Abstract Datatypes and Definite description

Frédéric Ruyer
Laboratoire de Mathématique, Université de Savoie, F-73376 Le Bourget-du-Lac cedex
frederic.ruyer@univ-savoie.fr *

Abstract

We present a programming language and its type system containing ML-like modules with abstract datatypes as first-class values, and a proof of type soundness. The result is based on a translation into an extension of Raffalli's system ST, a type system with subtyping that separates computational and logical parts of proofs. The great expressive power of ST, together with its realizability semantics, allow us to give an interpretation of abstract datatypes in terms of union, and to express the meaning of the "dot notation" by using the axiom of choice.

1 Introduction

The need for large developments has motivated the definition of a module language on the top of ML's family core languages (SML [MTH], Caml [OC] [CMP00],...). The idea is to cut a large program in small independent pieces called *modules*, making the whole structure easier to understand, modify, and allowing to reuse one piece in another structure. In ML's syntax a module (resp. a signature) is a collection of type declarations and value definitions (resp. types) enclosed between `struct` (resp. `sig`) and `end`. Module languages have two other functionalities: *functors* allow to define modules parametrized by other modules, and *abstract data types* (ADTs) can be given in a signature, allowing to hide some type definitions (some examples are given in Figure 1).

*The author thanks CSL's anonymous referees, and his director C. Raffalli for their numerous remarks that allowed to greatly improve the paper.

Module languages being separated from core languages, they can't be handled as arguments or results of ordinary functions. It is nevertheless desirable, for example to choose at run-time the most efficient representation of a data type (e.g. small or big hash-tables, ...). Various approaches have been proposed ([MMM91],[MP88],[DCH03],[Rus98]...) to solve this problem. But, as far as the author knows, they come with no proof of soundness.

We present in this paper a typed programming language that manipulates ADTs, and has the desired property of type soundness, in the sense that every well-typed program is an inhabitant of its type. In our approach, signatures *are* ordinary types, modules *are* records and functors *are* functions, the whole being included in the core language, with a very simple presentation (Figure 2 gives some examples of our syntax corresponding to those presented in Figure 1); section 2 gives the formal presentation of the language. We also add a feature called "program abstraction" that aims to mimic the reference to *self* in a recursive record.

Our work lies upon a very intuitive idea: if a program `p` has the abstract type `{type t; T(t)}`, it means that there exists a certain type `t` such that `p` has type `{T(t)}`; therefore, we can *choose* one of them, written almost as usual `(p:T(t)).t`; thus, the language of types must be able to "talk about" terms (`p.t` being a type containing the term `p`), and possess a form of the axiom of choice (AC). Moreover, many features of the module languages (ability of forgetting a field, behaviour of functors...) need subtyping.

ST type system ([Raf03a],[Raf03b]) is a type system with subtyping that makes a clear separation between computationally relevant and irrel-

2 modules having the same signature:

```
module Ord1=                    module Ord2=
struct                          struct
   type point=int                  type point=int
   let less x y = x<y              let less x y = ((x mod y)=0)
end                             end
              signature ORD=
              sig
                 type point=int
                 val less x y : point->point->bool
              end
```

Another signature and functor definition:

```
signature INTER=                    functor Inter(P:ORD):INTER=
sig                                 struct
   type point=int                      type point=int
   type interval=int*int               type interval=int*int
   val make:point->point->interval     let make x y= if (P.less x y)
end                                    then (x,y) else (y,x)
                                    end
```

Functor application :

```
I1 = Inter(Ord1) and I2 = Inter(Ord2)
```

A signature with type abstraction,
and a new module having this signature
(as well as Ord1 and Ord2):

```
signature ABTR_ORD=                 module Ord3=
sig                                 struct
   type point                          type point=bool
   val less : point->point->bool       let less x y = (not x) or y
end                                 end
```

Another abstract signature and functor:

```
signature                           functor
ABSTR_INTER=                        Interval(P:ABSTR_ORD):ABSTR_INTER=
sig                                 struct
   type point                          type point=P.point
   type interval                       type interval=P.point*P.point
   val make:point->point->interval     let make x y= if (P.less x y)
end                                    then (x,y) else (y,x)
                                    end
```

A module containing another module with
abstract type and its signature

```
module Prog=                    signature SigProg=
struct                          sig
   signature Sig =                 signature Sig =
    sig                             sig
     type set                        type set
     val e:set                       val e:set
    end                             end
   module A:Sig =                  module A:Sig
    struct
     type set=int
     val e=0
    end
     let neutral = A.e              val neutral:A.set
end                             end
```

Figure 1: Examples of ML's syntax for modules, functors and abstract type

2 records having the same type:

```
Ord1    =   {less = fun x -> fun y -> x<y}
Ord2    =   {less = fun x -> fun y -> ((x mod y)=0)}
ORD     =   {type point=int; less: point->point->bool}
```

A function definition and its type:

```
Inter=fun P ->
       {make = fun x -> fun y -> if (P.less x y) then (x,y) else (y,x)}
INTER={type point=int;type interval=int*int;make:point->point->interval}
Inter:∀P:ORD.INTER
```

Function application :

```
I1 = Interval(Ord1) and I2 = Interval(Ord2)
```

A type with type abstraction, and a new record having this type
(as well as Ord1 and Ord2):

```
ABSTR_ORD   =   {type point ; less:point->point->bool}
Ord3        =   {less = fun x -> fun y -> (not x) or y)}
```

Another abstract type and function:

```
ABSTR_INTER={type point;type interval; make:point->point->interval}
ABSTR_Inter=fun P ->
 {make = fun x -> fun y ->if (P.less x y) then (x,y) else (y,x)}
ABSTR_Inter:∀P:ABSTR_ORD.ABSTR_INTER
```

A record containing another record with
abstract type and its type

```
Prog      =   {p ; A = { e = 0 } ; neutral = p.A })
SigProg   =   {self s ; type sig ; A:sig ;
               neutral:(s.A:sig).set with sig={type set; e:set} }
```

Figure 2: Examples of our syntax for records, functions and abstract type

evant parts of a proof. This system allows to have a representation of terms in types; we show in this paper that ST+AC is consistent (theorem 1, section 3). So, it seemed that we had all the properties needed to handle ADTs. What was left to do was to translate the usual datatypes in this system (we use a new translation of product types, based on sum types and a kind of $\neg\neg$-translation, following an idea of C.Raffalli), and to "compile" the language into pure λ-calculus. Having checked the correctness of the translation (theorem 5, section 4), we get easily that our programming language is sound (corollary 2, section 4).

The paper is structured as follows: in section 2 we give the presentation of our language, section 3 is devoted to the ST type system, section 4 explains the translation; we make then a comparison with other works in section 5, and conclude in section 6.

Two appendices are added: appendix A contains ST's additional rules, and appendix B is the proof of theorem 5.

2 A typed programming language

2.1 Overview

We give here some rules of our system, illustrated by some examples. Although they aren't present in the basic language, we use in this section usual datatypes (`int,bool,int*int,...`) and operators (`+,*,sqrt,(.,.),...`) for an easier understanding.

The judgments are of two kinds: *Typing judgments* and *Subtyping judgments* indicated respectively by "$\vdash$" and "$\vdash_<$". "$\vdash$ `u:A`" means "the program `u` has type `A`" and "$\vdash_<$ `A<B`" means "every program of type `A` is of type `B`". Therefore, we have the following rule:

$$\frac{\Gamma \vdash \texttt{u:A} \qquad \vdash_< \texttt{A<B}}{\Gamma \vdash \texttt{u:B}}$$

Abstraction and *Application* rules bind the name of the argument in the result type, e.g.:

$$\frac{\texttt{x: int} \vdash \texttt{sqrt(x*x):float}}{\vdash \texttt{fun x->sqrt(x*x):}\forall\texttt{x:int.float.}}$$

and

$$\frac{\vdash \texttt{u:}\forall\texttt{x:\{type set;e:set\}.x.set} \quad \vdash \texttt{v:\{type set; e:set\}}}{\vdash \texttt{(uv):(v:\{type set; e:set\}).set}}$$

Records, *Records types* and *Abstract Types* have been introduced in section 1. To get a supertype of a record type, you can either take a supertype of a field, forget a field or abstract a type:

$$\frac{\vdash_< \texttt{string[10] < string}}{\vdash_< \begin{array}{c}\texttt{\{Name:string[10] ; Age:int ; Town:string\}} \\ < \\ \texttt{\{Name:string ; Age:int\}}\end{array}}$$

and

$$\frac{}{\vdash_< \begin{array}{c}\texttt{\{Name:string ; Age:int\}} \\ < \\ \texttt{\{type age ; Name:string ; Age:age\}}\end{array}}$$

(with `string[10]` the type of strings of length 10).

Sum types are union of types distinguished by constructors:

```
number = Arabic of int | Roman of string
```

can be a type for numbers, and we have:

$$\frac{\vdash \texttt{2005 : int}}{\vdash \texttt{Arabic 2005 :number}} \quad \text{and} \quad \frac{\vdash \texttt{"MMIV" : string}}{\vdash \texttt{Roman "MMIV":number}}$$

To handle values of sum types, we use a very simple pattern matching, which is case analysis:

$$\frac{\vdash \texttt{u:Circle of float | Square of float} \quad \ldots}{\vdash \texttt{cases u of Circle r->2*pi*r | Square a->4*a :float}}$$

Subtyping sum types is more intuitive than subtyping records types:

$$\frac{\vdash_< \texttt{int < float}}{\vdash_< \texttt{Circle of int < Circle of float|Square of float}}$$

Recursive Types are limited to positive type operators (the formal definition of positivity will be given in section 2.3). We can handle inductive types, written with a μ (we could also handle coinductive types, but we don't add them for sake of simplicity).
Take the paradigmatic example of naturals:

$$\texttt{nat=}\mu\texttt{a.(0 | S of a)}$$

We get from it the formula:

$$\frac{}{\vdash_< \texttt{nat = 0|S of nat}}$$

We can then define, for example, addition:

$$\frac{}{\vdash \begin{array}{c}\texttt{rec(fun f->fun x->fun y->cases x of 0->y|S n->S(f n y))} \\ : \\ \forall\,\texttt{x:nat}\;\forall\,\texttt{y:nat.nat}\end{array}}$$

