Realizability for programming languages.

Christophe Raffalli

Realizability for programming languages.

Christophe Raffalli*

29 mars au 4 avril 2010 à Chambéry

Abstract

We present a toy functional programming language inspired by our work on PML together with a criterion ensuring safety and the fact that non termination can only occur via recursive programs. To prove this theorem, we use realizability techniques and a semantical notion of types.

1 Introduction

1.1 Realizability

Realizability has been introduced by Kleene in 1945 [8] as a semantics of intuitionistic logic. The key idea is to interpret every proposition as a set of programs (or functions) whose elements, called realizers, compute witnesses for the corresponding proposition. Typically, a realizer of the proposition

$$\forall n{:}\,\mathbb{N}\ (A(n) \Rightarrow \exists m{:}\,\mathbb{N}\ B(n,m))$$

is a program that computes, from a natural number n and a realizer of $A(n)$, an ordered pair formed by an integer m (the 'witness') and a realizer of the formula $B(n,m)$ (the 'justification').

Although Kleene expressed realizers as natural numbers (that is: as codes of recursive functions), the very definition of realizability naturally calls for a functional programming language to express realizers. This is probably why Kreisel [9] replaced (codes of) recursive functions by closed λ-terms when he introduced the notion of modified realizability. The λ-calculus was introduced by Church [3] in the 30's as a universal programming language based on the sole notion of a function. Besides its applications to logic and realizability, the λ-calculus was the major source of inspiration of functional programming languages, from LISP to ML dialects (SML, Caml, Haskell).

The method of realizability has also been successfully used to prove the strong normalization of many typed λ-calculi. Girard's proof of strong normalization of system F constitutes a particular case of realizability, where types are interpreted as particular sets of programs called reducibility candidates. More recent proofs of strong normalization are based on the method of orthogonality, in which the allowed sets of realizers (i.e. the candidates) are defined from the

*email: raffalli@univ-savoie.fr

interaction between programs and evaluation contexts. The idea of orthogonality was already present (at least implicitly) in Tait's [23] and Girard's [6], but it was first explicitly used by Parigot in [17].

1.2 Typing in programming languages

However, realizability is a highly flexible technique that is not limited to strong normalization proofs, and it can be also used to establish the safety of programs written in realistic languages. By the safety of a program, we mean here the property that the evaluation of this program will never produce an error (typically by applying a primitive to wrong arguments), independently from the problem of its termination.

Traditionally, safety results of type systems based on the Damas-Hindley-Milner algorithm (HM) [4] are not based on realizability, but on the property of subject reduction expressing that typing is preserved during evaluation. However, the corresponding proofs are usually quite technical, and very sensitive to minor modifications or extensions of the type system. For this reason, one may argue that such proofs do not really help to understand type systems and that they are not an efficient guide to develop new ones.

Typing with constraints More recently, the use of constraints solving for typing [15, 18] was introduced. A very popular formalism for constraints solving applied to typing is HM(X), which is very well explained by Pottier and Rémy in [20]. The key idea of their work is to extract typing constraints from the program in such a way that these constraints and the basic properties of types lie in a decidable fragment of first-order predicate logic, namely: the existential fragment of first-order predicate logic.

In this setting, the fact that a program is typed can be written as a formula of the form

$$A \;\Rightarrow\; \exists t_1, \ldots, t_n \; C$$

where A is a conjunction of universal axioms like the injectivity of the arrow type

$$\forall t, u, t', u' \; (t \to u = t' \to u' \;\Rightarrow\; t = t' \wedge u = u')$$

and where C is the conjunction of all constraints extracted from the program. (For instance, if t_i, t_j and t_k are the type variables that are respectively attached to sub-expressions M, N and $(M\,N)$ within the program, then C will contain the constraint $t_i = t_j \to t_k$.) This more abstract view is much more helpful to design new type systems.

This simplistic description of HM(X) is not really faithful, due to the fact that witnesses cannot be extracted in general in the existential fragment of the predicate calculus (if classical logic is present for instance). In fact, the HM(X) algorithm also relies on the introduction of extra rules to solve the constraints, which rules are usually based on unification.

It is important to understand that in the HM(X) framework, constraints are solved in the syntactic models of types (using unification algorithms). This

is still the case in other works based on subtyping (see for instance [1, 19]) that are closer to what will be presented here.

Constraints and realizability With realizability in mind, it is no more necessary to express the solutions of the constraints in the algebra of syntactic types. Indeed, we only need to ensure that the constraints have a solution in a suitable realizability model. Since this problem is in general undecidable, we will introduce some sufficient (and non necessary) conditions together with an algorithm to check whether these conditions are satisfied.

These conditions will be checked in two phases: we will first *saturate* the constraints; then, we will require the constraints to be *inductive* (or *well-founded*), thus disallowing arbitrary recursive types. The latter condition is just designed to ensure the existence of a solution in the realizability model. The work presented here resembles those of Aiken, Wimmers and Pottier (among others) [1, 19] except for the definition of well-founded constraints and the simplifications of constraints which we omit (in fact they are already implemented in PML). These simplifications are necessary in practice when polymorphism is introduced, because we then copy the constraints at each generalization, and they should be simplified before copying.

1.3 Other trends in realizability

Nowadays, there is a great interest in realizability for non-constructive logic. Griffin [7] discovered that Felleisen's [5] `callcc` operator was related to Pierce's law. Krivine [10] and Parigot [16] applied realizability to classical analysis (i.e. classical second order arithmetic) using extensions of the λ-calculus (by adding a C operator in Krivine's work, or a μ-construction in Parigot's $\lambda\mu$-calculus). Later, this work was extended to ZF set theory [12], including the dependent axiom of choice [13, 22].

In non-constructive logic, the program extracted from a proof of an existential statement does not compute a correct witness in general. There are two view-points here:

- Realizability can be a semantical tool to analyze proof systems by studying the property shared be all programs realizing a given theorem. This was done for instance by Krivine for the completeness theorem [11].

  This is also interesting because it gives a formal meaning to sentences like "this theorem implies that theorem" of "this theorem is easier than that theorem", because this make sense on programs ("implies" could mean an easy reduction and "easier" could refer to computational complexity). For instance, the completeness theorem for minimal logic is trivial compared to the one for classical propositional logic (see [21]).

- We can also try to see when proofs using non-constructive principles can lead to programs that really computes the wanted witness. Most works in this area are typically based on Gödel or Friedman syntactic translations from classical to intuitionistic logic. But one may argue that realizability

leads to a better comprehension of the phenomena. A key tool here is the use of Krivine's storage [10] operator to deal with data types in non constructive proofs. (For the connection between Krivine's approach and negative translations, see [14].) This is specially true with the dependant axiom of choice. One interesting hope there, is the discovery of new algorithms by extracting programs from proofs [21]...

Current works try to extend realizability to the general axiom of choice or other strong axioms. There are also deep connections between Cohen's forcing and realizability that are currently being investigated.

1.4 What is done here

We present a toy programming language supporting higher-order functions, extensible records and polymorphic variants, as well as a sufficient condition (that can be tested in polynomial time) to ensure the existence of a realizability model that will guaranty that the evaluation is safe (you do not reach impossible case in the evaluation) and that looping programs have to unfold fixpoints infinitely often.

This property of looping programs is useful because it implies strong normalisation of programs without fixpoint and it also means that if you want to use some termination checker to prove the termination of your programs (which PML does), you just have to focus on fixpoints.

This toy programming language is actually the core of the current implementation of PML[1] which enjoys more complex features (like polymorphism, better typing of default case, ...) as well as a proof system that is based on the typing algorithm. These features will be briefly presented in the conclusion.

What is new here is: the simplicity of the algorithm compared to what it can do (this is arguable), the use of realizability, in a subtyping context, to prove that programs are safe in the above sense and the condition for the constraints to be inductive which naturally arises from realizability.

The benefit over HM(X) is (we think) simplicity: for instance, we can get for free extensible records, while they require polymorphism in HM(X). However, we lose type inference because we do not solve the constraints. Nevertheless, a rich language for types can be easily recovered using partial identity functions. This is described in the conclusion.

We also emphasize on the lattice of possible relations between type variables contrary to the lattice of types which is more usual with subtyping. Indeed, as we do not solve constraints, we have no types but only constraints between type variables. However, we found that a lot of interesting generalizations could be done by enlarging the lattice of relations between types. Usually there is only $\subset$, $\supset$, $=$ and the absence of relation written $\perp$. In this simple case, direct constraints between type names can be represented by a directed graph. All the algorithms presented with matrices with coefficients in the lattice of relations could be rephrased as some kind of computation of the transitive closure of a graph. But, we shall see that the presentation using the lattice with four

[1] `http://lama.univ-savoie.fr/tracpml`

elements $\{\bot, \subset, \supset, =\}$ is elegant, and in the conclusion, we will show a few possible generalizations with larger lattices.

1.5 Notes about this document

This document contains a full implementation of the algorithm described here in a *literate* programming style. The code is extracted from the LaTeX source and compiled. I also wrote some pretty printers for code and typing constraints. All the examples are generated by running the extracted code on examples with various options controlling what is printed.

Feel free to download this work to study and extend it. It is available using `darcs` via:

```
darcs get http://lama.univ-savoie.fr/~raffalli/repos/EJCIM
```

2 The toyML language

The language is defined from three denumerable sets of identifiers:

- the set of *variables* $\mathcal{V}_\lambda = \{x, y, \ldots\}$,
- the set of *constructor names* $\mathcal{V}_\Sigma = \{\mathbf{C}, \mathbf{Z}, \mathbf{S}, \mathbf{Cons}, \mathbf{Nil}, \ldots\}$; and
- the set of *label names* $\mathcal{V}_\Pi = \{l, \mathit{car}, \mathit{cdr}, \ldots\}$.

The set $\mathcal{P}$ of all programs is then defined using the following BNF:

$$\begin{array}{rcllll} P & := & x & | & \texttt{fun}\ x \to P \ \mid\ P\,P \ \mid\ \texttt{fix}\ x \to P \\ & | & \mathbf{C}[P] & | & \texttt{case}\ P\ \texttt{of}\ \mathbf{C}[id] \to P \mid \cdots \mid \mathbf{C}[id] \to P \\ & | & P.l & | & \{l = P; \ldots; l = P\} \end{array}$$

The first three constructs correspond to variables, functions, applications and fix-point. The second line corresponds to variants and case analysis on variants. Finally, the last line corresponds to projections and records.

There is one more restriction that is not shown in the BNF: in records and case analyzes, the variant names in the same case analysis and the labels in the same record must be pairwise distinct. In particular, the programs $\texttt{case}\ M\ \texttt{of}\ \mathbf{C}[x] \to N_1 \mid \mathbf{C}[x] \to N_2$ and $\{l = N_1; l = N_2\}$ are not allowed.

First, we precise some conventions and abbreviations:

- Priorities (to avoid parentheses) follow the usual ML rule: application is left-associative and has a higher priority than functions, fixpoints and cases. Cases are right associative when nested:
  - $M\ N\ P$ reads $(M\ N)\ P$.
  - $\texttt{fun}\ x \to M\ N$ reads $\texttt{fun}\ x \to (M\ N)$.
  - $\texttt{case}\ x\ \texttt{of}\ \mathbf{C}[x] \to \texttt{case}\ y\ \texttt{of}\ \mathbf{D}[y] \to M \mid \mathbf{E}[y] \to N$ reads $\texttt{case}\ x\ \texttt{of}\ \mathbf{C}[x] \to (\texttt{case}\ y\ \texttt{of}\ \mathbf{D}[y] \to M \mid \mathbf{E}[y] \to N)$
- $\texttt{fun}\ x_1 \ldots x_n \to M$ is a shortcut for $\texttt{fun}\ x_1 \to \ldots \texttt{fun}\ x_n \to M$

- `fix` $f\ x_1 \ldots x_n \to M$ denotes `fix` $f \to$ `fun` $x_1 \to \ldots$ `fun` $x_n \to M$
- $\mathbf{C}[r]$ is a shortcut for $\mathbf{C}[\{r\}]$ when r is a record without curly braces. Moreover, $\mathbf{C}[]$ reads $\mathbf{C}[x]$ in a `case` pattern where the variable x does not occur in the right-hand side of the pattern.
- $M[x \leftarrow N]$ represents the (capture avoiding) substitution of N for the variable x in M.

  Example: if M is `fun` $x\ y \to x\ z$ and if N is `fun` $y \to x$, then the notation $M[z \leftarrow N]$ refers to the program `fun` $x'\ y \to x'$ (`fun` $y \to x$).
- $M[(x_i \leftarrow N_i)_{i \in I}]$ denotes the simultaneous substitution of the variables x_i by the terms N_i in M, for all $i \in I$.
- `case` M `of` $(\mathbf{C}_i[x] \to N_i)_{i \in I}$ denotes the program
  `case` M `of` $\mathbf{C}_{i_1}[x] \to N_{i_1} \mid \ldots \mid \mathbf{C}_{i_n}[x] \to N_{i_n}$ (where $I = \{i_1, \ldots, i_n\}$).
- $\{(l_i = N_i)_{i \in I}\}$ denotes the program $\{l_{i_1} = N_{i_1}; \ldots; l_{i_n} = N_{i_n}\}$
  (where $I = \{i_1, \ldots, i_n\}$).

To understand how programs compute, we now give the *operational semantics* for our toy programming language. We will use the notation $M \succ_r N$ to express that the program M evaluates to N using one reduction rule named r. We will also use $M \succ N$ for the transitive closure of all possible reductions.

