Types in Functional Programming Languages

Andres Löh

Department of Information and Computing Sciences
Utrecht University

UU General Math Colloquium – May 8, 2008

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Overview

- What are types?
  - Advantages
  - Disadvantages
- What are functional programming languages?
  - Haskell
  - algebraic data types
  - parametric polymorphism
- Dependent types
  - Formulas vs. programs
  - Inductive families

Types

A **type** is a property of a program.

“The program yields an integer.”

“The program takes a list of natural numbers and produces another list that is a permutation of the original list.”

Type systems

Decidable type checking

There is an algorithm that, given a type t and a program P, can determine whether P has type t (written P :: t).

Type systems

Decidable type checking

There is an algorithm that, given a type t and a program P, can determine whether P has type t (written P :: t).

Static type checking

The algorithm for type checking does not require to produce any effects that the program might have, and is thus independent of a particular program run.

Type systems

Decidable type checking

There is an algorithm that, given a type t and a program P, can determine whether P has type t (written P :: t).

Static type checking

The algorithm for type checking does not require to produce any effects that the program might have, and is thus independent of a particular program run.

Type system

A language of static types for a specific programming language, together with an algorithm that allows decidable type checking.

Type system advantages

- Catch errors at compilation time
- Documentation / specification
- Optimisation

Typical situations prevented by static types

- Applying a function to the wrong number of arguments
- Permuting the arguments of a function
- Forgetting the application of a (conversion) function

Type system disadvantages

Classic type systems of mainstream languages (Pascal, C, Java, Fortran, Cobol) have a number of problems:

- Limited reuse of code
- Inflexibility
- Verbosity
- Types don't track effects
- Types reject correct programs (or accept bad programs)

Limited reuse

Select the last element of an array:

```
int last (int[] a) {
   return a[a.length − 1];
}
char last (char[] a) {
   return a[a.length − 1];
}
bool last (bool[] a) {
   return a[a.length − 1];
}
```

The same code has to be repeated for different types.

Inflexibility

Classic type systems are inspired by hardware:

- some numeric types
- arrays/strings (one-dimensional, i.e., contiguous blocks of memory)
- enumeration types (i.e., bit fields)
- records (sequence of values that is stored in a connected block of memory)
- first-order functions (blocks of code we can jump to)

Inflexibility (contd.)

But not all things map nicely to hardware:

- something that is either an integer or an exceptional value
- a higher-order function
- a tree
- a date vs. three natural numbers
- a credit card number vs. a phone number
- lists of functions from records to TEX documents

Verbosity

Often, a large part of the syntax is taken up by the administration of types:

```
int sum (int[] a) {
    int i;
    int temp = 0;
    for (i = 0; i < a.length; i++)
        temp = temp + a[i];
    return temp;
}
```

Verbosity

Often, a large part of the syntax is taken up by the administration of types:

```
int sum (int[] a) {
   int i;
   int temp = 0;
   for (i = 0; i < a.length; i++)
      temp = temp + a[i];
   return temp;
}
```

- All the types are “obvious”, but we have to write them down.
- The same code would work for other numeric types, but again we would have to repeat the definition.

Effects

```
int add0 (int x, int y) {
    return x + y;
}
```

Effects

```
int add0 (int x, int y) {
   return x + y;
}

int add1 (int x, int y) {
   launch_missiles (now);
   return x + y;
}
```

Effects

```
int add0 (int x, int y) {
    return x + y;
}

int add1 (int x, int y) {
    launch_missiles (now);
    return x + y;
}
```

Both functions have the same type!

Types are imprecise

There are many situations where the type system cannot see that a program that looks wrong will not fail:

- pointer arithmetic
- placing elements of different types together in a vector
- C's printf function:

printf ("%d", 7);	prints a single number
printf ("%d\n%d", 7, 9);	prints two numbers on two lines
printf ("%s", "foo");	prints a single string

The unfortunate result

Limitations of specific languages are blamed on the concept of static types.