2.2 Types, formulae and programs

We have chosen ML-like notations; the grammar for types ($\mathcal{T}$), sum types ($\mathcal{T}_+$), product types ($\mathcal{T}_\times$), pre-product types ($\mathcal{P}_\times$), abstract declarations ($\mathcal{A}$), bodies ($\mathcal{B}$), endings ($\mathcal{E}$), formulae ($\mathcal{F}$), programs ($\mathcal{P}$), cases ($\mathcal{C}$) , and records ($\mathcal{R}$) is defined by:

$$
\begin{array}{lcl}
\mathcal{T} & = & \mathcal{V}_\mathcal{T} \mid \mathcal{V}_{\mathcal{T}name} \mid (\mathcal{P}:\mathcal{T}).\mathcal{V}_{\mathcal{T}name} \mid \forall\mathcal{V}_\mathcal{P}{:}\mathcal{T}.\mathcal{T} \mid \\
 & & \forall\mathcal{V}_\mathcal{T}.\mathcal{T} \mid \mu\mathcal{V}_\mathcal{T}.\mathcal{T} \mid \mathcal{T}_\times \mid \mathcal{T}_+ \\
\mathcal{T}_+ & = & \mathcal{V}_\mathtt{C} \mid \mathcal{V}_\mathtt{C}\ \mathtt{of}\ \mathcal{T} \mid (\mathcal{T}_+ \mid \mathcal{T}_+) \\
\mathcal{T}_\times & = & \{\ \mathcal{P}_\times\ \} \mid \{\ \mathtt{self}\ \mathcal{V}_{\mathcal{P}name}{;}\mathcal{P}_\times\ \} \\
\mathcal{P}_\times & = & \mathcal{E} \mid \mathcal{A};\mathcal{E} \\
\mathcal{A} & = & \mathtt{type}\ \mathcal{V}_{\mathcal{T}name} \mid \mathcal{A};\mathcal{A} \\
\mathcal{B} & = & \mathcal{V}_\mathtt{C}\ :\ \mathcal{T} \mid \mathcal{B};\mathcal{B} \\
\mathcal{E} & = & \mathcal{B} \mid \mathcal{B}\ \mathtt{with}\ \mathcal{F} \\
\mathcal{F} & = & \mathcal{T}{<}\mathcal{T} \mid \mathcal{V}_{Pred_n}(\mathcal{T}^n) \mid (\mathcal{F}{=>}\mathcal{F}) \mid \forall\mathcal{V}_\mathcal{T}.\mathcal{F} \mid \forall_n\mathcal{V}_{Pred_n}.\mathcal{F} \\
\mathcal{P} & = & \mathcal{V}_\mathcal{P} \mid \mathcal{V}_{\mathcal{P}name} \mid \mathtt{fun}\ \mathcal{V}_\mathcal{P}\ \mathtt{->}\ \mathcal{P} \mid (\mathcal{P}\mathcal{P}) \mid \mathtt{rec}\ \mathcal{P} \mid \\
 & & \mathcal{V}_\mathtt{C} \mid \mathcal{V}_\mathtt{C}\mathcal{P} \mid \mathtt{cases}\ \mathcal{P}\ \mathtt{of}\ \mathcal{C} \mid \{\mathcal{R}\} \mid \mathcal{P}.\mathcal{V}_\mathtt{C} \\
\mathcal{C} & = & \mathcal{V}_\mathtt{C}\ \mathtt{->}\ \mathcal{P} \mid \mathcal{V}_\mathtt{C}\mathcal{V}_\mathcal{P}\ \mathtt{->}\mathcal{P} \mid \mathcal{C}|\mathcal{C} \\
\mathcal{R} & = & \mathcal{V}_\mathtt{C}\ \mathtt{=}\ \mathcal{P} \mid \mathcal{R};\mathcal{R}
\end{array}
$$

There are four kinds of variables: *Type variables* ($\mathcal{V}_\mathcal{T}$) will be written `a`, `b`, `c`,...; *Type Name variables* ($\mathcal{V}_{\mathcal{T}name}$) will be written *id* in rules, and with a name `point`, `set`, `sig`...in examples; *Program variables* ($\mathcal{V}_\mathcal{P}$) will be written `x`, `y`, `z`,...; *Program Name variables* ($\mathcal{V}_{\mathcal{P}name}$) will be written *s* (*s* stands for *self*) in rules, and with a name `s`...in examples; *n*-ary *Predicate variables* ($\mathcal{V}_{Pred_n}$) will be written $\mathtt{X}_n$, $\mathtt{Y}_n$, $\mathtt{Z}_n$,...; *Constructor names* ($\mathcal{V}_\mathtt{C}$) will be written `C` or $\mathtt{C}_i$ in rules, and with a name `less`, `Age`,...or with a symbol `+`,...in examples. *Constructor names* are never linked, and thus will simply be encoded by integers (see section 4.).
$\forall$, μ and `fun` are binders, as usual. `self` and `type` are binders too, thus we define the notions of free and bound (not free) occurrences of Type and Program names by:

Definition 1 (free names)

- *"id" (resp. "self") is free in "id" (resp. "self").*
- *free occurrences of "id" (resp. "self") in "T" are bound in "`type` id`;`T" (resp. "`self` self`;`T").*
- *"id" is bound in "`(p:T).id`" if "T=$\{P_1;\ldots;P_n\}$" and "P_i=`type` id" for some i.*
- *freeness and boundness are not modified by other patterns.*

Correct substitution of names is defined as usual to avoid name clashes.
A notion of legality of is necessary in order to be able to interpret correctly abstract types and programs:

Definition 2 (legal type) *A type is said legal if it contains no free names.*

Effectively, the meaning of the type {`type set;e:set`} is clear, but, `set` being a name, it seems not possible to give an interpretation of the type {`e:set`}.

Notation 1 *We add some syntactic sugar in the examples in order to make them more readable:*

{`type id=T;S`}	*stands for*	{`type id;S with id=T`}
∀`x:`{`type id;S`}`.B(x.id)`	*stands for*	∀`x:`{`type id;S`}`.B((x:`{`type id;S`}`).id)`
{`p ; P(p)`}	*stands for*	`rec (fun p -> `{`P(p)`}`)`
`T=T'`	*stands for*	`(T<T'`∧`T'<T)`*
`A -> B`	*stands for*	∀`x:A.B`**

**: `(A` ∧ `B)`≡ ∀$_0$$X_0$`((A=>B=>`$X_0$`)=>`$X_0$`)` the usual second order encoding of conjunction.*
***: when `x` is not free in `B`.*

2.3 Typing rules

Definition 3 (typing context and sequent) *A typing context* Γ *is a finite list of pairs of the form `x:A` with `x` a program variable, and `A` a legal type s.t. if `(p:T).id` occurs in `A`, then `p`'s free variables have been previously declared in* Γ*, each variable occurring at most once.*

A typing sequent has the form $\Gamma \vdash$ `p:T`*, with* Γ *a typing context,* `p` *a program, and* `T` *a type.*

Let's explain intuitively the two conditions for building a context. The notion of legality has already been discussed previously. For the second condition, the context

```
x:{ e:bool } , y:int -> ( x:{type set ; e:set} ).set
```

makes sense, because we can "interpret" the type of `y`, whereas

```
y:int -> ( x:{type set ; e:set} ).set
```

alone doesn't make sense.

The derivation rules for a typing sequent are given in figure 3.

$$\frac{}{\Gamma, \texttt{x:A} \vdash \texttt{x:A}}\ \text{Ax} \qquad \frac{\Gamma, \texttt{x:A} \vdash \texttt{p:T}}{\Gamma \vdash \texttt{fun x -> p:}\forall\texttt{x:A.T}}\ \text{Ab*}$$

$$\frac{\Gamma \vdash \texttt{p:}\forall\texttt{x:A.B} \qquad \Gamma \vdash \texttt{q:A}}{\Gamma \vdash \texttt{(pq): B[x:=q]}}\ \text{Ap}$$

$$\frac{\Gamma \vdash \texttt{u}_1\texttt{:A}_1 \ldots \Gamma \vdash \texttt{u}_n\texttt{:A}_n}{\Gamma \vdash \{\texttt{C}_1\texttt{= u}_1\texttt{;}\ldots\texttt{;C}_n\texttt{= u}_n\}\texttt{:}\{\texttt{C}_1\texttt{:A}_1\texttt{;}\ldots\texttt{;C}_n\texttt{:A}_n\}}\ \text{P-I}$$

$$\frac{\begin{array}{lll}\Gamma, \texttt{x:A}_1 & \vdash & \texttt{u}_1\texttt{:T}\ldots \\ \Gamma & \vdash & \texttt{u}_k\texttt{:T}\ldots \qquad \Gamma \vdash \texttt{p:C}_1\texttt{of A}_1 \mid \ldots \mid \texttt{C}_k \mid \ldots \mid \texttt{C}_n\texttt{of A}_n \\ \Gamma, \texttt{x:A}_n & \vdash & \texttt{u}_n\texttt{:T}\end{array}}{\Gamma \vdash \texttt{cases p of C}_1\ \texttt{x->u}_1 \mid \ldots \mid \texttt{C}_n\texttt{->u}_k \mid \ldots \mid \texttt{C}_n\ \texttt{x->u}_n\texttt{:T}}\ \text{S-E}$$

$$\frac{\Gamma \vdash \texttt{u:\{C:A\}}}{\Gamma \vdash \texttt{u.C:A}}\ \text{P-E} \qquad \frac{\Gamma \vdash \texttt{u:A}}{\Gamma \vdash \texttt{C u:C of A}}\ \text{S-I1} \qquad \frac{}{\Gamma \vdash \texttt{C : C}}\ \text{S-I2}$$

$$\frac{\Gamma \vdash \texttt{p:A}}{\Gamma \vdash \texttt{p:}\forall\texttt{a.A}}\ \forall\text{-I**} \qquad \frac{\Gamma \vdash \texttt{p:}\forall\texttt{a.A} \quad \texttt{T}\ \Gamma\text{-legal***}}{\Gamma \vdash \texttt{p:A[a:=T]}}\ \forall\text{-E}$$

$$\frac{\Gamma \vdash \texttt{p:\{type } id\texttt{; } \Theta(id)\texttt{ \}}}{\Gamma \vdash \texttt{p:}\{\Theta(\texttt{(p:\{type } id\texttt{; } \Theta(id)\texttt{ \}).}id)\}}\ \text{Reinforcement}$$

$$\frac{\Gamma \vdash \texttt{p:}\forall\texttt{a.((a->T)->(F(a)->T))} \qquad \vdash_{<} \texttt{Incr(F)}}{\Gamma \vdash \texttt{rec p:}\mu\texttt{a.F(a)->T}}\ \text{Recursion}$$

$$\frac{\begin{array}{lll}\Gamma & \vdash & \texttt{p}_1\texttt{:T}_1 \\ \Gamma, \texttt{x}_1\texttt{:T}_1 & \vdash & \texttt{f}_2\texttt{(x}_1\texttt{):T}_2\texttt{(x}_1\texttt{)} \\ \vdots & & \\ \Gamma, \texttt{x}_1\texttt{:T}_1, \ldots, \texttt{x}_i\texttt{:T}_i\texttt{(x}_1\texttt{,}\ldots\texttt{,x}_{i-1}\texttt{)} & & \\ & \vdash & \texttt{f}_{i+1}\texttt{(x}_1\texttt{,}\ldots\texttt{,x}_i\texttt{):T}_{i+1}\texttt{(x}_1\texttt{,}\ldots\texttt{,x}_i\texttt{)} \\ \vdots & & \\ \Gamma, \texttt{x}_1\texttt{:T}_1, \ldots, \texttt{x}_{n-1}\texttt{:T}_{n-1}\texttt{(x}_1\texttt{,}\ldots\texttt{,x}_{n-2}\texttt{)} & & \\ & \vdash & \texttt{f}_n\texttt{(x}_1\texttt{,}\ldots\texttt{,x}_{n-1}\texttt{):T}_n\texttt{(x}_1\texttt{,}\ldots\texttt{,x}_{n-1}\texttt{)}\end{array}}{\begin{array}{l}\Gamma \vdash \\ \texttt{rec (fun p -> \{C}_1\texttt{=p}_1\texttt{;C}_2\texttt{=f}_2\texttt{(p.C}_1\texttt{);}\ldots\texttt{;C}_n\texttt{=f}_n\texttt{(p.C}_1\texttt{,}\ldots\texttt{,p.C}_{n-1}\texttt{) \}):} \\ \texttt{\{self } s\texttt{;C}_1\texttt{:T}_1\texttt{;C}_2\texttt{:T}_2\texttt{(}s\texttt{.C}_1\texttt{);}\ldots\texttt{;C}_n\texttt{:T}_n\texttt{(}s\texttt{.C}_1\texttt{,}\ldots\texttt{,}s\texttt{.C}_{n-1}\texttt{) \}}\end{array}}\ \text{API}$$