Here are the reduction rules:

beta reduction $(\texttt{fun}\ x \to M)N \succ_\beta M[x \leftarrow N]$.

This rule corresponds to the replacement of the formal argument x in the body of the function by the real argument N the function is applied to.

fixpoint $\texttt{fix}\ x \to M \succ_\mu M[x \leftarrow (\texttt{fix}\ x \to M)]$.

The fixpoint rule allows for recursive definition (see the examples below).

projection $\{(l_i = P_i)_{i \in I}\}.l_j \succ_\pi P_j \quad \text{if } j \in I$

Here we project a record field. The implementation actually uses a more complex rule: $\{l_1 = P_1; \ldots; l_n = P_n\}.l_i \succ_\pi p_i[l_1 \leftarrow p_1, \ldots, l_{i-1} \leftarrow p_{i-1}]$ when $1 \leq i \leq n$. This extended rule allows fields of a given record to mention previously defined fields (without using a fixpoint).

Exercise 1 *Show how to express this feature using a fixpoint.*

case selection $\texttt{case}\ \mathbf{C}_j[N]\ \texttt{of}\ (\mathbf{C}_i[x] \to P_i)_{i \in I} \succ_\sigma P_j[x \leftarrow N] \quad \text{if } j \in I$

Here the case corresponding to the given variant is selected and the argument of the variant is substituted as expected.

congruence $M \succ_r M' \quad \text{implies} \quad E[M] \succ_r E[M']$

Here we use a *context* $E[]$, that is a program with one unique hole, to express the fact that the above reduction can be performed anywhere

inside a program. Contexts can bind variables, which means that we allow reduction under function abstractions or in the right-hand side of case analysis.

A program may also contain a *manifest error*, corresponding to a piece of code that cannot be reduced, and that would lead to a runtime error when executed. To indicate such errors, we define the notation $M \Uparrow$ as follows:

- $(\texttt{fun}\ x \to M).l \Uparrow$ $\qquad$ $\texttt{case fun}\ x \to M\ \texttt{of}\ \ldots \Uparrow$
- $\mathbf{C}[M].l \Uparrow$ $\qquad$ $\mathbf{C}[M]\, N \Uparrow$
- $\{\ldots\}N \Uparrow$ $\qquad$ $\texttt{case}\ \{\ldots\}\ \texttt{of}\ \ldots \Uparrow$
- $\texttt{case}\ \mathbf{D}[\ldots]\ \texttt{of}\ (\mathbf{C}_i[x] \to \ldots)_{i\in I} \Uparrow$ $\quad$ when $\mathbf{D} \neq \mathbf{C}_i$ for all $i \in I$.
- $\{(l_i = \ldots)_{i\in I}\}.k \Uparrow$ $\quad$ when $k \neq l_i$ for all $i \in I$.

Moreover, we require that every program that contains an erroneous piece of code is also considered as a manifest error:

- If $M \Uparrow$, then $E[M] \Uparrow$ (where $E[]$ is an arbitrary context with one hole).

Definition 2 (*safe*) *We say that a program M is safe when it cannot reduce to an error, that is: $M \succ M'$ implies $M' \not\Uparrow$.*

Here are some examples of programs and reductions:

omega The shortest looping λ-term (that reduces to itself):

$(\texttt{fun}\ x \to x\ x)\ (\texttt{fun}\ x \to x\ x)$

halving A partial function that computes the half of even numbers only (figure 1 gives an example of reduction using this program):

$\texttt{fix}\ \mathit{half}\ n \to$
$\quad \texttt{case}\ n\ \texttt{of}$
$\quad \mid \mathbf{Z}[] \to \mathbf{Z}[]$
$\quad \mid \mathbf{S}[n'] \to \texttt{case}\ n'\ \texttt{of}\ \mid \mathbf{S}[n''] \to \mathbf{S}[\mathit{half}\ n'']$

length The length of lists and an example of reduction in figure 2

$\texttt{fix}\ \mathit{length}\ l \to$
$\quad \texttt{case}\ l\ \texttt{of}\ \mid \mathbf{Nil}[] \to \mathbf{Z}[]\ \mid \mathbf{Cons}[c] \to \mathbf{S}[\mathit{length}\ c.\mathit{cdr}]$

The above examples were reduced using a specific strategy (*call-by-value*) which is implemented in the code given in this course (see appendix A). However, we will prove properties that are independent from the evaluation strategy. This is natural because the language is purely functional (no side-effect allowed) and therefore enjoys the Church-Rosser property :

Theorem 3 (*Church-Rosser property*) *If $M \succ M_1$ and $M \succ M_2$, then there exists a program M_0 such that $M_1 \succ M_0$ and $M_2 \succ M_0$.*

```
   (fix half n → ...) S[S[Z[]]]
≻μ (fun n →
      case n of
      | Z[] → Z[]
      | S[n'] → case n' of | S[n"] → S[(fix half n → ...) n"]) S[S[Z[]]]
≻β case S[S[Z[]]] of
   | Z[] → Z[]
   | S[n'] → case n' of | S[n"] → S[(fix half n → ...) n"]
≻σ case S[Z[]] of | S[n"] → S[(fix half n → ...) n"]
≻σ S[(fix half n → ...) Z[]]
≻μ S[(fun n →
        case n of
        | Z[] → Z[]
        | S[n'] → case n' of | S[n"] → S[(fix half n → ...) n"]) Z[]]
≻β S[case Z[] of
     | Z[] → Z[]
     | S[n'] → case n' of | S[n"] → S[(fix half n → ...) n"]]
≻σ S[Z[]]
```

Figure 1: Reduction of halving

```
   (fix length l → ...) Cons[car = A[]; cdr = Cons[car = B[]; cdr = Nil[];];
   ]
≻μ (fun l → case l of | Nil[] → Z[] | Cons[c] → S[(fix length l → ...) c.cdr])
     Cons[car = A[]; cdr = Cons[car = B[]; cdr = Nil[];];]
≻β case Cons[car = A[]; cdr = Cons[car = B[]; cdr = Nil[];];] of
   | Nil[] → Z[]
   | Cons[c] → S[(fix length l → ...) c.cdr]
≻σ S[(fix length l → ...) { car = A[]; cdr = Cons[car = B[]; cdr = Nil[];];
     }.cdr]
≻π S[(fix length l → ...) Cons[car = B[]; cdr = Nil[];]]
≻μ S[(fun l →
       case l of | Nil[] → Z[] | Cons[c] → S[(fix length l → ...) c.cdr])
     Cons[car = B[]; cdr = Nil[];]]
≻β S[case Cons[car = B[]; cdr = Nil[];] of
     | Nil[] → Z[]
     | Cons[c] → S[(fix length l → ...) c.cdr]]
≻σ S[S[(fix length l → ...) { car = B[]; cdr = Nil[]; }.cdr]]
≻π S[S[(fix length l → ...) Nil[]]]
≻μ S[S[(fun l →
         case l of | Nil[] → Z[] | Cons[c] → S[(fix length l → ...) c.cdr])
       Nil[]]]
≻β S[S[case Nil[] of | Nil[] → Z[] | Cons[c] → S[(fix length l → ...) c.cdr]]]
≻σ S[S[Z[]]]
```

Figure 2: Reduction of length

Proof: Realizability can not be used to prove this theorem (at least not yet). See [2] to see the proof techniques used for Church-Rosser.

Corollary 4 (Uniqueness of normal form) *If a term M reduces to a normal term (that is: a term where no further reduction can be performed), then this normal term M' is unique. If moreover the term M is safe, then its unique normal form M' contains no manifest error (i.e. $M' \not\lightning$).*

The listing 1 is the type definition in Caml for the *abstract syntax of programs*. The type of programs is parameterized by `'a` to be able to decorate programs with type names. Each constructor of the language, except variables, needs one parameter for its type, and each binding occurrence of a variable (`fun` and `case`) also needs a parameter for the type of the bound variable.

```
1  type id = string   type cid = string   type lid = string
2
3  type 'a program =
4  | Var of id
5  | App of 'a program * 'a program * 'a
6  | Fun of id * 'a * 'a program * 'a
7  | Fix of id * 'a * 'a program * 'a
8  | Cst of cid * 'a program * 'a
9  | Cas of 'a program * (cid * id * 'a program * 'a) list * 'a
10 | Rec of (lid * 'a program) list * 'a
11 | Pi of 'a program * lid * 'a
```

Listing 1: type for programs in program.ml

3 Realizability candidates

We shall now define and manipulate particular sets of programs: *realizability candidates*. To define them, we will use the method of orthogonality, which is a systematic way to define such candidates from the particular correctness invariant we want to convey throughout the proofs.

This correctness invariant is embodied as a particular set of programs written $\perp\!\!\!\perp$ (read: *double bottom*), that we define here as follows, first defining the notion of *bad reduction*:

Definition 5 *A reduction of a term M is* bad *if it yields an erroneous term or if it is infinite while using only finitely many times the fixpoint rules.*

The set $\perp\!\!\!\perp$ is defined as the set of all programs that do not have bad reduction. This means that $\perp\!\!\!\perp$ is the set of all safe programs whose infinite reductions use infinitely many times the fixpoint reduction rule.

This definition means that we are not only interested in the safety of programs, but also in the way programs may loop. The condition about infinite reduction sequences has been added here to ensure that every term $M \in \perp\!\!\!\perp$ that does not contain the `fix` construct is strongly normalizable. In particular, the looping term $(\texttt{fun}\, x \to x\ x)\ (\texttt{fun}\, x \to x\ x)$ does not belong to $\perp\!\!\!\perp$. (But the looping fixpoint $\texttt{fix}\, x \ \to\ x$ does.)

Note that by definition, the set $\perp\!\!\!\perp$ is closed under arbitrary reductions.

Exercise 6 (too difficult) *Show that $\perp\!\!\!\perp$ is not a recursive set of programs. It is probably not even recursively enumerable!*

Since the idea of orthogonality is based on the interaction between programs (seen as *players*) and their possible *opponents*, we first need to define what is an *opponent* of a program. Very often, opponents are taken as the evaluation contexts corresponding to the given reduction strategy. To preserve the independence from a particular reduction strategy, and due to our general form of case construct, we need to take arbitrary contexts as opponents:

Definition 7 (context) *A context E is an arbitrary program in $\perp\!\!\!\perp$, whose 'holes' are represented by the occurrences of a variable χ that has been fixed once and for all. (This variable may occur 0, 1 or more times in E). When M is a term and E is a context, we write $M \star E$ for $E[\chi \leftarrow M]$.*

Note that with this definition, contexts cannot *capture* variables. (This will be an important property in what follows.) Indeed, if E is $\texttt{fun}\ x \to \chi$ and if M is $\texttt{fun}\ y \to x$, then $M \star E$ denotes the program $\texttt{fun}\ x' \to \texttt{fun}\ y \to x$, and not the program $\texttt{fun}\ x \to \texttt{fun}\ y \to x$ where the variable x has been captured.

The same definition (with the same notation) will be also used to compose contexts. It is thus important to remark that

Lemma 8 *The operation $E \star F$ is associative.*

Proof: Let E, F, G be three contexts, then $(E \star F) \star G = G[\chi \leftarrow F[\chi \leftarrow E]]$ and $E \star (F \star G) = (G[\chi \leftarrow F])[\chi \leftarrow E]$. Both are equal without any hypothesis (which is rare for properties of substitution) because we substitute twice the same variable. This can be formally checked by induction on G. ■

Our notion of realizability candidates will only work if the set $\perp\!\!\!\perp$ is $\perp\!\!\!\perp$-saturated according to the definition below. This definition may seem complicated ... In fact, you can rediscover the definition by first trying to prove the adequacy lemma 26, and you will see that all the properties written here are just what is needed—and nothing more.

Definition 9 ($\perp\!\!\!\perp$-saturated sets) *We say that a set $\mathcal{A}$ of programs is $\perp\!\!\!\perp$-saturated if the following conditions holds (for all contexts E):*

1. *If $M[x \leftarrow P] \star E \in \mathcal{A}$ and $P \in \perp\!\!\!\perp$, then $(\texttt{fun}\ x \to M)\ P \star E \in \mathcal{A}$.*
2. *If $P_j[x \leftarrow M] \star E \in \mathcal{A}$, $M \in \perp\!\!\!\perp$ and $P_i \in \perp\!\!\!\perp$ for all $i \in I$, then $(\texttt{case}\ \boldsymbol{C}_j[M]\ \texttt{of}\ (\boldsymbol{C}_i[x] \to P_i)_{i \in I}) \star E \in \mathcal{A}$.*
3. *If $P_j \star E \in \mathcal{A}$ and $P_i \in \perp\!\!\!\perp$ for all $i \in I$, then $\{(l_i = P_i)_{i \in I}\}.l_j \star E \in \mathcal{A}$.*
4. *If for all $n \in \mathbb{N}$ and for a fresh variable x, we have $M^n(x) \star E \in \mathcal{A}$ and $M \in \perp\!\!\!\perp$, then $(\texttt{fix}\ x \to M) \star E \in \mathcal{A}$.*

(Writing $M^0(x) = x$ and $M^{n+1}(x) = M[x \leftarrow M^n(x)]$.)

Lemma 10 *The set $\perp\!\!\!\perp$ is $\perp\!\!\!\perp$-saturated.*

Proof: We treat each case separately, following a common pattern: we do a proof by coinduction: we prove that if there is a bad reduction from the term given in the conclusion, then we can construct a bad reduction in the initial term.

