\quad Cons :: a \rightarrow List\ a\ e \rightarrow List\ a\ Filled
\end{aligned}
$$

$$
\begin{aligned}
Nil &:: List\ a\ Empty\\
Cons\ True\ Nil &:: List\ Bool\ Filled
\end{aligned}
$$

$$
\begin{aligned}
Nil &\sim [\,]\\
Cons\ True\ Nil &\sim (:)\ True\ [\,]
\end{aligned}
$$

Relating $(List\ a\ \epsilon)$ and $[a]$

$$
\begin{array}{l}
\textbf{data}\ List\ a\ \epsilon\ \textbf{where} \\
\quad Nil \quad :: List\ a\ Empty \\
\quad Cons :: a \rightarrow List\ a\ \epsilon \rightarrow List\ a\ Filled
\end{array}
$$

$$
\begin{array}{ll}
Nil & :: List\ a\ Empty \\
Cons\ True\ Nil & :: List\ Bool\ Filled
\end{array}
$$

$$
\begin{array}{ll}
Nil & \sim\ [\,] \\
Cons\ True\ Nil & \sim\ (:)\ True\ [\,]
\end{array}
$$

Subtyping GADT Summary

$$
\begin{tikzcd}
List\ Int\ Filled \arrow[r, "\mathcal{S}_{\mathrm{T}}"] & [Int] \\
Cons\ 0\ Nil \arrow[u, "::"'] \arrow[r, "\mathcal{S}_{\mathrm{V}}"] & (:)\ 0\ [\,] \arrow[u, "::"']
\end{tikzcd}
$$

$$
\begin{aligned}
\mathcal{S}_{\mathrm{T}} &= \{List\ a\ e < [\,a\,]\} \\
\mathcal{S}_{\mathrm{V}} &= \{Nil < [\,],\ Cons < (:)\}
\end{aligned}
$$

Subtyping GADT Summary

$$\begin{tikzcd}
List\ Int\ Filled \arrow[r, "\mathcal{S}_{\mathrm{T}}"] & [Int] \\
Cons\ 0\ Nil \arrow[u, "::"] \arrow[r, "\mathcal{S}_{\mathrm{V}}"] & (:)\ 0\ [\,] \arrow[u, "::"']
\end{tikzcd}$$

Subtype rule

$$\mathcal{S}_{\mathrm{T}} = \{List\ a\ e < [a]\}$$
$$\mathcal{S}_{\mathrm{V}} = \{Nil < [\,],\ Cons < (:)\}$$

Sharing rules

Comparison

$$\begin{tikzcd}
Inch \arrow[r, leftrightarrow, "\mathcal{N}_{\mathrm{T}}"] & Double \\
Inch\ 2.5 \arrow[u, "::"'] \arrow[r, leftrightarrow, "\mathcal{N}_{\mathrm{V}}"] & 2.5 \arrow[u, "::"']
\end{tikzcd}
\qquad
\begin{tikzcd}
List\ Int\ Filled \arrow[r, "\mathcal{S}_{\mathrm{T}}"] & [Int] \\
Cons\ 0\ Nil \arrow[u, "::"'] \arrow[r, "\mathcal{S}_{\mathrm{V}}"] & (:)\ 0\ [\,] \arrow[u, "::"']
\end{tikzcd}$$

$$\mathcal{N}_{\mathrm{T}} = \{Inch \ \simeq\ Double\}$$
$$\mathcal{N}_{\mathrm{V}} = \{Inch\ x \ \simeq\ x\}$$

$$\mathcal{S}_{\mathrm{T}} = \{List\ a\ e < [\,a\,]\}$$
$$\mathcal{S}_{\mathrm{V}} = \{Nil < [\,],\ Cons < (:)\}$$

Newtypes

$\mathcal{N}_{\mathrm{T}}$, $\mathcal{N}_{\mathrm{V}}$ are bijections

Language Feature !!!

Subtyping (G)ADTs

$\mathcal{S}_{\mathrm{T}}$, $\mathcal{S}_{\mathrm{V}}$ are injections

Just a *CONCEPT* (until now!)