The unfortunate result

Limitations of specific languages are blamed on the concept of static types.

Many so-called **scripting languages** (Perl, Python, Ruby, ...) are specifically lauded by programmers for having no (static) types.

Are the problems solvable?

Not completely:

a decidable type system must necessarily reject sensible programs.

Are the problems solvable?

Not completely:

a decidable type system must necessarily reject sensible programs.

```
if (riemann_hypothesis_holds ()){
  print "Finally ...";
} else {
  return "42" / 0;
}
```

Checking if the else-branch is executed requires running the program. What if the program is interactive in some way?

However . . .

All is not lost. We can improve significantly in all areas. With a more powerful, lightweight, systematic type system we can

- allow more code reuse,
- reduce inflexibility,
- make most type annotations superfluous,
- track effects,
- be more precise.

Haskell

- Functional programming languages (most notably Haskell) have been ahead of imperative languages w.r.t. static type systems.
- Some features from functional programming languages have recently been incorporated into mainstream languages such as Java and C#.
- We will look at Haskell concepts that address the problems we described.

History of Haskell

Lisp	– McCarthy, 1960
SASL	– Turner, 1976
Scheme	– Sussman and Steele, 1978

History of Haskell

Lisp	– McCarthy, 1960
SASL	– Turner, 1976
Scheme	– Sussman and Steele, 1978
ML	– Milner, 1978
Miranda	– Turner, 1985

History of Haskell

Lisp	– McCarthy, 1960
SASL	– Turner, 1976
Scheme	– Sussman and Steele, 1978
ML	– Milner, 1978
Miranda	– Turner, 1985
Haskell	– Haskell Committee, 1990
Haskell 98	– Haskell Committee, 1999

History of Haskell

Lisp	– McCarthy, 1960
SASL	– Turner, 1976
Scheme	– Sussman and Steele, 1978
ML	– Milner, 1978
Miranda	– Turner, 1985
Haskell	– Haskell Committee, 1990
Haskell 98	– Haskell Committee, 1999
Haskell'	– Haskell Committee, 2009?

A few Haskell facts

- General purpose functional programming language
- Several implementations, all free and open-source software:
  - GHC (Glasgow University), now Microsoft Research – industrial strength compiler and interpreter, implemented in Haskell itself
  - Hugs, interpreter, written in C
  - EHC, in development at Utrecht
- Used for research and actual applications
- Can interface to other programming languages
- Lots of libraries available
- `http://haskell.org`

Haskell type system concepts

- Parametric polymorphism allows code reuse
- Algebraic datatypes, a systematic type language and higher-order functions get rid of most of the inflexibilities
- Type inference reduces verbosity
- All Haskell functions are pure – effects are tracked in the type sytstem

Parametric polymorphism

$\mathsf{last} :: \forall a.[a] \to a$
$\mathsf{last}\ xs = xs\ !!\ (\mathsf{length}\ xs - 1)$

Parametric polymorphism

$\mathsf{last} :: \forall a.[a] \to a$
$\mathsf{last\ xs} = \mathsf{xs}\ !!\ (\mathsf{length\ xs} - 1)$

$\mathsf{swap} :: \forall a.\forall b.(a, b) \to (b, a)$
$\mathsf{swap}\ (x, y) = (y, x)$

Parametric polymorphism

$\mathsf{last} :: \forall a.[a] \rightarrow a$
$\mathsf{last}\ xs = xs\ !!\ (\mathsf{length}\ xs - 1)$

$\mathsf{swap} :: \forall a.\forall b.(a, b) \rightarrow (b, a)$
$\mathsf{swap}\ (x, y) = (y, x)$

$\mathsf{id} :: \forall a.a \rightarrow a$
$\mathsf{id}\ x = x$

Algebraic data types

Next to the built-in numeric types, we can define our own datatype of Peano-style natural numbers as follows:

```
data ℕ where
   Zero :: ℕ
   Succ :: ℕ → ℕ
```