We first need to recall what is a proof by *coinduction*: It is a way to construct a sequence (finite of infinite) with a given property and at each step of the proof we need to check that we really extend the sequence (we say that each step is *productive*). In fact, this is too restrictive, we should allow some steps where the sequence stagnates, but we must prove that these steps cannot occur consecutively infinitely often.

1. We prove the following lemma: let E be a term where the variable χ_i occurs exactly once for all $i \in I$. Assume that $E[(\chi_i \leftarrow (\texttt{fun}\ x \rightarrow M_i)\, P_i)_{i\in I}] \notin \perp\!\!\!\perp$, and for all $i \in I$, $P_i \in \perp\!\!\!\perp$, then $E[(\chi_i \leftarrow M_i[x \leftarrow P_i])_{i\in I}] \notin \perp\!\!\!\perp$.

   Proving this allows us to deduce the property we want by renaming each occurrence of χ in the initial context with a different name $(\chi_i)_{i\in I}$ and taking $M_i = M$ and $P_i = P$ for all $i \in I$.

   Proving this lemma means that we can assume that we have a bad reduction (unsafe or infinite using finitely many time the fix-point rule) of $E[(\chi_i \leftarrow (\texttt{fun}\ x \rightarrow M_i)P_i)_{i\in I}]$ and construct by coinduction a bad reduction of $E[(\chi_i \leftarrow M_i[x \leftarrow P_i])_{i\in I}]$.

   There are only four possible cases for the first reduction (basically because there is no critical pair for the reduction of our language):

   - If it occurs in E: $E \succ_r E'$, then the variables χ_i may be erased or duplicated. We can replace duplicated variables in E' by distinct variables and ignore erased variables. This means that there is a context E'', a family of variables $\{\chi'_j\}_j \in J$ and a function ψ from J to I such that $E' = E''[(\chi'_j \leftarrow \chi_{\psi(j)})_{j\in J}]$ and all χ_j occur exactly once in E''. The function ψ_j is not surjective when some variables χ_i are erased in E' and it is not injective when they are duplicated. Then, we have, $E[(\chi_i \leftarrow M_i[x \leftarrow P_i])_{i\in I}] \succ_r E''[(\chi'_j \leftarrow M_{\psi(j)}[x \leftarrow P_{\psi(j)}])_{j\in J}]$ and we conclude by coinduction.
   - If the first reduction occurs in M_j; $M_j \succ_r M'_j$ for some j, then we write $M'_i = M_i$ if $i \neq j$ and we have $E[(\chi_i \leftarrow M_i[x \leftarrow P_i])_{i\in I}] \succ_r E[(\chi_i \leftarrow M'_i[x \leftarrow P_i])_{i\in I}]$ which allows to conclude by coinduction.
   - If the first reduction occurs in P_j; $P_j \succ_r P'_j$ for some j, then we write $P'_i = P_i$ if $i \neq j$ and we have $E[(\chi_i \leftarrow M_i[x \leftarrow P_i])_{i\in I}] \succ_r^* E[(\chi_i \leftarrow M_i[x \leftarrow P'_i])_{i\in I}]$. The symbol $\succ_r^*$ represents 0, 1 or more reduction step using the same rule r. If there is no reduction (which means that x does not occur free in M_j), then we need to remark that this case can not happens consecutively infinitely often, because that would mean that there is a bad reduction in one the terms $P_i \in \perp\!\!\!\perp$ which is impossible. Therefore this steps does not entail productivity.

- If the first reduction is the reduction of the redex substituted to χ_j for some $j \in I$: $(\texttt{fun}\ x \to M_j)P_j \succ_\beta M_j[x \leftarrow P_j]$. Then, we can take $E' = E[\chi_j \leftarrow M_j[x \leftarrow P_j]]$, and we do have $E[(\chi_i \leftarrow M_i[x \leftarrow P_i])_{i\in I}] = E'[\chi_i \leftarrow M_i[x \leftarrow P_i])_{i\in I\setminus\{j\}}]$. This step is not directly productive, but it can not happens infinitely often because the size of the index set I decreases while in the previous step, which in not productive either, I remains unchanged.

2. This case is similar to the previous case, but more cumbersome to write.

3. This case is still similar and even a bit simpler than the first case.

4. For the fixpoint case, assuming a bad reduction of $E[(\chi_i \leftarrow \texttt{fix}\ x \to M_i)_{i\in I}]$, we can take n to be the total number of reductions of the fixpoint rule and produce a bad reduction of $E[(\chi_i \leftarrow M_i^n(x))_{i\in I}]$ using the same technique. The productivity will come from the fact the the number of reductions of the fixpoint rule will decrease when we apply one. ■

We can now define orthogonality and realizability candidates:

Definition 11 (Orthogonality) *Let $\mathcal{M}$ be a set of programs and $\mathcal{E}$ a set of contexts. We define their orthogonal as follows:*

$$\begin{aligned} \mathcal{M}^\perp &= \{E \text{ contexts in } \perp\!\!\!\perp \text{ s.t. for all } M \in \mathcal{M}, M \star E \in \perp\!\!\!\perp\} \\ \mathcal{E}^\perp &= \{M \text{ programs in } \perp\!\!\!\perp \text{ s.t. for all } E \in \mathcal{E}, M \star E \in \perp\!\!\!\perp\} \end{aligned}$$

Lemma 12 *Orthogonality enjoys the following very general properties:*

1. *It is contravariant : if $\mathcal{M} \subset \mathcal{M}'$ then $\mathcal{M}'^\perp \subset \mathcal{M}^\perp$ (the same holds for context).*

2. *Double orthogonal is covariant.*

3. *$\mathcal{M} \subset \mathcal{E}^\perp$ if and only if $\mathcal{E} \subset \mathcal{M}^\perp$.*

4. *$\mathcal{E} \subset \mathcal{E}^{\perp\perp}$ and $\mathcal{E}^{\perp\perp\perp} = \mathcal{E}^\perp$ (the same holds for programs).*

5. *If $\mathcal{E}_i$ is a family of set of contexts indexed by $i \in I$, then*

$$\left(\bigcup_{i\in I} \mathcal{E}_i\right)^\perp = \bigcap_{i\in I} \mathcal{E}_i^\perp \text{ and } \bigcup_{i\in I} \mathcal{E}_i^\perp \subset \left(\bigcap_{i\in I} \mathcal{E}_i\right)^\perp$$

6. *If $\mathcal{E}$ is a set of contexts, then $\mathcal{E}^\perp \subset \emptyset^\perp = \perp\!\!\!\perp$.*

Proof:

1. Assume that $\mathcal{E} \subset \mathcal{E}'$ for two sets of contexts. Let $M \in \mathcal{E}'^\perp$, for all $E \in \mathcal{E}$, we have $E \in \mathcal{E}'$ and therefore $M \star E \in \perp\!\!\!\perp$. This means that $M \in \mathcal{E}^\perp$. The same works for set of programs.

2. Immediate from 12.1.

3. We prove the left to right implication, the converse having an identical proof. Let $\mathcal{E}$ be a set of contexts and $\mathcal{M}$ be a set of programs, assume that $\mathcal{M} \subset \mathcal{E}^\perp$ and let E be a context in $\mathcal{E}$. We want to show that $E \in \mathcal{M}^\perp$ so we take $M \in \mathcal{M} \subset \mathcal{E}^\perp$ and we have immediately $M \star E \in \perp\!\!\!\perp$.

4. By 12.3 $\mathcal{E} \subset \mathcal{E}^{\perp\perp}$ comes from $\mathcal{E}^\perp \subset \mathcal{E}^\perp$ and $\mathcal{E}^{\perp\perp\perp} \subset \mathcal{E}^\perp$ comes from $\mathcal{E} \subset \mathcal{E}^{\perp\perp} \subset \mathcal{E}^{\perp\perp\perp\perp}$.

5. Let $E \in \bigcup_{i\in I} \mathcal{E}_i^\perp$, then $E \in \mathcal{E}_j^\perp$ for some $j \in I$. Then, let M be in $\bigcap_{i\in I} \mathcal{E}_i$, we have $M \in \mathcal{E}_j$ which means that $M \star E \in \perp\!\!\!\perp$. This establishes

$$\bigcup_{i\in I} \mathcal{E}_i^\perp \subset \left(\bigcap_{i\in I} \mathcal{E}_i\right)^\perp$$

   For the orthogonal of union, we have $M \in \bigcap_{i\in I} \mathcal{E}_i^\perp$ if and only if $\forall i \in I, \forall E \in \mathcal{E}_i$, $M \star E \in \perp\!\!\!\perp$ which is equivalent to $\forall E \in \bigcup_{i\in I} \mathcal{E}_i$, $M \star E \in \perp\!\!\!\perp$. This exactly means $M \in \left(\bigcup_{i\in I} \mathcal{E}_i\right)^\perp$.

6. Let $\mathcal{E}$ be a set of contexts with no occurrence of χ, Then $M \star E = E \in \perp\!\!\!\perp$ for any program M. Therefore, $\mathcal{E}^\perp = \mathcal{P}$.

7. First, it is clear that $\{\chi\}^\perp = \perp\!\!\!\perp$: by definition, $M \in \{\chi\}^\perp$ if and only if $M \star \chi = M \in \perp\!\!\!\perp$. Let $\mathcal{E}$ be a set of contexts with at least one occurrence of χ in one element $E \in \mathcal{E}$. Then, $M \star E \in \perp\!\!\!\perp$ implies that $E[\chi \leftarrow M] \in \perp\!\!\!\perp$ which implies that $M \in \perp\!\!\!\perp$, because if M has a bad reduction, then $E[\chi \leftarrow M]$ has also one, because χ really occurs in E.

Definition 13 (*realizability candidates*) *Because of the above property, we will be only interested by set of programs that are the orthogonal of some set of contexts. We call this kind of sets* realizability candidates *and we write $\mathcal{C}$ the set of all realizability candidates.*

Lemma 14 (*property of realizability candidates*) *:*

1. *If $\mathcal{M} \in \mathcal{C}$ then $\mathcal{M}^{\perp\perp} = \mathcal{M}$.*
2. *Realizability candidates are closed by arbitrary intersections.*
3. *$\perp\!\!\!\perp$ is the largest realizability candidate.*
4. *The smallest realizability candidate is $\perp\!\!\!\perp^\perp$ and contains all variables. We will write it $\perp\!\!\!\perp_0$*
5. *Any realizability candidate $\mathcal{A}$ is $\perp\!\!\!\perp$-saturated and verifies $\perp\!\!\!\perp_0 \subset \mathcal{A} \subset \perp\!\!\!\perp$.*

Proof: The first property 14.1 is exactly 12.4 because candidates are themselves some orthogonal, 14.2 is 12.5, 14.3 is 12.6, 14.4 is immediate from 12.1. $\perp\!\!\!\perp_0$ contains all variables because $x \star E \in \perp\!\!\!\perp$ if and only if $E \in \perp\!\!\!\perp$.

The last property, is immediate from 14.3 and 14.4 and because $\perp\!\!\!\perp$ being saturated, all its subsets are saturated. ■

Finally, we end this section by defining *constructions* on candidates and sets of contexts that we will need to interpret types:

Definition 15

- *If $\mathcal{M}$ is a realizability candidate and $\mathcal{E}$ a set of contexts, then*

$$\mathcal{M}.\mathcal{E} = \{E[\chi \leftarrow \chi\ M]\ s.t.\ M \in \mathcal{M}\ and\ E \in \mathcal{E}\}$$

- *If $\mathcal{M}$ and $\mathcal{N}$ are realizability candidates then*

$$\mathcal{M} \to \mathcal{N} = (\mathcal{M}.\mathcal{N}^{\perp})^{\perp}$$

- *If $\mathcal{E}_i$ are contexts for all $i \in I$, then*

$$\overline{\Sigma}_{i=1}^{n}\overline{\boldsymbol{C}_i}[\mathcal{E}_i] = \{(\texttt{case}\ \chi\ \texttt{of}\ (\boldsymbol{C}_i[\chi] \to E_i)_{i\in I}) \star E\ s.t.\ \forall i \in I, E_i \star E \in \mathcal{E}_i\}$$

- *If $\mathcal{A}_i$ are realizability candidates for all $i \in I$, then*

$$\Sigma_{i=1}^{n}\boldsymbol{C}_i[\mathcal{A}_i] = (\overline{\Sigma}_{i=1}^{n}\overline{\boldsymbol{C}_i}[\mathcal{A}_i^{\perp}])^{\perp}$$

- *If $\mathcal{E}_i$ are contexts for all $i \in I$, then*

$$\overline{\Pi}_{i=1}^{n}\overline{l_i} : \mathcal{E}_i = \bigcup_{i\in I}\{\chi.l_i \star E_i\ s.t.\ E_i \in \mathcal{E}_i\}$$

- *If $\mathcal{A}_i$ are realizability candidates for all $i \in I$, then*

$$\Pi_{i=1}^{n} l_i{:}\,\mathcal{A}_i = (\overline{\Pi}_{i=1}^{n}\overline{l_i}{:}\,\mathcal{A}_i^{\perp})^{\perp}$$

4 Lattices and matrices

The main tool for the typing-checking algorithm are matrices with coefficients given in two *lattices*. The first lattice expresses the relation between *types*: it as four elements. But what is a type ... In fact, types will be just names (so we should say *type names*) that we will interpret using realizability candidates in the next section. Realizability candidates being sets, the possible relation between type names are:

- No relation is known : denoted 0
- $\alpha \subset \beta$ is known to hold : denoted i.
- $\alpha \supset \beta$ is known to hold : denoted i^*.
- $\alpha = \beta$ is known to hold : denoted 1.

