Verification of Functional Programs
Preliminary Concepts

Andrés Sicard-Ramírez

EAFIT University

Semester 2014-1

What is a Type?

- A type is a set of values (and operations on them).

What is a Type?

- A type is a set of values (and operations on them).
- Types as ranges of significance of propositional functions. Let $\varphi(x)$ be a (unary) propositional function. The type of $\varphi(x)$ is the range within which x must lie if $\varphi(x)$ is to be a proposition [Russell (1903) 1938, Appendix B: The Doctrine of Types].

  In modern terminology, Rusell's types are domains of propositional functions.

What is a Type?

- A type is a set of values (and operations on them).
- Types as ranges of significance of propositional functions. Let $\varphi(x)$ be a (unary) propositional function. The type of $\varphi(x)$ is the range within which x must lie if $\varphi(x)$ is to be a proposition [Russell (1903) 1938, Appendix B: The Doctrine of Types].

  In modern terminology, Rusell's types are domains of propositional functions.

  Example

  Let $\varphi(x)$ be the propositional function 'x is a prime number'. Then $\varphi(x)$ is a proposition only when its argument is a natural number.

$$\varphi : \mathbb{N} \to \{\mathrm{False}, \mathrm{True}\}$$
$$\varphi(x) = x \text{ is a prime number.}$$

What is a Type?

- 'A type is an approximation of a dynamic behaviour that can be derived from the form of an expression.' [Kiselyov and Shan 2008, p. 8]

What is a Type?

- ‘A type is an approximation of a dynamic behaviour that can be derived from the form of an expression.’ [Kiselyov and Shan 2008, p. 8]
- The propositions-as-types principle (Curry-Howard correspondence)

What is a Type?

- ‘A type is an approximation of a dynamic behaviour that can be derived from the form of an expression.’ [Kiselyov and Shan 2008, p. 8]
- The propositions-as-types principle (Curry-Howard correspondence)
- Homotopy Type Theory (HTT)

  Propositions are types, but not all types are propositions (e.g. higher-order inductive types)

What is a Type?

Example (some Haskell's types)

- Type variables: `a`, `b`
- Type constants: `Int`, `Integer`, `Char`
- Function types: `Int → Bool`, `(Char → Int) → Integer`
- Product types: `(Int, Char)`, `(a, b)`
- Disjoint union types:

```
data Sum a b = Inl a | Inr b
```

Type Systems

- Over-sized slogan:

  'Well-type programs cannot "go wrong".' [Milner 1978, p. 348]

Type Systems

- Over-sized slogan:

  'Well-type programs cannot "go wrong".' [Milner 1978, p. 348]

- 'A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.' [Pierce 2002, p. 1]

Referential Transparency

'We use [referential transparency] to refer to the fact of mathematics which says: The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value.' [Stoy 1977, p. 5].

Referential Transparency

'We use [referential transparency] to refer to the fact of mathematics which says: The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value.' [Stoy 1977, p. 5].

'A language that supports the concept that "equals can be substituted for equals" in an expression without changing the value of the expression is said to be *referentially transparent*.' [Abelson and Sussman 1996, p. 233].

Referential Transparency

Example

The following C program prints `hello, world` twice.

```
#include <stdio.h>

int
main (void)
{
  printf ("hello, world");
  printf ("hello, world");

  return 0;
}
```

Referential Transparency

Example

The following C program prints `hello, world` once.

```
#include <stdio.h>

int
main (void)
{
  int x;

  x = printf ("hello, world");
  x; x;

  return 0;
}
```

Referential Transparency

Example

The following Haskell program prints `hello, world` twice.

```haskell
main :: IO ()
main = putStr "hello, world" >> putStr "hello, world"
```

Referential Transparency

In Haskell, given

```
let x = exp
in  ... x ... x ...
```

the meaning of `... x ... x ...` is the same as `... exp ... exp ...`

Referential Transparency

In Haskell, given

```
let x = exp
in  ... x ... x ...
```

the meaning of `... x ... x ...` is the same as `... exp ... exp ...`

Example

The following Haskell program prints `hello, world` twice.

```
main :: IO ()
main = let x :: IO ()
           x = putStr "hello, world"
       in x >> x
```

Example

The following Haskell program prints `hello, world` twice.

```
main :: IO ()
main = x >> x
  where x :: IO ()
        x = putStr "hello, world"
```

Pure Functions

Side effects

'A side effect introduces a dependency between the global state of the system and the behaviour of a function... Side effects are essentially invisible inputs to, or outputs from, functions.' [O'Sullivan, Goerzen and Stewart 2008, p. 27].

