Lambda-calculus and types

Yves Bertot

May 2015

In these course notes, we review several aspects of λ-calculus and typing in functional languages. This knowledge makes it easier to understand functional programming languages such as Lisp, ML [13], Ocaml [16], and proof system like Coq [5] or Agda [11].

Paradoxically, we start with a reduced model of programming language without types, then we show several extensions of this language with type notions. In particular, we show how types can be used to represent logical formulas and typed terms can be used to represent proofs.

One of the most difficult points of this course is the introduction of dependent types, which make it possible to define functions whose type expresses very precisely the expected properties of inputs and the guaranteed properties of outputs. A type system with dependent types makes it possible to construct programs for which many programming errors can be detected at compile time. Seen as a tool to reason logically, such a type system can be used to construct tools for the verification of proofs on a computer, like the Coq system. Since we are describing together a programming language and a logical system, we obtain a tool with which we can prove the absence of certain classes of errors.

1 A quick overview of pure lambda-calculus

Studying λ-calculus makes it easy to understand on a minimal language some theoretical concepts that will also be relevant to richer languages. For example, we can reason ont the problem of terminating programs, equivalence between two programs, and on the relations between programing and logic. This language is the simplest example of a functional programing language and most of our studies will naturally translate to other functional languages (ML, Haskell, Scheme). λ-calculus also makes it possible to understand what happens in procedure calls and in recursion, even for non functional programming languages like C or Java.

1.1 Syntax

The syntax of λ-calculus is usually given by stating that it is the set of expressions e obtained in the following fashion:

$$e ::= \lambda x.\ e \mid e_1 e_2 \mid x$$

More precisely, we usually start by supposing the existence of an infinite set V whose elements are called variables, noted x, y, $\ldots$, x_i, y_i, f, g and the set of λ-terms is built up progressively in the following fashion:

1. If x is a variable and e is a λ-term, then $\lambda x.\ e$ is also a λ-term. Such a term is called an *abstraction*. Intuitively, $\lambda x.\ e$ represents the function that takes x as input and returns the value e.

2. If e_1 and e_2 are two λ-terms, then $e_1 e_2$ (the two terms side by side), is a λ-term. Such a term is called an *application*. Intuitively, this application represents the function e_1 applied to the argument e_2.

3. If x is a variable, then it is also a λ-term.

Parentheses can be added in the text of a λ term to make it easier to read. One notable difference with usual mathematic practice is that we don't necessarily write parentheses around the arguments of a function (especially if the argument is a single variable).

Here are a few examples of λ terms that are frequently used:

$\lambda x.\ x$	(This one is often called **I**),
$\lambda x.\ \lambda\ y.\ x$	(**K**),
$\lambda x.\ (x\ x)$	(Δ),
$\lambda x.\ \lambda y.\ \lambda z.((x\ z)(y\ z))$	(**S**),
$(\lambda x.\ \lambda f.(f(((xx)f))))(\lambda f.\ \lambda x.(f(((xx)f))))$	($\mathbf{Y}_T$)
$\lambda x.\ \lambda y.\ (x\ y)$.	

Intuitively an abstraction describes a function : $\lambda x.\ e$ represents *the function that maps x to e*. Thus the terms described in the previous paragraph are simple functions: $\widehat{\mathtt{I}}$ is the identity function, `K` represents a function that takes as input a value and returns a constant function, *Delta* is not easy to understand as a mathematical function: it receives a function as argument and applies it to itself. The argument must at the same time be a function and a piece of data.

Having functions that receive functions as arguments is not so uncommon in computer science. For instance, a compiler or an operating system receive programs as arguments. Applying a function to itself also happens, for instance when using a compiler to compile itself.

1.2 α-equivalence, free and bound variables

In the term $\lambda x.\ e$ the variable x can of course appear in e. We can replace x with another variable y, under the condition that all occurrences of x are replaced by y, and under the condition that y did not appear previously in e. The new expression represents the same function. We say in this case that the two expressions are *α-equivalent*. For the major part of this paper, discussions will be made modulo α-equivalence, in other terms, two α-equivalent terms will usually be considered equal.

The α-equivalence relation is an equivalence and a congruence[1].

Here are a few examples and counter examples of α-equivalent pairs:

1. $\lambda x.\ \lambda y.\ y$, $\lambda y.\ \lambda x.\ x$, and $\lambda x.\ \lambda x.\ x$ are α-equivalent.

2. $\lambda x.\ (x\ y)$ and $\lambda y.\ (y\ y)$ are not α-equivalent.

[1] two terms that differ only by a subterm being replaced by an α-equivalent term are also α-equivalent.

The abstraction construct is a binding construct, the instances of x that appear in the term $\lambda x.\ e$ are bound to this abstraction. It is possible to change the name of all the bound occurrences at the same time as the one introduced by the abstraction.

On the other hand, some variables occuring in an expression may be related to no binding abstraction. These variables are called *free*. Intuitively, a variable x occuring in a term is free only if it is part of no expression of the form $\lambda x.\cdots$. The notion of free variable is stable modulo α-equivalence.

For instance, the variable y is free in the terms $\lambda x.\ x\ y$, $\lambda x.\ y$, $\lambda x.\ y\ (\lambda y.\ y)$, and $\lambda x.\ (\lambda y.\ y)\ y$ (for the last example, it is the last occurrence of y that is free).

Exercices

1. *What are the α-equivalence classes among the following terms:* $\lambda x.\ x\ y$, $\lambda x.\ x\ z$, $\lambda y.\ y\ z$, $\lambda z.\ z\ z$, $\lambda z.\ z\ y$, $\lambda f.\ f\ y$, $\lambda f.\ f\ f$, $\lambda y.\ \lambda x.\ x\ y$, $\lambda z.\ \lambda y.\ y\ z$.