$$\frac{\Gamma \vdash \texttt{p:\{self } s\texttt{;C}_1\texttt{:T}_1\texttt{;C}_2\texttt{:T}_2\texttt{(}s\texttt{.C}_1\texttt{);}\ldots\texttt{;C}_n\texttt{:T}_n\texttt{(}s\texttt{.C}_1\texttt{,}\ldots\texttt{,}s\texttt{.C}_{n-1}\texttt{) \}}}{\Gamma \vdash \texttt{p:\{C}_1\texttt{:T}_1\texttt{;C}_2\texttt{:T}_2\texttt{(p.C}_1\texttt{);}\ldots\texttt{;C}_n\texttt{:T}_n\texttt{(p.C}_1\texttt{,}\ldots\texttt{,p.C}_{n-1}\texttt{) \}}}\ \text{APE}$$

$$\frac{\Gamma \vdash \texttt{p:\{C}_1\texttt{:T}_1\texttt{;}\ldots\texttt{;C}_n\texttt{:T}_n\ \texttt{with F\}}}{\Gamma \vdash \texttt{p:\{C}_1\texttt{:T}_1\texttt{;}\ldots\texttt{;C}_n\texttt{:T}_n\texttt{\}}}\ \text{SpecE-Ty}$$

* : with x not free in Γ **: with a not free in Γ
***: cf. section 2.4

Figure 3: Typing rules

All the rules are as usual, excepted Ap (application), API (Abstract program introduction), APE (Abstract program elimination), and SpecE-Ty for which we give some explanations:

- The type of a function $\forall$`x:A.B` is a kind of *dependent product* i.e. a type that depends on terms; when `x` is not free in `B`, it can be seen as the usual arrow type `A->B`. Therefore, the rule Ap is very similar to the modus ponens.
- The hypotheses of the rule API simply express a condition of acyclicity of dependencies, ensuring the well-foundedness of the recursion. In the conclusion, we could add a syntactic sugar to replace the recursor and the record by a succession of `let` definition, like in a ML file:

foo.ml
`let` $\texttt{C}_1$`=...`
⋮
`let` $\texttt{C}_n$`=...`

  , the corresponding type being the "interface file" (`.mli` in OCaml) where the reference to $self$ have been omitted:

foo.mli
$\texttt{C}_1\texttt{:T}_1$
⋮
$\texttt{C}_n\texttt{:T}_n\texttt{(C}_1\ldots\texttt{C}_{n-1}\texttt{)}$

  .
- The rule APE is the sister of the Reinforcement rule, in the sense that it allows to remove an abstraction. Having in mind that the type `{self s; T(s)}` has the intuitive meaning "I am the type of a program whose type contains its own name" helps to understand the conclusion of the sequent.
- SpecE-Ty allows to remove a formula that has no algorithmic content from a type.

2.4 Logical rules

Logical contexts C will always depend upon a typing context Γ, which will provide the interpretation of C's formulae, and afterwards of any formulae that can be derived from C.

Definition 4 (Γ-legality) *Let Γ be a context. A type `A` is said Γ-legal if Γ, `x:A` is a context. Otherwise, `A` is said illegal. A formulae is said Γ-legal if it doesn't contain any Γ-illegal type.*

Definition 5 (logical context and sequent) *A Γ-logical context C is a finite set of Γ-legal*

formulae. A Γ-logical sequent has the form $C \vdash^{\Gamma}_{<} F$*, with* C *a* Γ*-logical context, and* F *a formula.*

The following definition is a formalization of "is increasing":

Definition 6 *If n>0, a n-ary predicate is a function from* $\mathcal{T}^n$ *to* $\mathcal{T}$*. If* F *is a unary predicate, we define the formula* `Incr(F)` *by:*

$$\texttt{Incr(F)} = \forall \texttt{a}.\forall \texttt{b}.(\ \texttt{a<b => F(a)<F(b)}\ )$$

The rules for deriving a Γ-subtyping sequent are given in figure 4. We suppose given a typing context Γ, and note in the rules $C \vdash_{<} \texttt{F}$ for "C is a Γ subtyping context and $C \vdash^{\Gamma}_{<} \texttt{F}$".
The last logical rules are the rules that make logical judgments and typing judgments interact together. The rule SpecI allows to make a specification entering a type and the rule SpecE-Fo is a kind of elimination of conjunction. The others are clear.

2.5 Reduction rules

The reduction rules correspond to weak head reduction:

- Redex : `( (fun x->u) v ) >> u[x:=v]`
- Recursor : `rec p >> (p (rec p))`
- Access to a field of a record $(1 \leq i \leq n)$:
  $\{\texttt{C}_1\texttt{=u}_1\ \texttt{;...;}\ \texttt{C}_n\texttt{=u}_n\}\texttt{.C}_i\ \texttt{>>}\ \texttt{u}_i$
- Cases $(1 \leq i \leq n)$:
  `cases` $\texttt{C}_k$ `of` $\texttt{C}_1\texttt{x->f}_1\texttt{|...|C}_k\texttt{->f}_k\texttt{|...|C}_n\ \texttt{x->f}_n$ `>>` $\texttt{f}_k$
  `cases (`$\texttt{C}_i$ `u) of` $\texttt{C}_1\texttt{x->f}_1\texttt{(x)|...|C}_n\texttt{x->f}_n\texttt{(x)}$ `>>` $\texttt{f}_i\texttt{(u)}$
- Left : if `u>>u'` then `(u v)>>(u' v)`.
- Left-case : if `u>>u'` then
  `cases u of` $\mathcal{C}$ `>> cases u' of` $\mathcal{C}$
- Left-field : if `u>>u'` then `u.C >> u'.C`

$$\frac{}{C, \texttt{F} \vdash_{<} \texttt{F}}\ \text{Ax} \qquad \frac{}{C \vdash_{<} \texttt{T < T}}\ \text{Rfl} \qquad \frac{C \vdash_{<} \texttt{R < S} \qquad C \vdash_{<} \texttt{S < T}}{C \vdash_{<} \texttt{R < T}}\ \text{Tr}$$

$$\frac{C, \texttt{F} \vdash_{<} \texttt{G}}{C \vdash_{<} \texttt{F => G}}\ \text{Imp} \qquad \frac{C \vdash_{<} \texttt{F => G} \qquad C \vdash_{<} \texttt{F}}{C \vdash_{<} \texttt{G}}\ \text{MP}$$

$$\frac{C \vdash_{<} \texttt{F(a)}}{C \vdash_{<} \forall \texttt{a.F(a)}}\ \forall\text{-I}^{*} \qquad \frac{C \vdash_{<} \forall \texttt{a.F(a)}}{C \vdash_{<} \texttt{F(T)}}\ \forall\text{-E}^{**}$$

$$\frac{C \vdash_{<} \texttt{F}(X_n)}{C \vdash_{<} \forall_n X_n.\texttt{F}(X_n)}\ \forall^2\text{-I}^{***} \qquad \frac{C \vdash_{<} \forall_n X_n.\texttt{F}(X_n)}{C \vdash_{<} \texttt{F(P)}}\ \forall^2\text{-E}^{****}$$

$$\frac{C \vdash_{<} \forall \texttt{a}\forall \texttt{b.(a<b => F(a)<G(b))}}{C \vdash_{<} \mu\texttt{a.F(a)}<\mu\texttt{b.G(b)}}\ \text{AR}$$

$$\frac{C \vdash_{<} \texttt{Incr(F)}}{C \vdash_{<} \mu\texttt{a.F(a)=F[a:=}\mu\texttt{a.F(a)]}}\ \text{Fix}$$

$$\frac{C \vdash_{<} \forall \texttt{a.(}\ \texttt{F(a)<a => T<a}\ \texttt{)}}{C \vdash_{<} \texttt{T} < \mu\texttt{a.F(a)}}\ \mu\text{-r} \qquad \frac{C \vdash_{<} \texttt{T < F(T)} \qquad C \vdash_{<} \texttt{T < T'}}{C \vdash_{<} \mu\texttt{a.F(a) < T'}}\ \mu\text{-l}$$

$$\frac{C \vdash_{<} \texttt{A}_i\texttt{<A'}_i (\text{For i=1,...,n}) \qquad n \leq m}{C \vdash_{<} \{\texttt{C}_1\texttt{:A}_1\texttt{;}\ldots\texttt{C}_m\texttt{:A}_m\} < \{\texttt{C}_1\texttt{:A'}_1\texttt{;}\ldots\texttt{C}_n\texttt{:A'}_n\}}\ \text{Product}$$

$$\frac{C \vdash_{<} \texttt{A}_i\texttt{<A'}_i (\text{For i=1,...,n}) \qquad n \leq m}{C \vdash_{<} \texttt{C}_1\ \texttt{of}\ \texttt{A}_1\texttt{|}\ldots\texttt{|C}_n\ \texttt{of}\ \texttt{A}_n < \texttt{C}_1\ \texttt{of}\ \texttt{A'}_1\texttt{|}\ldots\texttt{|C}_m\ \texttt{of}\ \texttt{A'}_m}\ \text{Sum}$$

$$\frac{\sigma \text{ is a permutation}}{C \vdash_{<} \{\texttt{C}_1\texttt{:A}_1\texttt{;}\ldots\texttt{C}_n\texttt{:A}_n\} = \{\texttt{C}_1\texttt{:A}_{\sigma 1}\texttt{;}\ldots\texttt{C}_{\sigma n}\texttt{:A}_{\sigma n}\}}\ \text{PermutP}$$

$$\frac{\sigma \text{ is a permutation}}{C \vdash_{<} \texttt{C}_1\ \texttt{of}\ \texttt{A}_1\texttt{|}\ldots\texttt{|C}_n\ \texttt{of}\ \texttt{A}_n = \texttt{C}_{\sigma 1}\ \texttt{of}\ \texttt{A}_{\sigma 1}\texttt{|}\ldots\texttt{|C}_{\sigma n}\ \texttt{of}\ \texttt{A}_{\sigma n}}\ \text{PermutS}$$

$$\frac{}{C \vdash_{<} \{\Theta_{\times}\texttt{(T)}\} <\{\texttt{type}\ id;\ \Theta_{\times}(id)\ \}}\ \text{Abstract}$$

$$\frac{C \vdash_{<} \texttt{A'<A} \qquad C \vdash_{<} \texttt{B(x)<B'(x)}}{C \vdash_{<} \forall \texttt{x:A.B}<\forall \texttt{x:A'.B'}}\ \text{Function type}^{*****}$$

$$\frac{\Gamma \vdash \texttt{u:A} \qquad \vdash^{\Gamma}_{<} \texttt{A<B}}{\Gamma \vdash \texttt{u:B}}\ \text{Sub} \qquad \frac{\Gamma \vdash \texttt{u:}\{\Theta\} \qquad \vdash^{\Gamma}_{<} \texttt{F}}{\Gamma \vdash \texttt{u:}\{\Theta\ \texttt{with F}\}}\ \text{SpecI}$$

$$\frac{\Gamma \vdash \texttt{u:}\{\Theta\ \texttt{with F}\}}{\vdash^{\Gamma}_{<} \texttt{F}}\ \text{SpecE-Fo} \qquad \frac{\Gamma \vdash \texttt{u:A(T)} \qquad \vdash^{\Gamma}_{<} \texttt{T=T'}}{\Gamma \vdash \texttt{u:A(T')}}\ \text{Eq}$$

*: when a is not free in C **: for any Γ-legal type T
: when X_n is not free in C **: for any n-ary Γ-legal predicate P
****: when x is not free in C

Figure 4: Logical rules

2.6 Some more examples

Let's give a more mathematical example. A type for groups could be:

```
{group}={type set; e:set; +:set->set->set; -:set->set}
```

Then, we can build a function that computes the product of two groups (the explicitation of the term `f` is left to the reader):

```
⊢ f:∀g:group.∀h:group.{group with set = g.set*h.set}
```

If we have the conditional `if...then...else`, we can then build:

```
⊢ fun b->fun g->fun h-> if b then h else g :
        ∀b:bool∀g:group.∀h:group.group.
```

This function can be applied to various implementations of {`group`}.