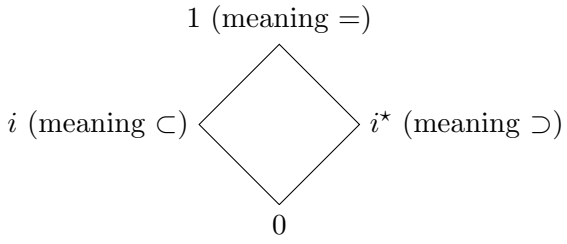

Figure 3: The lattice $\mathcal{B}$

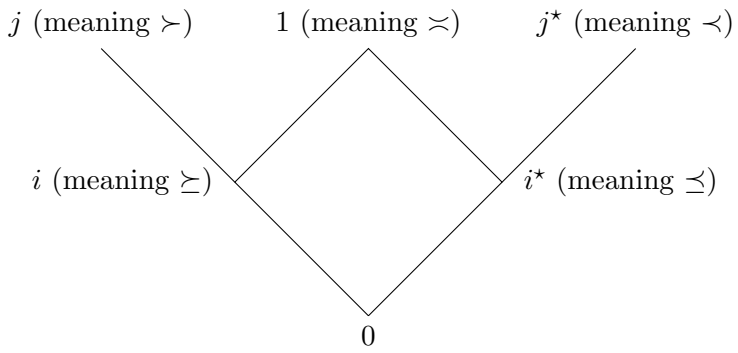

Figure 4: The lattice $\mathcal{B}'$

This means we have a set $\mathcal{B} = \{0, i, i^*, 1\}$ which can be given a lattice structure with $0 \leq i \leq 1$ and $0 \leq i^* \leq 1$ (see figure 3). This lattice is isomorphic to $(\mathbb{Z}/2\mathbb{Z})^2$ and we can define on it supremum (denoted $x \vee y$), infimum (denoted $x \wedge y$ or $x.y$), subtraction (denoted $x - y$ and defined as the smallest element such that $(x - y) \vee y = x \vee y$).

We will also need the *adjoin operation* denoted x^* exchanging i and i^* and keeping 0 and 1 unchanged.

We use a second inf-lattice $\mathcal{B}' = \{0, i, i^*, j, j^*, 1\} \supset \mathcal{B}$ whose lattice structure is given by figure 4. We see on this figure that not all supremum are defined in $\mathcal{B}'$. This lattice will be used to construct an ordering between type names that we will use to define the interpretation of types by induction on this order. Here are the intended meaning for the element of $\mathcal{B}'$:

- i represents $\alpha \preceq \beta$: α must be defined before or at the same time as β.
- i^* represents $\alpha \succeq \beta$: α must be defined after or at the same time as β.
- j represents $\alpha \prec \beta$: α must be defined before β.
- j^* represents $\alpha \succ \beta$: α must be defined after β.
- 1 represents $\alpha \asymp \beta$: α and β must be defined simultaneously.
- 0 represents the absence of constraints between the definitions of α and β.

With the lattice $\mathcal{B}'$, the product $x.y$ do not coincide with the infimum. The main property of the product is that it expresses transitivity. If x denotes a relation known to hold between α and β and y denotes a relation known to hold between β and γ, then, the relation denoted by $x.y$ is known to hold between α and γ.

Using this intuition, we define the product on $\mathcal{B}'$ as follows $j.1 = j \neq i = j \wedge 1$, $j^*.1 = j^* \neq i^* = j^* \wedge 1$, $j.i = j \neq i = j \wedge i$ and $j^*.i^* = j^* \neq i^* = j^* \wedge i^*$. In all other cases, we have $j.i = j \wedge i$.

Remark: The structure of $(\mathcal{B}, ., \vee)$ is very similar to the structure of tropical semiring which was introduced by Simon (a computer scientist from Brasil, hence the name) and which is now used in algebraic geometry. The only difference is that in tropical semiring, $(\mathcal{B}, .)$ is a group and here we just have a monoid.

The structure of $\mathcal{B}$ and $\mathcal{B}'$ is characterized by the following definitions:

Definition 16 (*relation lattice*) *$(\mathcal{B}, \leq, 0, 1, ., x \mapsto x^*)$ is a relational (resp. partial relational) lattice if*

- *$(\mathcal{B}, \leq)$ is a lattice (resp. an inf-lattice) with 0 as its bottom element.*
- *$.$ is a binary increasing operation which is commutative, associative with 1 as neutral element and distributing over sup when they are defined.*
- *$x \mapsto x^*$ is an idempotent and increasing unary operator. Moreover, $(x.y)^* = x^*.y^*$ and $x.x = x$.*
- *$x-y$, the least element such that $x \vee y = (x-y) \vee y$ satisfies $(x-y) \wedge y = 0$.*

Lemma 17 *The lattices $\mathcal{B}$ and (resp. inf-lattice $\mathcal{B}'$) are relational (resp. partial relational) lattice.*

Proof: Just a boring check on the table defining the lattices operations. ■

Here is the code for the basic operation on both lattice $\mathcal{B}$ and $\mathcal{B}'$ (we do not give separate implementation for both lattices):

```
1  type relation = Nothing | Leq | Le | Geq | Ge | Equal
2
3  exception SupUndefined
4
5  let sup a b = match a, b with
6  | Nothing, x | x, Nothing → x
7  | Leq, Le  | Le, Leq → Le
8  | Geq, Ge  | Ge, Geq → Ge
9  | Le, Equal | Equal, Le | Ge, Equal | Equal, Ge
10   | Ge, Le | Le, Ge | Le, Geq | Geq, Le
11   | Ge, Leq | Leq, Ge → raise SupUndefined
12 | Le, Le → Le | Ge, Ge → Ge
13 | Equal, x | x, Equal → Equal
14 | Leq, Geq | Geq, Leq → Equal
15 | Leq, Leq → Leq | Geq, Geq → Geq
16
17 let inf a b = match a, b with
```

```
18  | Nothing, x | x, Nothing → Nothing
19  | Le, Equal | Equal, Le | Le, Leq | Leq, Le → Leq
20  | Ge, Equal | Equal, Ge | Ge, Geq | Geq, Ge → Geq
21  | Le, Le → Le | Ge, Ge → Ge
22  | Equal, x | x, Equal → x
23  | Geq, Le | Le, Geq | Ge, Leq | Leq, Ge → Nothing
24  | Leq, Geq | Geq, Leq | Ge, Le | Le, Ge → Nothing
25  | Leq, Leq → Leq | Geq, Geq → Geq
26
27  let leq a b = inf a b = a
28  let le a b = a <> a && leq a b
29
30  let sub a b = match a, b with
31  | Equal, Leq → Geq | Equal, Geq → Leq
32  | Le, Leq → Le | Ge, Geq → Ge
33  | x, Nothing → x | _ → Nothing
34
35  let product a b = match a, b with
36  | Le, Equal | Equal, Le | Le, Leq | Leq, Le → Le
37  | Ge, Equal | Equal, Ge | Ge, Geq | Geq, Ge → Ge
38  | _ → inf a b
39
40  let adjoin a = match a with
41  | Le → Ge | Ge → Le
42  | Leq → Geq | Geq → Leq | x → x
```

Listing 2: definition of the lattices in lattice.ml

We will need to manipulate vectors and matrices with coefficients in the above lattices. We will mainly use sparse matrices and represent vector as list of pairs with the index and the coefficient, ordered by index. A representation as a map table with $O(\ln n)$ insertion and deletion would be better ... Then, matrices are just vectors of vectors:

```
1  open Lattice
2
3  type ('index, 'coef) vector = ('index * 'coef) list
4          (* ordered by index *)
5
6  type ('index1, 'index2, 'coef) matrix =
7    ('index1, ('index2, 'coef) vector) vector
```

Listing 3: vectors in matrix.ml

Now, we also need some operations on matrices and vectors like ordering (which is point-wise ordering), supremum (which is point-wise supremum) and multiplication defined as usual with product and supremum taking the place of addition. Here are some formal definitions (for these definitions, we just consider that vectors are column or line matrices and just give the definitions for matrices):

Definition 18 *Let $A = (A_{i,j})_{i\in I, j\in J}$, $A' = (A'_{i,j})_{i\in I, j\in J}$ and $B = (B_{j,k})_{j\in J, k\in K}$ be three matrices with index in the finite sets I, J, K and coefficients in a relational lattice (partial or total). Then*

- *$A \leq A'$ is true if for all $i \in I, j \in J$ we have $A_{i,j} \leq A'_{i,j}$.*

- $A \vee A'$ *is the matrix* $(A_{i,j} \vee A'_{i,j})_{i\in I, j\in J}$.
- $A - A'$ *is the matrix* $(A_{i,j} - A'_{i,j})_{i\in I, j\in J}$.
- $A.B = (C_{i,k})_{i\in I, k\in K}$ *with* $C_{i,k} = \bigwedge_{j\in J} A_{i,j}.B_{j,k}$.
- $A^* = (a^*_{j,i})_{j\in J, i\in I}$.
- *We will say that a square matrix* A *is an hermitian matrix if* $A = A^*$. *Remark: an hermitian matrix can only have the coefficients* 0 *and* 1 *on the diagonal.*

To define all operations on vectors and matrices we first define a *fold* operation on two vectors. It take four other arguments: three functions (called respectively when an index is bound in the first, second of both vectors) and the initial accumulator

Exercise 19 *Understand the code below.*

```
let fold2_vector
    : ('a → 'i → 'b → 'c → 'a) →
      ('a → 'i → 'b → 'a) → ('a → 'i → 'c → 'a)  →
      'a → ('i,'b) vector → ('i,'c) vector → 'a
  = fun bn ln rn a vector1 vector2 →
   let rec fn acc v1 v2 = match v1, v2 with
   | v1, [] →
       List.fold_left (fun acc (i,c) → ln acc i c) acc v1
   | [], v2 →
       List.fold_left (fun acc (i,c) → rn acc i c) acc v2
   | (i, c)::v1, (i', c')::v2 when i = i' →
       fn (bn acc i c c') v1 v2
   | (i, c)::v1, (i', c')::_ when i < i' →
       fn (ln acc i c) v1 v2
   | (i, c)::_ , (i', c')::v2 (*when i > i'*) →
       fn (rn acc i' c') v1 v2
   in fn a vector1 vector2
```

Listing 4: fold on vectors in matrix.ml

Using this folding function, we implement all the needed operations on vectors and matrices:

```
let sum_vector v1 v2 = List.rev (fold2_vector
  (fun acc i c c' → (i, sup c c')::acc)
  (fun acc i c → (i, c)::acc) (fun acc i c → (i, c)::acc)
  [] v1 v2)

let sub_vector v1 v2 = List.rev (fold2_vector
  (fun acc i c c' →
    let c'' = sub c c' in
    if c'' = Nothing then acc else (i, c'')::acc)
  (fun acc i c → (i, c)::acc) (fun acc i c → acc)
  [] v1 v2)

let sum_matrix m1 m2 =
  List.rev (fold2_vector
```

```
15      (fun acc i v1 v2 → (i, sum_vector v1 v2)::acc)
16      (fun acc i v → (i,v)::acc) (fun acc i v → (i,v)::acc)
17      [] m1 m2)
18
19  let sub_matrix m1 m2 =
20    List.rev (fold2_vector
21      (fun acc i v1 v2 →
22         let v = sub_vector v1 v2 in
23         if v = [] then acc else (i,v)::acc)
24      (fun acc i v → (i,v)::acc) (fun acc i v → acc)
25      [] m1 m2)
26
27  let dot_product v1 v2 = fold2_vector
28    (fun acc i c c' → sup acc (product c (adjoin c')))
29    (fun acc i c → acc) (fun acc i c → acc)
30    Nothing v1 v2
31
32  let matrix_product m1 m2 =
33    List.filter (fun (_, r) → r <> [])
34      (List.map (fun (i,v1) → i,
35        List.filter (fun (_, r) → r <> Nothing)
36          (List.map (fun (j,v2) → j, dot_product v1 v2) m2)) m1)
```

Listing 5: matrix and vector operations in matrix.ml

We introduce some notations for hermitian matrices with coefficients in the lattice $\mathcal{B}$ or $\mathcal{B}'$. Let $A = (a_{\alpha,\beta})_{\alpha,\beta\in I}$

- $A \models \alpha \subset \beta$ iff $a_{\alpha,\beta} \geq i$ (when A has coefficients in $\mathcal{B}$)
- $A \models \alpha \supset \beta$ iff $a_{\alpha,\beta} \geq i^*$ (when A has coefficients in $\mathcal{B}$)
- $A \models \alpha = \beta$ iff $a_{\alpha,\beta} \geq 1$ (when A has coefficients in $\mathcal{B}$)
- $A \models \alpha \preceq \beta$ iff $a_{\alpha,\beta} \geq i$ (when A has coefficients in $\mathcal{B}'$)
- $A \models \alpha \succeq \beta$ iff $a_{\alpha,\beta} \geq i^*$ (when A has coefficients in $\mathcal{B}'$)
- $A \models \alpha \asymp \beta$ iff $a_{\alpha,\beta} \geq 1$ (when A has coefficients in $\mathcal{B}'$)
- $A \models \alpha \prec \beta$ iff $a_{\alpha,\beta} \geq j$ (when A has coefficients in $\mathcal{B}'$)
- $A \models \alpha \succ \beta$ iff $a_{\alpha,\beta} \geq j^*$ (when A has coefficients in $\mathcal{B}'$)

Lemma 20 *Using the definition of multiplication and the above notation, we can give an equivalent to the definition of Hermitian matrix multiplication: if A, B are Hermitian matrices with coefficienst in $\mathcal{B}$. Then, A.B in the smallest matrix such that*

$$A \models \alpha \subset \beta \text{ and } B \models \beta \subset \gamma \text{ implies } A.B \models \alpha \subset \gamma$$

Proof: Immediate. ■

Exercise 21 *If A and B are hermitian matrices with coefficients in $\mathcal{B}'$ write a similar lemma with four cases, because you have to use $\preceq$ and $\prec$.*

Definition 22 (*multi-vector*) *A multi-vector is a vector where the same indices can be bound several times (to different values). Another way to say this is that a multi-vector with coefficients in a set X is a vector with coefficients in $\mathcal{P}(X)$. This is why when C is a multi-vector, we will write $c \in C_i$ and not $C_i = c$*

For the implementation, we define multi-vectors as vectors of lists:

```
1 type ('index, 'coef) multi_vector = ('index, 'coef list) vector
```

Listing 6: matrix and vector operations in matrix.ml

5 Typing constraints

The first (trivial) step of the typing algorithm is to *decorate* programs with type names.

