Projective ML

Didier Rémy
INRIA-Rocquencourt*

Apr 10, 1992

Abstract

We propose a projective lambda calculus as the basis for operations on records. Projections operate on elevations, that is, records with defaults. This calculus extends lambda calculus while keeping its essential properties . We build projective ML from this calculus by adding the ML Let typing rule to the simply typed projective calculus. We show that projective ML possesses the subject reduction property, which means that well-typed programs can be reduced safely. Elevations are practical data structures that can be compiled efficiently. Moreover, standard records are definable in terms of projections.

Introduction

The importance of records in programming languages is commonly accepted. There have been many proposals for adding records in strongly typed functional languages [Car84, Wan87, JM88, OB88, Oho90, Rém89, Rém91, CM89, Car91, HP90a, HP90b]. However the topic is still active and there is not yet a best solution. Even for the most popular of them, ML, each implementation extends the core language with records of a very different kind.

For experts of record calculi, the multitude of works converges continuously towards a better comprehension of records, but it appears as a jungle of proposals for the novice that can hardly understand their very insidious differences. There is a lack of a simple formalism in which evaluation of row expressions could be described concisely and precisely. Furthermore, in a typed language, the typing rules often add technical restrictions that increase the confusion. This work started as a modest attempt to find a simple untyped record calculus in which most classical operations of records could be described. It ended in yet another proposal, but one that subsumes some others.

In the simplest view of records, there are only two operations. A record is a finite collection of objects, each component being addressed by name. The creation of a record takes as many name-object pairs as there are components and creates the corresponding record. The names used to address components are called labels; a label together with its component is a field. Reading information from a record takes a label that defines a field in the record and returns the component of that field. Thus the access of a component in a record should only require that the label does define a field in the record. Some type systems are more drastic, and require that the labels of all other fields of the records be also given at access time. This makes it impossible to use the same function to access the same field in two records having that field in common, but differing by other fields — a feature that is highly desirable.

The most popular extension of simple records is the creation of a record from another one by adding one field. This operation is called *record extension*. If the component may already be defined in the argument the extension is *free*, otherwise it is *strict*. Conversely, *record restriction* creates a record from another one by removing one of its field. As for extension, restriction can be free or strict.

The most difficult operation to type is still the *concatenation of records* that creates a record by combining the fields of two others [Wan89, HP90b]. Again, record concatenation can be free or strict. There is also recursive concatenation that recursively merges the components of common fields, provided they are records themselves [OB88]. Record concatenation can be encoded with record extension, which gives one way of typechecking record concatenation [Rém92d]. However, none of the proposal for typing record concatenation is fully satisfactory.

Between extension and concatenation, there exists an intermediate operation that takes two records and a label and builds a record by copying all the fields of the first record except for the given label whose field is taken from the second one, whether it is defined or not. That is, either the label is undefined in both the second argument and the result or it is defined with

*Author's address: INRIA, B.P. 105, F-78153 Le Chesnay Cedex. Email: Didier.Remy@inria.fr

To appear in LFP'92

the same value in both records. This operation, called *modification*, is strictly more powerful than extension and restriction, but much easier to type than concatenation, since it involves only one field. Other constructions, such as the exchange or renaming of fields are less popular, though they easily typecheck in some systems.

We introduce a projective lambda calculus as the basis for designing functional languages with records. In the first section, we study the Projective Lambda Calculus, written $P\Lambda$, extends the lambda calculus while preserving the Church-Rosser property. There is a simple projective type system for this calculus, for which the subject reduction theorem holds. In the second section, we extend the simple projective type system with the ML Let typing rules and add concrete data types to the language: this defines the language we call projective ML. In the last section we elaborate on the significance of Projective ML from three different standpoints.

By lack of space, most of the proofs have been omitted, other are roughly sketched. See [Rém92b] for a more thorough presentation.

1 The projective lambda calculus

In this section we introduce the untyped projective lambda calculus. Then, we propose a simple type system for this calculus, we prove the subject reduction property and show that there are principal typings.

1.1 The calculus $P\Lambda$

The projective lambda calculus $P\Lambda$ is the lambda calculus extended with three constructions, namely the elevation, the modification and the projection. It is defined relatively to a denumerable collection of labels, written with letters a and b.