Options in Haskell

1st option (Code Reuse)

Conversion function that satisfies $\mathcal{S}_{\mathrm{T}}$ and $\mathcal{S}_{\mathrm{V}}$

$$
\begin{aligned}
&\mathit{fromList} :: \mathit{List}\ a\ e \rightarrow [\,a\,] \\
&\mathit{fromList}\ \mathit{Nil} \qquad\qquad\quad = [\,] \\
&\mathit{fromList}\ (\mathit{Cons}\ x\ xs) = (:)\ x\ (\mathit{fromList}\ xs)
\end{aligned}
$$

Problem: *O(n)* conversion overhead

2nd option (Efficiency)

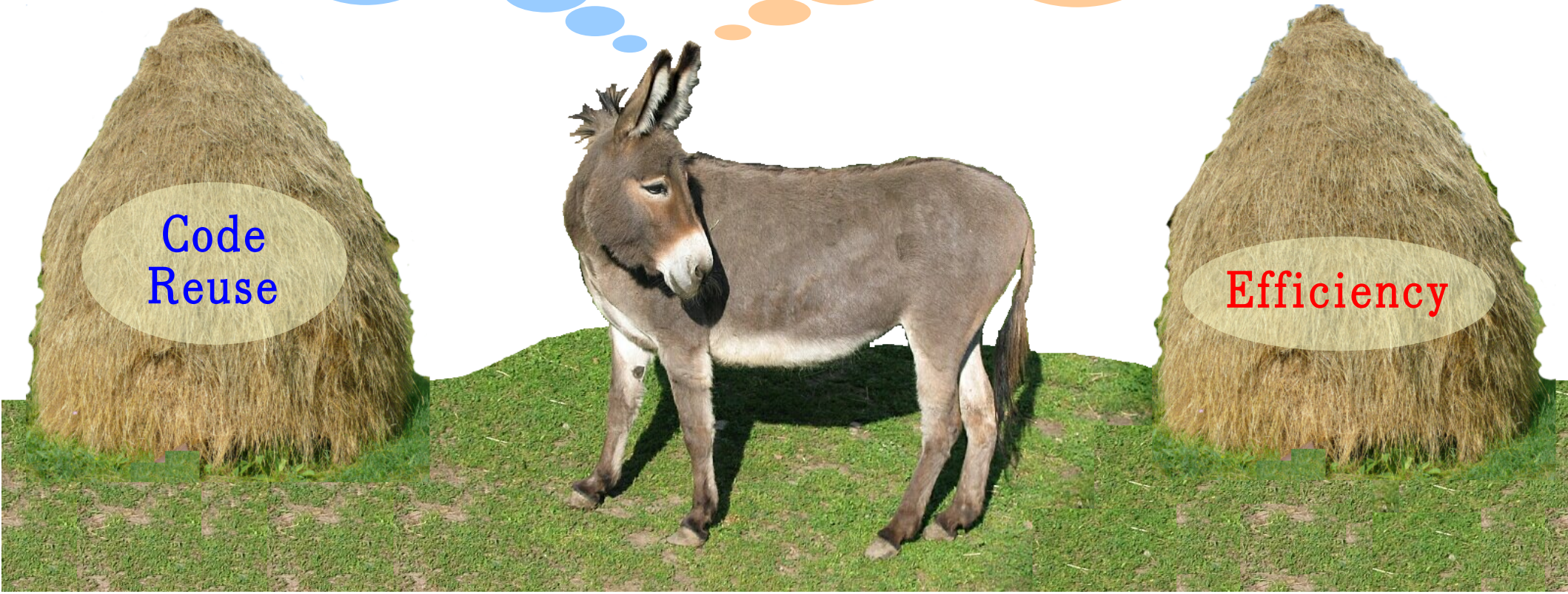
Define library functions for GADTs all again

Problem: code duplication

Dilemma

lose efficiency because of conversion overhead

lose code reuse writing same functions for each type



Features of Shared Subtypes

We must specify both

- subtype rule (types)
- sharing rules (constructors)

Check consistency

- not all SSubtype declaration make sense

We share representation (like newtypes)

We have implicit coercion (newtypes are explicit)

SSubtypes form hierarchy (as in OO)

Efficient code sharing and library reuse

SSubtype Syntax for GADT

$$
\begin{tikzcd}
List\ Int\ Filled \arrow[r, "\mathcal{S}_{\mathrm{T}}"] & [\,Int\,] \\
Cons\ 0\ Nil \arrow[u, "::"'] \arrow[r, "\mathcal{S}_{\mathrm{V}}"] & (:)\ 0\ [\,] \arrow[u, "::"']
\end{tikzcd}
$$

$$\mathcal{S}_{\mathrm{T}} = \{List\ a\ e < [\,a\,]\}$$
$$\mathcal{S}_{\mathrm{V}} = \{Nil < [\,],\ Cons < (:)\}$$