We consider that $\mathcal{T}$ is a countable set of such names and each subprograms is decorated with a different name in $\mathcal{T}$ except for variables: all occurrences of the same variable are decorated with the same name.

For the implementation, we use as names the positions given by the parser, keeping only the position of the binding occurrence for variables.

Here is the decorated version of the term omega :

$$((\texttt{fun}\ x^{\alpha_0} \to (x\ x)^{\rho_0})^{\varphi_0}\ \ (\texttt{fun}\ x^{\alpha_1} \to (x\ x)^{\rho_1})^{\varphi_1})^{\rho_2}$$

and the decorated version of the halving an even number, applied to unary representation of three (this program will fails):

$$\begin{array}{l}((\texttt{fix}\ \mathit{half}^{\alpha_1} \to \\ \quad (\texttt{fun}\ n^{\alpha_0} \to \\ \quad\quad (\texttt{case}\ n\ \texttt{of} \\ \quad\quad\quad \mid \mathbf{Z}[^{\beta_0}] \to \mathbf{Z}[^{\gamma_0}]^{\sigma_0} \\ \quad\quad\quad \mid \mathbf{S}[n'^{\beta_2}] \to (\texttt{case}\ n'\ \texttt{of}\ \mid \mathbf{S}[n''^{\beta_1}] \to \mathbf{S}[(\mathit{half}\ n'')^{\rho_0}]^{\sigma_1})^{\chi_0})^{\chi_1})^{\varphi_0})^{\varphi_1} \\ \mathbf{S}[\mathbf{S}[\mathbf{S}[\mathbf{Z}[^{\gamma_1}]^{\sigma_2}]^{\sigma_3}]^{\sigma_4}]^{\sigma_5})^{\rho_1}\end{array}$$

Definition 23 *The typing constraints extracted from a program consist in a triplet (C, R, D) where:*

- *C, the constructor constraints, is a vector and D, the destructor constraints, a multi-vector with index in $\mathcal{T}$ and coefficients in the disjoin sum of the following sets, using specific notations for an intuitive reading:*
  - *$\mathcal{T}^2$ writing those pairs $\alpha \to \beta$*
  - *Finite maps from $\mathcal{V}_\Sigma$ to $\mathcal{T}$, writing the map associating α_i to $\boldsymbol{C}_i$ for $i \in I$ as $\Sigma_{i\in I}\ \boldsymbol{C}_i[\alpha_i]$ or $\boldsymbol{C}_1[\alpha_1] + \boldsymbol{C}_2[\alpha_2] + \ldots$*
  - *Finite maps from $\mathcal{V}_\Pi$ to $\mathcal{T}$, writing the map associating the α_i to l_i for $i \in I$ as $\Pi_{i\in I}\ l_i : \alpha_i$ or $l_1 : \alpha_1 \times l_2 : \alpha_2 \times \ldots$*

- *R, the relational constraints, is an hermitian matrix with only* 1 *on the diagonal (recall that in general there could be* 0 *or* 1 *on the diagonal of such a matrix).*

Here C represents the constraints coming from data constructors in the code, D from data destructors and R the other constraints (coming only from the fixpoints and the right-hand side of case analysis). D is a multi-vector, because the same variable can be *destroyed* (applied to an argument for instance) more than once. This will be clear in definition 24

We introduce the following notations for constraints (similar to those introduced for matrices in the previous section):

- $(C, R, D) \models \alpha \subset \beta$ iff $R \models \alpha \subset \beta$
- $(C, R, D) \models \alpha \supset \beta$ iff $R \models \alpha \supset \beta$
- $(C, R, D) \models \alpha = \beta$ iff $R \models \alpha = \beta$
- $(C, R, D) \models \alpha \to \beta \subset \gamma$ iff $C_\gamma = \alpha \to \beta$
- $(C, R, D) \models \gamma \subset \alpha \to \beta$ iff $\alpha \to \beta \in D_\gamma$
- $(C, R, D) \models \Sigma_{i \in I}\, c_i[\alpha_i] \subset \gamma$ iff $\mathbf{C}_\gamma = \Sigma_{i \in I}\, \mathbf{C}_i[\alpha_i]$
- $(C, R, D) \models \gamma \subset \Sigma_{i \in I}\, \mathbf{C}_i[\alpha_i]$ iff $\Sigma_{i \in I}\, \mathbf{C}_i[\alpha_i] \in D_\gamma$
- $(C, R, D) \models \Pi_{i \in I}\, c_i : \alpha_i \subset \gamma$ iff $C_\gamma = \Pi_{i \in I}\, l_i : \alpha_i$
- $(C, R, D) \models \gamma \subset \Pi_{i \in I}\, l_i : \alpha_i :$ iff $\Pi_{i \in I}\, l_i : \alpha_i \in D_\gamma$

Here is the code to define data types for constraints (the triplet (C, R, D)). It does not exactly obey the above definition because we use the fact that the index set I above will always be a singleton for Σ in C and Π in D.

```
1  open ParserUtil
2  open Lattice
3  open Matrix
4  open Program
5
6  type constructor =
7  | CSum of cid * position
8  | CPro of (lid * position) list
9  | CFun of position * position
10
11 type destructor =
12 | DSum of (cid * position) list
13 | DPro of lid * position
14 | DFun of position * position
15
16 type constraints =
17   { constructors : (position, constructor) vector;
18     relations : (position, position, relation) matrix;
19     destructors  : (position, destructor) multi_vector; }
```

Listing 7: typing indices in typing.ml

Definition 24 (*extracted constraints*) *The initial typing constraints $K_0 = (C, R_0, D)$, extracted from an annotated program, are the smallest constraints (ordered point-wise for the matrix R_0 and by set inclusion for the vector C and multi-vector D, regarded as set of pairs) verifying:*

- *If $(M^{\varphi}\, N^{\alpha})^{\rho}$ is a sub-term of the program then $K_0 \models \varphi \subset \alpha \to \rho$*
- *If $(\mathtt{fun}\ x^{\alpha} \to M^{\rho})^{\varphi}$ is a sub-term of the program then $K_0 \models \alpha \to \rho \subset \varphi$.*
- *If $(\mathtt{fix}\ x^{\alpha} \to M^{\varphi})^{\rho}$ is a sub-term of the program then $K_0 \models \varphi \subset \alpha$ and $K_0 \models \varphi \subset \rho$*
- *If $\boldsymbol{C}[M^{\beta}]^{\sigma}$ is a sub-term of the program then $K_0 \models \boldsymbol{C}[\beta] \subset \rho$*
- *If $(\mathtt{case}\ M^{\sigma}\ \mathtt{of}\ (\boldsymbol{C}_i[x^{\beta_i}] \to N_i^{\rho_i})_{i\in I})^{\rho}$ is a sub-term or the program, then for all $i \in I$, $K_0 \models \rho_i \subset \rho$ and $K_0 \models \sigma \subset \Sigma_{i\in I}\, \boldsymbol{C}_i[\beta_i]$*
- *If $\{(l_i = M^{\rho_i})_{i\in I}\}^{\gamma}$ is a sub-term of the program then $K_0 \models \Pi_{i\in I}\, l_i : \rho_i \subset \gamma$*
- *If $(M^{\gamma}.l)^{\rho}$ is a sub-term or the program then $K_0 \models \gamma \subset l : \rho$*

Here are the initial constraints for the term omega, using the annotation given above:

$$
\begin{array}{l}
C = \{\alpha_0 \to \rho_0 \subset \varphi_0,\ \ \alpha_1 \to \rho_1 \subset \varphi_1\} \\
R_0 = \{\} \\
D = \{\alpha_0 \subset \alpha_0 \to \rho_0,\ \ \varphi_0 \subset \varphi_1 \to \rho_2,\ \ \alpha_1 \subset \alpha_1 \to \rho_1\}
\end{array}
$$

and those for halving applied to the unary representation of the natural number three:

$$
\begin{array}{l}
C = \{\alpha_0 \to \chi_1 \subset \varphi_0,\ \ \mathbf{Z}[\gamma_0] \subset \sigma_0,\ \ \subset \gamma_0,\ \ \mathbf{S}[\rho_0] \subset \sigma_1, \\
\qquad \mathbf{S}[\sigma_4] \subset \sigma_5,\ \ \mathbf{S}[\sigma_3] \subset \sigma_4,\ \ \mathbf{S}[\sigma_2] \subset \sigma_3, \\
\qquad \mathbf{Z}[\gamma_1] \subset \sigma_2,\ \ \subset \gamma_1\} \\
R_0 = \{\varphi_0 \subset \varphi_1,\ \ \varphi_0 \subset \alpha_1,\ \ \sigma_0 \subset \chi_1,\ \ \chi_0 \subset \chi_1,\ \ \sigma_1 \subset \chi_0\} \\
D = \{\varphi_1 \subset \sigma_5 \to \rho_1,\ \ \alpha_1 \subset \beta_1 \to \rho_0,\ \ \alpha_0 \subset \mathbf{Z}[\beta_0]\ \ +\ \ \mathbf{S}[\beta_2], \\
\qquad \beta_2 \subset \mathbf{S}[\beta_1]\}
\end{array}
$$

Definition 25 (*interpretation*) *An interpretation satisfying the typing constraints $K_0 = (C, R_0, D)$ is a function $\alpha \mapsto ||\alpha||$, associating realizability candidates to type names appearing in K_0 satisfying the conditions given below. Moreover, we will write $|\alpha|$ for $||\alpha||^{\perp}$. Here are the conditions:*

- *If $K_0 \models \varphi \subset \alpha$ then $||\varphi|| \subset ||\rho||$*
- *If $K_0 \models \varphi \subset \alpha \to \rho$ then $||\varphi|| \subset ||\alpha|| \to ||\rho||$*
- *If $K_0 \models \alpha \to \rho \subset \varphi$ then $||\alpha|| \to ||\rho|| \subset ||\varphi||$*
- *If $K_0 \models \Sigma_{i\in I}\, \boldsymbol{C}_i[\beta_i] \subset \rho$ then $\Sigma_{i\in I}\, \boldsymbol{C}_i[||\beta_i||] \subset ||\rho||$*
- *If $K_0 \models \rho \subset \Sigma_{i\in I}\, \boldsymbol{C}_i[\beta_i]$ then $||\rho|| \subset \Sigma_{i\in I}\, \boldsymbol{C}_i[||\beta_i||]$*

- *If* $K_0 \models \Pi_{1\leq i\leq n}\, l_i : \rho_i \subset \gamma$ *then* $\Pi_{1\leq i\leq n}\, l_i : ||\rho_i|| \subset ||\gamma||$
- *If* $K_0 \models \gamma \subset \Pi_{1\leq i\leq n}\, l_i : \rho_i$ *then* $||\gamma|| \subset \Pi_{1\leq i\leq n}\, l_i : ||\rho_i||$

Theorem 26 (*adequation lemma*) *If there is an interpretation for an annotated program M satisfying the typing constraints K_0 extracted from M, then $M \in \perp\!\!\!\perp$ which implies that M is safe and that all infinite reductions use the fixpoint rule infinitely often.*

Proof: Let M be an annotated program, (C, R_0, D) be the constraints extracted from M and assume that we have an interpretation satisfying these constraints. If N is a subterm of M we write $\Gamma(N) = x_1^{\alpha_1}, \ldots, x_n^{\alpha_n}$ the set of free variables that are bound above N with their type annotation (all the free variables of N are mentioned in $\Gamma(N)$). We say that a substitution $\sigma = [(x_i \leftarrow u_i)_{i\in\{1,\ldots,n\}}]$ satisfies $\Gamma(N)$ if for all $1 \leq i \leq n$ we have $u_i \in ||\alpha_i||$.

Next, we show by induction on the structure of M, that for all subterm N^β, if σ is a substitution satifying $\Gamma(N)$, then $N\sigma \in ||\beta||$.

