A short overview of Type Theory

Yves Bertot

June 2015

Motivation for types

- You know types, for instance in C `int x = 3;`
- Type errors are detected at *compile-time*
- Type verification removes errors from *run-time* errors
- Not powerful enough to remove all errors
- Type Theory: catch more errors at compile-time
- Comes from research on the foundations of mathematics
- This talk: simplified accound of types from mathematics and functional programming languages
- Conclusion: where Coq comes from, a demo of this system

λ-calculus

- A small-scale model of programming languages,
- Extremely simple
  - Three constructs
    - function descriptions, function calls, variables
  - Only one input to functions
  - Only one output to functions
- No complications
  - Higher order: programs are values
  - No control on memory usage
  - several possible evaluation strategies

Syntax ofλ-calculus

- $\lambda x.e$ is the function that maps x to e
- A function is applied to an argument by writing it on the left
- $a\ e_1\ e_2 = (a\ e_1)\ e_2$,
  - Several argument functions are a particular case
- if *plus* is the adding function and $\boxed{1}$ and $\boxed{2}$ numbers, then *plus* $\boxed{1}\ \boxed{2}$ is a number, and *plus* $\boxed{1}$ is a function
- notation $\lambda xy.e$ for $\lambda x.\lambda y.e$,
- numbers, pairs, and data lists can be modeled.

Computing with the λ-calculus

- the value of $(\lambda x.e)\ a$ is the same as the value of e where all occurrences of x are replaced by a,
- exemple: $(\lambda x.plus\ 1\ x)\ 2 = plus\ 1\ 2$,
- beware of bound variables: they are the occurrences of x that should be replaced when computing e

$$
\begin{aligned}
(\lambda x.\ plus((\lambda x.x)\ 1)\ x)\ 2 &= (\lambda x.\ plus\ 1\ x)\ 2 \\
(\lambda x.\ plus((\lambda x.x)\ 1)\ x)\ 2 &= plus\ ((\lambda x.x)\ 1)\ 2
\end{aligned}
$$

- the occurrences of x in e of $\lambda x.e$ are called the bound variables,
- the free occurrences of x in e are bound in $\lambda x.e$.

recursion and infinite computation

- A recursive program can call itself
- Example $x! = 1$ (si $x = 0$) ou bien $x! = x * (x - 1)!$
- In other words, there exists a function F such that $f = F\ f$
- For *fact*,
  $fact = \lambda x.$ if $x = 0$ then 1 else $x * fact(x - 1)$
  fact is a fixed point of
  $\lambda f\ x.$ if $x = 0$ then 1 else $x * f(x - 1)$
- In pure λcalculus, there exists $Y_T = (\lambda xy.y(xxy))\lambda xy.y(xxy)$, so that $Y\ F = F(Y\ F)$
- Y_T can be used to construct recusive functions
- Be careful for the evaluation strategy in presence of Y_T
  $Y_T\ F \to F(Y_T\ F) \to F(F(Y_T\ F)) \to \cdots$

A detailed explanation of fixed point computation

Name $\theta = (\lambda xy.y(xxy))$

$$
\begin{aligned}
\theta\theta F &= (\lambda xy.y(xxy))\theta F \\
&= (\lambda y.y(\theta\theta y))F \\
&= (\lambda y.y(\theta\theta y))F \\
&= F(\theta\theta F)
\end{aligned}
$$

Usual theorems about λ-calculus

Church Rosser property: if $t \xrightarrow{*} t_1$ and $t \xrightarrow{*} t_2$, then there exists t_3 such that $t_1 \xrightarrow{*} t_3$ and $t_2 \xrightarrow{*} t_3$,

Uniqueness of normal forms: if $t \xrightarrow{*} t'$ and t' can not be reduced further, then t' is unique,

Réduction standard: the strategy "outermost-leftmost" reaches the normal form when it exists.

- Beware that some terms have no normal form $(\lambda x.xx)\lambda x.xx \to (\lambda x.xx)\lambda x.xx \to \cdots$.

Representing data-types

- Boolean: T is encoded as $\lambda xy.x$, F as $\lambda xy.y$, If as $\lambda bxy.b\ x\ y$,
- pairs P: $\lambda xyz.z\ x\ y$, and projections π_i: $\lambda p.p\ (\lambda x_1\ x_2.x_i)$,
- Church encoding of numbers: n is represented by $\lambda fx.\overbrace{f(\cdots f}^{n}\ x)\cdots)$,
- addition: $\lambda nm.\lambda fx.n\ f\ (m\ f\ x)$,
  multiplication: $\lambda nm.\lambda f.n\ (m\ f)$,
- comparison to 0 (let's call it Q): $\lambda n.n\ (\lambda x.F)\ T$,
- predecessor: $\lambda n.\pi_1(n\ (\lambda p.\ P\ (\pi_2\ p)(add\ 1\ (\pi_2\ p)))(P\ 0\ 0))$,
- factorial: $Y_T \lambda fx.If\ (Q\ x)\ 1\ (mult\ x\ (f\ (pred\ x)))$.