SSubtype Syntax for GADT

$$
\begin{array}{l}
\textbf{data}\ List\ a\ e < [\,a\,]\ \textbf{where} \\
\quad Nil \quad < [\,] \ :: List\ a\ Empty \\
\quad Cons < (:) :: a \rightarrow List\ a\ e \rightarrow List\ a\ Filled
\end{array}
$$

$$
\begin{tikzcd}
List\ Int\ Filled \arrow[r, "\mathcal{S}_{\mathrm{T}}"] & {[\,Int\,]} \\
Cons\ 0\ Nil \arrow[u, "::"'] \arrow[r, "\mathcal{S}_{\mathrm{V}}"] & (:)\ 0\ [\,] \arrow[u, "::"']
\end{tikzcd}
$$

$$
\begin{array}{l}
\mathcal{S}_{\mathrm{T}} = \{List\ a\ e < [\,a\,]\} \\
\mathcal{S}_{\mathrm{V}} = \{Nil < [\,],\ Cons < (:)\}
\end{array}
$$

SSubtype Syntax for GADT

Subtype rule

$$\textbf{data}\ List\ a\ e < [a]\ \textbf{where}$$

$$Nil < [\,] :: List\ a\ Empty$$

$$Cons < (:) :: a \rightarrow List\ a\ e \rightarrow List\ a\ Filled$$

Sharing rules

$$\begin{tikzcd}
List\ Int\ Filled \arrow[r, "\mathcal{S}_{\mathrm{T}}"] & [Int] \\
Cons\ 0\ Nil \arrow[u, "::"] \arrow[r, "\mathcal{S}_{\mathrm{V}}"] & (:)\ 0\ [\,] \arrow[u, "::"]
\end{tikzcd}$$

$$\mathcal{S}_{\mathrm{T}} = \{List\ a\ e < [a]\}$$

$$\mathcal{S}_{\mathrm{V}} = \{Nil < [\,],\ Cons < (:)\}$$

Code Reuse for SSubtype

We can reuse library function on lists

$$length :: [\,a\,] \rightarrow Int$$

Values of $[a]$

$$length\ [\,] \rightsquigarrow 0$$

$$length\ ((:)\ True\ [\,]) \rightsquigarrow 1$$

Values of *List a e*

$$length\ Nil \rightsquigarrow 0$$

$$length\ (Cons\ True\ Nil) \rightsquigarrow 1$$

Code Reuse for SSubtype

Subtype rule

$$\textbf{data}\ List\ a\ e < [a]\ \textbf{where}$$

$$Nil < [\,] :: List\ a\ Empty$$

$$Cons < (:) :: a \rightarrow List\ a\ e \rightarrow List\ a\ Filled$$

Sharing rules

$$length\ [\,] \rightsquigarrow 0$$

$$length\ ((:)\ True\ [\,]) \rightsquigarrow 1$$

$$length\ Nil \rightsquigarrow 0$$

$$length\ (Cons\ True\ Nil) \rightsquigarrow 1$$

Consistency of SSubtypes

Not all SSubtype declaration make sense

Example: sharing between different arities

$$
\begin{array}{l}
\textbf{data}\ List''\ a\ e < [a]\ \textbf{where} \\
\quad Nil'' \quad < (:) :: List''\ a\ Empty \\
\quad Cons'' < [\,] \ :: a \rightarrow List''\ a\ e \rightarrow List''\ a\ Filled
\end{array}
$$

$$\mathcal{S}''_{\mathrm{V}} \ = \ \{\underline{Nil'' < (:)},\ \underline{Cons'' < [\,]}\}$$

What are the criteria for consistency?

And, how shall we check them?

Consistency of SSubtypes

Not all SSubtype declaration make sense

Example: sharing between different arities

$$\textbf{data}\ List''\ a\ e < [a]\ \textbf{where}$$
$$Nil'' < (:) :: List''\ a\ Empty$$
$$Cons'' < [] :: a \rightarrow List''\ a\ e \rightarrow List''\ a\ Filled$$

$$\mathcal{S}''_{\mathrm{V}} = \{\underline{Nil'' < (:)},\ \underline{Cons'' < []}\}$$

What are the criteria for consistency?

And, how shall we check them?