Algebraic data types

Next to the built-in numeric types, we can define our own datatype of Peano-style natural numbers as follows:

```
data ℕ where
  Zero :: ℕ
  Succ :: ℕ → ℕ
```

$$\frac{}{\mathsf{Zero} :: \mathbb{N}} \qquad \frac{\mathsf{n} :: \mathbb{N}}{\mathsf{Succ}\ \mathsf{n} :: \mathbb{N}}$$

Algebraic data types

Next to the built-in numeric types, we can define our own datatype of Peano-style natural numbers as follows:

```
data ℕ where
  Zero :: ℕ
  Succ :: ℕ → ℕ
```

$$\frac{}{\mathsf{Zero} :: \mathbb{N}} \qquad \frac{\mathsf{n} :: \mathbb{N}}{\mathsf{Succ\ n} :: \mathbb{N}}$$

```
plus :: ℕ → ℕ → ℕ
plus m Zero     = m
plus m (Succ n) = Succ (plus m n)
```

Higher-order functions

$$
\begin{array}{l}
\mathsf{foldNat} :: \forall r.r \to (r \to r) \to \mathbb{N} \to r \\
\mathsf{foldNat}\ z\ s\ \mathsf{Zero} \qquad\quad = z \\
\mathsf{foldNat}\ z\ s\ (\mathsf{Succ}\ n) = s\ (\mathsf{foldNat}\ z\ s\ n) \\
\mathsf{plus} :: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\mathsf{plus}\ m = \mathsf{foldNat}\ m\ \mathsf{Succ}
\end{array}
$$

Higher-order functions

```
foldNat :: ∀r.r → (r → r) → ℕ → r
foldNat z s Zero      = z
foldNat z s (Succ n) = s (foldNat z s n)

plus :: ℕ → ℕ → ℕ
plus m = foldNat m Succ
```

```
mult :: ℕ → ℕ → ℕ
mult m = foldNat Zero (plus m)

exp :: ℕ → ℕ → ℕ
exp m = foldNat (Succ Zero) (mult m)
```

Lists

Lists are built-in because they have special syntax, but essentially are defined via:

```
data [] :: * → * where
   []  :: ∀a.[a]
   (:) :: ∀a.a → [a] → [a]
```

Lists

Lists are built-in because they have special syntax, but essentially are defined via:

$$
\begin{array}{l}
\textbf{data}\ [\,]\ ::\ * \to * \ \textbf{where} \\
\quad [\,]\ \ ::\ \forall a.[a] \\
\quad (:)\ ::\ \forall a.a \to [a] \to [a]
\end{array}
$$

$$(\to)\ ::\ * \to * \to *$$

Lists

Lists are built-in because they have special syntax, but essentially are defined via:

$$\begin{array}{l}\textbf{data}\ [\,]::*\to *\ \textbf{where}\\ \quad [\,]\ ::\forall a.[a]\\ \quad (:)::\forall a.a\to[a]\to[a]\end{array}$$

$$(\to)::*\to *\to *$$

$$\begin{array}{rl} & 1:(2:(3:(4:5:[\,])))\\ \equiv & 1:2:3:4:5:[\,]\\ \equiv & [1,2,3,4,5]\\ \equiv & [1\,..\,5]\end{array}$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Lists (contd.)

$$
\begin{aligned}
&\mathsf{map} :: \forall a\ b.(a \to b) \to [a] \to [b] \\
&\mathsf{map}\ f\ [\,] \qquad\quad = [\,] \\
&\mathsf{map}\ f\ (x : xs) = f\ x : \mathsf{map}\ f\ xs
\end{aligned}
$$

Lists (contd.)