- If $N^\beta = x^\beta$ the result is immediate from the definition of substitution satisfying $\Gamma(x^\beta)$.
- If $N^\beta = N_1^\gamma\, N_2^\phi$, we know that $||\gamma|| \subset ||\phi|| \to ||\beta||$ and by induction hypothesis, we have $N_1\sigma \in ||\gamma||$ and $N_2\sigma \in ||\phi||$. Let us choose $E \in |\beta|$, we must prove $N \star E \in \perp\!\!\!\perp$. We have $N\sigma \star E = N_1\sigma \star E[\chi \leftarrow \chi\, N_2\sigma]$, $E[\chi \leftarrow \chi\, N_2\sigma] \in ||\phi||.|\beta|$ and $N_1\sigma \in ||\phi|| \to ||\beta|| = (||\phi||.|\beta|)^\perp$ which gives the wanted result.
- If $N^\beta = (\texttt{fun}\ y^\phi \to N_1^\gamma)^\beta$, we have $||\phi|| \to ||\gamma|| \subset ||\beta||$ and by induction hypothesis, for σ' satisfying $\Gamma(N_1)$, we have $N_1\sigma' \in ||\gamma||$. Therefore, it is enough to choose $E \in ||\phi||.|\gamma|$ and show that $N\sigma \star E \in \perp\!\!\!\perp$ (this gives $N\sigma \in (||\phi||.|\gamma|)^\perp = ||\phi|| \to ||\gamma|| \subset ||\beta||$). From $E \in ||\phi||.|\gamma|$ we deduce that $E = E'[\chi \leftarrow \chi\, P]$ with $P \in ||\phi||$ and $E' \in |\gamma|$. We have $N\sigma \star E = E'[\chi \leftarrow (\texttt{fun}\ y \to N_1)\sigma\, P]$. Up to a renaming of y, we assume that y is not free in the domain and images of σ and we have $E'[\chi \leftarrow (\texttt{fun}\ y \to N_1)\sigma\, P] = E'[\chi \leftarrow (\texttt{fun}\ y \to N_1\sigma)\, P] \succ_\beta N_1\sigma[y \leftarrow P] \star E'$. Then, by the definition of saturated sets and because $P \in ||\phi|| \subset \perp\!\!\!\perp$, we just need to prove that $N_1\sigma[y \leftarrow P] \in ||\gamma||$, which is immediate because $P \in ||\phi||$ implies that $\sigma \circ [y \leftarrow P]$ is a substitution satisfying $\Gamma(N_1) = y^\phi, \Gamma(N)$.
- If $N^\beta = (\texttt{fix}\ x^\phi \to N_1^\gamma)^\beta$, we have $||\gamma|| \subset ||\phi||$ and $||\gamma|| \subset ||\beta||$. As in the previous case, we assume x not to be free in the domain nor the images of σ. We prove by induction that for all $n \in \mathbb{N}$ we have $(N_1\sigma)^n(x) \in ||\gamma||$. If $n = 0$, this is immediate because $\sigma' = \sigma \circ [x \leftarrow x]$ satisfies $\Gamma(N_1)$ because $x \in \perp\!\!\!\perp_0 \subset ||\phi||$. For the induction case, we assume $(N_1\sigma)^n(x) \in ||\gamma||$ and prove $(N_1\sigma)^{n+1}(x) \in ||\gamma||$. The hypothesis $||\gamma|| \subset ||\phi||$ implies that $\sigma \circ [x \leftarrow (N_1\sigma)^n(x)]$ is a substitution satisfying $\Gamma(N_1)$. Then, let $E \in |\gamma|$, we have $(N_1\sigma)^{n+1}(x) \star E = N_1\sigma[x \leftarrow (N_1\sigma)^n(x)] \star E \in \perp\!\!\!\perp$ by the induction hypothesis on N_1.

Finally, by definition of saturated set, we know that $(N_1\sigma)^n(x) \star E \in \perp\!\!\!\perp$ for all $n \in \mathbb{N}$ implies $(\texttt{fix}\ x \to N_1\sigma) = N\sigma \star E \in \perp\!\!\!\perp$ which is what we wanted (we do have $N_1\sigma \in ||\gamma|| \subset \perp\!\!\!\perp$).

- If $N^\beta = \mathbf{C}[N_1^\phi]^\beta$, we have $\mathbf{C}[||\phi||] = (\overline{\mathbf{C}}[|\phi|])^\perp \subset ||\beta||$. Let $E \in \overline{\mathbf{C}}[|\phi|]$ be, which means that E can be written $(\texttt{case}\ \chi\ \texttt{of}\ \mathbf{C}[\chi] \to E_1) \star E'$ where $E_1 \star E' \in ||\phi||^\perp$. By induction hypothesis we deduce $N_1\sigma \star (E_1 \star E') \in \perp\!\!\!\perp$ which implies by saturation that $N \star E \in \perp\!\!\!\perp$.

- If $N^\beta = \texttt{case}\ P^\gamma\ \texttt{of}\ (\mathbf{C}_i[x_i^{\phi_i}] \to N_i^{\beta_i})_{i\in I}$, we have for all $i \in I$, $||\beta_i|| \subset ||\beta||$ and $||\gamma|| \subset \Sigma_{i\in I}\mathbf{C}_i[||\phi_i||]$. Let E be a context in $|\beta|$. The induction hypothesis and $||\beta_i|| \subset ||\beta||$ implies that for any term $Q_i \in ||\phi_i||$, we have $N_i\sigma[x_i \leftarrow Q_i] \star E = Q_i \star (N_i\sigma[x_i \leftarrow \chi] \star E) \in \perp\!\!\!\perp$ for all $i \in I$. This means that $N_i\sigma[x_i \leftarrow \chi] \star E \in ||\phi_i||^\perp = |\phi_i|$. Thus by definition of $\overline{\Sigma}$, this means that $(\texttt{case}\ \chi\ \texttt{of}\ (\mathbf{C}_i[x_i] \to N_i\sigma)_{i\in I}) \star E \in \overline{\Sigma}_{i\in I}\overline{\mathbf{C}_i}[|\phi_i|] \subset ||\gamma||^\perp$ which gives, together with $P\sigma \in ||\gamma||$, $N\sigma \star E = P\sigma \star (\texttt{case}\ \chi\ \texttt{of}\ (\mathbf{C}_i[x_i] \to N_i\sigma)_{i\in I}) \star E \in \perp\!\!\!\perp$.

- If $N^\beta = \{(l_i = N_i^{\gamma_i})_{i\in I}\}^\beta$, we have $\Pi_{i\in I} l_i : ||\gamma_i|| \subset ||\beta||$ and by induction hypothesis, $N_i\sigma \in ||\gamma_i||$. Let $E \in \overline{\Pi}_{i\in I}\overline{l_i} : |\gamma_i|$, E can be written $\chi.l_j \star E_j$ for some j such that $E_j \in |\gamma_j|$. Therefore, $N_j\sigma \star E_j \in \perp\!\!\!\perp$ which implies by saturation that $(N\sigma).l_j \star E_j = N\sigma \star E \in \perp\!\!\!\perp$.

- If $N^\beta = (N_1^\phi.l)^\beta$, we have $||\phi|| \subset l : ||\beta||$ and, by induction hypothesis, $N_i\sigma \in ||\phi||$. Let $E \in |\beta|$, by definition of $\overline{\Pi}$, we have $E[\chi \leftarrow \chi.l] \in \overline{\Pi}\,\overline{l} : |\beta| \subset |\phi|$. Therefore, $N\sigma \star E = N_i\sigma \star E[\chi \leftarrow \chi.l] \in \perp\!\!\!\perp$. ■

Here is the code extracting the constraints from an annotated program. It starts with a few functions adding constraints in the triplet (C, R_0, D), not forgetting to ensure that R_0 has only 1 on its diagonal:

```
let add_constructor info i c =
  { info with
    constructors = List.rev (fold2_vector
       (fun acc i c c' → assert false)
       (fun acc i c → (i, c)::acc)
       (fun acc i c → (i, c)::acc)
       [] info.constructors [i,c]);
    relations = sum_matrix info.relations [i, [i, Equal]]; }

let add_destructor info i d =
  { info with
    destructors = List.rev (fold2_vector
      (fun acc i c c' → (i, c @ c')::acc)
      (fun acc i c → (i, c)::acc)
      (fun acc i c → (i, c)::acc)
      [] info.destructors [i,[d]]);
    relations = sum_matrix info.relations [i, [i, Equal]]; }

let add_coef m i1 i2 =
  sum_matrix m
    (sum_matrix
      (sum_matrix [i1, [i1, Equal]] [i2, [i2, Equal]])
```

```
23        (sum_matrix [i1, [i2, Leq]] [i2, [i1, Geq]]))
24
25  let add_relation info i1 i2 =
26    { info with relations = add_coef info.relations i1 i2 }
```

Listing 8: adding to constraints in typing.ml

Then, we have a small function extracting the annotation of the root of a program:

```
1  let get_index env p = match p with
2  | App(_,_,idx) | Fun(_,_,_,idx) | Fix(_,_,_,idx) | Cst(_,_,idx)
3  | Cas(_,_,idx) | Rec(_,idx) | Pi(_,_,idx) → idx
4  | Var(name) →
5      try List.assoc name env
6      with Not_found →
7        Printf.fprintf stderr "Unbound identifier %s\n" name; exit 1
```

Listing 9: extracting the root annotation in typing.ml

Now, we have the extraction of constraints itself. It uses an environment `env` to store the decoration of free variables:

```
1  let extract_matrices (p : position program) =
2    let rec fn infos env p = match p with
3      | Var(name) → infos
4      | Fun(name, arg, p, all) →
5          let env = (name, arg)::env in
6          let infos = add_constructor infos all
7            (CFun(arg, get_index env p)) in
8          fn infos env p
9      | Fix(name, arg, p, all) →
10         let res = get_index env p in
11         let infos = add_relation infos res arg in
12         let infos = add_relation infos res all in
13         let env = (name, arg)::env in
14         fn infos env p
15     | App(p1, p2, res) →
16         let infos = add_destructor infos (get_index env p1)
17           (DFun(get_index env p2, res)) in
18         fn  (fn infos env p1) env p2
19     | Cst(cid,p,all) →
20         let infos = add_constructor infos all
21           (CSum(cid, get_index env p)) in
22         fn infos env p
23     | Cas(p, cases, res) →
24         let matched = get_index env p in
25         let infos = fn infos env p in
26         let infos =
27           add_destructor infos matched
28             (DSum(List.map (fun (cid,_,_,arg) → cid,arg) cases))
29         in
30         List.fold_left (fun infos (cid, name, p', arg) →
31           let env = (name,arg)::env in
32           let infos = add_relation infos (get_index env p') res in
33           fn infos env p') infos cases
34     | Pi(p,lid,res) →
35         let all = get_index env p in
36         let infos =
```

```
37            add_destructor infos all (DPro (lid, res)) in
38          fn infos env p
39      | Rec(record, all) →
40          let _, infos, ll =
41            List.fold_left (fun (env,infos,ll) (lid, p) →
42              let arg = get_index env p in
43              let ll = (lid,arg)::ll in
44              let infos = fn infos env p in
45              let env = (lid,arg)::env in
46              env, infos, ll) (env, infos, []) record
47           in
48           add_constructor infos all (CPro ll)
49    in fn { constructors = []; destructors = []; relations = []} [] p
```

Listing 10: constraints extraction in typing.ml

6 Checking the constraints

The first step when checking the constraints is saturation. To define it, we need to define a new product $C.R.D$ where C are constructor constraints, R is a matrix with coefficient in $\mathcal{B}$ and D are destructor constraints. This product is partial and the fact that it is undefined will be a *type error*. Here is the definition:

Definition 27 (*ternary product*) *$R' = C.R.D$ is the smallest hermitian matrix for lattice ordering such that:*

- *If $(C, R, D) \models \alpha \to \beta \subset \gamma$, $(C, R, D) \models \gamma \subset \gamma'$ and $(C, R, D) \models \gamma' \subset \alpha' \to \beta'$ then $R' \models \alpha' \subset \alpha$ and $R' \models \beta \subset \beta'$*
- *If $(C, R, D) \models \Sigma_{i\in I}\, \boldsymbol{C}_i[\alpha_i] \subset \gamma$, $(C, R, D) \models \gamma \subset \gamma'$ and $(C, R, D) \models \gamma' \subset \Sigma_{j\in J}\, \boldsymbol{C}_j[\alpha'_j]$ then, if $I \subset J$, for all $i \in I$ $R' \models \alpha_i \subset \alpha'_i$. If $I \not\subset J$, then the product is undefined (here we assume that $\boldsymbol{C}_i = \boldsymbol{C}_j$ implies $i = j$).*
- *If $(C, R, D) \models \Pi_{j\in J}\, l_j : \alpha_j \subset \gamma$, $(C, R, D) \models \gamma \subset \gamma'$ and $(C, R, D) \models \gamma' \subset \Pi_{i\in I}\, l_i : \alpha'_i$ then, if $I \subset J$, for all $i \in I$ $R' \models \alpha_i \subset \alpha'_i$. If $I \not\subset J$, then the product is undefined (here we assume that $l_i = l_j$ implies $i = j$).*
- *If $(C, R, D) \models L \subset \gamma$, $(C, R, D) \models \gamma \subset \gamma'$ and $(C, R, D) \models \gamma' \subset R$ and (L, R) matches one of the following pairs: $(\Sigma_{\_}, \_ \to \_), (\Pi_{\_}, \_ \to \_), (\_ \to \_, \Sigma), (\_ \to \_, \Pi), (\Sigma_{\_}, \Pi_{\_})$, or $(\Pi_{\_}, \Sigma_{\_})$ then $R' = C.R.D$ is undefined*

Remark: this definition really resembles two matrix products where the summation indexes are γ and γ' which do not appear as index in the result matrix.