Consistent SSubtype decl's

No cycles in SSubtype hierarchy

- Cycle means equivalence ($T \leq U \wedge T \geq U \Rightarrow T = U$)
- Newtype is the feature for equivalent types

No two subtype data constructors can share the same supertype constructor

Sharing rules are consistent with the subtype rule (see next slide)

No higher order types (no function args)

- additional restriction at present (future work)

Sharing rules are consistent with Subtype rule

Related data constructors must have same arity

Argument types and result types must be in subtype relation (+reflexivity for args)

We don't worry about contravariance at present because we don't have higher order types yet

Sharing rules are consistent with Subtype rule

Related data constructors must have same arity

Argument types and result types must be in subtype relation (+reflexivity for args)

$$\begin{array}{l} \textbf{data}\ [a]\ \textbf{where} \qquad \text{-- GADT} \\ \quad [\,] \ :: [a] \\ \quad (:) :: a \to [a] \to [a] \end{array}$$

$$\begin{array}{l} \textbf{data}\ \mathit{List}\ a\ e < [a]\ \textbf{where} \\ \quad \mathit{Nil} \quad < [\,] \ :: \mathit{List}\ a\ \mathit{Empty} \\ \quad \mathit{Cons} < (:) :: a \to \mathit{List}\ a\ e \to \mathit{List}\ a\ \mathit{Filled} \end{array}$$

Consistent SSubtype decl'

Related data constructors must have same arity

Argument types and result types must be in subtype relation (+reflexivity for args)

$$\frac{List\ a\ Empty \prec [a]}{Nil < []}$$

$$\frac{\dfrac{a \preccurlyeq a \qquad List\ a\ e \preccurlyeq [a] \qquad List\ a\ Filled \prec [a]}{a \to List\ a\ e \to List\ a\ Filled \ \prec\ a \to [a] \to [a]}}{Cons < (:)}$$

SSubtype Syntax for ADT

$$
\begin{aligned}
\textbf{data}\ Exp &= Lit\ Int \\
&\mid\ Add\ Exp\ Exp \\
&\mid\ Mul\ Exp\ Exp \\
\textbf{data}\ Term < Exp &= (Const < Lit)\ Int \\
&\mid\ (Times < Mul)\ Term\ Term
\end{aligned}
$$

Since *Term* is a SSubtype of *Exp*, we can reuse functions defined on *Exp* for *Term*.

SSubtype Syntax for ADT

$$
\begin{aligned}
\mathbf{data}\ Exp \quad & = Lit\ Int \\
& \mid\ Add\ Exp\ Exp \\
& \mid\ Mul\ Exp\ Exp \\
\mathbf{data}\ Term < Exp & = (Const < Lit)\ Int \\
& \mid\ (Times < Mul)\ Term\ Term
\end{aligned}
$$

If we have $eval :: Exp \to Int$ then

$eval\ (Const\ 2\ \text{`}Times\text{`}\ Const\ 3) \rightsquigarrow 6$

SSubtype Syntax for ADT

$$
\begin{aligned}
\textbf{data}\ Exp \quad &= Lit\ Int \\
&\mid Add\ Exp\ Exp \\
&\mid Mul\ Exp\ Exp \\
\textbf{data}\ Term < Exp &= (Const < Lit)\ Int \\
&\mid (Times < Mul)\ Term\ Term
\end{aligned}
$$

If we have $eval :: Exp \rightarrow Int$ then

$$eval\ \underbrace{(Const\ 2\ `Times`\ Const\ 3)}_{::\ Term} \rightsquigarrow 6$$

Finite Subsets of Integers

Foreign Function Interface can be
both safe and efficient using SSubtypes

$$
\begin{array}{lllll}
\textbf{data}\ Bool & < Int = & True & < 1 \\
 & \quad\ \mid & False & < 0 \\
\textbf{data}\ Signal & < Int = & SIGTERM & < 15 \\
 & \quad\ \mid & SIGSEGV & < 11 \\
 & \quad\ \mid & \ldots\ \ldots &
\end{array}
$$

