Haskabelle

Converting Haskell98 to Isar/HOL

Tobias-Christian Rittweiler

Technische-Universität München

2008-10-05

Outline

Outline

What's Haskabelle about?

Isar/HOL

- FP + logic
- typed λ calculus
- `fun`, `datatype`
- syntactic sugar: `case`, etc.

Haskell

- "practical" FP
- pure, lazy
- monads
- gaining momentum

What's Haskabelle about?

Isar/HOL

- FP + logic
- typed λ calculus
- `fun`, `datatype`
- syntactic sugar: `case`, etc.

Haskell

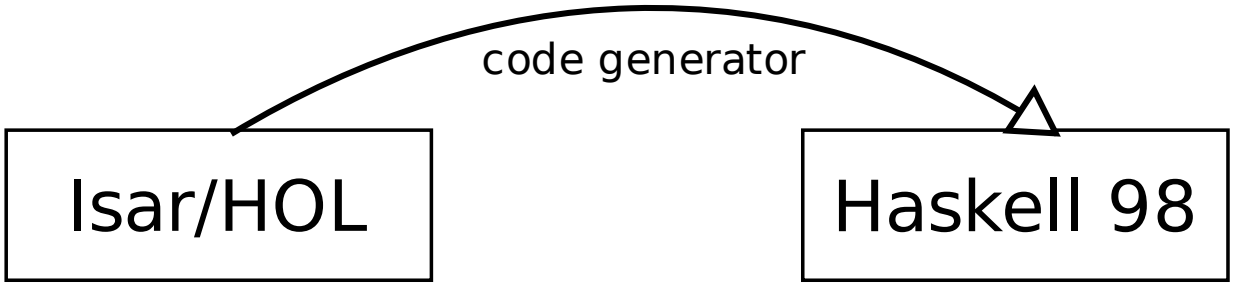
- "practical" FP
- pure, lazy
- monads
- gaining momentum

What's Haskabelle about?

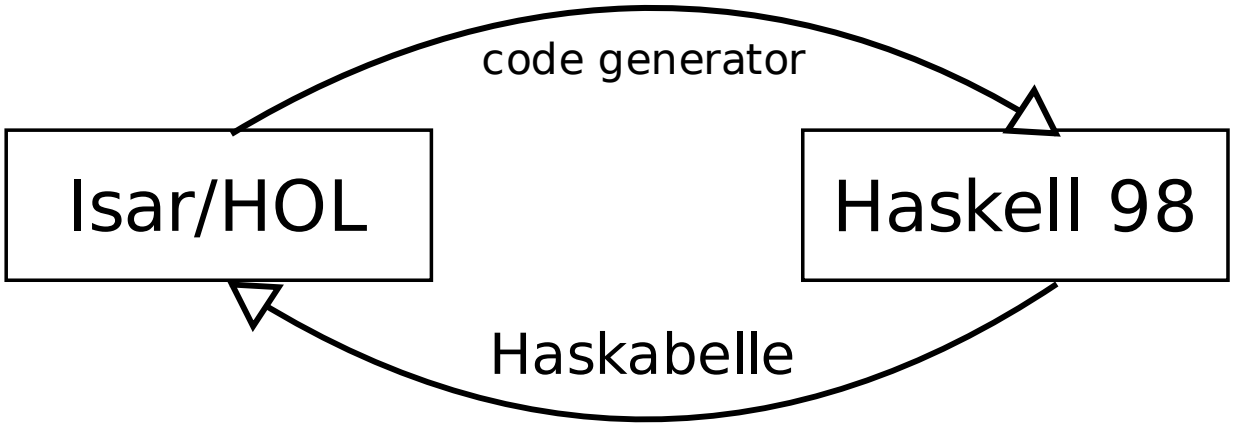
Isar/HOL

Haskell 98

What's Haskabelle about?