$M ::= x$	Variable
$\mid \lambda\, x.\, M$	Abstraction
$\mid M\ M$	Application
$\mid [M]$	Elevation
$\mid M[a = M]$	Modification
$\mid M/a$	Projection

The intended meaning of these constructions is given by the reduction rules of the projective lambda calculus. Namely, the rules are the classical β rule:

$$(\lambda\, x.\, M)\ N \longrightarrow M[x := N] \qquad (\beta)$$

plus the following projective rules (P):

$$[M]/a \longrightarrow M \qquad (\textsc{Default})$$
$$M[a = N]/a \longrightarrow N \qquad (\textsc{Access})$$
$$M[b = N]/a \longrightarrow M/a \qquad (\textsc{Skip})$$

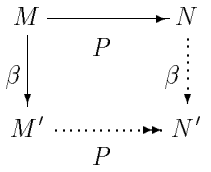
As opposed to records, elevations can be projected on all labels.

The compatible closure of $\longrightarrow$ is written $\longrightarrow\!\!\!\!\rightarrow$. The transitive closure of $\longrightarrow\!\!\!\!\rightarrow$ is written $\longrightarrow\!\!\!\!\!\!\rightarrow$ and call βP-reduction.

Theorem 1 (Church-Rosser) *The calculus βP is Church-Rosser.*

This means that if M βP-reduces to N and N', then there exists a term M' such that both N and N' βP-reduces to M'.

Proof: The reductions β and P are Church-Rosser. The reduction P is a rewriting system that has no critical pair and is nœtherien, thus it is Church-Rosser. The reductions β and P commute, since the diagram



commutes (this is checked by considering the relative positions of β- and P-redexes). ■

1.2 Projective types

Projective types extend the record types that have been introduced in [Rém90, Rém92c] in order to get a type system for the record extension of ML presented in [Rém90, Rém91].

Record types are based on the idea that types of records should carry information on all fields saying for every label either the field is present or absent [Rém89]. The way to deal with an infinite collection of labels is to give explicit information for a finite number of fields and gather all information about other fields in a template, called a row. Record types allow sharing between the same fields of two rows, but do not allow sharing between all fields of the same row (except for ground rows). When a type is coerced to a row, all projections must be shared for the same reason that lambda bound variables in ML cannot have polymorphic types.

$\tau ::=$	type	τ and σ
α	type variable	α and β
$\mid \tau \rightarrow \tau$	arrow type	
$\mid [\rho]$	projection type	
$\rho ::=$	row type	ρ and θ
φ	row variable	φ and ψ
$\mid \rho \Rightarrow \rho$	arrow row	
$\mid a : \tau\ ;\ \rho$	defined row	
$\mid \partial\, \tau$	shared row	

In fact, rows are sorted according to the set of labels that they cannot define. We omit this distinction here.

The reader is referred to [Rém92c] for a more thorough presentation.

The equality on types is defined by the following axioms. *Left commutativity*:

$$a : \alpha \ ; \ (b : \beta \ ; \ \varphi) = b : \beta \ ; \ (a : \alpha \ ; \ \varphi)$$

simply means that the order of definition of rows does not matter. *Replication*:

$$\partial\, \alpha = a : \alpha \ ; \ \partial\, \alpha$$

means that shared rows are the same as rows defining the same type on all labels. Distributivity of arrows:

$$\partial\, \alpha \Rightarrow \partial\, \beta = \partial\, (\alpha \rightarrow \beta)$$

and

$$(a : \alpha \ ; \ \varphi) \Rightarrow (a : \beta \ ; \ \psi) = a : (\alpha \rightarrow \beta) \ ; \ (\varphi \Rightarrow \psi)$$

means that arrow rows are truly rows of arrows.

Lemma 1 *The theory of projective types is regular, unitary unifying and has a decidable unification algorithm.*

Hint: The regularity directly follows from the shape of the axioms. The theory of projective types is shown syntactic by extending the method developed in [Rém92c] for simple record terms. This is the difficult part of the proof. It is a consequence that the rewrite rules given in the appendix A are sound and complete. The termination of the algorithm is quite standard. Then, since the rewrite rules never introduce any disjunction, the theory is unitary. ■

The unification algorithm is described in the appendix A.