We can also build a record that contains a group and some elements of the product; we suppose that we have ⊢ `g:{group}` :

```
⊢ {p ; G=g ; E'= p.G.e,p.G.e}:
{self s;G:{group};E':(s.G:{group}).set*(s.G:{group}).set}
```

3 Theoretical background: ST type system

3.1 System ST: syntax, rules and semantics

We will follow the presentation of [Raf03a] and [Raf03b], excepted that we add a family of new operator Δ_n^i, named *Definite Description Operators* to the expressions. The definite description operator has been introduced in [RB05]. It's intended meaning is to enable us to give a name to an object that satisfies a property that is true for a certain element. We use here a strong form of definite description, which is related to the axiom of choice, as it will be explained in the semantics.

Definition 7 (sorts) *The set of Sorts ($\mathcal{S}$) is defined by the grammar:*

$$\mathcal{S} = o \mid \tau \mid \mathcal{S} \to \mathcal{S}$$

Notation 2 *We note* $\tau^n \to o$ *for* $\underbrace{(\tau \to (\ldots \to (\tau}_{n} \to o)\ldots))$.

Definition 8 (expressions) *The set of expressions is the set of simply typed lambda terms, using Sorts as simple types, written using the following sorted constants:*

$$\Rightarrow_\epsilon : \epsilon \to \epsilon \to \epsilon \text{ for } \epsilon \in \{o, \tau\} \quad \twoheadrightarrow : o \to \tau \to \tau$$
$$\subset : \tau \to \tau \to o \quad \Delta_n^i : (\tau^n \to o) \to \tau(*)$$
$$\forall_\epsilon^s : (s \to \epsilon) \to \epsilon \text{ for } \epsilon \in \{o, \tau\} \text{ and any sort } s.$$
$$(*) : n, i \in \mathbb{N} \text{ and } i \leq n$$

The intended meaning of τ and o are respectively "the set of types" and "the set of propositions".

Definition 9 (context) *A context Γ is a finite set composed of terms of sort o and pairs of the form $x : A$ with A of sort τ and x a lambda variable, where each lambda variable is declared at most once.*

Notation 3 *We omit the sorts when they are given by the context; We write $\forall x.A$ instead of $\forall \lambda x.A$; $\Rightarrow_\epsilon$ arrows are right-associative; the priorities are given in this order: $\forall > \twoheadrightarrow > \Rightarrow_\tau > \subset$. Propositions (terms of sort o) and Types (terms of sort τ) will often be written respectively P, Q, R,... and A, B, C,...*

Notation 4 *We use the following syntax for λ-terms: a term is either a variable $(x, y, z..)$, an application (uv), or an abstraction $\lambda x.t$. The occurrences of x in t are bound in $\lambda x.t$, and λ doesn't bind maximally to the right: for example, the rightmost occurrence of x in $(\lambda x.(xy)\ x)$ is free.*

We give here the basic rules. Additional rules and axioms are given in appendix A.

3.1.1 Semantics.

We take the same realizability semantics as in [Raf03a]:

Definition 10 *We associate to each sort s an interpretation domain $\overline{s}$ inductively defined by:*

$\overline{\tau} = \mathcal{P}_{\beta\eta}(\Lambda)$*: the set of parts of Λ closed for $\beta\eta$*
$\overline{o} = \{\emptyset, \Lambda\}$
$\overline{s \to s'} = \overline{s} \to \overline{s'}$*: the set of total functions from $\overline{s}$ to $\overline{s'}$.*

Definition 11 *An interpretation $\mathcal{I}$ is a mapping associating to a variable x of sort s an element $\mathcal{I}(x)$ of $\overline{s}$.*

Subtyping rules:

$$\frac{}{\Gamma \vdash_{ST} A \subset A} \qquad \frac{\Gamma \vdash_{ST} t : A \qquad \Gamma \vdash_{ST} A \subset B}{\Gamma \vdash_{ST} t : B}$$

$$\frac{\Gamma \vdash_{ST} A \subset B \qquad \Gamma \vdash_{ST} B \subset C}{\Gamma \vdash_{ST} A \subset C}$$

Type's implication rules:

$$\frac{\Gamma, x : A \vdash_{ST} t : B}{\Gamma \vdash_{ST} \lambda x.t : A \Rightarrow B} \qquad \frac{\Gamma \vdash_{ST} t : A \Rightarrow B \qquad \Gamma \vdash_{ST} u : A}{\Gamma \vdash_{ST} (tu) : B}$$

$$\frac{\Gamma \vdash_{ST} A \subset B \qquad \Gamma \vdash_{ST} C \subset D}{\Gamma \vdash_{ST} B \Rightarrow C \subset A \Rightarrow D}$$

Type's quantification rules:

$$\frac{\Gamma \vdash_{ST} B \subset A(y)}{\Gamma \vdash_{ST} B \subset \forall x.A(x)}\ (*) \qquad \frac{}{\Gamma \vdash_{ST} \forall x.A(x) \subset A(v)}\ (**)$$

$$\frac{\Gamma \vdash_{ST} t : A(y)}{\Gamma \vdash_{ST} t : \forall x.A(x)}\ (*) \qquad \frac{\Gamma \vdash_{ST} t : \forall x.A(x)}{\Gamma \vdash_{ST} t : A(v)}\ (**)$$

Proposition's quantification rules:

$$\frac{\Gamma \vdash_{ST} P(y)}{\Gamma \vdash_{ST} \forall x.P(x)}\ (*) \qquad \frac{\Gamma \vdash_{ST} \forall x.P(x)}{\Gamma \vdash_{ST} P(v)}\ (**)$$

Proposition's implication rules:

$$\frac{\Gamma, P \vdash_{ST} Q}{\Gamma \vdash_{ST} P \Rightarrow Q} \qquad \frac{\Gamma \vdash_{ST} P \Rightarrow Q \qquad \Gamma \vdash_{ST} P}{\Gamma \vdash_{ST} Q}$$

Special implication rules:

$$\frac{\Gamma, P \vdash_{ST} t : A}{\Gamma \vdash_{ST} t : P \twoheadrightarrow A} \qquad \frac{\Gamma \vdash_{ST} t : P \twoheadrightarrow A \quad \Gamma \vdash_{ST} P}{\Gamma \vdash_{ST} t : A}$$

$$\frac{\Gamma, P \vdash_{ST} A \subset B}{\Gamma \vdash_{ST} A \subset P \twoheadrightarrow B} \qquad \frac{\Gamma \vdash_{ST} A \subset B \qquad \Gamma \vdash_{ST} P}{\Gamma \vdash_{ST} (P \twoheadrightarrow A) \subset B}$$

Definite description:

$$\frac{}{\Gamma \vdash_{ST} \begin{array}{c} \forall P(\exists x_1, \ldots, x_n.P(x_1, \ldots, x_n)) \\ \Rightarrow P(\Delta_n^1(P), \ldots, \Delta_n^n(P)) \end{array}}\ (***)$$

(*): y not free in the conclusion of the rule.
(**): v is an expression having the same sort as x.
(***): $\exists x_1, \ldots, x_n.P(x_1, \ldots, x_n)$ is defined by $\forall K(\forall x_1, \ldots, x_n(P(x_1, \ldots, x_n) \Rightarrow K) \Rightarrow K)$.

Figure 5: Rules of system ST

Let Δ_n be a choice operator on $\overline{\tau}^n$, in the sense that if P is a property on $\overline{\tau}^n$ that is true for a certain element, then $\Delta_n(P)$ is one of those members of $\overline{\tau}^n$ for which P is true, otherwise $\Delta_n(P)$ is any element of $\overline{\tau}^n$.
If $x = (x_1, \ldots, x_n) \in \overline{\tau}^n$, we define $\pi_i(x) = x_i$.

Definition 12 *We define inductively the value of a term t in the interpretation $\mathcal{I}$, written $|t|^{\mathcal{I}}$ by:*

$$\begin{cases} |x|^{\mathcal{I}} = \mathcal{I}(x) \qquad |\lambda x.u|^{\mathcal{I}} = \lambda a |u|^{\mathcal{I}}[x := a] \\ |(uv)|^{\mathcal{I}} = |u|^{\mathcal{I}}(|v|^{\mathcal{I}}) \\ |\Rightarrow_\epsilon|^{\mathcal{I}} = \lambda a \lambda b \{t | \forall u \in a(t) u \in b\} \text{ for } \epsilon = o \text{ or } \tau \\ |\twoheadrightarrow|^{\mathcal{I}} = \lambda a \lambda b (\text{if } a = \Lambda \text{ then } b \text{ else } \Lambda) \\ |\subset|^{\mathcal{I}} = \lambda a \lambda b (\text{if } a \subset b \text{ then } \Lambda \text{ else } \emptyset) \\ |\forall^s_\epsilon|^{\mathcal{I}} = \lambda a \cap_{b \in \overline{s}} a(b) \text{ for } \epsilon = o \text{ or } \tau \\ |\Delta^i_n|^{\mathcal{I}} = \lambda P(\pi_i(\Delta_n(P))) \end{cases}$$

Note that the values of the constants do not depend on the interpretation. The following theorem states that the theory has a model:

Theorem 1 *Let $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, $P_1, \ldots, P_q$ be a context. For any interpretation $\mathcal{I}$, and any substitution σ, if $\sigma(x_i) \in |A_i|^{\mathcal{I}}$ for $i = 1, \ldots, n$ and $|P_j|^{\mathcal{I}} = \Lambda$ for $j = 1, \ldots, q$, then:*

$$\Gamma \vdash t : A \text{ implies } t[x_i := \sigma(x_i)] \in |A|^{\mathcal{I}} \text{ and}$$

$$\Gamma \vdash P \text{ implies } |P|^{\mathcal{I}} = \Lambda$$

proof:

By induction on the size of proofs. The only new case, compared to [Raf03a] is the rule "Definite description": one has to check that if $|\exists \vec{x}.P(\vec{x})|^{\mathcal{I}} = \Lambda$, then there exists an $\vec{a} \in \overline{\tau}^n$ s.t. $|P|^{\mathcal{I}}(\vec{a}) = \Lambda$.