Related Feature: F# enum types are just for this
purpose to interact with .NET

Example from FPH (ICFP'08)

by Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones

Syntax of Types in the paper

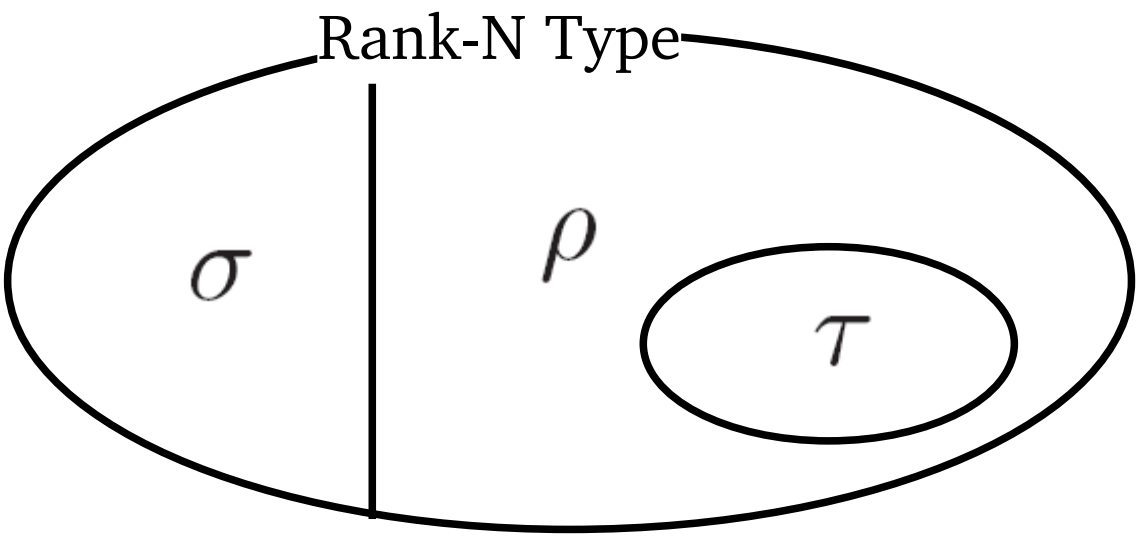
$$
\begin{array}{lcl}
\sigma & ::= & \forall \overline{a} \,.\, \rho \\
\rho & ::= & \tau \mid \sigma \to \sigma \mid \mathtt{T}\,\overline{\sigma} \\
\tau & ::= & a \mid \tau \to \tau \mid \mathtt{T}\,\overline{\tau}
\end{array}
$$

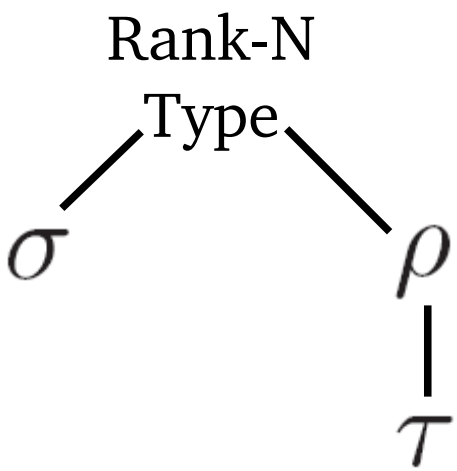
Example from FPH (ICFP'08)

by Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones

Syntax of Types in the paper

$$
\begin{array}{lcl}
\sigma & ::= & \forall \overline{a} \,.\, \rho \\
\rho & ::= & \tau \mid \sigma \to \sigma \mid \mathtt{T}\,\overline{\sigma} \\
\tau & ::= & a \mid \tau \to \tau \mid \mathtt{T}\,\overline{\tau}
\end{array}
$$