1.3 A type system for $P\Lambda$

There are two kinds of typing judgements. A type assertion is the binding of a variable x to a type, written $x :^T \tau$ and a row assertion is the binding of a variable x to a row ρ, written $x :^R \rho$. A context is a list of assertions with rightmost priority. Mixed contexts contain both type and row assertions. Row contexts only contain row assertions. Concatenation of contexts is written by juxtaposition.

The judgement $H \vdash^T M : \tau$ means that in the mixed context H, the program M has type τ. The judgement $H, K \vdash^R M : \rho$ means that in the mixed context H and the row context K, the program M has row ρ. The first set of typing rules are the ones of the simply-typed lambda calculus:

$$\frac{x :^T \tau \in H}{H \vdash^T x : \tau} \quad \text{T-Var}$$

$$\frac{H[x :^T \tau] \vdash^T M : \sigma}{H \vdash^T \lambda\, x.\ M : \tau \rightarrow \sigma} \quad \text{T-Fun}$$

$$\frac{H \vdash^T M : \sigma \rightarrow \tau \qquad H \vdash^T N : \sigma}{H \vdash^T M\ \ N : \tau} \quad \text{T-App}$$

The next set of rules deals with the elevations:

$$\frac{H \vdash^T M : [a : \sigma \ ; \ \rho] \qquad H \vdash^T N : \tau}{H \vdash^T M[a = N] : [a : \tau \ ; \ \rho]} \quad \text{Modify}$$

$$\frac{H \vdash^T M : [a : \tau \ ; \ \rho]}{H \vdash^T M/a : \tau} \quad \text{Project}$$

$$\frac{H, \emptyset \vdash^R M : \rho}{H \vdash^T [M] : [\rho]} \quad \text{Elevate}$$

The first two rules are quite standard with record calculi. The last one describes the typing of an elevation. The elevated expression must be assigned a row. The row context shall binds variables that will be introduced during the typing of the current elevation, while previously bound variables are in the mixed context H. All expressions can be elevated, thus we need to assign rows to applications and abstractions as well:

$$\frac{x :^R \rho \in K}{H, K \vdash^R x : \rho} \quad \text{R-Var}$$

$$\frac{H, K[x :^R \rho] \vdash^R M : \theta}{H, K \vdash^R \lambda\, x.\ M : \rho \Rightarrow \theta} \quad \text{R-Fun}$$

$$\frac{H, K \vdash^R M : \theta \Rightarrow \rho \qquad H, K \vdash^R N : \theta}{H, K \vdash^R M\ \ N : \rho} \quad \text{R-App}$$

Sometimes, one might get a type when a row is required. For instance, when a type derivation of $\lambda\, x.\ [x]$, the variable x will be assigned a type τ, but a row will be expected when typing x in the elevation. The type τ can be lifted to a shared row.

$$\frac{H K \vdash^T M : \tau}{H, K \vdash^R M : \partial\, \tau} \quad \text{Lift}$$

Conversely, a variable bound to the row $\partial\, (\tau)$ can be used with type τ:

$$\frac{x :^R \partial\, \tau \in H}{H \vdash^T x : \tau} \quad \text{Drop-Var}$$

Finally, since types are taken modulo E-equality:

$$\frac{H \vdash^T M : \sigma \qquad \sigma =_E \tau}{H \vdash^T M : \tau} \quad \text{T-Equal}$$

$$\frac{H \vdash^R M : \theta \qquad \theta =_E \rho}{H \vdash^R M : \rho} \quad \text{R-Equal}$$

We presented the previous set of rules (RT) since there are simple and very intuitive. There is a smaller and more regular set (S), given in the appendix B, that are equivalent to the rules (RT). The judgements of (S) are $H, K \vdash^S \rho$ were both H and K are row contexts (where superscript ^{R} is omitted).

Lemma 2 *The judgement $H \vdash^T M : \tau$ is derivable if and only if the judgement $H, \emptyset \vdash M : \partial\,\tau$ is derivable where $x :^T \tau$ in T is translated as $x : \partial\,\tau$ in S.*

Hint: The proof is by successive transformations of (RT) into equivalent systems ending with (S). The first step converts every type assertion $x :^T \tau$ in contexts into row assertions $x :^R \partial\,(\tau)$, replacing in the derivations, every occurrence of the rule T-VAR by a rule DROP-VAR. Rule T-VAR is removed. The converse of the LIFT rule:

$$\frac{H, K \vdash^R M : \partial\,\tau}{H\,K \vdash^T M : \tau} \qquad \text{DROP}$$

is derivable in (RT), by an easy induction on the size of the derivation of the premise and by cases on the last rule of the derivation. It is added to (RT).