2. *Find an α-equivalent term where each binder introduces a variable with a different name:*
$$\lambda x.\ ((x\ (\lambda y.\ x\ y))(\lambda x.\ x))(\lambda y.\ y\ x)$$

1.3 Application

Application represents the application of a function to an argument. It is written by placing the function on the left of the argument, with enough space so that the function and the argument can be distinguished.

λ-calculus provides only one kind, there is no need for a specific construct to express the application of a function to several arguments, because this can be described using application to one argument in a simple way. The trick is to have functions whose returned value is a function, which can in turn be applied to another argument.

For instance, we shall see later that we can "model" an addition function in pure λ-calculus. It is a function of one argument that returns another function. Adding two arguments x and y can be written the following way:

$$(plus\ x)\ y$$

The function $plus\ x$ is an offset-by-x function: it expects an input n and it returns $x+n$. For instance, we can write the function that computes the double of a number in the following way:

$$\lambda x.((plus\ x)\ x)$$

In what follows, we will avoid the excessive use of parenthesis by considering that it not necessary to use parentheses when a function meant as a several argument functions is applied to several arguments. The doubling function is written in the following way:

$$\lambda x.\ plus\ x\ x$$

The fact that we refrain from using parentheses in this case does not mean that application is associative. When parentheses appear on the right, they are usually meaningful. The following expression has a very different meaning:

$$\lambda x.\ plus\ (x\ x)$$

In this expression, $(x\ x)$ means: apply x to itself, and $plus\ (x\ x)$ expects $(x\ x)$ to compute a number which is given as first argument to $plus$.

So the take-home message is that you should not put parentheses just to group the arguments of a function accepting several arguments: it makes it believe that the first argument is actually a function applied to the other arguments.

Concerning notations, we will also note with a single λ functions of several arguments, so that $\lambda xyz.\ e$ is used instead of $\lambda x.\ \lambda y.\ \lambda z.\ e$. We will also avoid placing parentheses around the body of an abstraction, considering that this body extends as far as possible. thus the following term

$$(\lambda x.(plus\ x\ x))(plus\ y\ y)$$

will be written more succinctly

$$(\lambda x.\ plus\ x\ x)(plus\ y\ y).$$

This term actually computes four times y.

1.4 Substitution

It is possible to replace all the occurrences of a free variable by a λ-term, but one should be careful to make sure that this operation is stable modulo α-equivalence. More precisely, we note $e[e'/x]$ the term obtained by replacing all free occrrences of x by e' in e, making sure that all free occurrence of a variable in e' are still free in the final term. One approach is to perform two steps:

1. First construct a term e'' that is α-equivalent to e, but where no binding abstraction uses x or any of the free variables in e',
2. Then replace all occurrence of x by e' in e'', with no need to pay attention to binders.

It is also possible to describe recursively the substitution operation using the following equations:

- $x[e'/x] = e'$,
- $y[e'/x] = y$, if $y \neq x$,
- $(e_1\ e_2)[e'/x] = e_1[e'/x]\ e_2[e'/x]$,
- $(\lambda x.\ e)[e'/x] = \lambda x.\ e$,
- $(\lambda y.\ e)[e'/x] = \lambda y.(e[e'/x])$, if y is not free in e',
- $(\lambda y.\ e)[e'/x] = \lambda z.((e[z/y])[e'/x])$, if z is not free in e and e'.

The last equation can also be applied when the previous two can, but one should use the other two when possible. Using the last equation when the others can also be applied simply produces an α-equivalent term with more bound variables whose name changes.

1.5 Execution in the λ-calculus

λ-calculus commes with a notion called β-reduction based on substitution. The intuition is that every function applied to an argument can be unrolled. This behavious is simply written in the followin fashion, using the relation $\rightsquigarrow$:

$$(\lambda x.\ e)\ e' \rightsquigarrow e[e'/x]$$

Every instance of the left member is called a *β-redex*, or more simply a redex. This rule can be used on all possible instances in a term, wherever they are located in the term.

In the study of λ-calculus, it is customary to consider chains of elementary reduction, sometimes called *derivations*, or even *reductions*. We shal note $\rightarrow^\star$ the derivation relation.

A term that contains no redex is called a *normal form*. We shall say that a term has a normal form when there exists a derivation starting in this term and leading to a normal form.

Here are a few examples of reductions:

- $((\lambda x.\lambda y.\ x)\lambda x.\ x)\ z \rightsquigarrow (\lambda y.\lambda x.\ x)\ z \rightsquigarrow \lambda x.\ x$,
- $\mathbf{K}\ z(y\ z) = (\lambda xy.\ x)z(y\ z) \rightsquigarrow (\lambda y.\ z)\ (y\ z) \rightsquigarrow z$,
- $\mathbf{S}\ \mathbf{K}\ \mathbf{K} = (\lambda xyz.\ x\ z(y\ z))\mathbf{K}\mathbf{K} \rightsquigarrow (\lambda yz.\ \mathbf{K}z(y\ z))\mathbf{K} \rightsquigarrow (\lambda yz.\ z)\mathbf{K} \rightsquigarrow \lambda z.\ z = \mathbf{I}$
- $\Delta\ \Delta = (\lambda x.\ x\ x)\ \Delta \rightsquigarrow \Delta\ \Delta \rightsquigarrow \Delta\Delta$, the term $\Delta\Delta$ is often called Ω.
- $\mathbf{Y}_T \rightsquigarrow \lambda f.f\ (\mathbf{Y}_T f) \rightarrow^\star \lambda f.f\ (f\ (\mathbf{Y}_T f)) \rightsquigarrow \ldots$,
- $\mathbf{K}\ \mathbf{I}\ \Omega \rightsquigarrow \mathbf{K}\ \mathbf{I}\ \Omega \rightsquigarrow \ldots$,
- $\mathbf{K}\ \mathbf{I}\ \Omega \rightarrow^\star \mathbf{I}$.