Assume $|\exists \vec{x}.P(\vec{x})|^{\mathcal{I}} = \Lambda$; since $|\exists \vec{x}.P(\vec{x})|^{\mathcal{I}} = \cap_{k \in \overline{o}} |\forall \vec{x}(P(\vec{x}) \Rightarrow K) \Rightarrow K|^{\mathcal{I}[K:=k]}$, it follows that $|\forall \vec{x}(P(\vec{x}) \Rightarrow \emptyset) \Rightarrow \emptyset| = \Lambda$ (we note $|F(\emptyset)|$ for $|F(K)|^{\mathcal{I}[K:=\emptyset]}$). But $|\forall \vec{x}(P(\vec{x}) \Rightarrow \emptyset) \Rightarrow \emptyset| = \{u \in \overline{o}; \forall t \in |\forall \vec{x}(P(\vec{x}) \Rightarrow \emptyset)|, (u)t \in \emptyset\}$. Therefore, we must have $|\forall \vec{x}(P(\vec{x}) \Rightarrow \emptyset)| = \emptyset$. Now $|\forall \vec{x}(P(\vec{x}) \Rightarrow \emptyset)| = \cap_{\xi_1 \in \overline{\tau}}(\ldots(\cap_{\xi_n \in \overline{\tau}} |P(\vec{x}) \Rightarrow \emptyset|^{\mathcal{I}[\vec{x}:=\vec{\xi}]})\ldots) = \cap_{\xi \in \overline{\tau}^n} |P(\vec{x}) \Rightarrow \emptyset|^{\mathcal{I}[\vec{x}:=\vec{\xi}]}$. Suppose that for every $\xi \in \overline{\tau}^n$, $|P|(\xi) = \emptyset$; in this case, $|P(\xi) \Rightarrow \emptyset| = \Lambda$, and we have a contradiction. □

Remark: In the following, for sake of readability, we will use a the simplest form of definite description, limited to a single-argument predicate, that is $\Delta(P)$ will stand for $\Delta^1_1(P)$.

3.2 Some useful operators

3.2.1 Type application and lambda-abstraction.

These features allow to describe internally the singletons.

Definition 13 *Type abstraction and lambda-abstraction are defined by:*

$$A[B] = \forall X((A \subset B \Rightarrow X) \twoheadrightarrow X)$$
$$\Lambda X.T(X) = \forall X(X \Rightarrow T(X))$$

Now we have a way to give an interpretation from terms in types:

Definition 14 *Given a mapping ϕ associating each λ-variable to a type, the interpretation of a term t in ϕ is inductively defined by:*

$$|x|^\phi = \phi(x) \qquad |(uv)|^\phi = |u|^\phi[|v|^\phi]$$
$$|\lambda x.t|^\phi = \Lambda X.|t|^{\phi[x:=X]}$$

...which is substantiated by the following theorem that establishes a precise correspondence between typing and subtyping (proved in [Raf03a]):

Theorem 2 *If $\Gamma = x_1 : A_1, ..., x_n : A_n, P_1, ...P_q$ and $\phi(x_i) = A_i$ for i=1,...,n, then $\Gamma \vdash_{ST} t : A$ is derivable iff $\Gamma \vdash_{ST} |t|^\phi \subset A$ is derivable.*

3.2.2 Union, pair, etc.

Definition 15 *the union of a family $A_1, ..., A_n$ of types, of a family $(A(x))_{x\in s}$, disjunction of a family $P_1, ..., P_n$ of propositions, restriction, pairs, conjunction, falsities and non emptiness are defined by:*

$$A_1 \cup ... \cup A_n = \forall K((A_1 \subset K) \twoheadrightarrow ...(A_n \subset K) \twoheadrightarrow K)$$
$$\cup x A(x) = \forall K(\forall x(A(x) \subset K) \twoheadrightarrow K)$$
$$P_1 \vee ... \vee P_n = \forall Q((P_1 \Rightarrow Q) \Rightarrow ...(P_n \Rightarrow Q) \Rightarrow Q)$$
$$A \restriction P = \forall X((A \subset P \twoheadrightarrow X) \twoheadrightarrow X)$$
$$A \times B = \forall X^\tau((A \Rightarrow B \Rightarrow X) \Rightarrow X)$$
$$P \wedge Q = \forall X^o((P \Rightarrow Q \Rightarrow X) \Rightarrow X)$$
$$\bot_\tau = \forall X X \qquad \bot_o = \forall X \forall Y\, X \subset Y$$
$$A \neq \emptyset = A \subset \bot_\tau \Rightarrow \bot_o$$

It is helpful to make the comparison between the usual second-order encodings of logical connectors and the corresponding types, especially for union and disjunction, or for restriction and conjunction. The link between the union of a family of types and the existential connector, as defined below the rules at section 3.1, is also very instructive.

3.3 Singleton types, β-reduction and abstract types

Definition 16 *The property "being a singleton" is defined by:*

$$\mathcal{S}_\tau(A) = A \neq \emptyset \wedge \forall B((A \subset \cup X B(X)) \Rightarrow \exists X(A \subset B(X)))$$

This is a "key" property: it allows to prove subject reduction and makes a link between "Sums" (or Unions) and "Existentials".

Proposition 1 *Let Γ be a context of the shape: $x_1 : Y_1, ..., x_n : Y_n, \mathcal{S}_\tau(Y_1), ..., \mathcal{S}_\tau(Y_1), P_1, ..., P_q$. If $\Gamma \vdash t : A$, then $\Gamma \vdash \mathcal{S}_\tau(|t|^\phi)$, with ϕ as in Theorem 2.*

proof: It is a corollary of propositions 30 and 31 of [Raf03b]. □

Theorem 3 *System ST enjoys subject reduction for β: if $\Gamma \vdash_{ST} t : A$ and $t >_\beta t'$ then $\Gamma \vdash_{ST} t' : A$.*

proof: See [Raf03b]. □

Proposition 2 *Let Γ be a context of the shape: $x_1 : Y_1, ..., x_n : Y_n, \mathcal{S}_\tau(Y_1), ..., \mathcal{S}_\tau(Y_1), P_1, ..., P_q$. Then:*

$$\Gamma \vdash_{ST} u : \cup X^\tau T(X) \Longleftrightarrow$$
$$\Gamma \vdash_{ST} u : T(\Delta(\lambda X(|u|^{\phi[x_i:=Y_i]} \subset T(X))))$$

proof: The right-left implication is trivial. Assume $\Gamma \vdash_{ST} u : \cup X^\tau T(X)$. By theorem 2, $\Gamma \vdash_{ST} |u|^{\phi[x_i:=Y_i]} \subset \cup X^\tau T(X)$, and by proposition 1, we have $\mathcal{S}_\tau(|u|^{\phi[x_i:=Y_i]})$. Thus, applying the definition of $\mathcal{S}_\tau$ leads to $\exists X(|u|^{\phi[x_i:=Y_i]} \subset T(X))$. The Definite Description rule applied to $P = \lambda X(|u|^{\phi[x_i:=Y_i]} \subset T(X))$ entails $|u|^{\phi[x_i:=Y_i]} \subset T(\Delta(P))$, and another use of theorem 2 gives the final result. □

This proposition will allow us to give a meaning to `(p:T).t`. Effectively, as it will be formally described in section 4, the translation of the type for example `{ type t ; C:t }` will be the union of all the `{ C:t }`'s.
Now, if we infer $\vdash$ `p:{type t ; C:t}`, we have (informally) by the previous proposition:

$$\vdash \texttt{p: \{C:(p:\{C:t\}).t \}},$$

with `(p:{C:t}).t` $\equiv \Delta($`(λt.(|p|<{C:t})`$.$.

3.4 A basic result about integers

Before defining our translation, we need a basic result to ensure that the manipulation of constructors, which will be encoded as Church integers, is correct:

We give an extension of our language similar to Krivine's AF2[1] ([Kri93]). We enrich the language with two new sorts nat and $bool$, the following sorted constants: 0^{nat}, $s^{nat\to nat\to nat}$, $natrec^{\tau\to(\tau\to\tau)\to nat\to\tau}$, $True^{bool}$, $False^{bool}$, $boolrec^{\tau\to\tau\to bool\to\tau}$, and $Eq^{nat\to nat\to bool}$. The new interpretation domains are $\overline{nat}=\mathbb{N}$ and $\overline{bool}=\{0;1\}$, and we extend the interpretation of terms by:

$$\begin{cases} |0^{nat}|^{\mathcal{I}} = 0 \qquad |s^{nat\to nat\to nat}|^{\mathcal{I}}=\lambda n.n+1 \\ |True^{bool}|^{\mathcal{I}} = 1 \qquad |False^{bool}|^{\mathcal{I}}=0 \\ |Eq|^{\mathcal{I}} = \lambda n\lambda m(\text{if } n=m \text{ then 1 else 0}) \\ |boolrec|^{\mathcal{I}} = \lambda a\lambda b\lambda x(\text{if } x=1 \text{ then } a \text{ else } b) \\ |natrec|^{\mathcal{I}} = \lambda a\lambda f\lambda n(\text{the } n\text{-th term of the sequence } u \text{ defined by } u_0=a \text{ and } u_{n+1}=f(u_n)) \end{cases}$$

Notation 5 *In the following, the symbol "$=$" in an equation stands for Leibnitz's equality: $a=b$ is $\forall X(Xa\Rightarrow Xb)$.*

Lemma 1 *The following rules are acceptable (i.e. they do not modify the result of theorem 1):*

$$\frac{\frac{\frac{\frac{\frac{\frac{\frac{}{\vdash_{ST}\forall X\forall F(natrec\,X\,F\,0=X)}}{\vdash_{ST}\forall X\forall F\forall n(natrec\,X\,F\,(sn)=F(natrec\,X\,F\,n))}}{\vdash_{ST}\forall X\forall Y(boolrec\,X\,Y\,True=X)}}{\vdash_{ST}\forall X\forall Y(boolrec\,X\,Y\,False=Y)}}{\vdash_{ST}Eq\,0\,0=True}}{\vdash_{ST}\forall n(Eq\,0\,(sn)=False)}}{\vdash_{ST}\forall n(Eq\,(sn)\,0=False)}}{\vdash_{ST}\forall n\forall m(Eq\,(sn)\,(sm)=Eq\,n\,m)}$$

proof: Easy using the semantics. □

Definition 17 *We define the formulas (of sort τ) Nn and Bb by:*

$$Nn=\forall X(X0\Rightarrow\forall y(Xy\Rightarrow X(sy))\Rightarrow Xn)$$
$$Bb=\forall X(X\,True\Rightarrow X\,False\Rightarrow Xb)$$

The following theorem has for consequence, roughly speaking, that the integers are only inhabited by Church integers. It is surprising to see that it does not require to use an elementary extension of $\mathbb{N}$ as in [Kri93].

[1] The results could be expressed internally in ST, but we give here a short and easy presentation.