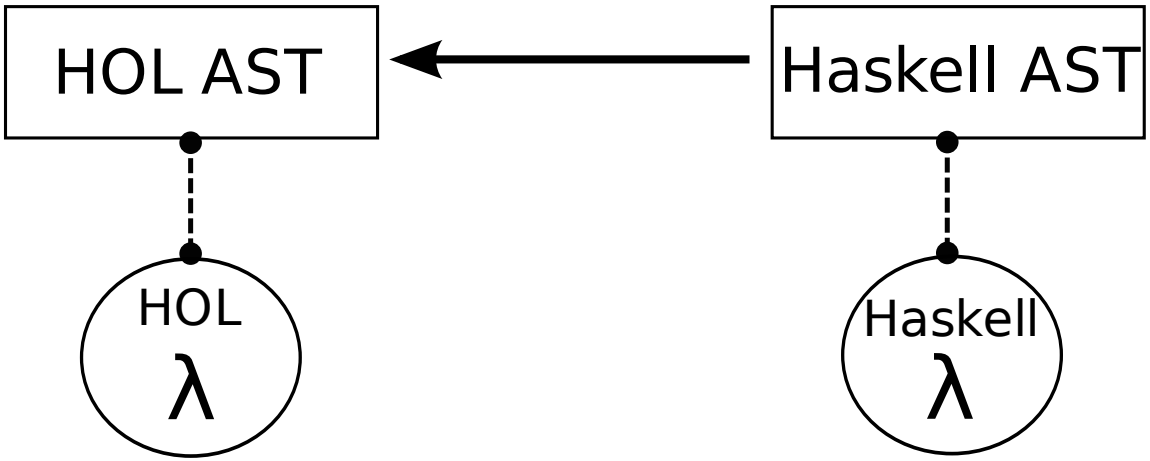


What's Haskabelle about?



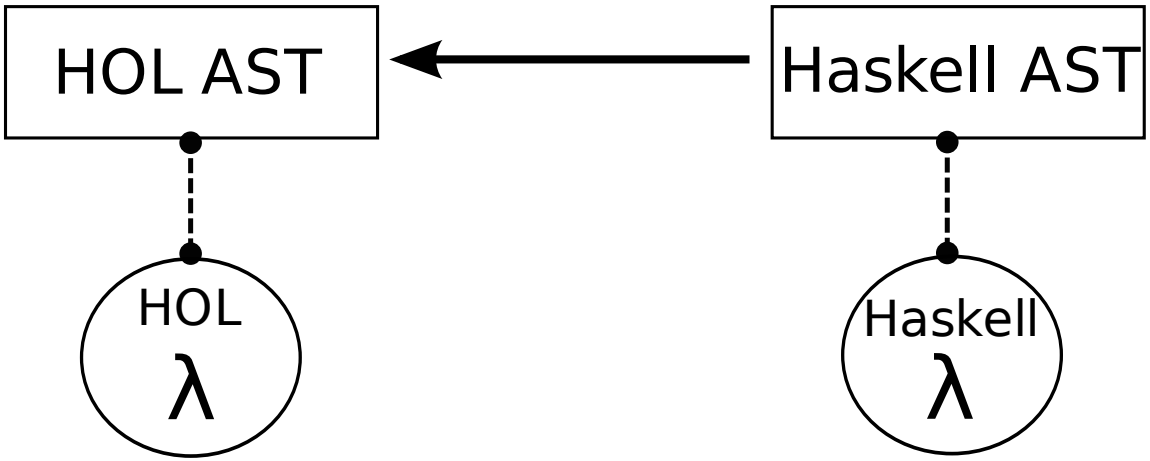
Outline

What does Haskabelle do?



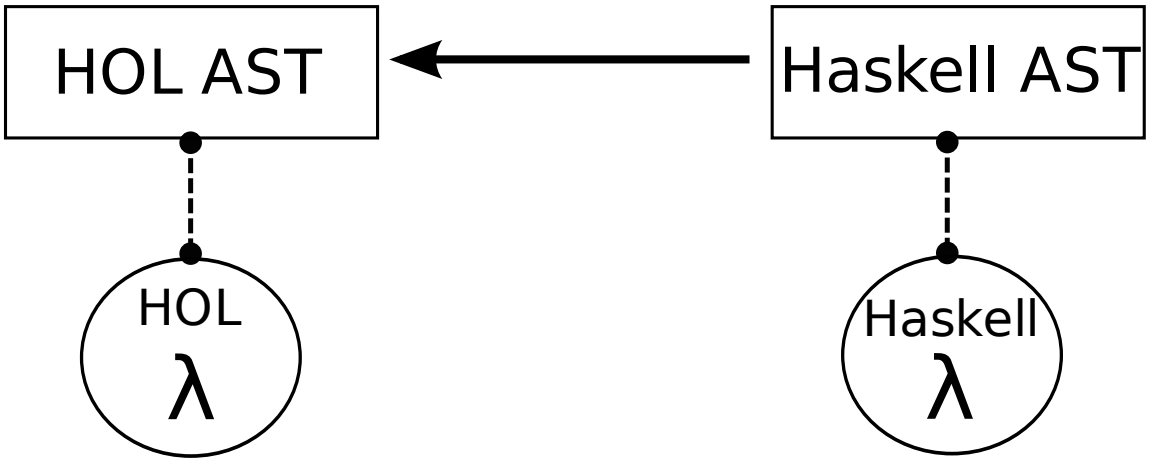
- "dumb tool", works on Abstract Syntax Trees only.
- e.g. no type inference
- we delegate the hard work to Isabelle
- Conclusion: Only because the conversion succeeded, does not mean that Isabelle won't choke...

