

Implementation of a Pragmatic Translation from Haskell into Isabelle/HOL

Patrick Bahr
pa-ba@arcor.de

NICTA Sydney, TU Wien

December 17, 2008

Outline

1 Introduction

- Haskell vs. Isabelle/HOL
- Motivation
- Goals

2 Translating Haskell into Isabelle/HOL

- Haskell vs. Isabelle/HOL
- Implementation

3 Conclusions

Outline

1 Introduction

- Haskell vs. Isabelle/HOL
- Motivation
- Goals

2 Translating Haskell into Isabelle/HOL

- Haskell vs. Isabelle/HOL
- Implementation

3 Conclusions

Haskell vs. Isabelle/HOL

Haskell in a nutshell

- **purely functional** programming language
- **non-strict** semantics (mostly implemented by **lazy evaluation**)
- comprehensive type system: **Hindley-Milner** (restricted F_ω) + **type classes**
- uses **monads** to allow side effects

Haskell vs. Isabelle/HOL

Haskell in a nutshell

- **purely functional** programming language
- **non-strict** semantics (mostly implemented by **lazy evaluation**)
- comprehensive type system: **Hindley-Milner** (restricted F_ω) + **type classes**
- uses **monads** to allow side effects

Isabelle/HOL in a nutshell

- Isabelle: **generic theorem prover**
- HOL: Isabelle formulation of classical **higher-order logic**
- based on **simply typed lambda calculus** (system F_1)
 \rightsquigarrow comparatively weak type system
- extended with **type classes**

Haskell vs. Isabelle/HOL

Haskell in a nutshell

- **purely functional** programming language
- **non-strict** semantics (mostly implemented by **lazy evaluation**)
- comprehensive type system: **Hindley-Milner** (restricted F_ω) + **type classes**
- uses **monads** to allow side effects

Isabelle/HOL in a nutshell

- Isabelle: **generic theorem prover**
- HOL: Isabelle formulation of classical **higher-order logic**
- based on **simply typed lambda calculus** (system F_1)
 \rightsquigarrow comparatively weak type system
- extended with **type classes**

more details when we come to the implementation

Motivation

Program verification

- Haskell's semantics allows comparatively **easy reasoning**
- there is **no theorem prover** for Haskell!
 \rightsquigarrow translate Haskell into language of a generic theorem prover

Motivation

Program verification

- Haskell's semantics allows comparatively **easy reasoning**
- there is **no theorem prover** for Haskell!
 \rightsquigarrow translate Haskell into language of a generic theorem prover

Example: I4.verified project

- aim: formalisation and verification of a microkernel
- prototype implementation in Haskell
- translation into Isabelle/HOL \rightsquigarrow executable model
- reasoning about executable model in Isabelle/HOL

Translation

Goals

- cover a **large subset** of Haskell's syntax
- result should be **easily readable**
 - ▶ preserve syntactic structure as much as possible
 - ▶ translate syntactic sugar as well
- keep **reasoning simple** \rightsquigarrow Isabelle/HOL

Translation

Goals

- cover a **large subset** of Haskell's syntax
- result should be **easily readable**
 - ▶ preserve syntactic structure as much as possible
 - ▶ translate syntactic sugar as well
- keep **reasoning simple** \rightsquigarrow Isabelle/HOL

\rightsquigarrow Translation is neither sound nor complete!

Translation

Goals

- cover a **large subset** of Haskell's syntax
- result should be **easily readable**
 - ▶ preserve syntactic structure as much as possible
 - ▶ translate syntactic sugar as well
- keep **reasoning simple** \rightsquigarrow Isabelle/HOL

\rightsquigarrow Translation is neither sound nor complete!