Theorem 4 *The following sequents are derivable:*

$$\vdash_{ST}\forall n(Nn\subset\Lambda X\Lambda F(natrec\,X\,(\lambda x.F[x])\,n))$$
$$\vdash_{ST}\forall b(Bb\subset\Lambda X\Lambda Y(boolrec\,X\,Y\,b))$$

proof: By definition of Λ, we must show $\vdash_{ST} Nn\subset(X\Rightarrow F\Rightarrow natrec\,X\,(\lambda x.F[x])\,n))$. For any A, for showing that $\vdash_{ST} Nn\subset A$, it is sufficient to find an X' such that $\vdash_{ST}(X'0\Rightarrow\forall y(X'y\Rightarrow X'(sy))\Rightarrow X'n)\subset A$. In our special case, using the properties of co- and contra-variance of $\Rightarrow$, it suffices to show three goals:

1. $X\subset X'0$
2. $F\subset\forall y(X'y\Rightarrow X'(sy))$
3. $natrec\,X\,(\lambda x.F[x])\,n)\subset X'n$

We take $X'=\lambda n.natrec\,X\,(\lambda x.F[x])\,n)$, which solves immediately the goals 1. and 3. Using [Raf03a], Fact 31, we have to show that $\vdash_{ST} F[X'y]\subset X'sy$ which is again immediate.
The proof for booleans is similar. □

To make more precise the link between Church integers and the theorem, suppose we have $\vdash_{ST} t:N(sss0)$. We are now able to affirm that $\vdash_{ST} t:\Lambda X\Lambda F natrec\,X\,(\lambda x.F[x])\,(sss0))$. Therefore, $\vdash_{ST} t:\Lambda X\Lambda F.F[F[F[x]]]$.
Now, by corollary 35 of [Raf03a] $|\Lambda X\Lambda F.F[F[F[x]]]|$ is the $\beta\eta$ equivalence class of $\lambda x.\lambda f.(f(f(fx)))$, thus $t=_{\beta\eta}\lambda x.\lambda f.(f(f(fx)))$.

Corollary 1 *There is a lambda term $=_{nat}$ such that for every $n,m\in\mathbb{N}$, for every $u,v\in\Lambda$ such that $\vdash_{ST} u:N(s^n0)$ and $\vdash_{ST} v:N(s^m0)$:*
If $n=m$ then $(=_{nat}\ uv)=_{\beta\eta}\lambda x\lambda yx$ else $(=_{nat}\ uv)=_{\beta\eta}\lambda x\lambda yy$. Moreover, $=_{nat}$ head-reduces to its normal form on Church integers.

proof: Take for $=_{nat}$ the term $\lambda n.(n\ T_0\ T_s)$ with:

$$\begin{cases} T_0=\lambda m.(m\ \ T\ \ \lambda\rho.F) \\ T_s=\lambda r.\lambda m.(m\ \ \lambda c.(cF\hat{0})\ \ \lambda\rho.\lambda c.(c\ (r(\rho F))\ (\hat{s}\rho T))F \\ \text{With } \hat{0}=T=\lambda x.\lambda y.x \text{ and } F=\lambda x.\lambda y.y \\ \text{and } \hat{s}=\lambda n.\lambda x.\lambda f.(f\ (nxf)) \end{cases}$$

We have $\vdash_{ST} t:\forall n\forall m(Nn\Rightarrow Nm\Rightarrow B(Eq\,n\,m))$. The result comes from the previous theorem and theorem 2, and a recurrence for head-normalization. □

4 Translation of the language

The goal of this section is to define a translation $\tilde{.}$ of our programming language into system ST such that the following theorem holds:

Theorem 5 *If* $\vdash \mathtt{p:T}$ *then* $\vdash_{ST} \tilde{\mathtt{p}} : \widetilde{\mathtt{T}}$,

$$\text{and} \begin{cases} \text{if } \mathtt{p>>p'} \text{ then } \tilde{\mathtt{p}} \succ \tilde{\mathtt{p}}\mathtt{'} \\ \text{if } \tilde{\mathtt{p}} \succ t, \text{ then there exists a program } \mathtt{p'} \\ \text{such that } t \succ \tilde{\mathtt{p}}\mathtt{'} \text{ and } \mathtt{p>>p'} \end{cases}$$

with $\succ$ *head-reduction of* λ*-calculus.*

This theorem entails the following soundness property:

Corollary 2 *For every program* $\mathtt{p}$ *and every type* $\mathtt{T}$, *if* $\vdash \mathtt{p:T}$, *then* $\tilde{\mathtt{p}} \in |\widetilde{\mathtt{T}}|$.

proof: It is a consequence of Theorems 1 and 5 $\square$

The first part of the theorem is a corollary of the following proposition:

Proposition 3 *For every context* Γ, *there exists a translation* $\tilde{.}^{\Gamma}$ *such that:*
if $\Gamma \vdash \mathtt{p:T}$ *then* $\tilde{\Gamma} \vdash_{ST} \tilde{\mathtt{p}} : \widetilde{\mathtt{T}}^{\Gamma}$, *and*
if $\mathtt{C} \vdash^{\Gamma}_{<} \mathtt{F}$, *then* $\widetilde{\mathtt{C}}^{\Gamma}, \tilde{\Gamma} \vdash_{ST} \tilde{\mathtt{F}}$.

4.1 Arrow types, recursive types, formulae, simple terms and contexts.

We identify type, type name and program name variables with variables of sort τ, program variables with λ variables, and n-ary predicate variables with variables of sort $\tau^n \to o.\underbrace{(\tau \to (\ldots \to (\tau}_{n} \to o)\ldots))$.

Definition 18 *We define the translation* $\tilde{.}^{\Gamma}$ *by:*

Types :

$$\begin{cases} \widetilde{\mathtt{a}} = \mathtt{a} \\ \widetilde{\forall \mathtt{x:A.B}} = \forall X(\mathcal{S}_\tau(X) \twoheadrightarrow X \subset \widetilde{\mathtt{A}}^{\Gamma} \twoheadrightarrow X \Rightarrow_\tau \widetilde{\mathtt{B}}^{\Gamma}\ [\tilde{\mathtt{x}} := X]) \\ \widetilde{\forall \mathtt{a.T}} = \forall \mathtt{a} \widetilde{\mathtt{T}}^{\Gamma} \\ \widetilde{\mu \mathtt{a.T}} = \forall \mathtt{a}((\widetilde{\mathtt{T}} \subset \mathtt{a}) \twoheadrightarrow \mathtt{a}) \end{cases}$$

Formulae :

$$\begin{cases} \widetilde{\mathtt{A<B}}^{\Gamma} = \widetilde{A}^{\Gamma} \subset \widetilde{B}^{\Gamma} \quad \widetilde{\mathtt{X_n(A_1,\ldots,A_n)}}^{\Gamma} = \mathtt{X_n}\ \widetilde{\mathtt{A_1}}^{\Gamma} \ldots \widetilde{\mathtt{A_n}}^{\Gamma} \\ \widetilde{\mathtt{F=>G}}^{\Gamma} = \widetilde{F}^{\Gamma} \Rightarrow_o \widetilde{G}^{\Gamma} \qquad \widetilde{\forall \xi F(\xi)}^{\Gamma} = \forall X \widetilde{F}^{\Gamma}[\xi := X] \end{cases}$$

Simple terms :

$$\begin{cases} \tilde{\mathtt{x}} = x \qquad \widetilde{\mathtt{fun\ x\ ->t}} = \lambda x.\tilde{\mathtt{t}} \qquad \widetilde{\mathtt{(uv)}} = (\tilde{\mathtt{u}}\tilde{\mathtt{v}}) \\ \widetilde{\mathtt{rec\ f}} = (\lambda x.(\tilde{f}(xx))\ \ \lambda x.(\tilde{f}(xx))) \end{cases}$$

Contexts :

$$\begin{cases} \tilde{\emptyset} = \emptyset \\ \widetilde{\Gamma, \mathtt{x:T}} = \tilde{\Gamma}, x : Y, \mathcal{S}_\tau(Y), Y \subset \widetilde{T}^{\Gamma} \quad \textit{with } Y \textit{ fresh} \end{cases}$$

The translation $\tilde{.}^{\Gamma}$ doesn't seem to be necessary at this stage. Its meaning will be clear when we will give the translation of the dotted types .

4.2 Constructors, Sum , Product and Abstract Types

We suppose given an injective mapping $\mathtt{C} \to \widehat{\mathtt{C}}$ from constructor names to Church integers.

Definition 19 *We extend the translation by:*

Sum Types :

$$\begin{cases} \widetilde{\mathtt{C}} & = & |\widehat{\mathtt{C}}| \times \Lambda X.X \\ \widetilde{\mathtt{C\ of\ T}}^{\Gamma} & = & |\widehat{\mathtt{C}}| \times \widetilde{T}^{\Gamma} \\ \widetilde{\mathtt{S_1/\ldots/S_n}}^{\Gamma} & = & \widetilde{\mathtt{S_1}}^{\Gamma} \cup \ldots \cup \widetilde{\mathtt{S_n}}^{\Gamma} \end{cases}$$

Sum terms :

$$\begin{cases} \widetilde{\mathtt{C}} = \lambda c.((c\ \widehat{\mathtt{C}})\lambda x.x) \\ \widetilde{\mathtt{C\ u}} = \lambda c.((c\ \widehat{\mathtt{C}})\tilde{\mathtt{u}}) \\ \widetilde{\mathtt{cases\ t\ of\ C_1 x->f_1/\ldots/C_k x->f_k}} = \\ \tilde{\mathtt{t}}\ \lambda n.\lambda u.(\theta_1^{n,u}(\theta_2^{n,u}(\ldots\theta_k^{n,u})\ldots)) \\ \textit{with } \theta_i^{n,u} = ((=_{nat}\ n)\ \widehat{\mathtt{C}_i})(\widetilde{\mathtt{fun\ x\ ->f_1}}\ u) \end{cases}$$

Record Types (without abstraction) :

$$\begin{cases} \widetilde{\{\mathtt{C_1\ :\ T_1;\ldots;C_n\ :\ T_n}\}}^{\Gamma} = \\ \forall\ \mathtt{a.(\ (C_1\ of\ T_1->a\ |\ldots|C_n\ of\ T_n->a)->a)} \\ \widetilde{\{\mathtt{T\ with\ F}\}}^{\Gamma} = \widetilde{\{\mathtt{T}\}}^{\Gamma} \restriction \widetilde{F}^{\Gamma} \end{cases}$$

Record terms :

$$\begin{cases} \widetilde{\{\mathtt{C_1\ =\ u_1;\ldots;C_n\ =\ u_n}\}} = \\ \mathtt{fun\ f\ ->cases\ f\ of\ \widetilde{C_1 g->\ g\ u_1/\ldots/C_n g->g\ u_n}} \\ \widetilde{\mathtt{t.C}} = \tilde{\mathtt{t}}\ \widetilde{\mathtt{(C\ (fun\ x->x))}} \end{cases}$$

Abstract types :

$$\begin{cases} \widetilde{\{\mathtt{type\ id;}\Theta(id)\}}^{\Gamma} & = & \cup X\ \widetilde{\{\Theta(id)\}}^{\Gamma}[id := X] \\ \widetilde{\{\mathtt{self\ s;}\Theta(s)\}}^{\Gamma} & = & \\ \cup X X \restriction (\mathcal{S}_\tau(X) \wedge (X \subset \widetilde{\{\Theta(s)\}}^{\Gamma}[s := X])) \end{cases}$$