What does Haskabelle do?



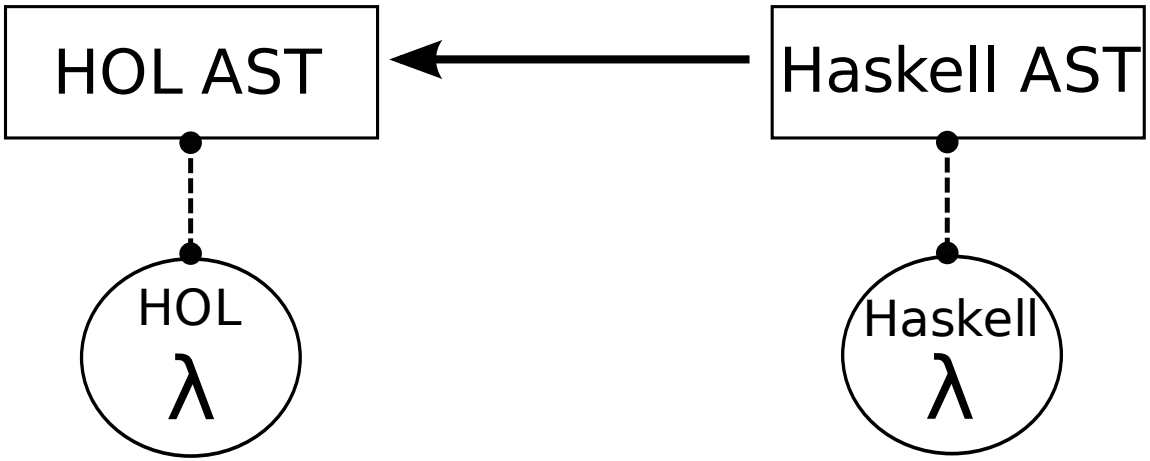
- "dumb tool", works on Abstract Syntax Trees only.
- e.g. no type inference
- we delegate the hard work to Isabelle
- Conclusion: Only because the conversion succeeded, does not mean that Isabelle won't choke...

What does Haskabelle do?



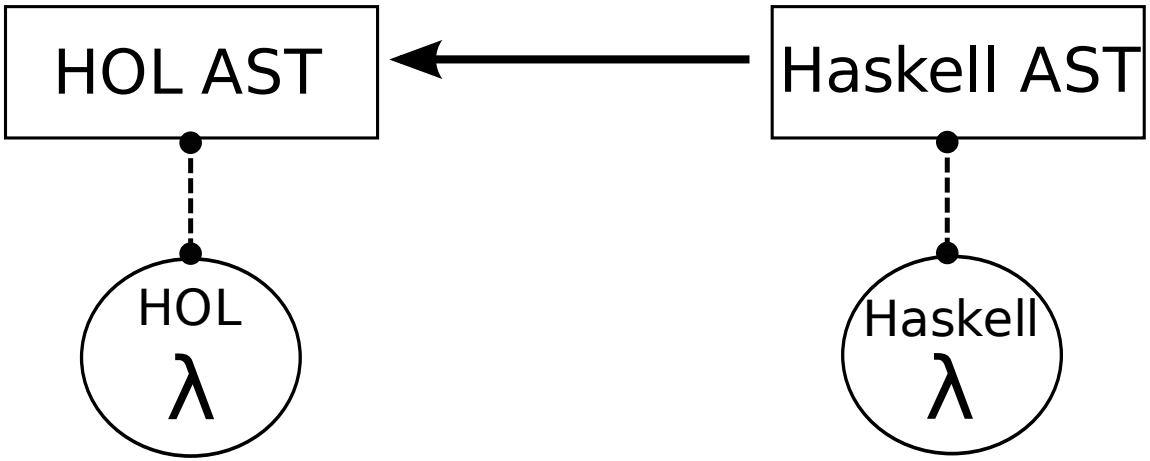
- "dumb tool", works on Abstract Syntax Trees only.
- e.g. no type inference
- we delegate the hard work to Isabelle
- Conclusion: Only because the conversion succeeded, does not mean that Isabelle won't choke...