These examples show that there exist terms without a normal form and that there also exist terms having a normal form, but which can also be the starting point of an infinite derivation.

λ-calculus contains enough primitives to represent the usual constructs needed for programming. For instance, we can adopt a convention to represent natural numbers, n being represented by

$$\lambda fx.\ \overbrace{f(\cdots(f\ x)\cdots)}^{n \text{ times}}.$$

Addition *add* and multiplication *mult* can then be represented by the following functions:

$$\lambda mnfx.\ m\ f(n\ f\ x) \qquad \lambda mn.\ m\ (n\ f)\ x.$$

Boolean values `true` and `false` can be represented by the expressions

$$\lambda xy.\ x \qquad \lambda xy.\ y,$$

The conditional construct if-then-else can be represented by

$$\lambda bxy.\ b\ x\ y.$$

The pair of two expressions can be represented by the following term:

$$\lambda z.\ z\ e_1\ e_2,$$

and the functions π_1 and pi_2 that take a pair as argument and return respectively the first and second component of this pair can be written:

$$\lambda c.\ c\ \lambda xy.\ x \qquad \lambda c.\ c\ \lambda xy.\ y.$$

The function *pred* that returns the predecessor of a natural number (or 0 if the input is already 0) can be written in the following manner:

$$\lambda n.\ \pi_1(n\ (\lambda cz.\ z\ (\pi_2\ c)(plus\ 1\ (\pi_2\ c)))\lambda z.\ z\ 0\ 0).$$

The function $eq0$ that tests whether an integer is 0 can be written:

$$\lambda n.\ n\ (\lambda x.\ false)\ true.$$

We have shown how to represent the data-type of natural numbers, with the usual operations and and a test function. In a similar way, one could describe the data-type of lists, with projections to get the first element of a list, the remainder of this list, and a test function to check whether a list is empty.

Even more generally, we can also produce recursive functions. The recursive functions that satisfies the equation $f\ x = e$ (f and x are free in e) can be represented by the expression:

$$Y_T\lambda fx.\ e.$$

This gives enough expressive power to represent the factorial function by the following text:

$$Y_T\lambda fx.\mathsf{if}\ (eq0\ x)\ 1\ (mult\ x\ (pred\ x)).$$

This expressive power is strong enough that we can claim λ-calculus to be a programming language.

2 Simply typed λ-calculus

As a programming language, λ-calculus has several drawbacks. The first drawback is that it is too easy to make programming mistakes. The second drawback is that this language does not use efficiently the data-structures already provided by computers, like numbers.

We shall now describe how λ-calculus was extended to add a notion of types to functions. Types make it possible to express what is the expected input data for each function and to express what is the expected data returned by each computation. This makes it possible to reduce the errors that can be made while programming and to include efficient operations on the data-types that are natively provided by the computer. We shall also see that in some contexts, types make it possible to guarantee that computations always terminate.

2.1 A language of types

Let's assume the existence of set P of names for primitive types. For instance, P can contain types `int`, `bool`, `float`. We consider the type language constructed in the following manner:

- Every primitive type from P is a type in T,
- if t_1 and $t2$ are two types, then $t_1 * t_2$ is a type,
- if t_1 and $t2$ are two types, then $t_1 \to t_2$ is a type.

Types of the form $t_1 * t_2$ are used to describe the type of pairs, we shall use the notation $\langle e_1, e_2 \rangle$ to describe the construction of a pair. The types of the form $t_1 \to t_2$ are used to describe the type of functions that take an argument of type t_1 and return a value of type t_2.

For instance, the type $\texttt{int} * \texttt{int}$ represents the type of pairs of integer values, while the type $\texttt{int} \to \texttt{int}$ represents the type of functions that take an integer as argument and return an integer. A function with two integer arguments and returning an integer can either be described as a function that takes a single argument of type pair of integers and returns an integers, or as a function that takes a single integer as input and returns another function that takes an integer as input and returns an integer. These two types are distinct, but a function named $\texttt{curry}_{int}$ can help going from one form to the other. Its type has the following shape:

$$((\texttt{int} * \texttt{int}) \to \texttt{int}) \to (\texttt{int} \to (\texttt{int} \to \texttt{int}))$$

In functional programming it is customary to define functions of several arguments without using pairs. One obtains function whose type is made of a longue sequence of arrows. The custom is to not print the parentheses when they occur on the right. Thus, the type of $\texttt{curry}_{int}$ is better written in the following fashion:

$$((\texttt{int} * \texttt{int}) \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

2.2 Annotating λ-terms with types

We shall now concentrate on a variant of λ-calculus where each function defined using the abstraction construct contains type information on the expected type of arguments. Expressions of this variant have the following shape:

$$e ::= x \mid \langle e_1, e_2 \rangle \mid \lambda x : t.\ e \mid e_1\ e_2 \mid \mathit{fst} \mid \mathit{snd}$$

In practice, it is useful to determine the type of expressions in this language, but to be able to do that, we need to know the types of all free variables. This information is collected in a *context*, a sequence of pairs combining a variable name and the corresponding type. These contexts will be noted with the variable Γ, the empty context will be noted $\emptyset$, and the context Γ to which one adds the pair asociating the variable x and the type t will be noted $\Gamma, x : t$.