$$
\begin{aligned}
&\mathsf{map} :: \forall a\ b.(a \to b) \to [a] \to [b]\\
&\mathsf{map}\ f\ [\,] \qquad\quad = [\,]\\
&\mathsf{map}\ f\ (x : xs) = f\ x : \mathsf{map}\ f\ xs
\end{aligned}
$$

$$\mathsf{map}\ f\ xs = [f\ x \mid x \leftarrow xs]$$

Lists (contd.)

```
map :: ∀a b.(a → b) → [a] → [b]
map f []       = []
map f (x : xs) = f x : map f xs
```

```
map f xs = [f x | x ← xs]
```

```
filter :: ∀a.(a → Bool) → [a] → [a]
filter p []       = []
filter p (x : xs)
    | p x       = x : filter p xs
    | otherwise =     filter p xs
```

Lists (contd.)

$$
\begin{aligned}
&\mathsf{map} :: \forall a\ b.(a \to b) \to [a] \to [b]\\
&\mathsf{map}\ f\ [\,] \qquad\quad = [\,]\\
&\mathsf{map}\ f\ (x : xs) = f\ x : \mathsf{map}\ f\ xs
\end{aligned}
$$

$$\mathsf{map}\ f\ xs = [f\ x \mid x \leftarrow xs]$$

$$
\begin{aligned}
&\mathsf{filter} :: \forall a.(a \to \mathsf{Bool}) \to [a] \to [a]\\
&\mathsf{filter}\ p\ [\,] \qquad\quad = [\,]\\
&\mathsf{filter}\ p\ (x : xs)\\
&\qquad \mid p\ x \qquad\quad = x : \mathsf{filter}\ p\ xs\\
&\qquad \mid \mathsf{otherwise} = \quad\ \ \mathsf{filter}\ p\ xs
\end{aligned}
$$

$$\mathsf{filter}\ p\ xs = [x \mid x \leftarrow xs, p\ x]$$

More datatypes

```
data Maybe :: * → * where
  Nothing :: ∀a.Maybe a
  Just    :: ∀a.a → Maybe a
```

More datatypes

```
data Maybe :: * → * where
   Nothing :: ∀a.Maybe a
   Just    :: ∀a.a → Maybe a
```

```
last :: ∀a.[a] → Maybe a
last [] = Nothing
last xs = Just (xs !! (length xs − 1))
```

More datatypes (contd.)

```
data Expr where
   Num :: ℤ → Expr
   Add  :: Expr → Expr → Expr
   Mul  :: Expr → Expr → Expr
```

More datatypes (contd.)

```
data Expr where
  Num :: ℤ → Expr
  Add  :: Expr → Expr → Expr
  Mul  :: Expr → Expr → Expr
```

```
eval :: Expr → ℤ
eval (Num x)       = x
eval (Add e₁ e₂) = eval e₁ + eval e₂
eval (Mul e₁ e₂) = eval e₁ ∗ eval e₂
```

Type inference

All the function type signatures are optional! If omitted, they will be inferred.

```
swap (x, y) = (y, x)

mult m = foldNat Zero (plus m)

eval (Num x)      = x
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2

sum []       = 0
sum (x : xs) = x + sum xs
```

Function sum will get type $\forall a.\mathrm{Num}\ a \Rightarrow [a] \rightarrow a$, indicating that it works for all **numeric types**.

Type inference (contd.)

Type inference is particularly convenient for local functions:

```
sort [] = []
sort (x : xs) = insert x xs
  where
    insert x []       = [x]
    insert x (y : ys)
      | x ⩽ y        = x : y : ys
      | otherwise    = y : insert x ys
```

Type inference (contd.)

Type inference is particularly convenient for local functions:

```
sort []     = []
sort (x:xs) = insert x xs
  where
    insert x []     = [x]
    insert x (y:ys)
      | x ⩽ y      = x : y : ys
      | otherwise  = y : insert x ys
```

Inference determines:

sort :: $\forall a.\mathsf{Ord}\ a \Rightarrow [a] \to [a]$
insert :: $\forall a.\mathsf{Ord}\ a \Rightarrow a \to [a] \to [a]$

Hindley-Milner type system

- Haskell can be mapped to a (variant of the) lambda calculus
- For some lambda calculi, efficient type inference algorithms exist
- Damas-Hindley-Milner is one such system, on which also ML is based