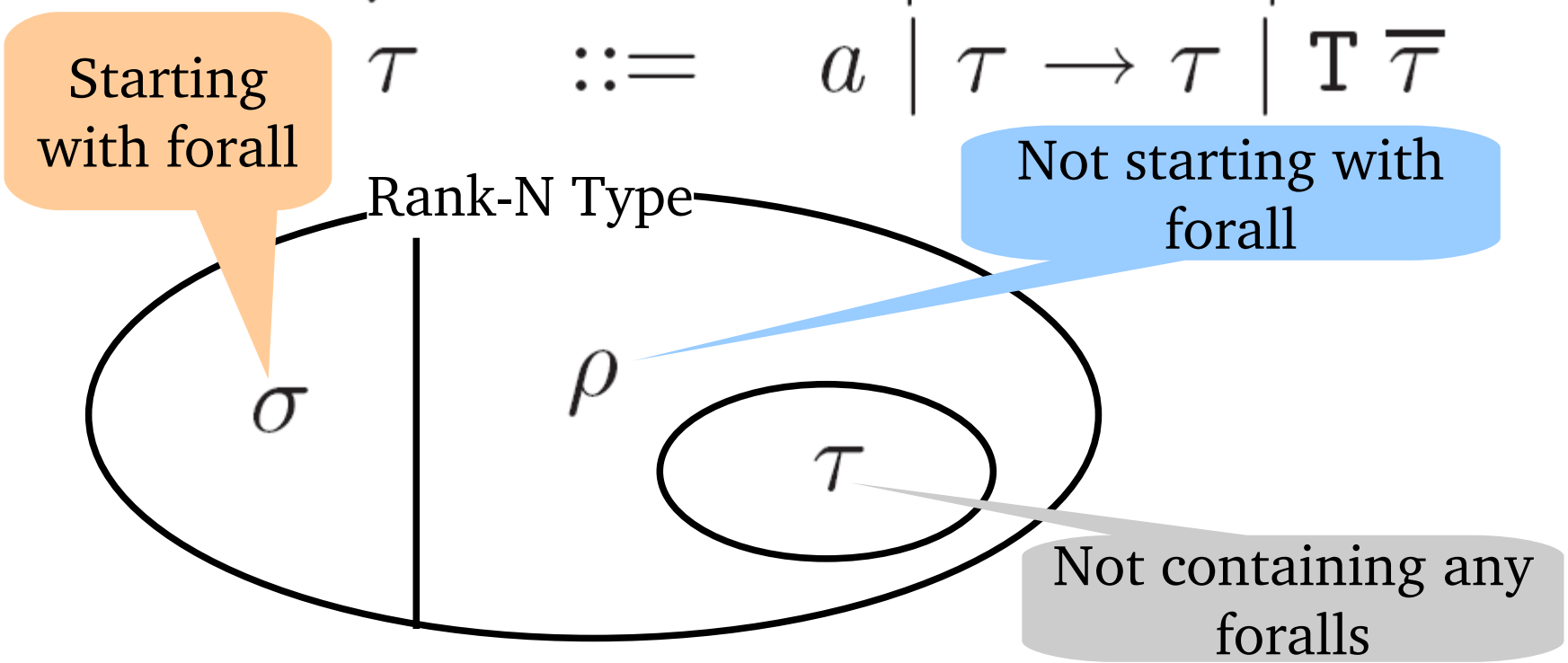


Example from FPH (ICFP'08)

by Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones

Syntax of Types in the paper

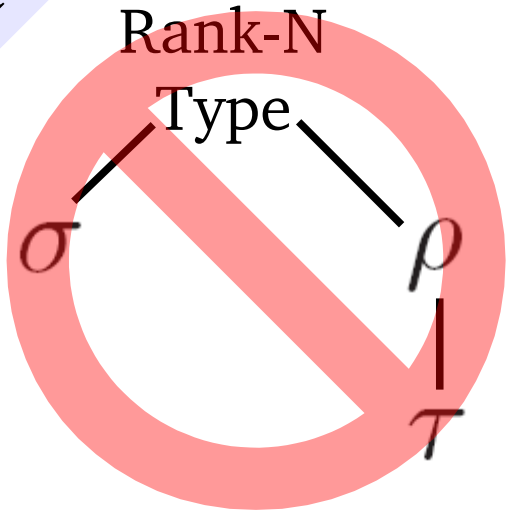
$$
\begin{array}{lcl}
\sigma & ::= & \forall \overline{a} \,.\, \rho \\
\rho & ::= & \tau \mid \sigma \to \sigma \mid \mathtt{T}\ \overline{\sigma} \\
\tau & ::= & a \mid \tau \to \tau \mid \mathtt{T}\ \overline{\tau}
\end{array}
$$



Example from FPH (ICFP'08)

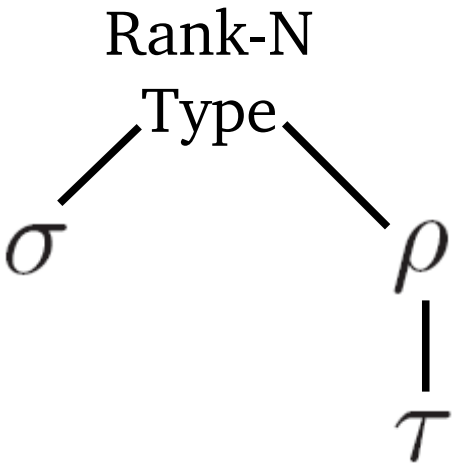
```
data Type = ForAll [Var] Rho
          | Fun Sigma Sigma
          | TyCon Name [Sigma]
          | TyVar Var

type Sigma = Type
type Rho   = Type
type Tau   = Type
```