It is customary to describe typing rules using the style of inference rules already used in logic and proof theory. To describe how one can decide whether a typed λ-term is well formed, one uses the following seven rules:

$$\frac{}{\Gamma, x : t \vdash x : t}\ (1) \qquad \frac{\Gamma \vdash x : t \qquad x \neq y}{\Gamma, y : t \vdash x : t}\ (2)$$

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

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

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

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

When typing rules are presented in this manner, it is also possible to describe the whole typing process for a complete expression as a big figure, which we call a *derivation tree*.

For instance, checking the type of $\lambda f : \texttt{int} * \texttt{int} \to \texttt{int}.\lambda x : \texttt{int}.\lambda y : \texttt{int}.\ f\langle x, y\rangle$ can be represented by the following derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \cdots, x : \texttt{int} \vdash x : \texttt{int}}\ (1)}{\Gamma, \cdots \vdash x : \texttt{int}}\ (2) \quad \dfrac{}{\Gamma, \cdots, y : \texttt{int} \vdash y : \texttt{int}}\ (1)}{\Gamma, \cdots \vdash \langle x, y\rangle : \texttt{int} * \texttt{int}}\ (3)}{\Gamma, \cdots \vdash f : \texttt{int} * \texttt{int} \to \texttt{int}}\ (2)}{\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}, x : \texttt{int}, y : \texttt{int} \vdash f\langle x, y\rangle}\ (5)}{\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}, x : \texttt{int} \vdash \lambda y : \texttt{int}.\ f\langle x, y\rangle : \texttt{int} \to \texttt{int}}\ (4)}{\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int} \vdash \lambda x : \texttt{int}.\lambda y : \texttt{int}.\ f\langle x, y\rangle : \texttt{int} \to \texttt{int} \to \texttt{int}}\ (4)}{\Gamma \vdash \lambda f : \texttt{int} * \texttt{int} \to \texttt{int}.\lambda x : \texttt{int}.\lambda y : \texttt{int}.\ f\langle x, y\rangle : (\texttt{int} * \texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}}\ (4)$$

2.3 Logical uses of types

The expressions of the type language can be read as if they were logical formulas: primitive types from P are propositional variables, the arrow type can be read as an implication and cartesian product (the type of pairs) as a conjunct. For instance, the logical proposition *if the sentence "A and B" is true then the sentence "B and A" is true too* will be represented by the following type:

$$A * B \to B * A.$$

This way of reading types as logical formulas is justified by an extra remark: if there exist an expression with the type t in the empty context, then this logical formula is a tautology (this formula is always true). In this case, the type is called *inhabited*, and any term in this type describes a proof of the corresponding logical formula. Moreover, when clearing the λ-terms that appear in a typing derivation and when consider the contexts appearing in each context simply as propositional variables assumed to be true, what one obtains is simply a proof from sequent calculus, a language invented to study proof theory in the first half of the XXth century.

For instance, the type $A * B \to B * A$ is inhabited by the following term:

$$\lambda x : A * B, \langle snd\ x, fst\ x \rangle$$

it is this use of types as logical formulas that provides the foundations of proofs systems in type theory like Coq [5]. The correspondence between types and logical formulas on one side and functional programs and proofs on the other side is often called the *Curry-Howard correspondence*[2].

All inhabited types are tautological formulas, but not all logical tautologies are inhabited types. For instance, it is well known that the following formula can be verified by a method of truth tables but there is no λ-term with this type in an empty context (this formulas is called Peirce's formula):

$$((A \to B) \to A) \to A.$$

The difference between formulas provable using typed λ-calculus and formulas provable using truth tables is known as the difference between *intuitionistic* and *classical* logic. The main difference revolves around the law of *excluded middle*, which simply expresses that *every formula is either true or false.*

Exercices

3. *Build a proof of*$(A \to B \to C) \to (B \to A \to C)$,
4. *Build a proof of*$(A \to B \to C) \to A * B \to C$,
5. *Build a proof of* $((((A \to B) \to A) \to A) \to B) \to B$, *Please note that this formula is very close to Peirce's law.*

2.4 Typed reduction

To describe computation in typed λ-calculus, we simply use λ-term reduction as in untyped λ-calculus, using β-reduction. An important property of reduction is a stability property: when reducing a λ-term, the type is preserved. This can be proved by induction on the size of expressions. This theorem is also called the "subject-reduction theorem."

We shall not give all the details of this proof. but only mention two lemmas.

First if e_1 has the type $t \to t'$ and reduces in e_1' with the same type, then the expression $e_1\ e_2$ is also well typed if and only $e_1'\ e_2$ is well-typed.

Second if e_1 has the type t in the context $\Gamma, x : t$ and if e_2 has the t in the context Γ, then the expression $(\lambda x : t.\ e_1)e_2$ is well typed and has the type t' in the context Γ. It remains to verify that $e_1[e_2/x]$ is well typed in the context Γ, which holds for the following two reasons:

1. In $e_1[e_2/x]$ the variable x does not appear anymore, and therefore the declaration $x : t$ is not necessary to verify that the expression is well typed,
2. the variable of type x of type t is replaced everywhere by an expression e_2 of type t.

[2]or isomorphism

2.5 Reduction termination

If we observe the rule 5 which describes how the application of a function is typed, one can see that the type of the function is a term that is strictly larger than the type of the argument. For this reason, it is not possible that an expression of the form $x\ x$ is well-typed. the expression $\Delta \equiv \lambda x.\ x\ x$ is not typable, and so neither the expression $\Omega \equiv \Delta\Delta$, nor the expression $Y = (\lambda zx.\ x(z\ z\ x))\lambda zx.\ x(z\ z\ x)$. These expression, which we saw in the section on pure λ-calculus, are involved in infinite derivations, but they are not typable. This just an instance of a very important theorem: *all typable expressions in simply typed λ-calculus are strongly normalizing.* In other words, all derivations involving typed λ-terms are finite. A proof of this theorem can be found in [7], a generalization can also be found in [9].