Expressions

Programs/expressions:

$e ::= x$	variables
$\mid (e_1\ e_2)$	application
$\mid \lambda x \rightarrow e$	abstraction
$\mid$ **let** $x = e_1$ **in** e_2	local definitions (non-recursive)

Expressions

Programs/expressions:

$e ::= x$	variables
$\mid\ (e_1\ e_2)$	application
$\mid\ \lambda x \rightarrow e$	abstraction
$\mid\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2$	local definitions (non-recursive)

Reduction:

$$\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \leadsto e_2[x \mapsto e_1]$$
$$((\lambda x \rightarrow e_1)\ e_2) \leadsto e_1[x \mapsto e_2]$$

Hindley-Milner types

Types:

$$
\begin{array}{rcll}
t & ::= & C & \text{constants} \\
  & \mid & a & \text{variables} \\
  & \mid & t_1 \to t_2 & \text{function types}
\end{array}
$$

Type schemes:

$$
\begin{array}{rcl}
\sigma & ::= & \forall a.\sigma \\
       & \mid & t
\end{array}
$$

All quantification happens on the outside.

Type system

Typing judgements are of the form:

$$\Gamma \vdash \mathsf{e} :: \sigma$$

I.e., typing is a 3-place relation between an environment Γ mapping variables to type schemes, an expression e and a type scheme σ.

Type rules

$$\frac{x :: \sigma \in \Gamma}{\Gamma \vdash x :: \sigma} \qquad \frac{\begin{array}{l}\Gamma \vdash e_1 :: t_1 \to t_2 \\ \Gamma \vdash e_2 :: t_1\end{array}}{\Gamma \vdash (e_1\ e_2) :: t_2}$$

$$\frac{\Gamma, x :: t_1 \vdash e :: t_2}{\Gamma \vdash \lambda x \to e :: t_1 \to t_2} \qquad \frac{\begin{array}{l}\Gamma \vdash e_1 :: \sigma_1 \\ \Gamma, x :: \sigma_1 \vdash e_2 :: \sigma_2\end{array}}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 :: \sigma_2}$$

$$\frac{\begin{array}{l}a \text{ does not appear free in } \Gamma \\ \Gamma \vdash e :: \sigma\end{array}}{\Gamma \vdash e :: \forall a.\sigma} \qquad \frac{\Gamma \vdash e :: \forall a.\sigma}{\Gamma \vdash e :: \sigma[a \mapsto t]}$$

Linking types and reduction

Theorem (Preservation)

If $\Gamma \vdash \mathsf{e}_1 :: \sigma$ *and* $\mathsf{e}_1 \rightsquigarrow \mathsf{e}_2$, *then* $\Gamma \vdash \mathsf{e}_2 :: \sigma$.

Linking types and reduction

Theorem (Preservation)

If $\Gamma \vdash \mathsf{e}_1 :: \sigma$ *and* $\mathsf{e}_1 \rightsquigarrow \mathsf{e}_2$*, then* $\Gamma \vdash \mathsf{e}_2 :: \sigma$*.*

Theorem (Progress)

If $\varepsilon \vdash \mathsf{e} :: \sigma$*, then* e *is a lambda abstraction or it can be reduced.*

Progress theorems state that the reduction for well-typed terms does not get stuck in unexpected situations, and is more interesting for slightly richer lambda calculi.

Linking types and reduction

Theorem (Preservation)

If $\Gamma \vdash \mathsf{e}_1 :: \sigma$ *and* $\mathsf{e}_1 \rightsquigarrow \mathsf{e}_2$, *then* $\Gamma \vdash \mathsf{e}_2 :: \sigma$.

Theorem (Progress)

If $\varepsilon \vdash \mathsf{e} :: \sigma$, *then* e *is a lambda abstraction or it can be reduced.*

Progress theorems state that the reduction for well-typed terms does not get stuck in unexpected situations, and is more interesting for slightly richer lambda calculi.