Dotted types :

$$\begin{cases} \widetilde{\mathtt{(p:\{type\ id;}\Theta(id)\})\mathtt{.id}}^{\Gamma} = \Delta(\lambda X.(|\tilde{p}|^{\phi} \subset \theta(X)) \\ \textit{where} \qquad \tilde{\Gamma} = x_1 : Y_1, ..., x_n : Y_n, ... \\ \textit{and} \qquad \phi(x_i) = Y_i \\ \textit{and} \qquad \theta(X) = \widetilde{\{\Theta(\mathtt{a})\}}^{\Gamma}[\mathtt{a} := X] \end{cases}$$

5 Comparison with other works

[MP88], [CL90], and [Rus98] have modeled ADT with existentials, and their approach of modules is very similar to ours, despite the fact that we model ADTs by unions and use the notion of singletons to retrieve existentials (section 3.3). Russo proposes a sketched proof of soundness in his thesis; his method seems comparable to ours in the sense that he uses a dynamic semantics, which is also a way to identify saturated parts of the set of expressions. The main advantage of our method, compared to his method is that we work with a small language (λ-calculus) and with a type system which has already a semantics, and that the features of our language are only "macros".
The notions of opacity and transparency are discussed in [HL94] and [DCH03](which also uses the notion of singletons), with the notion of translucent sums and first-order modules, and in [Ler95], without first-order modules. Our presentation is "strictly opaque" in the sense that once a type is abstracted, it is not possible to retrieve the hidden type; it is the reason why we use the construction "`with` *id*`=...`", which enables us to retrieve a weak form of projectability of types.
Another language that uses ADTs in core language is Quest: [CL91] gives a PER interpretation of types that ensures the soundness of the language, except for recursive types; we haven't been able to find out the continuation of this work concerning this feature. Our system enjoys recursive types.
[AGW04] contains an approach of ADTs with Hilbert's ϵ, trying to give a computational meaning to the instantiation of an ADT during a computation in order to imitate dynamic linking, which can be linked to transparent type naming propagation during program evaluation; we don't have this feature in our language, but our use of definite description is more liberal than in their work, for which a strong condition of *subordination* seems necessary to obtain soundness. It would be interesting to see if a semantics like the one we uses could ameliorate the understanding of the mechanisms behind their calculus.
[Cou97] has build a general theory of modules placed on top of PTSs, and his purpose is not really the one we purchase because we want first order modules; [Ch03] follows his approach in an implementation for the Coq system.
We are looking for an extension of our language that could be very similar to Extended ML[KST97], a framework that adds specifications in modules, which formal soundness is not yet established.

6 Further research and conclusion

The specifications we have presented in the examples are quite poor; in fact, the construction `with` and the rules can be extended to be closer to ST, and allow richer specifications as those presented. We are working on it, together with another rule for API that could allow to "speak from" properties of the fields of the record. For example, we could give a better specification for groups:

```
                self s;
                type set;
{group}={       e:set;          }
                +:set->set->set;
                -:set->set
                with F
```

with `F=...`$\wedge$ $\forall$`x:set(s.e+x=x)` $\wedge$ `...`
We are sure of the properties of type preservation through program reduction ("subject reduction"). But what about termination? We already know that we have lost the property of strong and weak normalization in ST. But we have found out type properties that express normalizability, and could ensure a control of this property.
Having at most the expressive power of system F, we know that we have lost decidability of type checking. We are looking for a semi-decidable typing algorithm together with a syntax for annotating programs that would allow a programmer to type-check his program with respect to a specification.
The ability to handle recursive records and abstract programs will maybe allow us to imitate some features of object programming languages.

Conclusion: We have presented a toy-programming language that enjoys good properties: it is very expressive, mimics many features of modern languages, and has a well established semantics. It seems to be a good basis for developing a more realistic programming language.

A ST's additional rules and axioms

Extra rules **Left rules:**

$$\frac{\Gamma, x:A \vdash_{ST} A \subset B}{\Gamma \vdash_{ST} A \subset B} \qquad \frac{\Gamma, x:A \vdash_{ST} t:B \qquad \Gamma \vdash_{ST} A' \subset A}{\Gamma, x:A' \vdash_{ST} t:B}$$

$$\frac{\Gamma \vdash_{ST} t:(P \twoheadrightarrow \bot_\tau) \Rightarrow \bot_\tau}{\Gamma \vdash_{ST} P} \qquad \frac{\Gamma, x:A(y) \vdash_{ST} t:B}{\Gamma, x:\cup_y A(y) \vdash_{ST} t:B}$$

Axioms:

1. $\vdash_{ST} \forall A,B(\forall x(A(x) \Rightarrow B(x)) \subset \forall x A(x) \Rightarrow \forall x B(x)$
2. $\vdash_{ST} \forall P \forall A,B(P \twoheadrightarrow A \Rightarrow B \subset A \Rightarrow P \twoheadrightarrow B)$
3. $\vdash_{ST} \forall A,B(A \subset \bot_\tau \Rightarrow B)$
4. $\vdash_{ST} \forall A,B(\mathcal{S}_\tau(B) \Rightarrow \forall X(A[X] \subset B[X]) \Rightarrow (A \subset B))$
5. $\vdash_{ST} \forall A(A \subset \cup X(X \restriction (\mathcal{S}_\tau(X) \wedge (X \subset A))))$
6. $\vdash_{ST} \forall F \forall A(\forall x(F(x) \Rightarrow A) \subset \cup x F(x) \Rightarrow A)$
7. $\vdash_{ST} \forall X \forall B(\mathcal{S}_\tau(X) \Rightarrow ((X \Rightarrow \cup Y B(Y)) \subset \cup Y(X \Rightarrow B(Y))))$
8. $\vdash_{ST} \forall A,B(A \subset B \cup B^c)$ With $B^c = \cup X(X \restriction (X \cap B \subset \bot_\tau)$
9. $\vdash_{ST} \forall A,B(A \subset B \Rightarrow B \subset A \Rightarrow A = B)$

B Proof of theorem 5

B.1 Evaluation

Assume that `p>>p'`. We make an induction by analyzing each case of reduction:

- Redex, Recursor and Left cases are trivial.
- `p` is a Case:
  `p=cases(`$\mathtt{C}_i$ `u)of` $\mathtt{C}_1$`x->`$\mathtt{f}_1$`(x)|...|`$\mathtt{C}_n$`x->`$\mathtt{f}_n$`(x)`
  and `p'=`$\mathtt{f}_i$`(u)`. We note $\hat{\mathtt{C}}_i = C$ and $\tilde{\mathtt{u}} = u$.
  $\widetilde{\mathtt{p}} = (\lambda c.((cC)u))\ \lambda n.\lambda u.(\theta_1^{n,u}(\theta_2^{n,u} \dots \theta_n^{n,u})\dots)$
  $\succ (\theta_1^{C,u}(\theta_2^{C,u} \dots \theta_n^{C,u})\dots)$
  $\succ (\theta_i^{C,u} \dots \theta_n^{C,u})\dots)$ by corollary 1
  $\succ \widetilde{\texttt{fun x ->}}\ \mathtt{f}_1\ u$ which is treated in the Redex case.
- `p` is an access to a field of a record :
  `p={`$\mathtt{C}_1$`=`$\mathtt{u}_1$ `;...; `$\mathtt{C}_n$`=`$\mathtt{u}_n$`}.`$\mathtt{C}_i$ and `p'=`$\mathtt{u}_i$.
  $\widetilde{\mathtt{p}} =$
  `fun f->cases f of` $\widetilde{\mathtt{C}_1\mathtt{g}\texttt{->}\mathtt{gt}_1}$ `...`$\mathtt{C}_n$`g->`$\mathtt{gt}_n$ $\mathtt{C}_i\widetilde{\texttt{(fun x->x)}}$
  $\succ$ `cases`$\mathtt{C}_i$`(fun x->x)` $\widetilde{\texttt{of }\mathtt{C}_1\texttt{g->}\mathtt{gu}_1}$ `...`$\mathtt{C}_n$`g->`$\mathtt{gu}_n$
  $\succ \widetilde{\mathtt{u}_i}$ by the case Cases.
- `p` is a Left-case:
  `p=cases u of` $\mathtt{C}_1$`x->`$\mathtt{f}_1$`(x)|...|`$\mathtt{C}_n$`x->`$\mathtt{f}_n$`(x)` ,
  `p'=cases u' of` $\mathtt{C}_1$`x->`$\mathtt{f}_1$`(x)|...|`$\mathtt{C}_n$`x->`$\mathtt{f}_n$`(x)` ,
  and `u>>u'`.
  $\widetilde{\mathtt{p}} = \widetilde{\mathtt{u}}\ \ \lambda n.\lambda u.(\theta_1^{n,u}(\theta_2^{n,u} \dots \theta_n^{n,u})\dots)$
  $\succ \widetilde{\mathtt{u'}}\ \ \lambda n.\lambda u.(\theta_1^{n,u}(\theta_2^{n,u} \dots \theta_n^{n,u})\dots)$ by induction hypothesis.
- `p` is a Left-field:
  `p=u.C`, `p'=u'.C`, and `u>>u'`.
  $\widetilde{\mathtt{p}} = \widetilde{\mathtt{u}}\ \ \lambda c.((cC)\lambda x.x)$
  $\succ \widetilde{\mathtt{u'}}\ \ \lambda c.((cC)\lambda x.x)$ by induction hypothesis.

Assume that $\widetilde{\mathtt{p}} \succ t$. It is sufficient to remark that each step of `>>`-reduction corresponds to one or more steps of head-reduction, and that `>>`-normal terms are also head-normal terms. It ensures that $t \succ \widetilde{\mathtt{p'}}$ for some `p'`.

B.2 Type correctness

As stated at the beginning of section 4, we prove proposition 3 by induction on the derivation. Let's remark that the conditions of legality and Γ-legality are sufficient to ensure the fact that the translation is well defined.

B.2.1 Logical rules

We treat the rules Sum, Product, PermutP,Permut S and Abstract. All the other rules are trivially mirrored by one or more rules of ST, excepted Fix, that is treated in Fact 49 of [Raf03a]:

- Product is treated by the negative position of the sum type and the positive positions of the $\mathtt{A}_i$'s.
- Sum is a consequence of the fact that $\lambda X.\mathtt{C\ of}\ X$ is increasing, and of trivial properties of union.
- PermutP and Permut S are immediate from the translation and properties of union.
- Abstract is an instance of the first Union rule of Fact 28 of [Raf03a].
- SpecI and SpecE-Fo are some Restriction rules of Fact 22 of [Raf03a].
- Eq is a consequence of 9th additional axiom.