What does Haskabelle do?



- "dumb tool", works on Abstract Syntax Trees only.
- e.g. no type inference
- we delegate the hard work to Isabelle
- Conclusion: Only because the conversion succeeded, does not mean that Isabelle won't choke...

What does Haskabelle do?



- "dumb tool", works on Abstract Syntax Trees only.
- e.g. no type inference
- we delegate the hard work to Isabelle
- Conclusion: Only because the conversion succeeded, does not mean that Isabelle won't choke...

Outline

An Overview (1)

- Module Resolution
- Declarations:
  - functions (`fun`)
  - constants (`definition`)
  - algebraic data types (`datatype`)
  - classes & instances (`class`, `instantiation`)
- Linearization of declarations

An Overview (2)

- Expressions:
  - literals (integers, strings, characters)
  - applications, incl. infix applications and sections
  - lambda abstractions
  - if, let, case
  - pattern guards
  - list comprehensions

Outline

Toplevel Function Declarations

.hs

```
revappend []     ys = ys
revappend (x:xs) ys = revappend xs (x:ys)
```

.thy

```
fun revappend
where
  "revappend Nil ys = ys"
| "revappend (x # xs) ys = revappend xs (x # ys)"
```

Toplevel Function Declarations

.hs

```
revappend []     ys = ys
revappend (x:xs) ys = revappend xs (x:ys)
```

.thy

```
fun revappend
where
  "revappend Nil ys = ys"
| "revappend (x # xs) ys = revappend xs (x # ys)"
```

Local Function Declarations (1)

.hs

```
square_list xs = map square xs
  where square x = x * x
```

.thy

```
fun square0
where
  "square0 x = (x * x)"

fun square_list
where
  "square_list xs = map square0 xs"
```

Local Function Declarations (1)

.hs

```
square_list xs = map square xs
  where square x = x * x
```

.thy

```
fun square0
where
  "square0 x = (x * x)"

fun square_list
where
  "square_list xs = map square0 xs"
```

Local Function Declarations (2)

- Free variables in local function declarations can't be converted..

.hs

```
scale_list n xs = map scale xs
  where scale x = n * x    -- Error: closes over n.
```

- If applicable, use lambda expressions instead..

.hs

```
scale_list n xs = map (\x -> n * x) xs
```

.thy

```
fun scale_list
where
  "scale_list n xs = map (% x . n * x) xs"
```

Local Function Declarations (2)

- Free variables in local function declarations can't be converted..

.hs

```
scale_list n xs = map scale xs
  where scale x = n * x    -- Error: closes over n.
```

- If applicable, use lambda expressions instead..

.hs

```
scale_list n xs = map (\x -> n * x) xs
```

.thy

```
fun scale_list
where
  "scale_list n xs = map (% x . n * x) xs"
```

Local Function Declarations (2)

- Free variables in local function declarations can't be converted..

.hs

```
scale_list n xs = map scale xs
  where scale x = n * x    -- Error: closes over n.
```

- If applicable, use lambda expressions instead..

.hs

```
scale_list n xs = map (\x -> n * x) xs
```

.thy

```
fun scale_list
where
  "scale_list n xs = map (% x . n * x) xs"
```

Algebraic Data Types

.hs

```
data Nat = Zero | Suc Nat

add_nat Zero     y = y
add_nat (Suc x) y = Suc (add_nat x y)
```

.thy

```
datatype Nat = Zero
             | Suc Nat

fun add_nat
where
  "add_nat Zero y = y"
| "add_nat (Suc x) y = Suc (add_nat x y)"
```

Algebraic Data Types

.hs

```
data Nat = Zero | Suc Nat

add_nat Zero     y = y
add_nat (Suc x) y = Suc (add_nat x y)
```