Such theorems are proved by mostly boring induction on the structure of expressions or the structure of the typing derivations.

Type inference algorithm

There is an efficient algorithm that can compute a type scheme for an expression and an environment: $\Gamma \vdash \mathsf{e} ::\uparrow \sigma$.

Theorem (Soundness)

If $\Gamma \vdash \mathsf{e} ::\uparrow \sigma$, *then* $\Gamma \vdash \mathsf{e} :: \sigma$.

Type inference algorithm

There is an efficient algorithm that can compute a type scheme for an expression and an environment: $\Gamma \vdash \mathsf{e} ::\uparrow \sigma$.

Theorem (Soundness)

If $\Gamma \vdash \mathsf{e} ::\uparrow \sigma$, *then* $\Gamma \vdash \mathsf{e} :: \sigma$.

Theorem (Completeness)

If $\Gamma \vdash \mathsf{e} :: \sigma_1$, *then* $\Gamma \vdash \mathsf{e} ::\uparrow \sigma_2$ *such that* σ_1 *is an instance of* σ_2.

Every well-typed expression has a **principal type**.

Type inference algorithm

There is an efficient algorithm that can compute a type scheme for an expression and an environment: $\Gamma \vdash \mathsf{e} ::\uparrow \sigma$.

Theorem (Soundness)

If $\Gamma \vdash \mathsf{e} ::\uparrow \sigma$, *then* $\Gamma \vdash \mathsf{e} :: \sigma$.

Theorem (Completeness)

If $\Gamma \vdash \mathsf{e} :: \sigma_1$, *then* $\Gamma \vdash \mathsf{e} ::\uparrow \sigma_2$ *such that* σ_1 *is an instance of* σ_2.

Every well-typed expression has a **principal type**.

Again, the proofs are by induction on the derivations.

Effects

Effectful computations in Haskell are tagged by the type-system.

$$\begin{aligned}
&\mathrm{add0} :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}\\
&\mathrm{add0}\ x\ y = x + y\\
&\mathrm{add1} :: \mathbb{Z} \to \mathbb{Z} \to \mathrm{IO}\ \mathbb{Z}\\
&\mathrm{add1}\ x\ y = \mathrm{launch\_missiles} \gg \mathrm{return}\ (x + y)
\end{aligned}$$

Effects

Effectful computations in Haskell are tagged by the type-system.

$$\begin{aligned}
&\text{add0} :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}\\
&\text{add0 x y} = \text{x} + \text{y}\\
&\text{add1} :: \mathbb{Z} \to \mathbb{Z} \to \text{IO } \mathbb{Z}\\
&\text{add1 x y} = \text{launch\_missiles} \gg \text{return } (\text{x} + \text{y})
\end{aligned}$$

$$\begin{aligned}
&(\gg) \;\;\, :: \forall \text{a.IO a} \to \text{IO a} \to \text{IO a}\\
&(\gg\!=) :: \forall \text{a.IO a} \to (\text{a} \to \text{IO b}) \to \text{IO b}
\end{aligned}$$

Effects

Effectful computations in Haskell are tagged by the type-system.

$$\mathrm{add0} :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$$
$$\mathrm{add0}\ x\ y = x + y$$
$$\mathrm{add1} :: \mathbb{Z} \to \mathbb{Z} \to \mathrm{IO}\ \mathbb{Z}$$
$$\mathrm{add1}\ x\ y = \mathrm{launch\_missiles} \gg \mathrm{return}\ (x + y)$$

$$(\gg)\ \ :: \forall a.\mathrm{IO}\ a \to \mathrm{IO}\ a \to \mathrm{IO}\ a$$
$$(\gg\!=) :: \forall a.\mathrm{IO}\ a \to (a \to \mathrm{IO}\ b) \to \mathrm{IO}\ b$$

$$\begin{aligned} &\mathrm{greet} :: \mathrm{IO}\ ()\\ &\mathrm{greet} = \mathrm{putStr}\ \texttt{"Who are you? "} \gg\\ &\qquad\quad \mathrm{getLine} \gg\!= \lambda x \to\\ &\qquad\quad \mathrm{putStrLn}\ (\texttt{"Hello "} +\!\!+\ x)\end{aligned}$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Imprecision