Having eliminated causes for non-termination may seem a great progress, but in doing so we also loose the possibility to define recursive functions. A first solution is to re-introduce recursion while preserving typing by adding a constant Y in the language with an extra reduction rule:

$$Y\ f \rightsquigarrow f(Y\ f).$$

To preserve typing and typing stability (the subject reduction theorem), Y must have a function type of the form $\theta \to \psi$ for some θ and ψ. According to the right hand side, the argument must be a function, so we must have $\theta = \sigma \to \sigma'$. Last, to ensure the type of $Y\ f$ is the same as the type of $f\ (Y\ f)$ it is necessary $\sigma' = \psi$ and then $\sigma = \psi$. Altogether Y must have the type $(t \to t) \to t$, for any type t.

For instance, when working in the following context:
$\Gamma, \mathit{plus}, \mathit{sub}, \mathit{mult} : \texttt{int} \to \texttt{int} \to \texttt{int}, \mathit{le} : \texttt{int} \to \texttt{int} \to \texttt{bool}$,

$\mathit{if} : \texttt{bool} \to \texttt{int} \to \texttt{int} \to \texttt{int}$

We can define the factorial function, in a well-typed manner, by using the same approach as in pure λ-calculus. We first devise a functional $factF$ of type

$$(factF : (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$$

and defined in the following manner:

$$factF \equiv \lambda f : \texttt{int} \to \texttt{int}.\lambda x : \texttt{int}.\ \texttt{if}(\texttt{le}\ x\ 0)\ 1\ (\texttt{mult}\ x\ (f\ (\texttt{sub}\ x\ 1)))$$

The factorial function is then the following value:

$$fact \equiv Y\, factF.$$

The operator Y makes it possible to bring general recursion back in the language, but it also restores the posibility for programs to loop for ever. This approach with a fix-point operator can model what happens in most typed functional programming languages, including `ML`, `OCaml`, and `Haskell`.

2.6 Recursive types and structural recursion

Another approach to introduce recursion, but which preserves the property that computations always terminate is to introduce data structures representing trees of finite height

and recursive functions that compute on these trees by authorizing recursive calls only on immediate sub-trees of the argument at each step of recursion. In other words, we introduce at the same time the recursive data-type and a function that describes recursive computation only for this type. The constraints that recursive calls are restricted to immediate sub-trees can be expressed using types.

As a first illustration, we study an example drawn from Gödel's system T, as presented in [7]. The type of natural numbers can be described as a data structure named `nat` with three constants `O:nat`, `S:nat` $\to$ `nat`, and `rec_nat`. The first two constants are used to represent numbers as data. The constant `O` represents 0 and when `x` represents the number n, then `S x` represents $n+1$. Thus, in the extended programming language, there are numbers, but 3 is never written, one write `S (S (S O))` instead.

The constant `rec_nat` is used to describe all kinds of functions of type `nat` $\to t$ for an arbitrary type t. Instead of having just one type, it has any type of the following shape[3]:

$$\texttt{rec\_nat} : t \to (\texttt{nat} \to t \to t) \to \texttt{nat} \to t.$$

The constant `rec_nat` must also come with the following reduction rules:

$$\texttt{rec\_nat} : t \to (\texttt{nat} \to t \to t) \to \texttt{nat} \to t.$$

The constant `rec_nat` makes it possible to construct the usual functions of arithmetic, while preserving the property that computations always terminate. The expression `rec_nat` $v\ f\ x$ which appears on the right-hand side of the second reduction rule corresponds to the only authorized recursive call, and this recursive call happens on a sub-term of the argument on the left-hand side of the rule, `S` x. Thus, when a function defined with the help of `rec_nat` computes on a number n, it can proceed recursively only on $n-1$, then $n-2$, et this necessarily stops when reach 0. Moreover, there is a guarantee that every normal form of type `nat` in the empty context is solely composed of a finite amount of applications of `S`, ultimately to `O`.

For instance, addition can be represented by the following function:

$$\lambda x\ y.\ \texttt{rec\_nat}\ y\ (\lambda p\ r.\texttt{S}\ r)\ x.$$

Indeed, when adding 0 to y, the result is y and if one adds `S` p to y, the result must be `S` $(p+y)$.

The constants `S` and `O` are called the *constructors* of this type `nat` and we shall call `rec_nat` the *recursor* associated to this type.

We can construct other types with recursion. Each time, it is enough to provide a collection of constructors, which are always functions whose result type is the data type being defined, or constants in this type. The arguments of the constructors can be in any existing type or in the type that is being defined (and in this case, the type exhibits recursion). The recursor is a function with $n+1$ arguments when the type has n constructors. The n first arguments correspond to a case-by-case analysis of values in this type. The first argument explains what should happen when the argument to the recursive function is obtained with the first constructor. In each case, if the constructor

[3]We already experimented with this kind schematic type when observing the constant `Y` in a previous section.

has k arguments, among which l are in the recursive type itself, the function given for this case is a function with $k + l$ arguments. For instance, the second constructor of `nat` has one argument, among which one is in `nat`. The expected type for the second argument of `rec_nat` is the type for a function with two arguments. In each case, the l extra arguments correspond to the values of recursive calls on the sub-terms in the recursive type. In the case of nat, this is illustrated by the right-hand side of the second reduction rule:

$$\texttt{rec\_nat}\ v_0\ f\ (\texttt{S}\ n) \rightsquigarrow f\ n\ (\texttt{rec\_nat}\ v_0\ f\ n)$$

In this reduction rule ($\texttt{rec\_nat}\ v\ 0\ (f\ n)$) represents the recursive call.