.thy

```
datatype Nat = Zero
             | Suc Nat

fun add_nat
where
  "add_nat Zero y = y"
| "add_nat (Suc x) y = Suc (add_nat x y)"
```

Type Classes (1)
Class declarations

.hs

```
class Monoid a where
  nothing :: a
  plus    :: a -> a -> a
```

.thy

```
class Monoid = type +
  fixes nothing :: 'a
  fixes plus :: "'a => 'a => 'a"
```

Type Classes (1)
Class declarations

.hs

```
class Monoid a where
  nothing :: a
  plus    :: a -> a -> a
```

.thy

```
class Monoid = type +
  fixes nothing :: 'a
  fixes plus :: "'a => 'a => 'a"
```

Type Classes (2)
Instance declarations

.hs

```
instance Monoid Nat where
  nothing = Zero
  plus    = add_nat
```

.thy

```
instantiation Nat :: Monoid
begin
  definition nothing_Nat :: "Nat" where
    "nothing_Nat = Zero"
  definition plus_Nat :: "Nat => Nat => Nat" where
    "plus_Nat = add_nat"
instance ..
```

Type Classes (2)
Instance declarations

.hs

```
instance Monoid Nat where
  nothing = Zero
  plus    = add_nat
```

.thy

```
instantiation Nat :: Monoid
begin
  definition nothing_Nat :: "Nat" where
    "nothing_Nat = Zero"
  definition plus_Nat :: "Nat => Nat => Nat" where
    "plus_Nat = add_nat"
instance ..
```

Type Classes (3)
Class context annotations

.hs

```
summ :: (Monoid a) => [a] -> a
summ []     = nothing
summ (x:xs) = plus x (summ xs)
```

.thy

```
fun summ :: "('a :: Monoid) list => ('a :: Monoid)"
where
  "summ Nil = nothing"
| "summ (x # xs) = plus x (summ xs)"
```

Type Classes (3)
Class context annotations

.hs

```
summ :: (Monoid a) => [a] -> a
summ []     = nothing
summ (x:xs) = plus x (summ xs)
```

.thy

```
fun summ :: "('a :: Monoid) list => ('a :: Monoid)"
where
  "summ Nil = nothing"
| "summ (x # xs) = plus x (summ xs)"
```

Outline

Work In Progress

- Infix Declarations
- Records
- Monads, and "do"
- Mapping between Haskell identifiers and their Isabelle pendants
- Documentation