Implementation

- implementation language: Haskell
- based on existing work from TU Munich

Outline

- 1 Introduction
 - Haskell vs. Isabelle/HOL
 - Motivation
 - Goals
- 2 Translating Haskell into Isabelle/HOL
 - Haskell vs. Isabelle/HOL
 - Implementation
- 3 Conclusions

Haskell vs. Isabelle/HOL – Non-strictness/Partiality

- in Isabelle/HOL **only total functions** are definable
 \rightsquigarrow recursive definitions need termination proof
- Haskell is **Turing-complete** \rightsquigarrow partial functions definable
- Haskell's semantics is **non-strict**

Haskell vs. Isabelle/HOL – Non-strictness/Partiality

- in Isabelle/HOL **only total functions** are definable
 \rightsquigarrow recursive definitions need termination proof
- Haskell is **Turing-complete** \rightsquigarrow partial functions definable
- Haskell's semantics is **non-strict**

Example (Haskell)

```
from :: Int -> [Int]
from n = n : from (n+1)
```

Haskell vs. Isabelle/HOL – Non-strictness/Partiality

- in Isabelle/HOL **only total functions** are definable
 \rightsquigarrow recursive definitions need termination proof
- Haskell is **Turing-complete** \rightsquigarrow partial functions definable
- Haskell's semantics is **non-strict**

Example (Haskell)

```
from :: Int -> [Int]
from n = n : from (n+1)
```

- from does **not terminate** for any input
 \rightsquigarrow **not definable** in Isabelle/HOL
- due to non-strictness this function is still usable in Haskell

Haskell vs. Isabelle/HOL – Non-strictness/Partiality

- in Isabelle/HOL **only total functions** are definable
 \rightsquigarrow recursive definitions need termination proof
- Haskell is **Turing-complete** \rightsquigarrow partial functions definable
- Haskell's semantics is **non-strict**

Example (Haskell)

```
from :: Int -> [Int]
from n = n : from (n+1)
```

- from does **not terminate** for any input
 \rightsquigarrow **not definable** in Isabelle/HOL
- due to non-strictness this function is still usable in Haskell

Example (Haskell)

```
nPrimes :: Int -> [Int]
nPrimes n = take n (filter isPrime (from 1))
```


Haskell vs. Isabelle/HOL – Non-strictness/Partiality

- in Isabelle/HOL **only total functions** are definable
 \rightsquigarrow recursive definitions need termination proof
- Haskell is **Turing-complete** \rightsquigarrow partial functions definable
- Haskell's semantics is **non-strict**

Example (Haskell)

```
from :: Int -> [Int]
from n = n : from (n+1)
```

- from does **not terminate** for any input
 \rightsquigarrow **not definable** in Isabelle/HOL
- due to non-strictness this function is still usable in Haskell

Example (Haskell)

```
nPrimes :: Int -> [Int]
nPrimes n = take n (filter isPrime (from 1))
```

Definitions that depend on non-strictness have to be avoided!

Haskell vs. Isabelle/HOL – Local Function Definitions

- **Haskell** allows recursive function definitions in **local contexts** (using `let` or `where`)
- in **Isabelle/HOL** recursive function definitions are only allowed at the **top level**

Haskell vs. Isabelle/HOL – Local Function Definitions

- Haskell allows recursive function definitions in **local contexts** (using `let` or `where`)
- in Isabelle/HOL recursive function definitions are only allowed at the **top level**

Example (Haskell)

```
sumLen :: Int -> [a] -> [a] -> Int
sumLen s l1 l2 = let len [] = 0
                  len (x:xs) = len xs + s
                  in len l1 + len l2
```

Haskell vs. Isabelle/HOL – Local Function Definitions II

- local function definitions have to be moved to the top level
- closures have to be made explicit

Haskell vs. Isabelle/HOL – Local Function Definitions II

- local function definitions have to be moved to the top level
- closures have to be made explicit

Example (Isabelle/HOL)

```
fun len1
where
  "len1 _ Nil = 0"
| "len1 s (x # xs) = len1 s xs + s"

fun sumLen :: "int  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  int"
where
  "sumLen s l1 l2 = (let len = len1 s
                      in len l1 + len l2)"
```

Haskell vs. Isabelle/HOL – Local Function Definitions II

- local function definitions have to be moved to the top level
- closures have to be made explicit

Example (Isabelle/HOL)

```
fun len1
where
  "len1 _ Nil = 0"
| "len1 s (x # xs) = len1 s xs + 1"

fun sumLen :: "int ⇒ 'a list ⇒ 'a list ⇒ int"
where
  "sumLen s l1 l2 = (let len = len1 s
                      in len l1 + len l2)"
```

Our implementation is able to make these transformations!