Pure Functions

Side effects

'A side effect introduces a dependency between the global state of the system and the behaviour of a function... Side effects are essentially invisible inputs to, or outputs from, functions.' [O'Sullivan, Goerzen and Stewart 2008, p. 27].

Pure functions

'Take all their input as explicit arguments, and produce all their output as explicit results.' [Hutton 2007, p. 87].

Pure Functions

Are the following GHC 7.8.2 functions, pure functions?

```
maxBound :: Int     -- Prelude
os       :: String  -- System.Info
```

*From: `https://wiki.haskell.org/Referential_transparency`, 2014-02-25.

Pure Functions

Are the following GHC 7.8.2 functions, pure functions?

```
maxBound :: Int     -- Prelude
os       :: String  -- System.Info
```

'One perspective is that Haskell is not just one language (plus `Prelude`), but a family of languages, parametrized by a collection of implementation-dependent parameters. Each such language is RT, even if the collection as a whole might not be. Some people are satisfied with situation and others are not.' *

*From: `https://wiki.haskell.org/Referential_transparency`, 2014-02-25.

Functions are First-Class Citizens

Source: Abelson and Sussman [1996]

- They can be passed as arguments and they can be returned as results (higher-order functions)
- They can be assigned to variables
- They can be stored in data structures

Bottom

Working with functions how handle undefined values yielded by partial functions or non-terminating functions?

Example

```
head :: [a] → a
head (x : _) = x

head [] = ?
```

Bottom

Working with functions how handle undefined values yielded by partial functions or non-terminating functions?

Example

```haskell
head :: [a] → a
head (x : _) = x

head [] = ?
```

Example

```haskell
fst :: (a, b) → a
fst (x, _) = x

ones :: [Int]
ones = 1 : ones

fst (ones, 10) = ?
```

Bottom

The $\perp$ symbol represents the undefined value.
($\perp$ is represented in Haskell by the **`undefined`** keyword)

Example (first version)

```
head []          = undefined
fst (ones, 10) = undefined
```

*See 'Hussling Haskell types into Hasse diagrams' from Edward Z. Yang's blog on December 6, 2010.

Bottom

The $\perp$ symbol represents the undefined value.
($\perp$ is represented in Haskell by the **undefined** keyword)

Example (first version)

```
head []         = undefined
fst (ones, 10) = undefined
```

Remark

The $\perp$ value is polymorphic in Haskell.

Remark

The Haskell types are lifted types.*

*See 'Hussling Haskell types into Hasse diagrams' from Edward Z. Yang's blog on December 6, 2010.

Example (second version)

$$\text{head } [] = \bot_{\mathsf{a}}$$
$$\text{fst } (\text{ones}, 10) = \bot_{[\mathsf{Int}]}$$

Therefore, `head []` $\neq$ `fst (ones, 10)`.

Bottom

Example

```haskell
foo :: Int → Int
foo 0 = 0

bar :: Int → Int
bar n = bar (n + 1)

foobar :: Int → Int
foobar n = if foo n == 0 then 1 else 2
```

Example

```
foo :: Int → Int
foo 0 = 0

bar :: Int → Int
bar n = bar (n + 1)

foobar :: Int → Int
foobar n = if foo n == 0 then 1 else 2
```

Can we replace `foo` by `bar` in `foobar`?

Bottom

Example

```
foo :: Int → Int
foo 0 = 0

bar :: Int → Int
bar n = bar (n + 1)

foobar :: Int → Int
foobar n = if foo n == 0 then 1 else 2
```

Can we replace `foo` by `bar` in `foobar`? Only for $n \neq 0$.

Lazy Evaluation

See slides for the chapter 12 on the book by Hutton [2007]:
http://www.cs.nott.ac.uk/~gmh/book.html.

Strict and Non-Strict Functions

Definition

Let f be a unary function. If f $\perp = \perp$ then f is a **strict** function, otherwise it is a **non-strict** function. The definition generalise to n-ary functions.