Outline

Overview over the Implementation

5 Phases:

- Parsing
- Preprocessing
- Converting
- Adapting
- Printing

Overview over the Implementation

5 Phases:

- Parsing
- Preprocessing
- Converting
- Adapting
- Printing

Overview over the Implementation

5 Phases:

- Parsing
- Preprocessing
- Converting
- Adapting
- Printing

Overview over the Implementation

5 Phases:

- Parsing
- Preprocessing
- Converting
- Adapting
- Printing

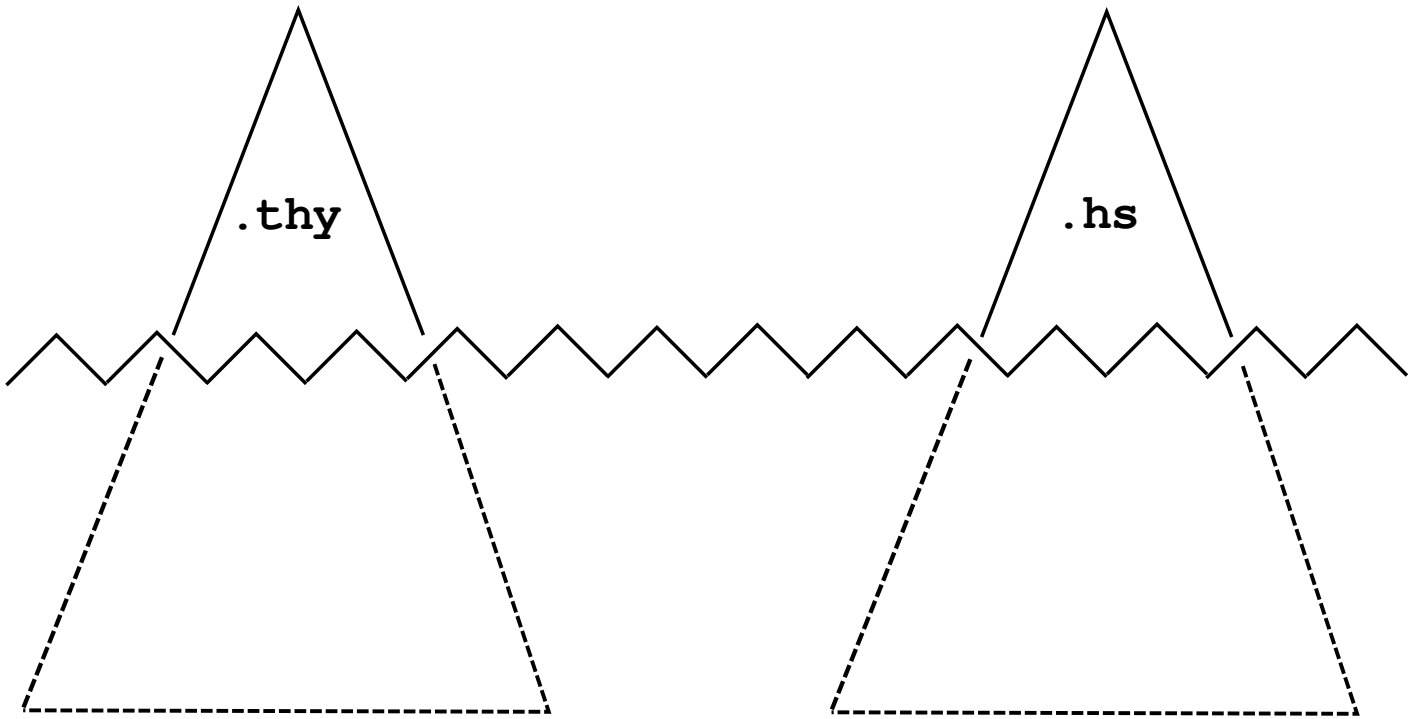
Overview over the Implementation

5 Phases:

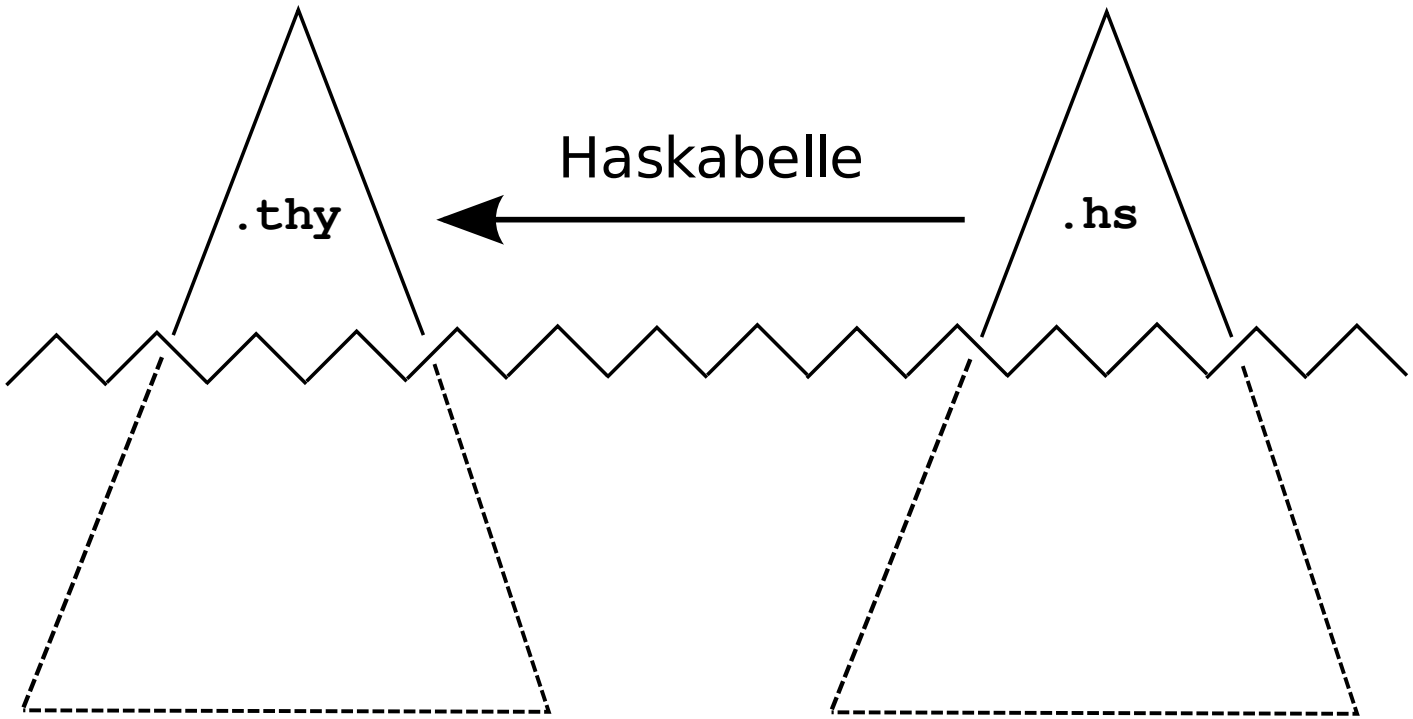
- Parsing
- Preprocessing
- Converting
- Adapting
- Printing