Haskell vs. Isabelle/HOL – Order of Definitions

- in **Haskell** definitions can appear in **any order**
- in **Isabelle/HOL**:
 - ▶ an identifier has to be **defined before usage**
 - ▶ **mutual recursive** definitions have to be made **in parallel**

Haskell vs. Isabelle/HOL – Order of Definitions

- in **Haskell** definitions can appear in **any order**
- in **Isabelle/HOL**:
 - ▶ an identifier has to be **defined before usage**
 - ▶ **mutual recursive** definitions have to be made **in parallel**

Our implementation reorders definitions accordingly!

Haskell vs. Isabelle/HOL – Polymorphism

- **Haskell**: polymorphism over type constructors (of arbitrary kind)
- **Isabelle/HOL**: polymorphism over types only

Example (type constructors)

- types (constructors of kind $*$): `Int`, `[Bool]`, `Int -> Bool`, ...
- type constructors of first-order kind: `list ([]: * -> *)`, `sum`
(**Either**: $* \rightarrow (* \rightarrow *)$)
- type constructor of higher-order kind: `Tree: (* -> *) -> (* -> *)`

```
data Tree c a = Node a (c (Tree c a))
```

Haskell vs. Isabelle/HOL – Ad Hoc Polymorphism

- Haskell: type classes + constructor classes
- Isabelle/HOL: type classes only

Haskell vs. Isabelle/HOL – Ad Hoc Polymorphism

- Haskell: type classes + constructor classes
- Isabelle/HOL: type classes only

Example (classes)

- type class:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*)      :: a -> a -> a
  negate            :: a -> a
                    :
```

Haskell vs. Isabelle/HOL – Ad Hoc Polymorphism

- Haskell: type classes + constructor classes
- Isabelle/HOL: type classes only

Example (classes)

- type class:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*)      :: a -> a -> a
  negate            :: a -> a
  ...
```

- constructor class:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Haskell vs. Isabelle/HOL – Ad Hoc Polymorphism II

- monad class is **not definable** in Isabelle/HOL!
- monads are crucial for practical **Haskell** programs
- monads can be used to describe computations with side effects

Haskell vs. Isabelle/HOL – Ad Hoc Polymorphism II

- monad class is **not definable** in Isabelle/HOL!
- monads are crucial for practical **Haskell** programs
- monads can be used to describe computations with side effects

Our solution

- Translate only **instances** of the class **Monad**!
- each monad instance has to use **different names** for the operation
e.g. one monad uses `>>=`, `return`; another one uses `>>='`, `return'`
- **type inference** has to be performed to rename the operations correctly
- not full type inference is used, only a simple heuristics

Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns

Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns

Example

In Haskell:

```
f :: [Int] -> [Int]
f l@(_:_) = 0 : l
f l@([])  = 1 : l
```


Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns

Example

In Haskell:

```
f :: [Int] -> [Int]
f l@(_:_) = 0 : l
f l@([])  = 1 : l
```

In Isabelle/HOL:

```
fun f where
  "f (a0 # a1)
   = (let l = (a0 # a1)
        in 0 # l)"
| "f Nil = (let l = Nil
              in 1 # l)"
```

Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns
- labelled fields in data types

Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns
- labelled fields in data types

Example (Haskell)

```
data MyRecord = A { aField1 :: String,
                   common1  :: Bool,
                   common2  :: Int }
              | B { bField1  :: Bool,
                   bField2  :: Int,
                   common1  :: Bool,
                   common2  :: Int }
              | C Bool Int String
```

Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns
- labelled fields in data types

Example (Haskell)

```
data MyRecord = A { aField1 :: String,
                   common1  :: Bool,
                   common2  :: Int }
              | B { bField1  :: Bool,
                   bField2  :: Int,
                   common1  :: Bool,
                   common2  :: Int }
              | C Bool Int String
```

↪ This is reduced to an ordinary data type!

Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns
- labelled fields in data types
- guards

Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns
- labelled fields in data types
- guards

Example (Haskell)

```
insert :: Int -> [Int] -> [Int]
insert n [] = [n]
insert n (m:ms)
  | n < m      = n:m:ms
  | otherwise = m: insert n ms
```

Haskell vs. Isabelle/HOL – Misc.

