Dependently typed functional languages

Andrés Sicard-Ramírez

(Last modification: May 25, 2011)

Examination

Homework	30%
Paper presentation	30%
Project	40%

What are dependent types?

Types that depend on element of other types.

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.[1]

  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.

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

  Related reading: R. L. Constable. The triumph of types: *Principia Mathematica's* impact on computer science. Presented at the *Principia Mathematica* anniversary symposium, 2010.

[1] B. Russell. *The Principles of Mathematics*. W. W. Norton & Company, Inc, 2 edition, 1938.

What is a type? (cont.)

- “A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute”.[2]
- A type is an approximation of a dynamic behaviour that can be derived from the form of an expression.[3]

[2] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002. p. 1.

[3] O. Kiselyov and C. Shan. Interpreting types as abstract values. Formosan Summer School on Logic, Language and Computacion (FLOLAC 2008), 2008.

What is a type? (cont.)

- BHK (Brouwer, Heyting, Kolmogorov) interpretation: A type is a set, a proposition, a problem, a specification.

Type A	Term $a : A$	
A is a set	a is an element of the set A	$A \neq \emptyset$
A is a proposition	a is a proof (construction) of the proposition A	A is true
A is a problem	a is a method of solving the problem A	A is solvable
A is a specification	a is a program than meets the specification A	A is satisfiable

Applications of dependent types

- To reduce the semantic gap between programs and their properties
- To carry useful information for programs optimisation
- The propositions-as-types-principle: Computational interpretation of logical constants
- Unified programing logics (programs, specification, satisfaction relation)

Dependently typed systems

(See the `Wikipedia` article on dependent types for a more complete list)

- The ALF-family (Gothenburg - Sweden)
  - ALF
  - Agda
  - Alfa. Graphical interface for Agda
  - AgdaLight. Experimental version of Agda
  - Agda 2. Based on Martin-Löf type theory. Direct manipulation of proofs-objects. Backends to Haskell or Epic. Written in Haskell.

Dependently typed systems (cont.)

- Cayenne (Gothenburg - Sweden). An Haskell-like language with dependent types. Written in Haskell.
- Coq (INRIA - France). Based on the Calculus of Inductive Constructions. Tactic-based. Extraction of programs to Objective Caml, Haskell or Scheme. Written in Objective Caml (with a bit of C).
- DML (USA). An extension of ML with a restricted form of dependent types. Written in Objective Caml.
- Epigram 2 (England). Based on a closed type theory. Direct manipulation of proofs-objects. The language is in development.
- NuPrl (Cornell - USA). Based on Computational Type Theory. Tactic-based. Written in ML.

Dependent types

$B(x)$: Dependent type for $x : A$

Definition (Dependent product type (Pi types)).

$\prod_{x:A} B(x)$ is the type of functions f with $f\ a : B(a)$ if $a : A$.

Note:

If $B(x) = B$ for all $x : A$, then $\prod_{x:A} B(x) \equiv A \to B$.

(Agda notation: $(x : A) \to B$)

Dependent types (cont.)

$B(x)$: Dependent type for $x : A$

Definition (Dependent sum type (Sigma types)).

$\sum\limits_{x:A} B(x)$ is the type of pairs (a, b) with $a : A$ and $b : B(a)$.

Note:

If $B(x) = B$ for all $x : A$, then $\sum\limits_{x:A} B(x) \equiv A \times B$.

The propositions-as-types principle

(The Curry-Howard isomorphism)

(The Brouwer - Heyting - Kolmogorov - Schönfinkel - Curry - Meredith - Kleene - Feys - Gödel - Läuchli - Kreisel - Tait - Lawvere - Howard - de Bruijn - Scott - Martin-Löf - Girard - Reynolds - Stenlund - Constable - Coquand - Huet - . . . - isomorphism)[4]

[4] M.-H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006, p. viii.

The propositions-as-types principle (cont.)