Code snippet from their reference implementation (with slight simplification)



SSubtype Hierarchy for FPH

$$
\begin{aligned}
&\textbf{data}\ Sigma < Type \\
&\quad = (Forall < ForAll)\ Var\ Rho \\
&\textbf{data}\ Rho \quad < Type \\
&\quad = (Arrow < Fun)\ Type\ Type \\
&\quad \mid (Con < TyCon)\ Name\ [Type] \\
&\quad \mid (Var < TyVar)\ Var \\
&\textbf{data}\ Tau \quad < Rho \\
&\quad = (A < Arrow)\ Tau\ Tau \\
&\quad \mid (C < Con)\ Name\ [Type] \\
&\quad \mid (V < Var)\ Var
\end{aligned}
$$



Limitations

We may lose some static properties when we reuse the functions on the supertype

$$map :: (a \to b) \to [a] \to [a]$$
$$map\ id\ (Cons\ 0\ Nil) :: [Int] \text{ -- not } List\ a\ Filled$$
$$map\ id\ Nil :: [Int] \text{ -- not } List\ a\ Empty$$

In order to preserve such static properties, we have to write such functions again

$$mapList :: (a \to b) \to List\ a\ e \to List\ b\ e$$
$$mapList\ f\ Nil = Nil$$
$$mapList\ f\ (Cons\ x\ xs) = Cons\ x\ (mapList\ f\ xs)$$

Related Work

Language Features

Datatype Subtyping in O'Haskell (Nordlander)

Refinement Types (Freeman & Pfenning)

Theory

Subtyping Recursive Types (Amadio & Cardelli)

Implicit Calculus of Constructions (Miquel)

Related Work

Language Features

Datatype Subtyping in O'Haskell (Nordlander)

- Similar syntax and idea with SSubtypes
- Extends existing ADTs (c.f. SSubtype restricts)
- Shares constructor names between related types
- Does not cover recursive types or GADTs

Related Work

Language Features

Refinement Types (Freeman & Pfenning)

- Based on intersection types
- More powerful than subtyping (e.g. *map*, ++)
- Types can get bulky
- Separate compilation is questionable
- Polymorphic variants are related to this too

Related Work

Theory

Subtyping Recursive Types (Amadio & Cardelli)

- Build types from ground types using $\times$, $+$, $\rightarrow$
- Recursive type equation using μ notation
- Our work conforms to the rules in their work

Related Work

Theory

- Implicit Calculus of Constructions (Miquel)
  - Defines subtype relation as a Macro
  - Data constructors are encoded as λ–terms, and subtype relation means that these terms coincide
  - SSubtype restores this lost relation in (G)ADTs

Future Work

(G)ADTs containing higher order values

relax from Contravariance Free types
to Variance Consistent types

Complete proof that covers all uses

Harmonize SSubtypes with Type Classes

Possibility of adopting O'Haskell features

Non-variable parameters in subtype rules

Implementation

Conclusion

SSubtype is subtyping by sharing representations

- c.f. Usual subtyping (as in OO) is by sharing interface of different representation

We can use SSubtypes when we want to

- Restrict number of data constructors
- Ensure static property using type indexes
- reuse library without efficiency loss

Separate compilation (tradeoff for the limitation)

GADTs makes this idea more powerful

Thanks to

Everyone in the Acknowledgment section in the paper.

Also, those who gave useful comments on the practice talk (Andrew P. Black, Emerson Murphy Hill, Brian Huffman, Mark P. Jones).

And, YOU for listening.

Questions?

Name Sharing

$$
\begin{aligned}
\textbf{data}\ Type = {} & ForAll\ [\,Var\,]\ Rho \\
\mid\ & Fun\ Type\ Type \\
\mid\ & TyCon\ Name\ [\,Type\,] \\
\mid\ & TyVar\ Var
\end{aligned}
$$