Here is the code defining this ternary product:

```
1  exception Type_Error of position*constructor*position*destructor
2
3  let ternary_product constructors relations destructors =
4    fold2_vector (fun acc i c v →
5      fold2_vector
```

```
6         (fun acc j (ds:destructor list) r →
7            if leq Leq r then
8              List.fold_left (fun acc (d:destructor) →
9                try match c, d with
10               | CFun(arg,res), DFun(arg',res') →
11                   add_coef (add_coef acc res res') arg' arg
12               | CSum(cid,arg), DSum(l) →
13                   add_coef acc arg (List.assoc cid l)
14               | CPro(l), DPro(lid,arg) →
15                   add_coef acc (List.assoc lid l) arg
16               | _ → raise Not_found
17               with Not_found →
18                 raise (Type_Error(i,c,j,d))) acc ds
19           else acc)
20        (fun acc j _ → acc)  (fun acc j _ → acc) acc
21        destructors
22        v)
23      (fun acc i _ → acc) (fun acc i _ → acc) []
24      constructors
25      relations
```

Listing 11: ternary product in typing.ml after extraction

Definition 28 (*saturated constraints*) *We say that the typing constraints (C, R, D) are saturated if and only if $R \geq R.R \vee C.R.D$.*

Exercise 29 *Write a more intellegible (but longer) definition of saturated constraints using the notation $(C, R, D) \models \ldots$, lemma 20 and the definition 27 of ternary product.*

We need to compute the smallest saturated constraints (C, R, D) such that $R \geq R_0$. A first way is to define $R_{n+1} = R_n.R_n \vee C.R_n.D$. Recall that R_0 has only 1 on the diagonal. This property is preseved for all the matrices R_n because 1 is the top element of the lattice $\mathcal{B}$ and this implies $R_n.R_n \geq R_n$. The fact that the sequence is increasing implies that it converges toward the wanted matrix because the lattice $\mathcal{B}$ is of finite height 2. This means that the limit is reached before $n = 2N^2$ where N is the number of lines of the matrix R, that is the number of type names mentioned in the initial constraints. When the above sequence reaches its limit R, we have $R = R.R \vee C.R.D$.

However, the abobe definition would lead to a bad complexity. A better sequence is:

- $S_0 = R'_0 = R_0$
- $T_n = R'_n.S_n \vee S_n.R'_n \vee C.S_n.D$
- $R'_{n+1} = T_n \vee R'_n$
- $S_{n+1} = T_n - R'_n$

With this definition, we clearly have for all n, $S_n \leq T_n \leq R'_n$ and by induction we get $R'_n \leq R_n$. The sequence R'_n is still increasing, and by definition of the subtraction, we have $R'_n = \bigvee_{0 \leq i \leq n} S_n$. Then, we deduce $R'_{n+1} \geq S_i.S_j$

for $0 \leq i, j \leq n$ and $R'_{n+1} \geq C.S_i.D$ for $0 \leq i \leq n$, which implies by summing the inequalities $R'_{n+1} \geq R'_n.R'_n \vee C.R'_n.D$ and therefore we have $R'_n = R_n$ by induction.

Remark: when R'_n reaches its limit, we have $S_n = 0$ (because $T_n \leq R'_n$, otherwise the limit would not be reached) which is an easy stopping condition.

To compute the complexity, let us define $|S_n|$ the number of non zero coefficients of S_n and N the number of types names in the annotated program M which produced the initial constraints (C, R_0, D). The definition of subtraction implies that $S_{n+1} \wedge R_n = 0$ (the matrix with only 0). This means that the total number of non zero coefficients in all the matrices S_n ($\Sigma_{1 \leq i \leq n} |S_n|$) is less then $2.N^2$. The product $R'_n.S_n$, $S_n.R'_n$ and $C.S_n.D$ needs a computing time which is less than $O(N|S_n|)$ (the N for the product $C.S_n.D$ is there because D is a multi-vector). We have also the same upper bound for the number of non-zero coefficients $O(N|S_n|)$ in the resulting matrices. This means that the two supremums and the subtraction can be done in time $O(N|S_n|)$ too. Therefore, globally, we have a complexity of $\Sigma_{1 \leq i \leq 2N^2} O(N|S_n|) \simeq O(N^3)$.

Remark: the analysis here assumes a constant time access to vectors and matrix coefficients. A more reasonnable complexity would be $O(N^3 \ln^2(N))$.

Exercise 30 *Find the best representation of each matrix to get a complexity* $O(N^3 \ln^2(N))$ *(or better, but in this case, let me know ... especially if you drop the* 3 *exponent.).*

This means that we proved the following theorem:

Theorem 31 *Let* (C, R_0, D) *be the constraints extracted from an annotated program* M*, the smallest saturated constraints* (C, R, D) *such that* $R \geq R_0$ *can be computed (if it exists) in polynomial time (*$O(N^3 \ln^2(N))$*).*

It is also trivial, from the fact that $R \geq R_0$*, that is we can find an interpretation satisfying the constraint* (C, R, D)*, it also satisfies* (C, R_0, D)*.*

Here is the code computing the saturated constraints:

```
1  let saturate verbose info =
2    let rec fn n _Rn _Sn =
3      let _Tn = sum_matrix (sum_matrix
4        (matrix_product _Rn _Sn) (matrix_product _Sn _Rn))
5        (ternary_product info.constructors _Sn info.destructors)
6      in
7      let _Sn_plus_one = sub_matrix _Tn _Rn in
8      if _Sn_plus_one = [] then _Rn else
9        let _Rn_plus_one = sum_matrix _Tn _Rn in
10       fn (n+1) _Rn_plus_one _Sn_plus_one
11   in
12   let _R = fn 0 info.relations info.relations in
13   { info with relations = _R }
```

Listing 12: constraint saturation in typing.ml

Unfortunately, even if (C, R, D) can be computed from (C, R_0, D), it does not imply that the constraints are satisfiable. We need another criteria: the well-foundedness of the constraints. This is here that we will use the lattice $\mathcal{B}'$:

Definition 32 *We say that the saturated constraints (C, R, D) are well-founded if there exists an hermitian matrix W with coefficients in $\mathcal{B}'$ and 1 on the diagonals such that:*

- *If $(C, R, D) \models \alpha \subset \beta$ then $W \models \alpha \preceq \beta$.*
- *If $(C, R, D) \models \alpha \to \beta \subset \gamma$ then $W \models \alpha \prec \gamma$ and $W \models \beta \preceq \gamma$.*
- *If $(C, R, D) \models \Sigma_{i \in I}\ \boldsymbol{C}_i[\alpha_i] \subset \gamma$ then $\forall i \in I$, we have $W \models \alpha_i \prec \gamma$.*
- *If $(C, R, D) \models \Pi_{i \in I}\ l_i : \alpha_i \subset \gamma$ then $\forall i \in I$, we have $W \models \alpha_i \prec \gamma$.*
- *$W \geq W.W$*

Theorem 33 *We can compute the matrix W (when it exists) and test if the constraints (C, R, D) are well-founded in polynomial time complexity.*

Proof: We can easily (in time $O(N^2 \ln^2 N)$) produce an initial matrix $W_0 \geq R$ (we use $\mathcal{B} \subset \mathcal{B}'$ to write this) satisfying all conditions but the last one and then, we can define a sequence computing the smallest matrix W such that $W \geq W_0$ and $W \geq W.W$ using an algorithm similar to the previous one. ■

Example: for the term omega, even the initial matrix W_0 is undefined because it should satisfy $W_0 \models \alpha_1 \prec \phi_1$ (because C for omega contains $\alpha_1 \to \rho_1 \subset \phi_1$) and $W_0 \models \phi_1 \preceq \alpha_1$ (because $S_5 \models \phi_1 \subset \alpha_1$).

Here is the code checking for well-foundedness, with a first function adding the initial constraints $\alpha \prec \beta$ coming from the constraints like $K \models \alpha \to \alpha' \subset \beta$:

```
1  let add_coef_strict m i1 i2 =
2    sum_matrix m (sum_matrix
3        (sum_matrix [i1, [i1, Equal]] [i2, [i2, Equal]])
4        (sum_matrix [i1, [i2, Le]] [i2, [i1, Ge]]))
5
6  let order_saturate verbose infos =
7    let rec fn n _Wn _Sn =
8      if verbose then begin
9        print_matrix_as_judgment print_relation_in_Bp_as_tex
10         ("S_"^string_of_int n) _Sn;
11       print_as 0 "\\cr";
12       print_newline ()
13     end;
14     let _Tn = sum_matrix (matrix_product _Wn _Sn)
15       (matrix_product _Sn _Wn) in
16     let _Sn_plus_one = sub_matrix _Tn _Wn in
17     if _Sn_plus_one = [] then _Wn else
18       let _Wn_plus_one = sum_matrix _Tn _Wn in
19       fn (n+1) _Wn_plus_one _Sn_plus_one
20   in
21   let _W0 = List.fold_left (fun m (i, c) →
22     match c with
23       | CSum(cid,j) → add_coef m j i
24       | CPro(l) →
25           List.fold_left (fun m (lid, j) → add_coef m j i) m l
26       | CFun(j,k) → add_coef (add_coef_strict m j i) k i)
27     infos.relations infos.constructors
```

```
28    in
29    fn 0 _W0 _W0
```

Listing 13: well-foundedness check in typing.ml

Now, we can state our last theorem:

Theorem 34 *If the saturated constraints $K = (C, R, D)$ are well founded then, they are satisfiable which means that there is an interpretation using definition 25 satisfying the constraints.*

Proof: Let W be the hermitian matrix witnessing the well-foundedness of the saturated constraints (C, R, D). We can directly define the interpretation as follows:

$$\begin{aligned} ||\alpha|| &= \bigvee\{||\beta|| \text{ s.t. } K \models \beta \subset \alpha\} \\ &\vee \bigvee\{||\beta|| \to ||\gamma|| \text{ s.t. } K \models \beta \to \gamma \subset \alpha\} \\ &\vee \bigvee\{\Sigma_{i\in I}\mathbf{C}_i[||\beta_i||] \text{ s.t. } K \models \Sigma_{i\in I}\mathbf{C}_i[\beta_i] \subset \alpha\} \\ &\vee \bigvee\{\Pi_{i\in I} l_i : ||\beta_i||] \text{ s.t. } K \models \Pi_{i\in I} l_i : \beta_i \subset \alpha\} \end{aligned}$$

The use of the well foundedness constraints is to show that this definition makes sense. For this, we need to remark that by definition of W, if β is used in the definition of α, then we have $W \models \beta \preceq \alpha$ and moreover, if β is used in the definition of α at the right of an implication, then $W \models \beta \prec \alpha$.

This means that if $W \models \beta \preceq \alpha$ but $W \not\models \beta \asymp \alpha$ then, we can define $||\beta||$ strictly before $||\alpha||$. Next, we remark that $W \models \beta \asymp \alpha$ is an equivalence relation and for $\{\beta_1, \ldots, \beta_n\}$ an equivalence class of this relation, we know that $||\beta_i||$ is only used in a covariant position in the definitions of $||\beta_j||$ for $1 \leq i, j \leq n$. This means that $||\beta_1||, \ldots, ||\beta_n||$ can be defined simultaneously as a smallest fixpoint using the above definition.

Next, we prove that this interpretation satisfies the constraints (C, R, D). For C and R this is immediate because they are directly included in the definition. To treat the destructors contraints D, we first remark that the above definition is equivalent to:

$$\begin{aligned} ||\alpha|| &= \bigvee\{||\beta|| \to ||\gamma|| \text{ s.t. } \exists\alpha', K \models \beta \to \gamma \subset \alpha' \text{ and } K \models \alpha' \subset \alpha\} \\ &\vee \bigvee\{\Sigma_{i\in I}\mathbf{C}_i[||\beta_i||] \text{ s.t. } \exists\alpha', K \models \Sigma_{i\in I}\mathbf{C}_i[\beta_i] \subset \alpha' \text{ and } K \models \alpha' \subset \alpha\} \\ &\vee \bigvee\{\Pi_{i\in I} l_i : ||\beta_i||] \text{ s.t. } \exists\alpha', K \models \Pi_{i\in I} l_i : \beta_i \subset \alpha' \text{ and } K \models \alpha' \subset \alpha\} \end{aligned}$$

This is true, just by replacing the $||\beta||$'s in $\bigvee\{||\beta|| \text{ s.t. } K \models \beta \subset \alpha\}$ by their definition.