B.2.2 Typing rules

We treat all the rules excepted Ax, ∀-I, ∀-E, which are trivial, and recursion, treated in Theorem 52 of [Raf03a]:

- Ab: We assume that $\widetilde{\Gamma, \mathtt{x:A}} \vdash_{ST} \widetilde{\mathtt{p}} : \widetilde{\mathtt{T}}^{\Gamma,x:A}$; by definition, we have:

$$\widetilde{\Gamma}, x:Y, \mathcal{S}_\tau(Y), Y \subset \widetilde{\mathtt{A}}^{\Gamma} \vdash_{ST} \widetilde{\mathtt{p}} : \widetilde{\mathtt{T}}^{\Gamma,x:A}$$

with Y fresh. It follows:

$$\widetilde{\Gamma} \vdash_{ST} \lambda x.\widetilde{\mathtt{p}} : \mathcal{S}_\tau(Y) \twoheadrightarrow (Y \subset \widetilde{\mathtt{A}}^{\Gamma}) \twoheadrightarrow Y \Rightarrow \widetilde{\mathtt{T}}^{\Gamma,x:A}$$

By an easy induction on the structure of T, we have $\widetilde{\mathtt{T}}^{\Gamma,x:A} = \widetilde{\mathtt{T}}^{\Gamma}[\widetilde{\mathtt{x}} := Y]$, and the freshness of Y gives the right to universally quantify it, and so:

$$\widetilde{\Gamma} \vdash_{ST} \widetilde{\mathtt{fun\ x->p}}^{\Gamma} : \widetilde{\forall \mathtt{x:A.T}}^{\Gamma}$$

- Ap: We assume that $\widetilde{\Gamma} \vdash_{ST} \widetilde{\mathtt{q}} : \widetilde{\mathtt{A}}^{\Gamma}$. By proposition 1 and Theorem 2, we have $\widetilde{\Gamma} \vdash_{ST} \mathcal{S}_\tau(\widetilde{\mathtt{q}})$ and $\widetilde{\Gamma} \vdash_{ST} |\widetilde{\mathtt{q}}|^{\phi} \subset \widetilde{\mathtt{A}}^{\Gamma}$. The other premise of the rule is $\widetilde{\Gamma} \vdash_{ST} \widetilde{\mathtt{p}} : \widetilde{\forall \mathtt{x:A.B}}^{\Gamma}$. The translation of the product type together with some immediate rules give $\widetilde{\Gamma} \vdash_{ST} \widetilde{\mathtt{p}} : |\widetilde{\mathtt{q}}|^{\phi} \Rightarrow B[x := |\widetilde{\mathtt{q}}|^{\phi}]$. We conclude with Theorem 2 and $\Rightarrow$-elimination.
- P-I is a consequence of S-E.
- S-E is a consequence of the two following derivable rules:

$$\frac{\Gamma \vdash_{ST} f : A_i \Rightarrow T \qquad i = 1 \ldots n}{\Gamma \vdash_{ST} f : (A_1 \cup \ldots \cup A_n) \Rightarrow T} \ (1)$$

$$\frac{\Gamma \vdash_{ST} f_i : A_i \Rightarrow T \qquad i = 1 \ldots k}{\Gamma, x : C_i \times A_i \vdash_{ST} (x\ \theta) : T} \ (2)$$

With $\theta = \lambda n.\lambda u.(\theta_1^{n,u}(\theta_2^{n,u} \ldots \theta_k^{n,u}) \ldots)$ defined as in section 4.2.
(1) can be derived from the rule $\vdash_{ST} \forall A, B, C((A \subset (B \Rightarrow C)) \Rightarrow (B \subset A^{-1}[C]))$ with $A^{-1}[C] = \cup X(X \upharpoonright (A \subset (X \Rightarrow B)))$, proved in fact 30 of [Raf03a], and trivial reasonings on union.
(2) can be derived from the rules $\vdash_{ST} \forall A, B(A \times B \subset \Lambda C.C[A][B])$ and $\vdash_{ST} \forall A, B((\Lambda X.A(X))[B] \subset A(B))$, proved respectively in Theorem 41 and Fact 44 of [Raf03a], and from the second point of Theorem 4.

- P-E is a consequence of:

$$\vdash_{ST} \lambda c.((c\hat{C})\lambda x.x) : C \times (A \Rightarrow A)$$

- S-I 1 and 2 are the usual rule of pairing.
- Reinforcement: we assume that:

$$\widetilde{\Gamma} \vdash_{ST} \widetilde{\mathtt{p}} : \{\mathtt{type}\ \widetilde{id;\Theta_\times(id)}\}^{\Gamma}$$

with $\{\mathtt{type}\ \widetilde{id;\Theta_\times(id)}\}^{\Gamma} = \cup X\ \{\widetilde{\Theta(id)}\}^{\Gamma}[id := X]$.
Considering the definition of $\widetilde{\Gamma}$, we are in the conditions of application of Theorem 2, which gives:

$$\widetilde{\Gamma} \vdash_{ST} \widetilde{\mathtt{p}} : \{\widetilde{\Theta_\times(id)}\}^{\Gamma}[id := \delta]$$

with $\delta = \Delta(\lambda X; (|\widetilde{\mathtt{p}}| \subset \{\widetilde{\Theta(id)}\}^{\Gamma}[id := X])$, which is the wanted result.

- API : we assume all the hypotheses, and define the programs ϕ_i and the types θ_i by:

$$\begin{cases} \phi_1 = \mathtt{p}_1 & \phi_{i+1} = \mathtt{f}_{i+1}(\phi_1, \ldots, \phi_i) \\ \theta_1 = \mathtt{T}_1 & \theta_{i+1} = \mathtt{T}_{i+1}(\phi_1, \ldots, \phi_i) \end{cases}$$

From the hypotheses, we get that:

$$\Gamma \vdash \{\mathtt{C}_i = \phi_i\} : \{\mathtt{C}_i : \theta_i\}$$

($\{\mathtt{C}_i = (\text{resp.}:)\psi_i\}$ for $\{\mathtt{C}_1 = (\text{resp.}:)\psi_1; \ldots; \mathtt{C}_n = (\text{resp.}:)\psi_n\}$)
Thus, we have by induction hypothesis and a few rules:

$$\widetilde{\Gamma} \vdash_{ST} \widetilde{\{\mathtt{C}_i = \phi_i\}}^{\Gamma} : \widetilde{\{\mathtt{C}_i : \theta_i\}}^{\Gamma}$$

Take $t = \widetilde{\mathtt{p}}$ with `p=rec (fun p -> {C1=p1;...Cn=fn(p.C1,...,p.Cn)}`. Clearly, $t =_\beta \widetilde{\{\mathtt{C}_i = \phi_i\}}$, and $\widetilde{\mathtt{p.C}_i} =_\beta \widetilde{\phi_i}$. Since β-equality of terms implies equality of the corresponding singleton types, we have $|\widetilde{\mathtt{p.C}_i}| = |\widetilde{\phi_i}|$, and thus

$$\widetilde{\Gamma} \vdash_{ST} \widetilde{\mathtt{p}} : \widetilde{\mathtt{T(p)}}^{\Gamma}$$

with $\mathtt{T(p)} = \{\mathtt{C}_1 : \mathtt{T}_1; \ldots; \mathtt{C}_n : \mathtt{T}_n(\mathtt{p.C}_1, \ldots \mathtt{p.C}_n)\}$.
An application of Proposition 1 and rules of restriction (Fact 22 of [Raf03a]) gives the conclusion.

- APE: assume

$$\widetilde{\Gamma} \vdash_{ST} \widetilde{\mathtt{p}} : \cup X.X \upharpoonright (\mathcal{S}_\tau(X) \wedge X \subset \widetilde{\{\mathtt{T(p)}\}}^{\Gamma}[p := X])$$

with T(p) defined as above. By the definition of singleton, Proposition 1, and by application of Lemma 27 of [Raf03b], we get that:

$$\widetilde{\Gamma} \vdash_{ST} \widetilde{\mathtt{p}} : \widetilde{\mathtt{T(p)}}^{\Gamma}$$

which is sufficient for concluding.

- SpecE-Ty is one of the Restriction rules of Fact 22 of [Raf03a].

References

[AGW04] Abadi,Martìn, Gonthier,Georges and Werner,Benjamin. *Choice in dynamic linking.* Foundations of Software Science and Computation structures: 7th International Conference, FOSSACS 2004, Springer-Verlag (March 2004), 12-26.

[Ch03] Chrząszcz,Jacek. *Implementing Modules in the Coq System.* In "Theorem Proving in Higher Order Logic, Rome, Italy, September 8-13, 2003 proceedings", David Basin and Burkhart Wolff (Eds.): LNCS 2758, Springer-Verlag, pp. 270-286.

[CL90] Cardelli,Luca and Leroy,Xavier. *Abstract types and the dot notation.* Technical report 56, DEC SRC, Palo Alto, CA, March 1990.

[CL91] Cardelli,Luca and Longo,Giuseppe *A semantic basis for Quest.* Journal of functional programming, 1(4):417-458, October 1991.

[CMP00] Chailloux,Emmanuel, Manoury,Pascal and Pagano,Bruno. *Developing applications with Objective Caml.* O'Reilly, 2000.

[Cou97] Courant,Judicaël. *A module calculus for Pure Type Systems.* In Typed Lambda Calculi and Applications 97, Lecture Notes in Computer Science, pages 112 - 128. Springer-Verlag, 1997.

[DCH03] Dreyer,Derek, Crary,Karl and Harper,Robert. *A type system for higher-order modules.*2003 Symposium on Principles of Programming Languages.

[HL94] Harper,Robert and Lillibridge,Mark. *A type theoretic approach to higher-order modules with sharing.* In Twenty-first ACM Symposium on Principles of Programming languages, pages 123-137, Portland, OR, January 1994.

[HM93] Harper,Robert and Mitchell,John C. *On the type structure of Standard ML.* ACM Transactions on programming Languages and Systems, 15(2):211-252, April 1993.

[Kri93] Krivine,Jean-Louis. *Lambda-calculus, types and models.* Ellis Horwood (1993).

[KST97] Kahrs,Stefan, Sannella,Donald, and Tarlecki, Andrzej. *The definition of Extended ML: a gentle introduction.* TCS 173:445-484, 1997.

[Ler95] Leroy,Xavier. *Applicative functors and fully transparent higher-order modules.* In conference Record of POPL'95:ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages, pages 142-153, San Francisco, CA, January 1995.

[MT94] MacQueen,David B. and Tofte,Mads. *A semantics for higher-order functors.* In Donald T. Sannella, editor, Programming Languages and systems-ESOP'94, vol.788 of Lecture Notes in Computer Science, pages 409-483. Springer-Verlag,1994.

[MMM91] Mitchell,John, Meldal,Sigurd and Madhav,Neel. *An extension of standard ML modules with subtyping and inheritance.* In Eighteenth ACM Symposium on Principles of Programming languages, January 1991.

[MP88] Mitchell, John C. and Plotkin, Gordon D. *Abstract types have existential type.* ACM Transactions on Programming Languages and Systems, 10(3):470-502, July 1988.

[MTH] Milner,Robin, Tofte,Mads, Harper,Robert and MacQueen,David. *The definition of Standard ML(Revised).* MIT Press, 1997.

[OC] Objective Caml.`http://www.ocaml.org`

[Raf03a] Raffalli,Christophe. *System ST toward a type system for extraction and proofs of programs.* Archive for Pure and Applied Logic, 2003, volume 122, pages 107-130.

[Raf03b] Raffalli,Christophe. *System ST, beta-reduction and completeness*, presented at LICS 2003, publication IEEE, pages 21-32.

[RB05] Russell,Bertrand. *On denoting*; Mind,14:479-493, 1905.

[Rus98] Claudio V. Russo. *Types for modules.* PhD thesis, Edinburgh University, Edinburgh, Scotland, 1998. LFCS Thesis ECS-LFCS-98-389.

[Rus00] Claudio V. Russo. *First-class structures for standard ML.* Nordic Journal of computing, 7(4):348-374,2000.