Successively, rules FUN and APP are removed, record rules of (S) are added, then those of (RT) can be removed, rule VAR is added and rule DROP-VAR is removed. Last, DROP and LIFT are shown to be useful only at the end of a derivation. ■

Lemma 3 (Stability by substitution) *Typings are stable by substitution.*

This property is quite immediate in the case of the a simple calculus.

The type inference problem is: given a triple $H, K \triangleright M : \rho$, find all substitutions μ such that $\mu(H), \mu(K) \vdash M : \mu(\rho)$. The type system (S) has principal typings if the set of solutions of every type inference problem is either empty, or has a maximal element called a principal solution, and if, in addition, there exists an algorithm that takes a type inference problem as input and returns a principal solution or an indication of failure if no solution exists.

Theorem 2 (Principal typings) *The type system of $P\Lambda$ has principal typings.*

Hint: Type inference for $P\Lambda$ is in the general framework of extending the ML type system with an equational theory on types. The comma that splits the contexts into two parts is a detail, since the system (S) is still syntax directed. The principal type property for such a system holds in general whenever the axiomatic theory on types is regular, unitary unifying and as a decidable unification algorithm [Rém90].

Type inference is based on the syntacticness of the theory of projective types and the unification algorithm that follows. It proceeds exactly as for the language with record extension presented in [Rém91]. ■

The algorithm for type inference can be found in the Appendix C for the language PML presented in the next section.

1.4 Subject reduction

Subject reduction holds if reduction preserves typings: for any program M and N, if M has type τ in the context H, K and βP-reduces to N, then N has type τ in context H, K.

Theorem 3 (Subject reduction) *Subject reduction holds in $P\Lambda$.*

Hint: It is shown independently for all cases of reduction at the root, then it easily follows for deeper reductions. The difficult case is ELEVATE. It uses the lemma *if $HK, \emptyset \vdash M : (a : \tau; \theta)$ is derivable in S, then so is $HK, \emptyset \vdash M : \partial\,\tau$* which is proved with a little stronger hypothesis by induction on the length of the derivation of the premise and cases on the last rule that is not an equality rule. ■

2 The language PML

Since the simply typed projective lambda calculus behaves nicely, we extend it to a full language, PML, in two steps. We add the ML *Let* typing rule and then concrete data types. In each case we check that the principal type property and subject reduction still hold.

2.1 Let polymorphism

We extend the projective calculus with a *let* construction

$$M ::= \ldots \mid \mathit{let}\ x = M\ \mathit{in}\ N$$

The *let* is syntactic sugar for marked redexes

$$(\lambda\, x.\ N)^*\ \ N$$

Thus, there is no special reduction rule for *let* redexes but the (β) rule:

$$(\lambda\, x.\ M)^*\ \ N \longrightarrow (x \mapsto N)(M) \qquad (\beta)$$

Therefore the calculus remains Church-Rosser.

Types are extended with type schemes. Type schemes are pairs of a set of variables and a type or a row, written $\forall\, W \cdot \tau$ or $\forall\, W \cdot \rho$. Formally, variables should be annotated with their sorts, but the sorts can be recovered from the occurrences of variables in their scheme. We identify type schemes modulo α-conversion of bound variables, and elimination of quantification over variables that are not free.

Type assertions now bind variables to type schemes. The rules VAR are changed to:

$$\frac{x : \forall W \cdot \rho \in K \qquad dom(\mu) \subset W}{H, K \vdash x : \mu(\rho)}$$

$$\frac{x : \forall W \cdot \partial\, \tau \in H \setminus K \qquad dom(\mu) \subset W}{H, K \vdash x : \partial\, \mu(\rho)}$$

The LET rule is

$$\frac{H, K \vdash M : \rho \qquad H, K[x : \mathcal{V}(\rho) \setminus \mathcal{V}(HK)] \vdash N : \theta}{H, K \vdash \mathit{let}\ x = M\ \mathit{in}\ N : \theta} \quad \text{LET}$$

where $\mathcal{V}(\rho)$ is the set of free variables in ρ and $\mathcal{V}$ is naturally extended to contexts.