Then, we may verify that constraints in D are satisfied:

- If $K \models \alpha \subset \beta \to \gamma$, we have $||\alpha|| = \bigvee\{||\beta|| \to ||\gamma|| \text{ s.t. } \exists\alpha', K \models \beta' \to \gamma' \subset \alpha' \text{ and } K \models \alpha' \subset \alpha\}$. The other sets in the supremum are empty otherwise the ternary product $C.R.D$ would be undefined and this contradicts the hypothesis that (C, R, D) are saturated. Because of saturation, for all α', β', γ' such that $K \models \beta' \to \gamma' \subset \alpha'$ and $K \models \alpha' \subset \alpha$, we

also have $K \models \beta \subset \beta'$ and $K \models \gamma' \subset \gamma$. This implies $||\beta|| \subset ||\beta'||$ and $||\gamma'|| \subset ||\gamma||$ which implies $||\beta'|| \to ||\gamma'|| \subset ||\beta|| \to ||\gamma||$ and therefore $||\alpha|| \subset ||\beta|| \to ||\gamma||$

- If $K \models \alpha \subset \Sigma_{i \in I} \mathbf{C}_i[\beta_i]$ or $K \models \alpha \subset \Pi_{i \in I} l_i : \beta_i$, the proof is similar. ■

7 Further work

We describe here, extensions of the algorithm that are already implemented in PML but for which there is no theoretical result published (this should change in the near future).

7.1 Types

Consider the following program:

$$\texttt{fix}\ \mathit{id\_nat}\ n \to \texttt{case}\ n\ \texttt{of}\ |\ \mathbf{Z}[] \to \mathbf{Z}[]\ |\ \mathbf{S}[n'] \to \mathbf{S}[\mathit{id\_nat}\ n']$$

It is clear what it does: it is a function copying a unary natural number! In other terms, this is the partial identity function whose domain is exactly the unary natural numbers. But what does the type-checking algorithm when we use that function ? It creates constraints that ensures that the argument of the function `id_nat` is a unary natural number and constraints enforcing that the result can only be used as a natural number.

This means that we can use partial identity functions as types. We just need to decide a syntax for types, for instance:

$$\begin{array}{rcl} \mathit{Type} & := & \mathit{Type} \to \mathit{Type} \\ & | & [\mathbf{C}_1[\mathit{Type}] | \ldots | \mathbf{C}_n[\mathit{Type}]] \\ & | & \{l_1 : \mathit{Type}; \ldots; l_n : \mathit{Type}\} \\ & | & \texttt{fix}\ t \to T \end{array}$$

Which will be translated as partial identity functions as follows (if T is a type, then $\overline{T}$ is a partial identity function):

$$\begin{array}{rcl} \overline{T_1 \to T_2} & = & \texttt{fun}\ f \to \texttt{fun}\ x \to \overline{T_2}(f(\overline{T_1}x)) \\ \overline{[\mathbf{C}_1[T_1] | \ldots | \mathbf{C}_n[T_n]]} & = & \texttt{fun}\ s \to \texttt{case}\ s\ \texttt{of}\ (\mathbf{C}_i[x] \to \mathbf{C}_i[\overline{T_i}x])_{1 \le i \le n} \\ \overline{\{l_1 : T_1; \ldots; l_n : T_n\}} & = & \texttt{fun}\ r \to \{(l_i = \overline{T_i}\, r.l_i)_{1 \le i \le n}\} \\ \overline{\texttt{fix}\ t \to T} & = & \texttt{fix}\ t \to \overline{T} \end{array}$$

What remains to do is to show that the above translation from types to programs do correspond in some sense to identity functions. This is easy by induction for all cases but function types. For function types, we need the notion of η-equivalence. We also would like to have parametric types, which can be done by having a new kind of arrow in type constraints which is interpreted exactly as the set of identity functions and this will ensure that parametric types are used with the correct number of parameters. This approach even allows for higher-order parametric types which are not available in usual ML implementations.

7.2 Simplification of the constraints and Polymorphism

The current implementation of PML extends the algorithm presented here with an algorithm simplifying the typing constraints by removing type names which can not be accessed anymore (as in [19]). A type name is said to be accessible in a program if, by using this program, we can create new constraints on this name. This is very important to have a reasonable complexity in practice.

Moreover, it allows for a nice implementation of polymorphism that defines the least set of type names that must be generalized. Not taking the least set would result in a non terminating algorithm. This least set is a smallest set satisfying some closure property and containing all type names α that are accessible from the left and the right which means that the newly created constraints are respectively of the form $\_ \subset \alpha$ and $\alpha \subset \_$.

7.3 Larger lattices

Many applications require using an enlarged lattice for type constraints that will still be a relational lattice. Here are a few (the ones used in PML):

- The type constraints contain a lot of information useful for a compiler. Unfortunately, if we use types, we do not want to compile the corresponding identity function and we can loose some information. A solution is to replace our lattice $\mathcal{B}$ with $\mathcal{B} \times \mathcal{B}$ using one component for type constraints ignoring types and the other for types constraints with types. This basically allows for maximum sharing compared with an approach when one would type-check the program twice: with types and without types.

- The language presented here do not have a default case in case analysis nor extensible records. Both features require the typing constraints to know that some variant constructors or labels can be ignored. This can be done by replacing the lattice $\mathcal{B}$ with functions in $\mathcal{V}_\Sigma \cup \mathcal{V}_\Pi \rightarrow_{fin} \mathcal{B}$ (here $\rightarrow_{fin}$ means function which are constant except on a finite set). This allows to say that a constraint ignores or concerns only specific variants or labels. This is exactly what we need for default cases and extensible records.

- Operator overloading is a very nice feature of programming languages. When an operator is overloaded, we need to know what implementation to choose, one way is to replace the lattice $\mathcal{B}$ by the set of binary decision diagram having elements of $\mathcal{B}$ at their leafs and testing boolean *variables* at their nodes. These *variables* express specific choices for overloaded operators. For instance if the constant 0^α (written with its type annotation) can be interpreted by two programs $\texttt{zero\_int}^\beta$ and $\texttt{zero\_float}^{\beta'}$, the type constraints will be something like $\beta \subset_{\alpha=\texttt{zero\_int}} \alpha$ and $\beta' \subset_{\alpha=\texttt{zero\_float}} \alpha$ where $\alpha = \texttt{zero\_int}$ and $\alpha = \texttt{zero\_float}$ are boolean variables which indicates which version of 0 is chosen. These variables clearly need to satisfy the axioms $\neg(\alpha = \texttt{zero\_int} \wedge \alpha = \texttt{zero\_float})$ and $\alpha = \texttt{zero\_int} \vee \alpha = \texttt{zero\_float}$.

This was surprisingly easy to implement in PML ... However, this is clearly NP-complete. But other parts of traditional ML compilation have bad worst-case complexity (polymorphism is exponential, test for exhaustive pattern-matching is NP-complete, ...).

A Call by value semantics

We can also give some code for a *call-by-value* semantics. First, let us define values as a BNF :

$$\begin{array}{rcl} V & := & id \quad | \quad \texttt{fun}\, x \to P \\ & | & \mathbf{C}[V] \\ & | & \{l = V; \ldots\} \end{array}$$

Here is the type definition for values, with a few differences:

- It is parameterized, because `'a program` is parameterized.
- In the `fun` case, we give an environment holding the value for all variables. This avoid us to perform substitution. We are making a *closure*.

```
1  type 'a value =
2  | VFun of id * 'a program * 'a env
3  | VFix of 'a program * 'a env
4  | VCst of cid * 'a value
5  | VRec of (lid * 'a value) list
6  and 'a env = (id * 'a value) list
```

Listing 14: Values in program.ml

To program call by value we will use an abstract machine with a stack, and an environment. The stack will hold not only arguments for functions, but all pending destructors. This is reflected by the following type definition with stack constructors corresponding to application (`SApp` to hold an argument and `RApp` to remember the function while computing the argument), projection (`SPi`) and case analysis (`SCas`). The constructor `RCst` and `RRec` remembers the constructor and the rest of a record while computing one of the field.

```
1  type 'a stack =
2    SEmpty
3  | SApp of 'a value * 'a stack
4  | SPi of lid * 'a stack
5  | SCas of (cid * (id * 'a program * 'a env)) list * 'a stack
6  | RApp of 'a env * 'a program * 'a stack
7  | RCst of cid * 'a stack
8  | RRec of (id * 'a value) list * lid * 'a env *
9            (id * 'a program) list * 'a stack
```

Listing 15: Stacks in program.ml

Here is now the code for a call-by-value interpretor. You may notice that it basically consists in two mutually recursive functions: `eval` and `unfold`. The first is evaluating programs more or less as expected. The second function is applying the stack elements to a value.

We could have simplified the code but removing `RApp`, `RCst` and `RRec` if we used OCaml's stack. But we preferred to give a tail-rec evaluator nearer to the behavior of some compiled code.

Exercise 35 *(not easy and out of the scope of this course) prove that this code implements call by value evaluation.*

Exercise 36 *There is a small trick is the code below to implement some sort of recursive definition which is not in the definition of the operational semantics of the language ... Understand what this does enough to remove it!*

```
1  open ParserUtil
2  open Format
3
4
5  let cbv verbose p =
6    let rec eval did_reduce env p stack =
7      match p with
8      | Var(id) →
9          unfold None (List.assoc id env) stack
10     | App(p1,p2,pos) →
11         eval None env p2 (RApp(env,p1,stack))
12     | Fun(id,_,p,_) →
13         unfold None (VFun(id,p,env)) stack
14     | Fix(id,_,p,_) →
15         let rec env' = (id,VFix(p,env'))::env in
16         unfold None (VFix(p,env')) stack
17     | Cst(cid,p,pos) →
18         eval None env p (RCst(cid,stack))
19     | Cas(p,cases,_) →
20         let stack =
21           SCas(List.map (fun (cid,id,p,_) →
22             (cid,(id,p,env))) cases, stack) in
23         eval None env p stack
24     | Rec(record,pos) →
25         begin
26           match record with
27           | [] → unfold None (VRec []) stack
28           | (lid,p)::recp →
29               eval None env p (RRec([],lid,env,recp,stack))
30         end
31     | Pi(p,lid,_) →
32         eval None env p (SPi(lid,stack))
33   and unfold did_reduce v stack =
34     try match v, stack with
35     | v, SEmpty → v
36     | v, RApp(env,p,stack) → eval None env p (SApp(v,stack))
37     | v, RCst(cid,stack) → unfold None (VCst(cid,v)) stack
38     | v, RRec(recv,lid,env,recp,stack) →
39         begin
40           let recv = (lid, v)::recv in
41           match recp with
42             | [] → unfold None (VRec(List.rev recv)) stack
43             | (lid,p)::recp →
44               eval None env p (RRec(recv,lid,env,recp,stack))
```

```
45          end
46      | VFun(id,p,env), SApp(c,stack) →
47          eval (Some "\\beta") ((id,c)::env) p stack
48      | VRec(record), SPi(lid,stack) →
49          unfold (Some "\\pi") (List.assoc lid record) stack
50      | VCst(cid,v), SCas(cases,stack) →
51          let id, p, env = List.assoc cid cases in
52          eval (Some "\\sigma") ((id,v)::env) p stack
53      | VFix(p,env), stack → eval (Some "\\mu") env p stack
54      | _ → assert false
55      with Not_found → assert false
56    in
57    eval None [] p SEmpty
```

Listing 16: Call by calue in program.ml

References

[1] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 31–41, New York, NY, USA, 1993. ACM.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[3] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[4] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.

[5] M. Felleisen and D. Friedman. Control operators, the SECD machine and the λ-calculus. *Formal description of Programming Concepts III*, pages 131–141, 1986.

[6] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[7] Timothy G. Griffin. A formulae as-as-types notion of control. In *17th annual ACM symposiuom on Principle of Programming Language*, pages 47–58. Oxford University Press, 1990.

[8] S. C. Kleene. On the interpretation of intuitionistic number theory. *The Journal of Symbolic Logic*, 10(4):109–124, 1945.

[9] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types". *Constructivity in Mathematics*, pages 101–128, 1959.

[10] Jean-Louis Krivine. Classical logic, storage operators and second-order lambda-calculus. *Ann. Pure Appl. Logic*, 68(1):53–78, 1994.

[11] Jean-Louis Krivine. Une preuve formelle et intuitionniste du théorème de complétude de la logique classique. *Bulletin of Symbolic Logic*, 2(4):405–421, 1996.

[12] Jean-Louis Krivine. The curry-howard correspondence in set theory. In *LICS*, pages 307–308, 2000.

[13] Jean-Louis Krivine. Dependent choice, 'quote' and the clock. *Theor. Comput. Sci.*, 308(1-3):259–276, 2003.

[14] A. Miquel. Relating classical realizability and negative translation for existential witness extraction. In P.-L. Curien, editor, *TLCA*, volume 5608 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2009.

[15] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1):35–55, 1999.

[16] Michel Parigot. $\lambda\mu$-calculus an algorithmic interpretation of classical natural deduction. *Proceedings of Logic and Automatic Reasoning*, 1991. Lecture Notes in Computer Science Vol. 624.

[17] Michel Parigot. Proofs of strong normalisation for second order classical natural deduction. *J. Symb. Log.*, 62(4):1461–1479, 1997.

[18] François Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4):312–347, 2000.

[19] FranÃ§ois Pottier. Simplifying subtyping constraints: a theory. *Information & Computation*, 170(2):153–183, November 2001.

[20] FranÃ§ois Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[21] Christophe Raffalli. Getting results from programs extracted from classical proofs. *Theor. Comput. Sci.*, 323(1-3):49–70, 2004.

[22] Christophe Raffalli and Frédéric Ruyer. Realizability of the axiom of choice in hol. (an analysis of krivine's work). *Fundam. Inform.*, 84(2):241–258, 2008.

[23] W.W. Tait. A realizability interpretation of the theory of species. In R. Parikh, editor, *Logic Colloquium*, volume 453 of *Lectures Notes in Mathematics*, pages 240–251, Boston, 1975. Springer-Verlag.

Index