Example

The `three` function is non-strict.

```haskell
three :: a → Int
three _ = 3
```

```haskell
three undefined               = 3
three (head [])               = 3
three (fst (ones, 10))        = 3
three (putStr "hello, world") = 3
```

Strict and Non-Strict Functions

Example

```
three :: a → Int
three _ = 3
```

Non-strict reasoning...

$$(\forall x \in \mathsf{Int})(\forall y)(x + \mathsf{three}\ y = x + 3).$$

Example

(Why Haskell hasn't a predefined recursive data type for natural numbers?)

```
data Nat = Zero | Succ Nat
```

```
Zero :: Nat
Succ :: Nat → Nat
```

Strict and Non-Strict Functions

Example

(Why Haskell hasn't a predefined recursive data type for natural numbers?)

```
data Nat = Zero | Succ Nat

Zero :: Nat
Succ :: Nat → Nat
```

Is `Succ` a non-strict function?

Strict and Non-Strict Functions

Example

(Why Haskell hasn't a predefined recursive data type for natural numbers?)

```
data Nat = Zero | Succ Nat
```

```
Zero :: Nat
Succ :: Nat → Nat
```

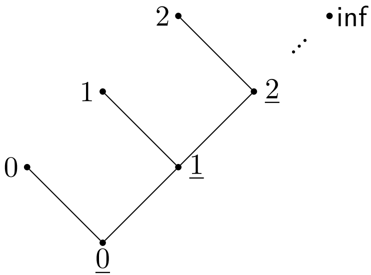
Is `Succ` a non-strict function?

We can define

```
inf :: Nat
inf = Succ inf
```

Example (cont.)

`Nat` represents the lazy natural numbers, that is, `Succ ⊥ ≠ ⊥` [Escardó 1993].



$$
\begin{aligned}
\underline{0} &= \bot, \\
\underline{n+1} &= \mathsf{Succ}\ n, \\
\mathsf{inf} &= \bigsqcup_{n \in \omega} \underline{n}
\end{aligned}
$$

Partially Ordered Sets

Definition

A **partially ordered set (poset)** $(D, \sqsubseteq)$ is a set D on which the binary relation $\sqsubseteq$ satisfies the following properties:

$$\begin{aligned} \forall x.\, x \sqsubseteq x & \qquad \text{(reflexive)} \\ \forall x\, \forall y\, \forall z.\, x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z & \qquad \text{(transitive)} \\ \forall x\, \forall y.\, x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y & \qquad \text{(antisymmetry)} \end{aligned}$$

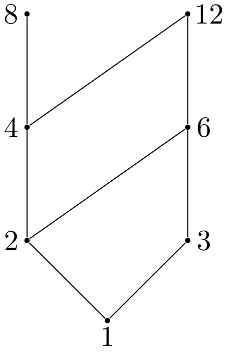
Partially Ordered Sets

Examples

- $(\mathbb{Z}, \leq)$ is a poset.
- Let $a, b \in \mathbb{Z}$ with $a \neq 0$. The divisibility relation is defined by $a \mid b := \exists c\ (ac = b)$. Then $(\mathbb{Z}^+, |)$ is a poset.
- $(P(A), \subseteq)$ is a poset.

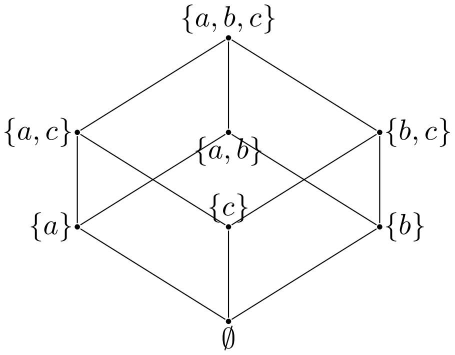
Example

Hasse diagram for the poset $(\{1, 2, 3, 4, 6, 8, 12\}, |)$.



Example

Hasse diagram for the poset $(\{a, b, c\}, \subseteq)$.



Monotone Functions

Definition

Let $(D, \sqsubseteq)$ and $(D', \sqsubseteq')$ be two posets. A function $f : D \rightarrow D'$ is **monotone** iff

$$\forall x \; \forall y. \; x \sqsubseteq y \Rightarrow f(x) \sqsubseteq' f(y).$$

Some Concepts of Fixed-Point Theory

Let D be a set, $(D, \sqsubseteq)$ be a poset and f be a function $f : D \to D$.