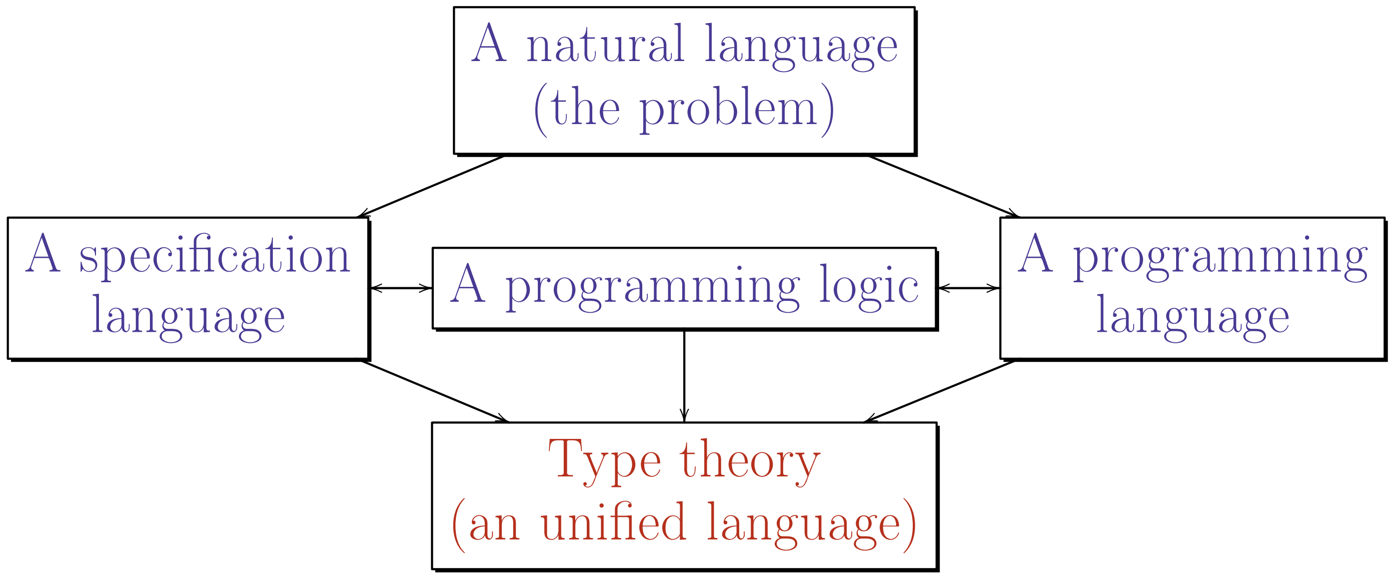
- A proposition corresponds to the type of its proofs.
- A proposition is true if the corresponding type is non-empty.

Logic	Dependently typed λ-calculus
false formula	bottom type
true formula	unit type
implication	function type
conjunction	product type
disjunction	sum type
universal quantification	dependent product type
existential quantification	dependent sum type

The propositions-as-types principle (cont.)

Logic	Dependently typed λ-calculus
$\bot$	$\bot$
$\top$	$\top$
$A \supset B$	$A \to B$
$A \wedge B$	$A \times B$
$A \vee B$	$A + B$
$\neg A$	$A \to \bot$
$\forall x.B(x)$	$\prod_{x:A} B(x)$
$\exists x.B(x)$	$\sum_{x:A} B(x)$

Reasoning about of programs



Reasoning about of programs (cont.)

Correctness of an program:

- Partial correctness: If the program terminates then its answer is correct respect to a specification.
- Total correctness = Partial correctness + termination proof.

Two approaches:

- Verification (external logic): The program is defined using a weak specification and we prove some theorems about it.
- Correct programs for construction (internal logic): The program is defined using a strong specification.

Example strong/weak specification

Strong specification for the greatest common divisor

```
_|_ : ℕ → ℕ → Set  -- Divisibility relation

gcd : (m n : ℕ) →
      Σ ℕ (λ r → r | m ∧
                 r | n ∧
                 ((r' : ℕ) → r' | m → r' | n → r ≥ r'))
gcd = ...
```

Example strong/weak specification (cont.)

Weak specification for the greatest common divisor

```
gcd : ℕ → ℕ → ℕ
gcd = ...

-- Some theorems to be proved

gcd-fst : (m n : ℕ) → gcd m n | m
gcd-fst = ...

gcd-ge : (m n r' : ℕ) → r' | m → r' | n → gcd m n ≥ r'
gcd-ge = ...
```

Limitations of type theory

We want: Dependent types + type checking decidability

Feature: Computations at the level of types

Problem: Non-terminating/partial computations

```
ntc : ℕ → ℕ
ntc n = ...

thm : ∀ n → ... n ... ≡ ntc n
thm zero     = ?
thm (succ n) = ?
```

Limitations of type theory (cont.)

Solution: All functions must be total

Restriction:

- Structural recursion: The recursive calls are only in structurally smaller elements (Martin-Löf type theory)
- Termination checker (Agda)

Research area: General recursion/partiality in type theory[5]

Consequence: Type theory is not a Turing-complete language

Discussion: Does it matter?

[5]Related reading: A. Bove, A. Krauss, and M. Sozeau. Partiality and recursion in interactive theorem provers. An overview. Accepted for publication at Mathematical Structures in Computer Science, special issue on DTP 2010, 2012.

General recursion: An alternative approach

Combining automatic and interactive proof in first order theories of combinators

- See slides for the Agda Implementors' Meeting XIII talk by Ana Bove and Peter Dybjer (`wiki.portal.chalmers.se/agda/pmwiki.php?n=AIMXIII.ProgramEtc`).
- See code related to the above talk (`www1.eafit.edu.co/asicard/code/fotc/`).