Simply typed λ-calculus

- Annotate the functions with information about their input,
  - provide documentation on programs,
- The consistency of programs can be verified without executing programs
- collections used in annotations are called *types*,
- notation: $\lambda x : t.\ e$,
- primitive types, `int`, `bool`, ... but also function types $t_1 \to t_2$ (convention: $t_1 \to (t_2 \to t_3) \equiv t_1 \to t_2 \to t_3$),

Data-types and primitive operations

- Typing can handle new data-types and primitive operations,
- Making sure that operations are applied to compatible data,
- For instance, we add pairs and projectors,

$$\langle e_1, e_2 \rangle \qquad fst\langle e_1, e_2 \rangle \rightsquigarrow e_1$$

- New type for pairs: $t_1 * t_2$, and for $fst : t_1 * t_2 \to t_1$,
- Also possible to have native intgers

Examples

$\lambda f : int \to int \to int.\lambda x : int.f\ x\ (f\ x\ x)$ well typed
$\lambda f : int \to int.\lambda g : int \to int.f\ (g\ (f\ x))$ well typed if $x : int$,
$\lambda f : int.\lambda x : int \to int.f\ x$ badly typed,
$f\ f$ badly typed, whatever the type of f.

Type verification

- First stage: choose types for free variables
- verify that functions are applied to expressions of the correct type,
- recursive traversal of terms
- An algorithm described using inference rules

Typing rules

$$\frac{}{\Gamma, x : t \vdash x : t}\ (1) \qquad \frac{\Gamma \vdash x : t \qquad x \neq y}{\Gamma, y : t' \vdash x : t}\ (2)$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \langle e_1, e_2 \rangle : t_1 * t_2}\ (3)$$

$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x : t.\ e : t \to t'}\ (4)$$

$$\frac{\Gamma \vdash e_1 : t \to t' \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\ e_2 : t'}\ (5)$$

$$\frac{}{\Gamma \vdash \mathit{fst} : t_1 * t_2 \to t_1}\ (6) \qquad \frac{}{\Gamma \vdash \mathit{snd} : t_1 * t_2 \to t_2}\ (7)$$

Interpretation for logic

- Primitive types should be read as propositional variables,
- Read function types $t_1 \rightarrow t_2$ as implications,
- Read pair types $t_1 * t_2$ as conjunctions ("and"),
- The type of closed well-formed term is *always* a tautology,
  - *Curry-Howard isomorphism*, *types-as-propositions*,
- For a type t, finding e with this type, this means proving that it is a tautology
- Beware, all tautologies are not provable
- example: $((A \rightarrow B) \rightarrow A) \rightarrow A$ (Peirce's formula).

Peirce's formula

A	B	$A \to B$	$(A \to B) \to A$	$((A \to B) \to A) \to A$
T	T	T	T	T
T	F	F	T	T
F	T	T	F	T
F	F	T	F	T

Types and logic

- $\lambda x : A * B.\langle \mathit{snd}\ x, \mathit{fst}\ x\rangle$ is a proof of $A \wedge B \Rightarrow B \wedge A$,
- Several proof systems are based on this principle Nuprl, Coq, Agda, Epigram,
- A type verification tool is a simple program
- Finding proofs is a difficult problem,
- Verifying proofs is easy,
- typed λ-calculus is also a small-scale model of a proof verification tool

Typed reduction

- Same computation rule as for pure λ-calculus,
- We can add a computation rule for pairs and projections
- Standard theorems:

  subject reduction theorem types are preserved during computation,

  weak normalization Every typed term has a normal form,

  strong normalization Every reduction chain is finite

A crossroad

- Toward programming languages
  - Type inference
  - Polymorphism
  - General recursion
- Towards proof systems
  - Universal quantification
  - Proofs by induction
  - Guaranteeing computation termination

Structural recursion

- Avoid infinite computations, which are “undefined”,
- Providing recursive computations only for some types,
- Generalize primitive recursion,
- Well-formed types represent provable formulas
- reference : Gödel’s system T (cf. Girard & Lafont & Taylor *Proofs and types*),