The extension of $P\Lambda$ with *let* binding does not interfere with projections, and the substitution lemma, and the principal typing property and subject reduction theorems easily extend to PML.

2.2 Concrete data types

The language is now parameterized by a finite collection of concrete data types. For sake of simplicity, we consider a single two-constructor data type. We shall make other simplifying assumptions on types below, but it is possible to generalize to arbitrary data types.

The data type that we consider could be declared in ML as:

$$\text{type } bar\,(\rho) = A \mid B \text{ of } \rho$$

The syntax is extended with:

$$\begin{array}{rl} M ::= & \ldots \\ & \mid A \mid B(M) \\ & \mid \mathit{match}\ M\ \mathit{with}\ A \Rightarrow M \mid B(y) \Rightarrow M \end{array}$$

The new reduction rules are:

$$\begin{array}{l} (\mathit{match}\ A\ \mathit{with}\ A \Rightarrow M \\ \qquad\qquad \mid B(y) \Rightarrow N) \longrightarrow M \\ (\mathit{match}\ B(L)\ \mathit{with}\ A \Rightarrow M \\ \qquad\qquad \mid B(y) \Rightarrow N) \longrightarrow (\lambda\, y.\ N)\ L \end{array}$$

These δ-reductions are CR and commute with βP. Therefore the language PML with sums is still Church-Rosser.

Types are also extended with a symbol *bar* of arity one.

$$\begin{array}{lll} \tau ::= & \ldots & \text{Old type} \\ & bar\,(\tau) & \text{bar type} \\ \rho ::= & \ldots & \text{Old row} \\ & bar\,(\rho) & \text{bar row} \end{array}$$

We should have used two different symbols for bar types and bar rows, but the context will distinguish them. The symbol *bar* obeys the two distributivity axioms:

$$\begin{array}{c} bar\,(a : \alpha; \varphi) = a : bar\,(\alpha); bar\,(\varphi) \\ \partial\,(bar\,(\alpha)) = bar\,(\partial\,\alpha) \end{array}$$

We add the three typing rules:

$$\frac{}{H, K \vdash A : bar\,(\rho)} \qquad \frac{H, K \vdash M : \rho}{H, K \vdash B(M) : bar\,(\rho)}$$

$$\frac{H, K \vdash L : bar\,(\theta) \qquad H, K \vdash M : \rho \qquad H, K \vdash \lambda\, y.\ N : \theta \Rightarrow \rho}{\mathit{match}\ L\ \mathit{with}\ A \Rightarrow M \mid B(y) \Rightarrow N : \rho}$$

Theorem 4 *The language PML with sums has principal typings.*

Theorem 5 *Subject reduction holds for PML with sums.*

3 The three views of PML

Projective ML is a practical language of records with default values. It is also a language in which all operations of classical records but concatenation are definable. Finally, computation inside elevations introduces a new kind of polymorphism.

3.1 Records with default values

To the author's knowledge, this feature has never been introduced in the literature before. Instead of starting with empty records that can be extended with new fields, projective ML initially creates records with the same default value on all fields. Then a finite number of fields can be modified. Thus, all fields are always defined and can be read.

The introductory examples below have been typechecked by a prototype typechecker written in Caml-Light [Ler90]. The first examples are:

```
#type unit = Unit;;
#let r = [Unit];;
r : shared [unit]

#r/a;;
it : shared unit

#type bool = True | False;;
#let s = r [a = True];;
s : shared [a : bool; unit]

#s/a;;
it : shared bool
```

The *a* field of *s* cannot be removed, but it can be reset to its default value. Whenever the types of fields are known statically, but not their presence, the attendance can be dynamically checked:

```
#type field (θ) = Absent | Present of θ;;
#let r = [Abs] [a = Present (True)]
               [b = Present (Unit)];;
r : shared [a : field (bool); b : field (unit); field (θ)]
```

```
#let check x =
  match x with Present y ⇒ y
      | Absent ⇒ failwith "Absent field";;
check : field (θ) ⇒ θ

#let v = check (r/a);;
v : shared bool
```

If the presence of fields is stactically known, the two-constructor data type can be replaced by two one-constructor data types, leaving the typechecker check attendances.

```
#type absent = Absent;;
#type present (θ) = Present (θ);;
#let get x = match x with Present y ⇒ y;;
get : present (θ) ⇒ θ

#[Absent][a = Present (true)][b = Present (unit)];;
it : shared [a : present (bool); b : present (unit); absent]

#let v = get (it/a);;
v : shared bool
```

Record with defaults are not just an untractable toy feature. They can be compiled very efficiently, as classical records [Rém92a].