For another illustration, let's consider a type of binary trees, which we call `bin`, with two constructors `leaf` and `node`, which have the following types:

1. $\texttt{leaf} : \texttt{bin}$,
2. $\texttt{node} : \texttt{nat} \to \texttt{bin} \to \texttt{bin} \to \texttt{bin}$.

Since this types has two constructors, the recursor `rec_bin` has three arguments. The first one is a constant like `leaf`, le second one is a function with 5 arguments, because `node` has three arguments, among which 2 are in the type `bin`. The two extra arguments correspond to results of recursive calls on the sub-terms of type `bin`. The type of `rec_bin` is as follows:

$$\texttt{rec\_bin} : t \to (\texttt{nat} \to \texttt{bin} \to t \to \texttt{bin} \to t \to t) \to \texttt{bin} \to t$$

and the reductions rules are as follows:

$$\texttt{rec\_bin}\ v\ f\ \texttt{leaf} \rightsquigarrow v$$

$$\texttt{rec\_bin}\ v\ f\ (\texttt{node}\ n\ t_1\ t_2) \rightsquigarrow f\ n\ t_1\ (\texttt{rec\_bin}\ v\ f\ t_1)\ t_2\ (\texttt{rec\_bin}\ v\ f\ t_2)$$

Intuitively, this describes recursive functions where recursive calls are only allowed on immediate sub-trees of a binary tree. These sub-trees are necessarily smaller than the initial tree, this ensures that the computations will always terminate. This approach of recursion where recursive calls are only allowed on direct subterms is called *structural recursion.*

For instance, we can describe the function that adds all the numbers in a binary tree in the following manner:

$$\texttt{rec\_bin}\ 0\ (\lambda n : \texttt{int}, t_1 : \texttt{bin}, v_1 : \texttt{int}, t_2 : \texttt{bin}, v_2 : \texttt{int}.(\mathit{plus}\ n(\mathit{plus}\ v_1\ v_2)))$$

There is a correspondence between the recursive functions that can be defined using the recursor associated to a recursive type and the recursive functions that can be defined using OCaml or Haskell thanks to the pattern matching capabilities. For instance, `bin` would be defined in OCaml in the following manner:

```
type bin = Leaf | Node of nat*bin*bin
```

A function g defined as $g = \texttt{rec\_bin}\ v\ f$ would be defined in OCaml in the following manner:

```
let rec g x = match x with
   Leaf -> v
 | Node n t1 t2 -> f n t1 (g t1) t2 (g t2)
```

3 Type inference

3.1 Untyped abstraction

Strongly typed programming gives trongs guarantees that programs behave as intended, but the necessity to provide types for all bound variables is a heavy borden. To ease this burden, we can extend the language with an untyped abstraction construct $\lambda x.\ e$ and give it the following typing rule: *for every types* t *and* t' *if* e *has the type* t' *in the context* $\Gamma, x : t$ *then* $\lambda x.\ e$ *has the type* $t \to t'$. In fact, we generalize for abstraction a facility that we already exploited for the pair projectors *fst* and *snd*, for the fixpoint operator Y, and for the recursors associated to recursive types.

When considering an expression containing several untyped abstractions, it is necessary to choose for each of these abstractions a type that makes the whole expression well-typed. A solution is to rely on a unification algorihm, after having given to each bound variable a variable type. The program that checks whether an expression is well-typed should replace the operation of checking whether two types are equal by the operation of adding an equality constraint in the set of constraints maintained by the unification algorithm. Of course, one should also give a variable type to all occurrences of polymorphic constants. For instance, every occurrence of *fst* is given a type $T_1 * T_2 \to T_1$, whereT_1 and T_2 are fresh variables. The same is done with *snd* and Y if this constant is part of the language. Then, the type verification is performed as usual, except that each the verification of each function application imposes a constraint, because it is necessary to verify that the input type of the function and the type of the argument are equal.

To illustrate this, we consider the expression $\lambda x.plus\ x\ x$. The first step is to add a type variable for the type of x and we actually verify that the expression $\lambda x : T, plus\ x\ x$ is well-typed in the context where *plus* has the type $\texttt{int} \to \texttt{int} \to \texttt{int}$.

1. the algorithm starts with an empty list of constraints,
2. *plus* has the type $\texttt{int} \to \texttt{int} \to \texttt{int}$.
3. x has the T in the context $\cdots .x : T$,
4. the aplication *plus* x is well typed if $\texttt{int} = T$, this constraint is added to the set of constraints. Moreover, *plus* x has the type $\texttt{int} \to \texttt{int}$,
5. the application *plus* $x\ x \equiv (plus\ x)\ x$ is well type if x if the constraint $\texttt{int} = T$ is satisfied. Moreover, this expression has the type $\texttt{int}$. Altogether, we have the set of constraints $\texttt{int} = T, \texttt{int} = T$. The solution of this set of constraints is obvious,
6. the expression $\lambda x.\ plus\ x\ x$ has the type $\texttt{int} \to \texttt{int}$.

The solution of all constraints relies on a unification algorithm, already known in computer-based proofs. For instance, unification algorithms are used in the definition of Prolog.