Structural recursion for integers

- A new type `nat`,
- Three new constants:
  - `0 : nat` (represents 0)
  - `S : nat` $\rightarrow$ `nat` (represents *successor*),
  - `rec_nat`
- `rec_nat` is a recursor, it makes it possible to define recursive functions,
- Execution by pattern-matching (`rec_nat` $v\ f$ is a recursive function)
  - `rec_nat` $v\ f$ `0` $= v$
  - `rec_nat` $v\ f$ (`S` n) $= f\ n$ (`rec_nat` $P\ v\ f\ n$)
- Accordingly the type of `rec_nat` is:
  - `rec_nat :` $t \rightarrow$ (`nat` $\rightarrow t \rightarrow t$) $\rightarrow$ `nat` $\rightarrow t$, for any type t,
- Termination of computation is again guaranteed by typing

Examples of recursive functions

- addition: $plus \equiv \lambda xy.\texttt{rec\_nat}\ y\ (\lambda nv.\texttt{S}\ v)\ x$,
- predecessor: $pred \equiv \texttt{rec\_nat}\ 0\ (\lambda nv.n)$,
- subtraction: $minus \equiv \lambda xy.\texttt{rec\_nat}\ x\ (\lambda nv.pred\ v)\ y)$,
  - subtraction is also a comparison test, $minus\ x\ y = 0$ si $x \leq y$,
- multiplication: $\lambda xy.\texttt{rec\_nat}\ \texttt{0}\ (\lambda nv.plus\ y\ v)$,
- any function for which we can predict the number of recursive calls (for instance division)
- Even functions that are not recursive primitive: Ackermann.

Example of binary trees (if time allows)

- Introduce a new type `bin`,
- Two constructors:
  - `leaf : bin`,
  - `node : nat` $\rightarrow$ `bin` $\rightarrow$ `bin` $\rightarrow$ `bin`,

Example of binary trees (2)

- The recursor is defined accordingly to the constructors
  - `rec_bin` has three arguments (2+1), `rec_bin` f_1 f_2 x, is well-typed if the type of f_1 (resp. f_2) is adapted to pattern-matching and recursion by `leaf` (resp. `node`).
  - f_1 is a value of type t,
  - f_2 has (3+2) arguments,
    - 3 is the number of arguments of `node`,
    - 2 is the number of arguments of `node` in type `bin`,
  - extra arguments are values for recursive calls

$$\texttt{rec\_bin}\ f_1\ f_2\ (\texttt{node}\ n\ t_1\ t_2) =$$
$$f_2\ n\ t_1\ (\texttt{rec\_bin}\ f_1\ f_2\ t_1)\ t_2\ (\texttt{rec\_bin}\ f_1\ f_2\ t_2)$$

Recursors and pattern-matching

- Ocaml :

```
let rec_nat fun v f x =
    match x with
        O -> v | S p -> f p (rec_nat v f p)
```

Dependent types: Type families

- Functions whose values are types,
- "Diagonal" function: each value is in a different type (determined by a type family)
- example: A_i a sequence of types represented by a the function $A : \mathtt{nat} \to \mathtt{Type}$, we can think of a function f such that:
  - $f\ 0$ has type $A\ 0$,
  - $f\ 1$ has type $A\ 1$,
  - $f\ 2$ has type$A\ 2$,
  - and so on,
- the type of f is noted $f : \Pi x : \mathtt{nat}.A\ x$.

dependent products

- A pair $A_1 \times A_2$ maps an index $i \in \{1, 2\}$ to a value of type A_i,
- More generally a sequence $(a_0, \ldots, a_n, \ldots)$ makes it possible to map an index $i \in \mathbb{N}$ to a value in A_i ,
- Such sequence is in $A_0 \times A_1 \times \cdots \times A_n \times \cdots = \Pi_{i \in \mathbb{N}} A_i$,
- This notation of indexed product is adapted to describe this notion of function with dependent type

Typing rules for dependent products

$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x : t.e : \Pi x : t.\ t'}$$

$$\frac{\Gamma \vdash e_1 : \Pi x : t.t' \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\ e_2 : t'[e_2/x]}$$

- The notation $A \to B$ is shorthand for $\Pi x : A.B$ when x does not occur in B,
- if $f : \Pi x : \mathtt{nat}.A\ x$ then $f\ 1 : A\ 1$.

logical interpretation of dependent products

- if B has type $A \rightarrow \mathtt{Type}$, the logical interpretation is that B is a *predicate*
- if $t : B\ i$, then t is a proof of $B\ i$,
- if $f : \Pi i : A.B\ i$, then f makes it possible to constructs proofs of $B\ i$ for every $i : A$,
- Read $\Pi i : A.B\ i$ as universal quantification
  and $f : \Pi i : A.B\ i$ as the proof of a universal quantification
- In Coq, one never writes $\Pi i : A.B\ i$ but always $\forall i : A, B\ i$.