While Haskell has even more type system features that allow it to type many programs, the type system is still imprecise:

$$\begin{array}{l}\text{last} :: \forall a.[a] \to a \\ \text{last xs} = \text{xs} \,!!\, (\text{length xs} - 1)\end{array}$$

Valid, but fails on empty lists.

Imprecision

While Haskell has even more type system features that allow it to type many programs, the type system is still imprecise:

$$\text{last} :: \forall a.[a] \to a$$
$$\text{last xs} = \text{xs} \mathbin{!!} (\text{length xs} - 1)$$

Valid, but fails on empty lists.

It is not easily possible in Haskell to describe types such as:

- lists of a specific number of elements (vectors)
- natural numbers between 3 and 7
- sorted lists
- . . .

Dependent types

Dependent types allow us to mix values and types:

> Vec :: $\forall a :: *. \forall n :: \mathbb{N}. *$

A vector is a type ($*$), parameterized by a type a and a natural number n.

Dependent types

Dependent types allow us to mix values and types:

$$\mathsf{Vec} :: \forall a :: *.\forall n :: \mathbb{N}.\ *$$

A vector is a type ($*$), parameterized by a type a and a natural number n.

$$\begin{aligned}&\mathsf{last} :: \forall a :: *.\forall n :: \mathbb{N}.\mathsf{Vec}\ a\ (\mathsf{Succ}\ n) \to a\\ &\mathsf{last}\ (x : [\,])\ = x\\ &\mathsf{last}\ (x : xs) = \mathsf{last}\ xs\end{aligned}$$

No case for the empty list is required. It would even be rejected by the type checker.

Agda

- Dependent types have a history in proof assistants, as a modelling language (Automath, NuPRL, ELF, Twelf, Coq).
- It is relatively new to consider dependent types for programming (for example: Cayenne 1998, Epigram 2004, Coq's Program extension 2007).
- Agda2 is an ongoing experimental rewrite of the Agda proof assistant specifically designed as a programming language. It has Haskell-inspired syntax.
- `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php` or google for Agda2

Program properties

With dependent types, you have a language on the type-level that allows you to describe arbitrary properties of your program.

$(\equiv) :: \forall a :: *. \forall x :: a. \forall y :: a. *$

Now you can write down properties such as

$\text{zero\_right\_neutral} :: \forall n :: \mathbb{N}. n \equiv n + 0$

$\text{reverse\_involutary} ::$
$\quad \forall a :: *. \forall n :: \mathbb{N}. \forall xs :: \text{Vec } a\ n. \text{reverse } (\text{reverse } xs) \equiv xs$

Program properties

With dependent types, you have a language on the type-level that allows you to describe arbitrary properties of your program.

$$(\equiv) :: \forall a :: *. \forall x :: a. \forall y :: a.\ *$$

Now you can write down properties such as

$$\text{zero\_right\_neutral} :: \forall n :: \mathbb{N}. n \equiv n + 0$$

$$\begin{aligned}&\text{reverse\_involutary} ::\\ &\quad \forall a :: *. \forall n :: \mathbb{N}. \forall xs :: \text{Vec}\ a\ n. \text{reverse}\ (\text{reverse}\ xs) \equiv xs\end{aligned}$$

Properties are types, programs are proofs:

$$\begin{aligned}\text{zero\_right\_neutral}\ 0 &= \text{refl}\\ \text{zero\_right\_neutral}\ (\text{Succ}\ n) &= \text{cong Succ}\ (\text{zero\_right\_neutral}\ n)\end{aligned}$$