3.2 Classical records

Continuing the example above, we show that classical records are definable in projective ML. Precisely, classical record operations are just syntactic sugar for:

$$\begin{aligned} \{\} &\equiv [Absent] \\ \{M \ with \ a = N\} &\equiv M[a = Present \ (N)] \\ (M.a) &\equiv get \ (M/a) \end{aligned}$$

Many other constructions are programmable as well, since projective ML allows the manipulation of fields whether they are present or absent.

$$\begin{aligned} M \setminus a &\equiv M[a = Absent] \\ \{M \ but \ a \ from \ N\} &\equiv M[a = N/a] \\ \{exchange \ a \ and \ b \ in \ M\} &\equiv let \ u = M/a \ in \\ &\quad\ let \ v = M/b \ in \\ &\quad\ M[a = v][b = u] \end{aligned}$$

Though efficiency is not our main goal here, it is important to emphasize that dealing explicitly with the presence of fields does not cost anything. Since both *abs* and *pre* data types have unique constructors, the constructors need not be represented explicitly. That is, the presence of fields can be statically computed by the typechecker. Even the default value *Absent* need not be represented, since it is the only value in its type. Thus the (very small) overhead for computing with elevations only costs when there are used.

Obviously, the projective implementation of standard records can be packed in an abstract data type or a module so that the two types *pre* and *abs* and their constructors are not visible outside, and the presence of fields cannot be manipulated by hand. But elevations and projections will remain visible, can be used whenever defaults values in records are desirable, or also to implement another variant of classical records.

3.3 Projection polymorphism

The last view of projective ML is quite unexpected. The elevations are assigned rows that are in fact "template" types. That is, they can be read on any component by taking a copy of the template; therefore the type of two projections will not be equal but isomorphic. For instance, with classical records as in [Rém91] (or using the syntactic sugar of the previous section) the function that reads the a field of a record has type:

$$[a : pre \ \tau; \varphi] \to \tau$$

But this type can also be seen as[1]:

$$[a : pre \ \tau; b : \alpha; \psi] \to \tau$$

With classical records, this polymorphism allows the finite representation of a potentially infinite product of types, and nothing more. In projective ML, we can fill the elevations with any value and even compute inside. The identity function elevation $[\lambda x.\ x]$ has type $[\varphi \Rightarrow \varphi]$. Taking its projection on two arbitrary fields gives twice the same value but with two isomorphic types $\alpha \to \alpha$ and $\beta \to \beta$. The program,

$$(\lambda x.\ x\ \ x)\ \ (\lambda x.\ x) \tag{1}$$

cannot be written in ML without a LET. In projective ML one can write:

$$(\lambda x.\ x/a\ \ x/b)\ \ [\lambda x.\ x] \tag{2}$$

which has type $\alpha \to \alpha$. It can be argued that this is not exactly the same program, and that, if program transformations are allowed, then the following ML program also computes the same result.

$$(\lambda xy.\ x\ \ y)\ \ (\lambda x.\ x)\ \ (\lambda x.\ x) \tag{3}$$

This is certainly true, but the program (3) is much bigger than the program (1) and duplicates some of the code. The expression (2) is almost as small as the expression (1) and takes less time to typecheck (for bigger example of course, since all examples here are too small to allow any comparison). In (3), the body of $\lambda x.\ x$ is typed twice, but it is typed only once in (2) before the resulting type is duplicated by unification.

Moreover, if we consider a variant of PML without the possibility of modifying elevations,

$$M ::= x \mid \lambda x.\ M \mid M\ M \mid [M] \mid M/a$$

[1] In [Rém92c] we define canonical forms and show that both type have the same canonical form, though they are not equal (the latter is less general).

then projections always access the default value of elevations (since they could not be modified). Elevation and projection can both be implemented as empty code. They only modifies the types (they are called retyping functions), and helps the typechecker as if they were type annotations. The elevation indicates that an expression may be used later with different types, and thus should be typed with a row. The projection requires the use of a copy of the row template instead of the row itself. The copy is kept inside the row for constraint propagation.