Definition

An element $d \in D$ is a **fixed-point** of f iff

$$f(d) = d.$$

Let D be a set, $(D, \sqsubseteq)$ be a poset and f be a function $f : D \to D$.

Definition

An element $d \in D$ is a **fixed-point** of f iff

$$f(d) = d.$$

Definition

The **least/greatest fixed-point** of f is least/greatest among the fixed-points of f.

Some Concepts of Fixed-Point Theory

Let D be a set, $(D, \sqsubseteq)$ be a poset and f be a function $f : D \to D$.

Definition

An element $d \in D$ is a **fixed-point** of f iff

$$f(d) = d.$$

Definition

The **least/greatest fixed-point** of f is least/greatest among the fixed-points of f.

That is, $d \in D$ is the least/greatest fixed-point of f iff:

- $f(d) = d$ and
- $\forall x. f(x) = x \Rightarrow d \sqsubseteq x$ / $\forall x. f(x) = x \Rightarrow x \sqsubseteq d$.

Some Concepts of Fixed-Point Theory

Theorem

Let $(D, \sqsubseteq)$ be a poset and $f : D \rightarrow D$ be monotone. Under certain conditions f has a least fixed-point [Winskel 1994] and a greatest fixed-point [Ésik 2009].

Some Concepts of Fixed-Point Theory

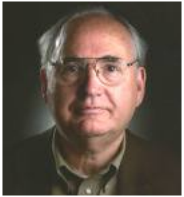
Theorem

Let $(D, \sqsubseteq)$ be a poset and $f : D \to D$ be monotone. Under certain conditions f has a least fixed-point [Winskel 1994] and a greatest fixed-point [Ésik 2009].

Notation

The least and greatest fixed-points of f are denoted by $\mu x.f(x)$ and $\nu x.f(x)$, respectively.

Motivation: Does λ-calculus have models?



'Historically my first model for the λ-calculus was discovered in 1969 and details were provided in Scott [1972] (written in 1971).' [Scott 1980, p. 226.].

Introduction to Domain Theory

Non-standard definitions

pre-domain, domain, complete partial order (cpo), ω-cpo, bottomless ω-cpo, Scott's domain, ...

Convention

domain $\equiv$ ω-complete partial order

ω-Complete Partial Orders

Definition

Let $(D, \sqsubseteq)$ be a poset. A ω-**chain** of D is an increasing chain

$$d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots$$

where $d_i \in D$.

ω-Complete Partial Orders

Definition

Let $(D, \sqsubseteq)$ be a poset. The poset D is a ω-**complete partial order** (ω-cpo) iff [Plotkin 1992]:

1. There is a least element $\bot \in D$, that is, $\forall x. \bot \sqsubseteq x$. The element $\bot$ is called *bottom*.
2. For every increasing ω-chain $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots$, the least upper bound $\bigsqcup_{n \in \omega} d_n \in D$ exists.

ω-Complete Partial Orders

Definition

Let A be a set. The symbol $A_\perp$ denotes the ω-cpo whose elements $A \cup \{\perp\}$ are ordered by

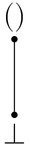
$$x \sqsubseteq y \quad \text{iff} \quad x = \perp \text{ or } x = y.$$

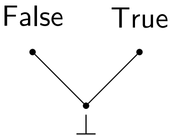
The ω-cpo $A_\perp$ is called A **lifted** [Mitchell 1996].

ω-Complete Partial Orders

Examples

The lifted unit type and the lifted Booleans $B_\perp$ are ω-cpos.





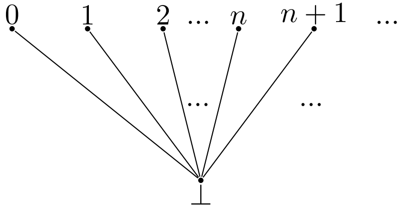

```
data () = ()
```

```
data Bool = True | False
```

ω-Complete Partial Orders

Example

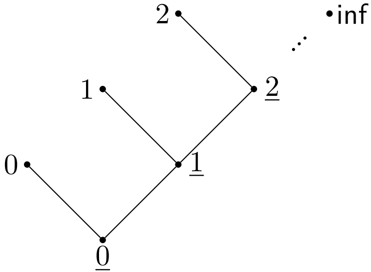
The lifted natural numbers $N_\perp$.