Curry-Howard correspondence

property	type
proof	program

Curry-Howard correspondence

property	type
proof	program
truth	inhabited type
falsity	uninhabited type
conjunction	pair
disjunction	union type
implication	function
negation	function to the uninhabited type
universal quantification	dependent function
existential quantification	dependent pair

Inductive families

```
data Fin :: ∀n :: ℕ. ∗  where
   fz :: ∀n :: ℕ.          Fin (Succ n)
   fs :: ∀n :: ℕ.Fin n → Fin (Succ n)
```

Inductive families

```
data Fin :: ∀n :: ℕ. ∗  where
   fz :: ∀n :: ℕ.             Fin (Succ n)
   fs :: ∀n :: ℕ.Fin n → Fin (Succ n)
```

```
data (⩽) :: ∀m :: ℕ.∀n :: ℕ. ∗  where
   leq_refl  :: ∀n :: ℕ.n ⩽ n
   leq_step :: ∀m :: ℕ.∀n :: ℕ.m ⩽ n → m ⩽ Succ n

decide_leq :: ∀m :: ℕ.∀n :: ℕ.m ⩽ n ∨ n ⩽ m
```

Inductive families

```
data Fin :: ∀n :: ℕ. ∗  where
   fz :: ∀n :: ℕ.              Fin (Succ n)
   fs :: ∀n :: ℕ.Fin n → Fin (Succ n)
```

```
data (⩽) :: ∀m :: ℕ.∀n :: ℕ. ∗  where
   leq_refl  :: ∀n :: ℕ.n ⩽ n
   leq_step :: ∀m :: ℕ.∀n :: ℕ.m ⩽ n → m ⩽ Succ n

decide_leq :: ∀m :: ℕ.∀n :: ℕ.m ⩽ n ∨ n ⩽ m
```

Inductive families let you define arbitrary relations as types and prove additional properties about the relation by induction on the derivation.

Capturing induction

Compare

$$\begin{aligned}
&\mathsf{foldNat} :: \forall r.r \to (r \to r) \to \mathbb{N} \to r\\
&\mathsf{foldNat}\ z\ s\ \mathsf{Zero} \qquad\ \ = z\\
&\mathsf{foldNat}\ z\ s\ (\mathsf{Succ}\ n) = s\ (\mathsf{foldNat}\ z\ s\ n)
\end{aligned}$$

and

$$\begin{array}{lll}
\mathsf{indNat} :: & \forall P :: \mathbb{N} \to *. & \text{“motive”}\\
& P\ \mathsf{Zero} \to & \text{base case}\\
& (\forall n :: \mathbb{N}.P\ n \to P\ (\mathsf{Succ}\ n)) \to & \text{induction step}\\
& \forall n :: \mathbb{N}.P\ n &
\end{array}$$

with the same definition.

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Dependently-typed programming

Huge amount of possibilities:

- state and prove properties of your program within a common framework
- devise algorithms that are correct by construction (run your proofs)
- type very complicated functions (printf, or a function that takes an abstract description of a language and produces a parser for that language)
- . . .

What about decidability and nontermination?

How will the typechecker determine whether

> $\mathsf{Vec\ a\ n} \equiv \mathsf{Vec\ a\ (loop\ n)}$

Running loop might not terminate ...

What about decidability and nontermination?

How will the typechecker determine whether

> Vec a n ≡ Vec a (loop n)

Running loop might not terminate . . .

- Allow only total terminating programs
- Flag possibly non-terminating programs in the type system (like IO in Haskell)

Research questions

- Efficient compilation of dependently-typed programs
- Presenting comprehensible error messages
- Organizing large amounts of theorems and properties
- Automating trivial proofs
- Datatype-generic programming
- . . .

Conclusions

- Haskell is a mature general-purpose language with a fantastic type system compared to mainstream languages.
- Haskell is very suitable to quickly try out ideas and get stuff done.
- One always wants more. Dependently-typed programming languages like Agda2 are the next step. They allow programs and verification to live side-by-side. But many interesting practical problems remain.