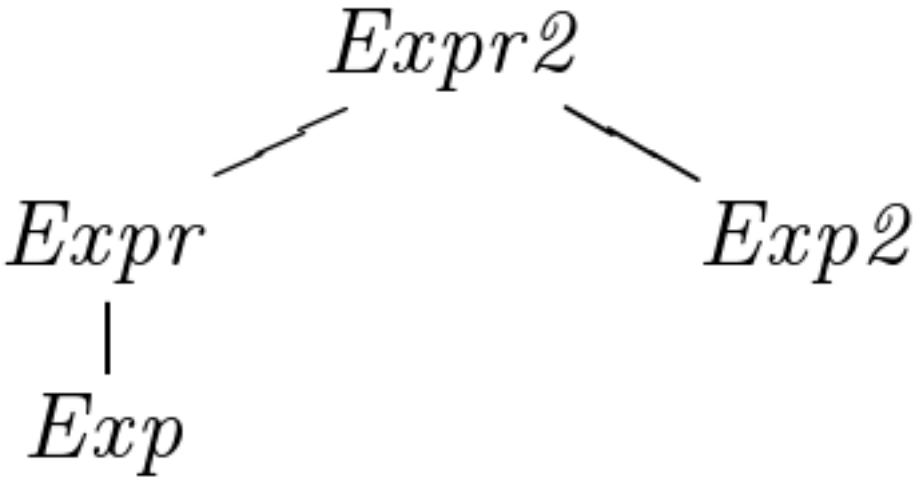
$$
\textbf{data}\ Sigma < Type = ForAll\ Var\ Rho
$$

$$
\begin{aligned}
\textbf{data}\ Rho \quad < Type = {} & Fun\ Type\ Type \\
\mid\ & TyCon\ Name\ [\,Type\,] \\
\mid\ & TyVar\ Var
\end{aligned}
$$

$$
\begin{aligned}
\textbf{data}\ Tau \quad < Rho\ = {} & Fun\ Tau\ Tau \\
\mid\ & TyCon\ Name\ [\,Type\,] \\
\mid\ & TyVar\ Var
\end{aligned}
$$

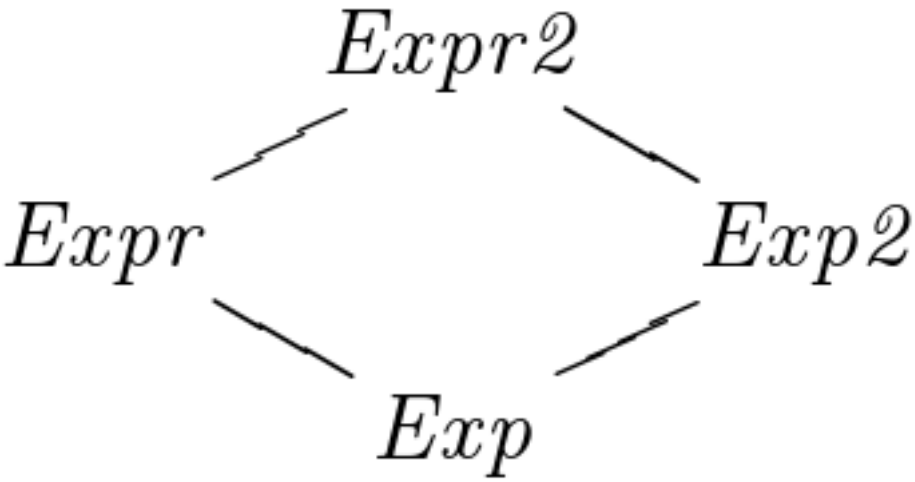
SSubtype Hierarchy

$$
\begin{array}{lll}
Expr2 & : +, *, & int, var, let \\
Expr & : +, & int, var, let \\
Exp2 & : +, *, & int \\
Exp & : +, & int
\end{array}
$$



SSubtype Hierarchy

$$
\begin{array}{lll}
Expr2 & : +, *, & int, var, let \\
Expr & : +, & int, var, let \\
Exp2 & : +, *, & int \\
Exp & : +, & int
\end{array}
$$



Type Inhabitation (§6.2)

$$\mathbf{data}\ IList\ a = (ConsI < (:))\ a\ (IList\ a)$$

$$\begin{array}{l}
\mathbf{data}\ U\ \ a\ \mathbf{where} \\
\quad D1 :: U\ (T\ \ a) \\
\quad D2 :: U\ (U\ \ a) \rightarrow U\ (U\ \ a) \\
\mathbf{data}\ T\ \ a < U\ \ a\ \mathbf{where} \\
\quad C1 < D1 :: T\ (T\ \ a) \\
\quad C2 < D2 :: T\ (U\ \ a) \rightarrow T\ (U\ \ a)
\end{array}$$