Breaking the expression (2), the subexpression $(\lambda\, x.\ x/a\ \ x/b)$ has type:

$$[a : \tau \to \sigma; b : \tau] \to \sigma \tag{4}$$

There are obvious similarities with conjunctive types [Cop80, Pie91]. This expression would have the conjunctive type

$$(\tau \to \sigma \wedge \tau) \to \sigma \tag{5}$$

Projective ML differs from conjunctive types by naming the conjunctions, but also in some deeper way. The projection, which correspond to the expansion in conjunctive types, is much more restrictive than the expansion. An interesting comparison would be with the decidable restriction of conjunctive types that has been recently proposed by Coppo and Denzianni [CG92].

There is an important limitation in the type system of projective ML: it is a two-level design. Elevations inside elevations get typed with shared rows and projective polymorphism is lost. A stratified version with types, rows, rows of rows, etc. composing an infinite row tower can be imagined. The author has actually worked on such a version but has not proved yet that it is correct.

Another form of this limitation of projective polymorphism is its failure to cross elevations. The best type for $\lambda\, x.\ [x]$ is $\partial\, \alpha \Rightarrow [\partial\, \alpha]$, while we would expect $\varphi \Rightarrow [\varphi]$. Variables in elevations that are bound outside of the current elevation in which they appear can only have shared rows.

Projective polymorphism combines nicely with generic polymorphism. The two concepts are orthogonal. Here is an example that combines both:

$$\begin{array}{l} \mathit{let}\ F = \lambda\, f.\ \lambda\, x, y.\ f/a\ \ x, f/b\ \ y\ \mathit{in} \\ F\ \ [I]\ \ (I, K), F\ \ [K]\ \ (I, K) \end{array}$$

where I and K are abbreviations for $\lambda\, x.\ x$ and $\lambda\, xy.\ x$. It is typeable in projective ML.

Conclusions

We have introduced Projective ML, and shown that it is a type-safe language. Projective ML exceeds ML on two opposite fields.

- Elevations, modifications and projections are extensible records with defaults. With only three operations that can be compiled very efficiently, they provide the ML language with enough power to define all variants of classical records.
- Projective ML brings in the type system a restricted form of conjunctive polymorphism.

The curiosity of Projective ML is that both features are almost independent but one still need the other. The most intriguing of the two is projective polymorphism, for which more investigation is still needed.

References

[Car84] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–68. Springer Verlag, 1984. Also in Information and Computation, 1988.

[Car91] Luca Cardelli. Extensible records in a pure calculus of subtyping. Private Communication, 1991.

[CG92] M. Coppo and P. Giannini. A complete type inference algorithm for simple intersection types. In *Proceedings of European Symposium On Programming*, volume 582 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[CM89] Luca Cardelli and John C. Mitchell. Operations on records. In *Fifth International Conference on Mathematical Foundations of Programming Semantics*, 1989.

[Cop80] Mario Coppo. An extended polymorphic type system for applicative languages. In *MFCS '80*, volume 88 of *Lecture Notes in Computer Science*, pages 194–204. Springer Verlag, 1980.

[HP90a] Robert W. Harper and Benjamin C. Pierce. Extensible records without subsumption. Technical Report CMU-CS-90-102, Carnegie Mellon University, Pittsburg, Pensylvania, February 1990.

[HP90b] Robert W. Harper and Benjamin C. Pierce. A record calculus based on symmetric concatenation. Technical Report CMU-CS-90-157, Carnegie Mellon University, Pittsburg, Pensylvania, February 1990.

[JM88] Lalita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes. In *Proceedings of the 1988 Conference on LISP and Functional Programming*, 1988.

[Kir86] Claude Kirchner. Computing unification algorithms. In *Proceeding of the first symposium on Logic In Computer Science*, pages 206–216, Boston (USA), 1986.

[KJ90] Claude Kirchner and Jean-Pierre Jouannaud. Solving equations in abstract algebras: a rule-based survey of unification. Research Report 561, Université de Paris Sud, Orsay, France, April 1990.