Outline

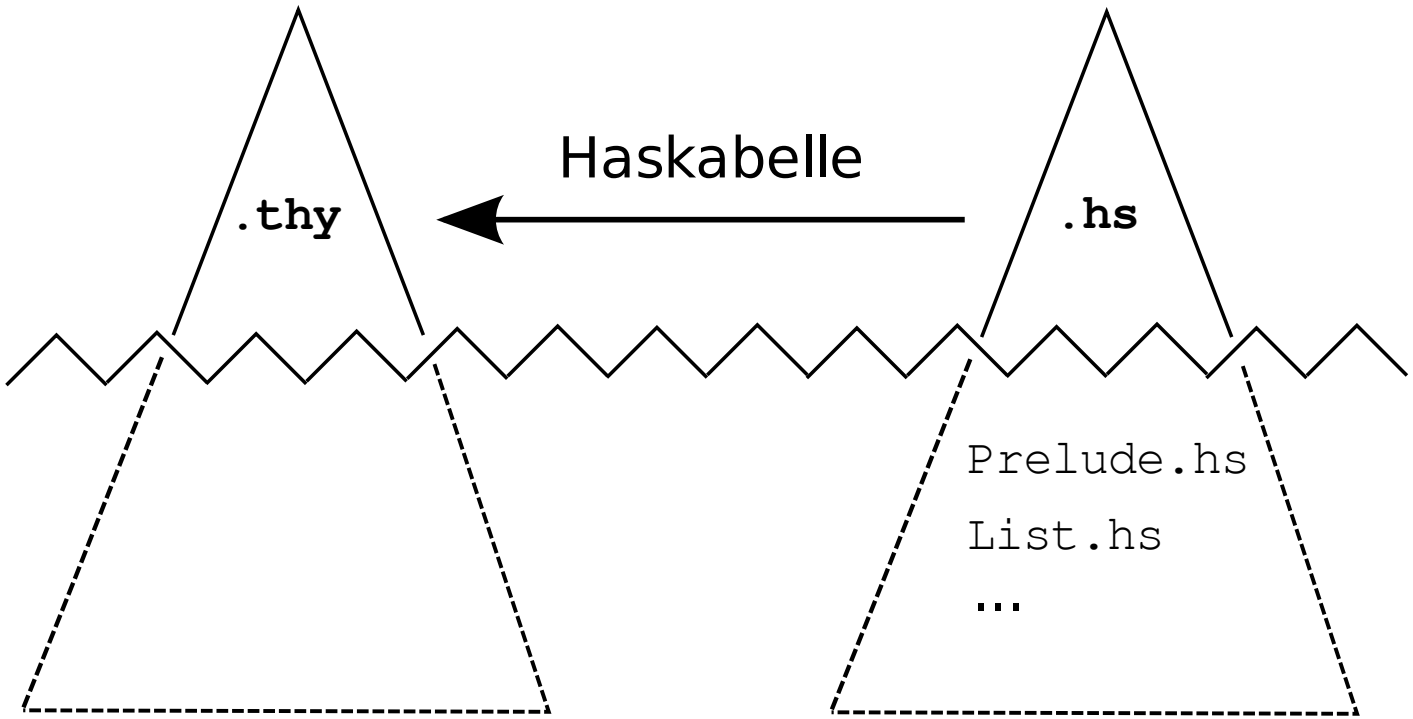
Adapting Haskell to Isabelle



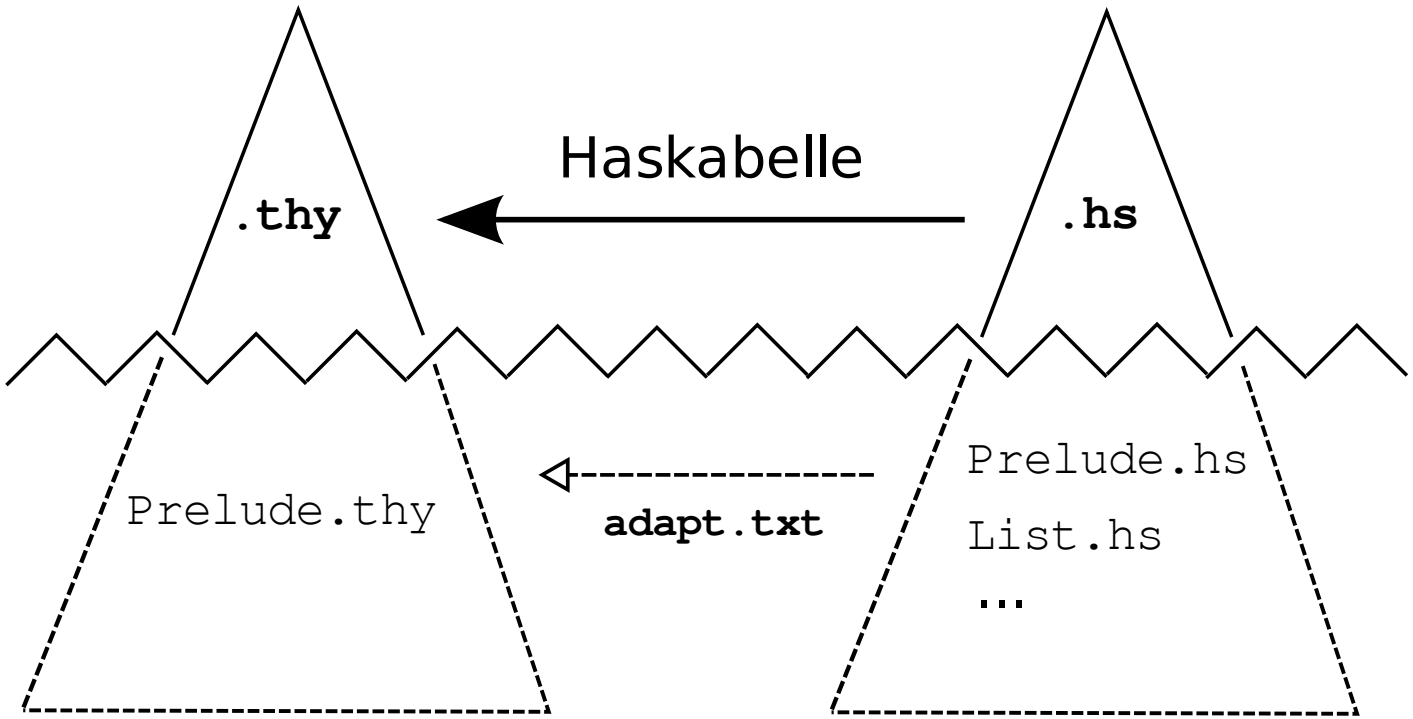
Adapting Haskell to Isabelle



Adapting Haskell to Isabelle



Adapting Haskell to Isabelle



Outline

lookup

.hs

```
lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key []     =  Nothing
lookup key ((x,y):xys)
    | key == x    =  Just y
    | otherwise   =  lookup key xys
```

.thy

```
fun lookup :: "('a :: heq) => ('a * 'b) list
                => 'b option"
where
  "lookup key Nil = None"
| "lookup key ((x, y) # xys)
    = (if heq key x then Some y
       else lookup key xys)"
```

lookup

.hs

```
lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key []     =  Nothing
lookup key ((x,y):xys)
    | key == x    =  Just y
    | otherwise   =  lookup key xys
```

.thy

```
fun lookup :: "('a :: heq) => ('a * 'b) list
                => 'b option"
where
  "lookup key Nil = None"
| "lookup key ((x, y) # xys)
     = (if heq key x then Some y
        else lookup key xys)"
```

Outline

Haskell is cool...

- ...but debugging can be a real PITA.