3.2 Polymorphic typing

With a value of type t and a function of type $t \to t$, it is sometimes needed to apply this function twice on this value, something that can be represented by the following λ-term:

$$\lambda v\ f.\ f(f\ v)$$

This function always work in the same manner, whatever the type t. It would be useful to be able to apply this function in two different places, one where the type t would instantiate on `int`, for instance, and one where it would instantiate on `bool`. Here is an illustration of an expression where this would happen:

$\lambda c.(\lambda g.\lambda b : \texttt{bool}, f_1 : \texttt{bool} \to \texttt{bool}, n : \texttt{int}, f_2 : \texttt{int} \to \texttt{int}.c\ (g\ b\ f_1)\ (g\ n\ f_2))$
$\lambda v\ f.f(f\ v)$.

This expression is not well typed, because one of the uses of g imposes that the type of v should be `bool`, while the other use imposes that the type of v should be `int`. Here, strict typing appears to impose code duplication.

To solve this problem, we shall generalize the solution we already used for *fst*, *snd*, or Y. Instead of expressing that the function $\lambda v\ f.\ f(f\ v)$ should have a unique type $T \to (T \to T) \to T$ for a given T, we want to express that this function has the type $t \to (t \to t) \to t$ for any type t. Thus, some functions can have a type that is universally quantified. Such a type is called a *polymorphic type*.

We will introduce a new language construct, which is used explicitely to introduce a sub-term that has a polymorphic type. This new construct has the following syntax:

$$\mathsf{let}\ x = e\ \mathsf{in}\ e'$$

Operationally, the meaning of this expression is the meaning of a redex:

$$\mathsf{let}\ x = e\ \mathsf{in}\ e' \sim (\lambda x.\ e')e$$

However, for typing purposes, polymorphic typing can be expressed with the following rule:

$$\frac{\Gamma \vdash e : \theta \quad \Gamma \vdash e'[e/x] : t}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : t}$$

To verify that an expression $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ is well typed in a context Γ, this rule says that one must verify if e is well-typed in this context, and then one should verify if each of the uses of e is well-typed in the context provided by e', each instance independently from the others. This typing rule performs the code duplication for its own typing purposes, thus avoiding that programmers need to do it.

This typing algorithm is not very efficient, because it relies on a substitution that may imply an excessive increase in the size of the term whose types should be verified. In [6], there is a more efficient algorithm, which is also detailed in G. Dowek's course[4].

4 Dependent types

Type theory becomes really complex when one adds dependent types, which make it possible to consider type families, indexed by data. With this extension of typed λ-calculus, it will be possible to express that some functions can be applied only on those

[4] `https://who.rocq.inria.fr/Gilles.Dowek/Cours/Tlp/`

arguments that satisfy certain properties. For instance, the function that fetches the element of rank n of a list can be designed in such a way that it can only be used for functions whose length is larger than n. Type theories with dependent types exist since the 1970s [7, 10] and may have been experimented with even earlier in the work of De Bruijn.

4.1 More syntax

The first stage to introduce dependent types is to make it possible to define functions that take values as arguments and return types. For these functions, the output type is a *type of type*. We shall give a different name to these kinds of types, they will be called *sorts*: a sort will be a type whose elements are types. In these course notes, we will suppose that there exists a sort `Type`. We will describe a type family indexed by elements in a type A by a function of type $A \to \texttt{Type}$.

When $f : A \to \texttt{Type}$ is a type family, we may want to consider functions that take as input elements x in A and produce as outputs elements of the type $f\ x$ for each x. The notation based on arrows is not adapted for this need: we need to give a name to the argument, even when simply describing the type of this function, because we need this name to express what is the output type. Researchers introduced a new notation. A popular choice is the notation of indexed product: $\Pi x\ A.\ f\ x$.

This notation can be explained intuitively: the cartesian product $A_1 \times A_2$ of two types contains pairs where each provides one value of type A_1 and one value of type A_2. The two values have different types. But a pair can also be viewed as a function with inputs in $\{1, 2\}$: when the input is 1 the output is the first component of type A_1, when the input is 2 the output is the second component of type A_2. The notation of indexed product generalizes in a simple manner this view of cartesian products. A function of type $\Pi x : A.\ f\ x$ makes it possible to obtain values of type $f\ a$ forall possible ways of choosing a in A, in the same manner as a cartesian product indexed by A.

4.2 Extended typing rules

When there are dependent types, the rules for typing λ-abstractions and applications must be modified to take into account the fact that a function may return a value whose type depends on the argument. The new typing rules take the following form:

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

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

In fact, the notation $A \to B$ can still be used when the type is not really dependent, i.e., as a shorthand for the type $\Pi x : A.\ B$ when x does not appear in B.

4.3 Logical point of view

In this section, we consider only a type theory that enjoys the property of terminating computations (possibly with recursive types and associated recursors, but without a general fixpoint operator like Y).

When using dependent types for logical purposes, it is practical to introduce a second sort, which will be used explicitely to describe logical formulas. In the Coq system, this second sort is called `Prop`.

In the first section, we saw that type variables correspond to propositional variables. When propositions are indexed by elements of type `A`, they become predicates on this type. For instance, let's consider the predicate $P : A \to$ `Prop`: for two different values x and y of type A, we shall have two distinct types $P\ x$ and $P\ y$, where one may be inhabited while the other is not, which means that we have two logical formulas, one of which is provable and the other is not.

To find a meaning to the new construction of dependent product, we should remember that all functions that we consider terminate. The following property is then guaranteed: if f has the type $\Pi x : A.\ P\ x$ then for every element x of A, the function f will produce an element $f\ x$ which is in type $P\ x$. In other words, $P\ x$ is always inhabited, always provable. Thus, the dependent product can be read as universal quantification. In what follows, I will often use the notation $\forall x : A,\ P\ x$ instead of $\Pi x : A,\ P\ x$ when P has the type $A \to$ `Prop`.