Example

The lazy natural numbers ω-cpo.

```haskell
data Nat = Zero | Succ Nat
```



$$
\begin{aligned}
\underline{0} &= \bot, \\
\underline{n+1} &= \mathsf{Succ}\ n, \\
\mathsf{inf} &= \bigsqcup_{n\in\omega} \underline{n}
\end{aligned}
$$

$$\bigsqcup_{n\in\omega} \underline{n} = \bot \sqsubseteq \mathsf{Succ}\ \bot \sqsubseteq \mathsf{Succ}\ (\mathsf{Succ}\ \bot) \sqsubseteq \cdots$$

Admissible Properties

Definition

Let D be a w-cpo. A property P (a subset of D) is w-**inductive (admissible)** iff whenever $\langle x_n \rangle_{n \in \omega}$ is an increasing sequence of elements in P, then $\bigsqcup_{n \in \omega} x_n$ is also in P, that is,

$$\forall n \in \omega.\, P(x_n) \Rightarrow P\left(\bigsqcup_{n \in \omega} x_n\right).$$

References

Abelson, Harold and Sussman, Gerald Jay (1996). Structure and Interpretation of Computer Programs. 2nd ed. MIT Press (cit. on pp. 11, 12, 23).

Escardó, Martín Hötzel (1993). On Lazy Natural Numbers with Applications to Computability Theory and Functional Programming. SIGACT News 24.1, pp. 61–67. DOI: 10.1145/152992.153008 (cit. on p. 38).

Ésik, Zoltán (2009). Fixed Point Theory. In: Handbook of Weighted Automata. Ed. by Droste, Manfred, Kuich, Werner and Vogler, Heiko. Monographs in Theoretical Computer Science. An EATCS Series. Springer. Chap. 2 (cit. on pp. 47, 48).

Hutton, Graham (2007). Programming in Haskell. Cambridge University Press (cit. on pp. 19, 20, 32).

Kiselyov, Oleg and Shan, Chung-chieh (2008). Interpreting Types as Abstract Values. Formosan Summer School on Logic, Language and Computacion (FLOLAC 2008) (cit. on pp. 5–7).

Milner, Robin (1978). A Theory of Type Polymorphism in Programming. Journal of Computer and System Science 17, pp. 348–375 (cit. on pp. 9, 10).

Mitchell, John C. (1996). Foundations for Programming Languages. MIT Press (cit. on p. 53).

References

O'Sullivan, Bryan, Goerzen, John and Stewart, Don (2008). Real World Haskell. O'Really Media, Inc. (cit. on pp. 19, 20).

Pierce, Benjamin C. (2002). Types and Programming Languages. MIT Press (cit. on pp. 9, 10).

Plotkin, Gordon (1992). Post-graduate Lecture Notes in Advance Domain Theory (Incorporating the "Pisa Notes"). Electronic edition prepared by Yugo Kashiwagi and Hidetaka Kondoh. URL: http://homepages.inf.ed.ac.uk/gdp/ (visited on 29/07/2014) (cit. on p. 52).

Russell, Bertrand [1903] (1938). The Principles of Mathematics. 2nd ed. W. W. Norton & Company, Inc (cit. on pp. 2–4).

Scott, Dana (1972). Continuous Lattices. In: Toposes, Algebraic Geometry and Logic. Ed. by Lawvere, F. W. Vol. 274. Lecture Notes in Mathematics. Springer, pp. 97–136. DOI: 10.1007/BFb0073967 (cit. on p. 49).

— (1980). Lambda Calculus: Some Models, Some Philosophy. In: The Kleene Symposium. Ed. by Barwise, Jon, Keisler, H. Jerome and Kunen, Kenneth. Vol. 101. Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, pp. 223–265 (cit. on p. 49).

Stoy, Joseph (1977). Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press (cit. on pp. 11, 12).

References

Winskel, Glynn (1994). The Formal Semantics of Programming Languages. An Introduction. Second printing. MIT Press (cit. on pp. 47, 48).