Further things that are taken care of in the translation

- as-patterns
- labelled fields in data types
- guards

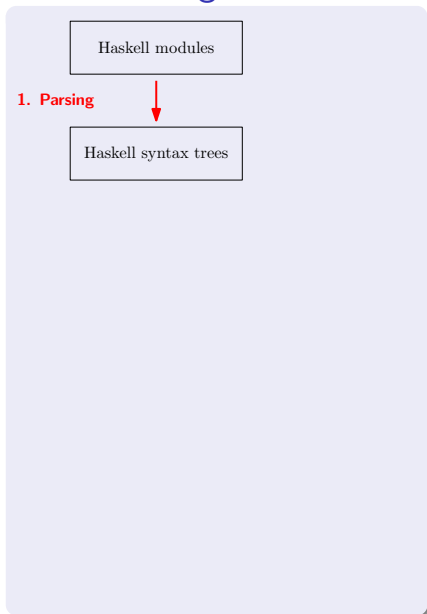
Example (Haskell)

```
insert :: Int -> [Int] -> [Int]
insert n [] = [n]
insert n (m:ms)
  | n < m      = n:m:ms
  | otherwise = m: insert n ms
```

↪ Guards are reduced to if-then-else expressions!

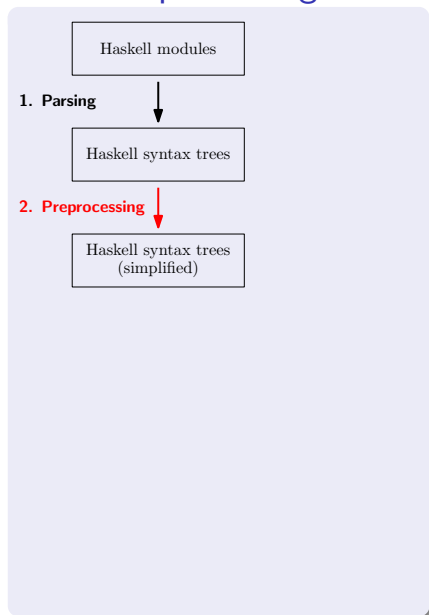
Overall Design of Implementation – Parsing

- parse each Haskell module to a **syntax tree**
- **imported modules** are located and parsed as well
- parser only verifies **context-free** part of the syntax
- syntactically correct Haskell program is assumed



Overall Design of Implementation – Preprocessing

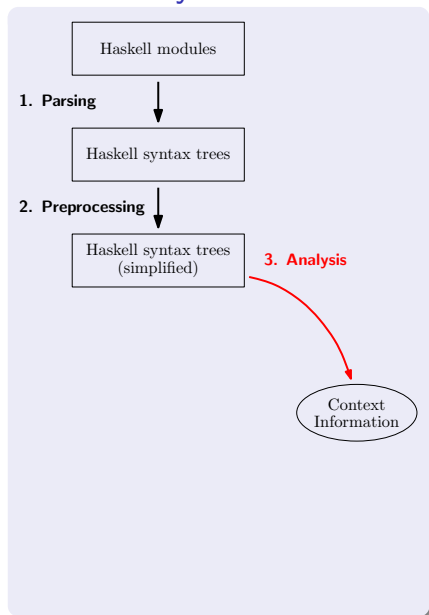
- **Guards** are transformed into **if-then-else** expressions.
- **Local function definitions** are transformed into top-level function definitions.
- **Keywords** and identifiers defined in the Isabelle/HOL **library** are renamed.



Overall Design of Implementation – Analysis

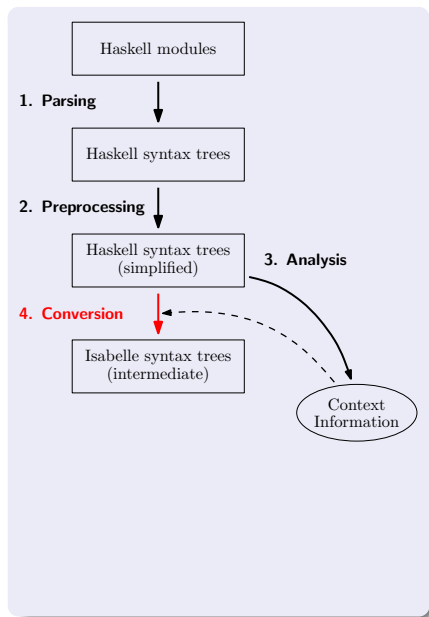
Some global information about the program is collected:

- type annotations
- the module where an identifier was defined
- what an identifier refers to (type, function etc.)
- associativity and precedence of defined operators



Overall Design of Implementation – Conversion

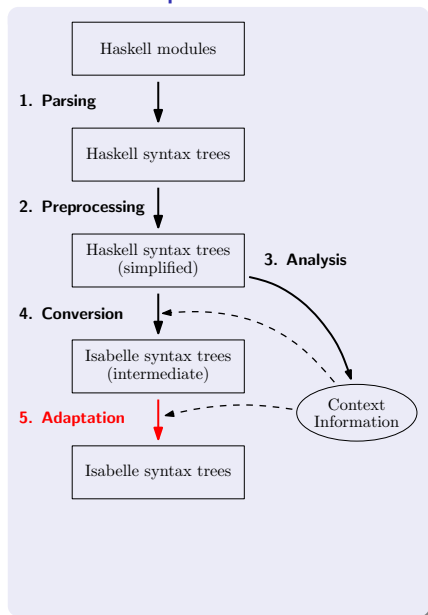
- Definitions are **reordered** according to their dependencies.
- Haskell syntax trees are translated into Isabelle/HOL syntax trees.



Overall Design of Implementation – Adaptation

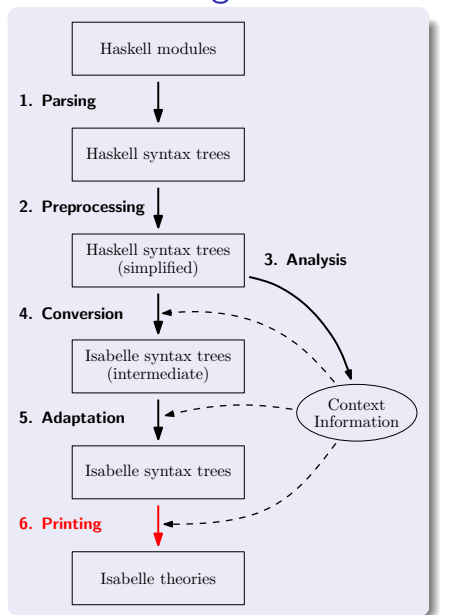
Renaming of predefined identifiers, e.g.:

- `Int` \mapsto `int`
- `[]` \mapsto `Nil`
- `++` \mapsto `@`



Overall Design of Implementation – Printing

- Isabelle/HOL syntax trees are written into theory files.



Outline

- 1 Introduction
 - Haskell vs. Isabelle/HOL
 - Motivation
 - Goals
- 2 Translating Haskell into Isabelle/HOL
 - Haskell vs. Isabelle/HOL
 - Implementation
- 3 Conclusions

Summary

Original implementation covered

- case, if-then-else, and let expressions
- list comprehensions
- where bindings
- as-patterns
- guards
- mutually recursive functions and data type definitions
- simple pattern bindings
- definitions and instantiations of type classes

Summary

Original implementation covered

- case, if-then-else, and let expressions
- list comprehensions
- where bindings
- ✗ as-patterns
- ✗ guards
- ✗ mutually recursive functions and data type definitions
- simple pattern bindings
- definitions and instantiations of type classes

Some parts of the translations were unsound!

Summary II

Our Contributions

- ✓ mutually recursive function and data type definitions
- ✓ as-patterns
- ✓ guards
 - data types with labelled fields
 - closures in local function definitions
 - monomorphic uses of monads

Summary II

Our Contributions

- ✓ mutually recursive function and data type definitions
- ✓ as-patterns
- ✓ guards
 - data types with labelled fields
 - closures in local function definitions
 - monomorphic uses of monads

What is missing

- constructor type classes \rightsquigarrow polymorphic uses of monads
- non-simple pattern bindings
- irrefutable patterns

Conclusions

What do we have

- translation is unsound!
- most of the Haskell 98 language can be translated
- resulting Isabelle/HOL formalisation is close to Haskell program
- comparatively easy reasoning in Isabelle/HOL
- adequate translation for most purposes \rightsquigarrow I4.verified

Conclusions

What do we have

- translation is unsound!
- most of the Haskell 98 language can be translated
- resulting Isabelle/HOL formalisation is close to Haskell program
- comparatively easy reasoning in Isabelle/HOL
- adequate translation for most purposes \rightsquigarrow I4.verified

Alternative Approach

- logic **HOLCF** is well suited to formalise partiality and non-strictness
- even constructor classes can be formalised
- reasoning in Isabelle/HOLCF is more complicated

Coping with Large Data Types

Dealing with **syntax trees** \Rightarrow dealing with **large data types**.