Expressions whose type is a logical formula are theorems. The typed calculus makes it possible to compose these expressions to obtain new theorems.

For instance, in some context we may have a predicate `even` and two constants `even0` and `even2` that have the following types:

$$\begin{array}{lcl} \texttt{even0} & : & \texttt{even O} \\ \texttt{even2} & : & \forall x : \texttt{nat.even}\ x \to \texttt{even(S (S}\ x\texttt{))} \end{array}$$

The term `even2 (S (S O)) (even2 O even0)` is a proof of the proposition

$$\texttt{even (S (S (S (S O))))}.$$

In other words it is a proof that 4 is even.

4.4 Dependent types and recursive types

Recursive types can also be defined in a manner that produces type families. For instance, data lists can be parameterized by the type of the elements. In this case, one obtains a type family represented by a function `list : Type` $\to$ `Type`. To construct an element in one of the types of this family, one may choose to construct an empty list or to add an element to an existing list. Even for an empty list, it is necessary to choose the type of elements to know in which type of lists this empty list will exist. Therefore, the constructor for empty lists is not a constant but a function that takes a type A as argument and returns an element in type `list` A:

$$\texttt{nil} : \forall A : \texttt{Type.list}\ A$$

To add an element in an existing list, it is necessary to know what is the type A, take an element in A, and take a list whose elements are in A. The constructor has a type with the following form:

$$\texttt{cons} : \forall A : \texttt{Type.}A \to \texttt{list}\ A \to \texttt{list}\ A$$

If one wants to define a recursive function with a dependent type $\Pi x : \texttt{nat}, A\ n$ with the recursor `rec_nat`, we must adapt the type of this recursor for this need. It is necessary that the value for `O` is in the type A `O`. and the function that computes the value for `S` x must return a value in A (`S` x) while potentially using a value for the recursive call that should be in type $A\ x$. All this is expressed by assigning the following type to the constant `rec_nat`:

$$\texttt{rec\_nat} : \Pi A : \texttt{nat} \to \texttt{Type}.A\ \texttt{O} \to (\Pi n : \texttt{nat}.A\ n \to A\ (\texttt{S}\ n)) \to \Pi n : nat.A\ n$$

When reading the type of `rec_nat` as a logical formula, we discover that this type is a well-known logical formula. It is the induction principle that we can use to prove formulas about natural numbers. Replacing A by a predicate P, this reads as: *forall predicate P, if P is satisfied in 0, and for every n P n implies P $(n+1)$, then P is satisfied for all natural numbers:*

$$\forall P : nat \to \texttt{Prop}.P\ \texttt{O} \to (\forall x : \texttt{nat}.P\ n \to P\ (\texttt{S}\ n)) \to \forall x : \texttt{nat}.P\ n$$

There are many other interactions between dependent types and recursive types. In particular, dependent types make it possible to extend the notion of recursive function beyond structural recursion, while maintaining the important property that all computations terminate. A good way to study these interactions is to experiment with systems based on type theory, like the Coq system [5], or Agda [11].

5 Further reading

These notes only give an overview of what can be done with type theory. There are several books that provide a deeper study. Pure λ-calculus is studied intensively in [1], which is the major reference on the topic. Several programming languages are derived from λ-calculus [3, 16, 15]. Many proof systems use λ-calculus and types in various form [8, 14, 4, 12]. One of the most advanced systems based on type theory is the Coq system, for which we wrote a book [2].

References

[1] Henk Barendregt. The Lambda Calculus, Its Syntax and Semantics. *Studies in Logic. North-Holland, 1984.*

[2] Yves Bertot and Pierre Castéran. Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions. *Springer-Verlag, 2004.*

[3] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. Développement d'applications avec Objective Caml. *O'Reilly and associates, 2000.*

[4] Robert Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harber, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. Implementing mathematics with the Nuprl proof development system. *Prentice-Hall, 1986.*

[5] *Coq development team.* The Coq Proof Assistant Reference Manual, version 8.4, *2012.* `http://coq.inria.fr`.

[6] *Luís Damas and Robin Milner. Principal type-schemes for functional programs. In* ninth ACM symposium on Principles of Programming Languages, *pages 207–212. ACM, 1982.*

[7] *Jean-Yves Girard, Yves Lafont, and Paul Taylor.* Proofs and types. *Cambridge University Press, 1989.*

[8] *Michael J. C. Gordon and Thomas F. Melham.* Introduction to HOL : a theorem proving environment for higher-order logic. *Cambridge University Press, 1993.*

[9] *Jean-Louis Krivine.* Lambda Calcul, types et modèles. *Masson, 1990.*

[10] *Per Martin-Löf.* Intuitionistic type theories. *Bibliopolis, 1984.*

[11] *Ulf Norell. Dependently typed programming in agda. In* Revised Lectures from 6th Int. School on Advanced Functional Programming, AFP 2008, *volume 5832 of* Lect. Notes in Comput. Sci. *Springer, 2009.*

[12] *Lawrence C. Paulson.* Logic and computation, Interactive proof with Cambridge LCF. *Cambridge University Press, 1987.*

[13] *Lawrence C. Paulson.* ML for the working programmer. *Cambridge University Press, 1991.*

[14] *Lawrence C. Paulson and Tobias Nipkow.* Isabelle : a generic theorem prover, *volume 828 of* Lecture Notes in Computer Science. *Springer-Verlag, 1994.*

[15] *Simon Thompson.* Haskell, the craft of functional programming. *Addison-Wesley, 1996.*

[16] *Pierre Weis and Xavier Leroy.* Le Langage Caml. *Dunod, 1999.*