Building proofs

- assume there exists a predicate `even` (in French *pair*)
- assume we have two theorems:
  - `even0 : even 0`,
  - `even2 :` $\forall x : \texttt{nat},$ `even` $x \rightarrow$ `even (S (S` x`))`,
- We can compose these theorems to prove that a number is even
- For instance: `even2 0 even0 : even (S (S O))` is a proof that 2 is even
- `even2 2 (even2 0 even0) : even 4` is a proof that 4 is even
- `even2 4 (even2 2 (even2 0 even0)) : even 6`, and so on. . .

Dependent products and explicit polymorphism

- A polymorphic function has type $T[\alpha]$ for every possible instance of α,
- This can be described explicitely by stating T as a type family $T : \mathtt{Type} \to \mathtt{Type}$,
- The polymorphic type is described by $\Pi x : \mathtt{Type}. T\ x$ (with an extra argument),
- For instance the type of pairs $t_1 * t_2$ can be described by a constant $\mathtt{prod} : \mathtt{Type} \to \mathtt{Type} \to Type$,
- The notation $\langle e_1, e_2\rangle$ is described by $pair : \Pi t_1 : \mathtt{Type}.\Pi t_2 : \mathtt{Type}.t_1 \to t_2 \to \mathtt{prod} t_1 t_2$,
- Because of explicit polymorphism, `pair` now has 4 arguments, `fst` 3 arguments).

Dependent product and recursion

- rec_nat is a polymorphic constant, behavior repeated here
  - rec_nat $P\ v\ f\ \mathtt{0} = v$
  - rec_nat $P\ v\ f\ (\mathtt{S}\ n) = f\ n\ (\mathtt{rec\_nat}\ P\ v\ f\ n)$
- rec_nat should also be usable to define functions with a dependent type
  - Need a type family $P : \mathtt{nat} \rightarrow \mathtt{Type}$,
  - The value for 0 must be in P 0,
  - The value for $S\ n$ must be in $P\ (S\ n)$,
  - The value of any recursive call on n must be in $P\ n$,
- rec_nat :
  $\Pi P : \mathtt{nat} \rightarrow \mathtt{Type}.P\ \mathtt{0} \rightarrow (\Pi n : \mathtt{nat}.P\ n \rightarrow P\ (\mathtt{S}\ n)) \rightarrow \Pi n : \mathtt{nat}.P\ n$
- Logical formula: induction principle for natural numbers!

Inductive types and dependence

- Families of recursive types
- Elements of T_i may have sub-terms in T_j,
- example: complete binary trees:
  - `hleaf : T 0`,
  - `hnode :` Πn`:nat. A` $\to$ `T n` $\to$ `T n` $\to$ `T (S n)`
- The type of each tree has information about the height,
- The constructor `hnode` states that both subterms must have the same height
- A recursor can be constructed automatically

Inductive predicates

- In inductive type families some instances may not be inhabited
- Example: `even` indexed by `nat`, with two constructors
  - `even0`: `even 0`,
  - `even2`: $\forall n$:`nat`. `even` $n \to$ `even (S (S` n`))`,
- En interprétation logique, le type of the recursor expresses that `even` is satisfied only by even numbers
- ```
  even_ind :
  ∀ P : nat → Prop,
    P 0 →
    (∀ n:nat, even n → P n → P (S (S n)))→
  ∀ n:nat, even n → P n
  ```
  ```

The Coq system: the calculus of inductive constructions

- Inductive predicates are very powerful
- In Coq, they are used to represent logical connectives, equality, existential quantification, except $\forall$ and $\rightarrow$
- There are rules that govern the construction of dependent products to avoid paradoxes (Russell, Burali-Forti)
- One can define a new property by quantifying over all properties (*impredicativity*),
- A type inductive must satisfy constraints
- Recursors are replaced by a general notion of structural recursion

Simple uses of Coq

- One can use Coq without knowing about dependent types,
  - Defining only simply typed functions
  - One uses universal quantifiactions only in logical formula
  - The only type families one considers are inductive predicates
  - Tactics take care of constructing the most complex terms
- Dependent types can also be used for safer programming
- Future research
  - Make types less cumbersome (esp. for equality)
  - Integrate automatic proof search
  - Applications in reliable software development