Data Types Defining Haskell Syntax Trees

- 500 lines of Haskell code
- 51 data types
- “largest” data type contains 45 constructors

Coping with Large Data Types

Dealing with **syntax trees** \Rightarrow dealing with **large data types**.

Data Types Defining Haskell Syntax Trees

- 500 lines of Haskell code
 - 51 data types
 - “largest” data type contains 45 constructors
-
- You **don't want to write all the code** for all those data types and each of their constructors!
 - If you have to write it you only want to **write it once!**

Coping with Large Data Types

Dealing with **syntax trees** \Rightarrow dealing with **large data types**.

Data Types Defining Haskell Syntax Trees

- 500 lines of Haskell code
 - 51 data types
 - “largest” data type contains 45 constructors
-
- You **don't want to write all the code** for all those data types and each of their constructors!
 - \Rightarrow **Generic Programming + Code Generation**
 - If you have to write it you only want to **write it once!**
 - \Rightarrow **Modularity**

Generic Programming

“Scrap Your Boilerplate”

Problem Addressed by SYB

- traverse a data structure to transform or query it
- only a few parts of the data structure are relevant

Generic Programming

“Scrap Your Boilerplate”

Problem Addressed by SYB

- traverse a data structure to transform or query it
- only a few parts of the data structure are relevant

Example

- compute free variables of an expression
- transform `where` clauses into `let` expressions

Generic Programming

“Scrap Your Boilerplate”

Problem Addressed by SYB

- traverse a data structure to transform or query it
- only a few parts of the data structure are relevant

Example

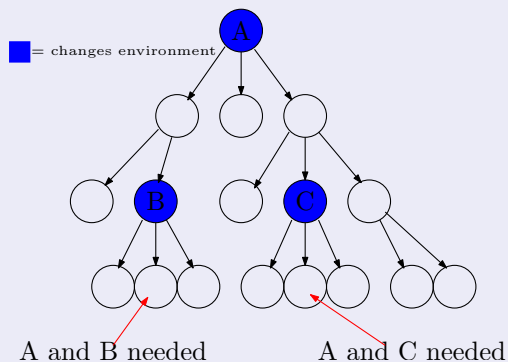
- compute free variables of an expression
- transform `where` clauses into `let` expressions

Difficulties when Applying SYB in our Setting

- often **context information** is necessary
- We want to define a piece of context information **only once**.

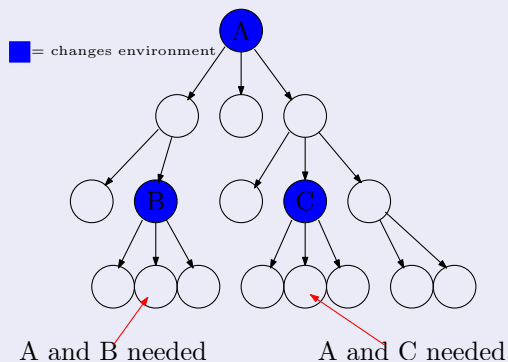
Environments

Data Structure as a Tree



Environments

Data Structure as a Tree



Defining Environments by $a \rightarrow (e \rightarrow e)$

- a is the type of the current node
- e is the type of the environment

Extending SYB by Environment Propagation

Extension to SYB

- allows to **define environments**
- allows to **combine environments**
- provides **traversal strategies** with environment propagation

Extending SYB by Environment Propagation

Extension to SYB

- allows to **define environments**
- allows to **combine environments**
- provides **traversal strategies** with environment propagation

Generalisation of Environment Propagation

- **non-uniform** propagation
- **monadic computations** to define an environment