[KK89] Claude Kirchner and François Klay. Syntactic theories and unification. CRIN & INRIA Loraine, Nancy (France), 1989.

[Ler90] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA-Rocquencourt, 1990.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[OB88] Atsushi Ohori and Peter Buneman. Type inference in a database langage. In *ACM Conference on LISP and Functional Programming*, pages 174–183, 1988.

[Oho90] Atsushi Ohori. Extending ML polymorphism to record structure. Technical report, University of Glasgow, 1990.

[Pie91] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, Pittsburg, Pensylvania, February 1991.

[Rém89] Didier Rémy. Records and variants as a natural extension of ML. In *Sixteenth Annual Symposium on Principles Of Programming Languages*, 1989.

[Rém90] Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objects Enregistrements dans les Langages Fonctionnels.* Thèse de doctorat, Université de Paris 7, 1990.

[Rém91] Didier Rémy. Type inference for records in a natural extension of ML. Technical Report 1431, Inria-Rocquencourt, May 1991. Also in [Rém90], chapter 4.

[Rém92a] Didier Rémy. Efficient representation of extensible records. In *Proceedings of the 1992 ML workshop*, 1992.

[Rém92b] Didier Rémy. Projective ML. Technical report, BP 105, F-78 153 Le Chesnay Cedex, BP 105, F-78 153 Le Chesnay Cedex, 1992. To appear.

[Rém92c] Didier Rémy. Syntactic theories and the algebra of record terms. Technical report, BP 105, F-78 153 Le Chesnay Cedex, BP 105, F-78 153 Le Chesnay Cedex, 1992. To appear. Also in [Rém90], chapter 2.

[Rém92d] Didier Rémy. Typing record concatenation for free. In *Nineteenth Annual Symposium on Principles Of Programming Languages*, 1992.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Second Symposium on Logic In Computer Science*, 1987.

[Wan89] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Fourth Annual Symposium on Logic In Computer Science*, pages 92–97, 1989.

A Unification on projective types

We describe the unification algorithm by transformation rules on unificands (multi-sets of equations). The formalism is the one of [Rém92c] in general, improved with existential unificands [KJ90]. A multi-equation is a multi-set of terms written $\tau_1 \doteq \ldots \tau_n$. A solution of a multi-equation is a substitution that unifies all the terms of the multi-equation. A multi-set of multi-equations is noted $U_1 \wedge \ldots U_p$. Its solutions are the substitutions that satisfy all the multi-equations. We also use existential unificands, written $\exists\alpha.U$, whose solutions are the restrictions of the solutions of U on variables distinct from α. Indeed, $\exists$ acts as a binder, and existential unificands are equal modulo α-conversion. Consecutive binders can be exchanged, and $\exists\alpha.U$ is equal to U whenever α is not free in U. We identify unificands modulo the previous equalities.

Two unificands U and U' are *equivalent*, and we write $U \equiv U'$ if they have the same set of solutions. The relation $\equiv$ is obviously an equivalence. It is also a congruence, that is, parts of unificands can be replaced by equivalent parts. We also write $\bot$ and $\top$ for unificands that are respectively equivalent to the empty set and the set of all substitutions.

The input of the unification algorithm is a multi-set of equations. The output will be failure or a most general solution of the input unificand. It proceeds in three steps. All of these steps are described by transformations of unificands that are equivalences.

Most of the transformations are valid for both types and rows. We write χ and ξ for terms and π for variables that can be of both kinds. The first step is the generalization:

$$\frac{e \doteq \chi[\xi]}{\exists\, \pi.\, e \doteq \chi[\pi] \wedge \pi \doteq \xi} \qquad \textsc{Generalize}$$

An iteration of this rule will transform any system into one that contains only small terms (terms of height at most one).

The second step is only defined on small unificands, and keeps them small. The mutation of unificands is one of the four following transformations (f is a symbol of arity p and I is the segment of integers $[1, p]$):

$$\frac{a : \tau; \rho \doteq f(\theta_i)_I}{\exists\, (\alpha_i)_I\ (\varphi_i)_I\, .\ \bigwedge \begin{cases} \tau \doteq f\,(\alpha_i)_I \\ \rho \doteq f\,(\varphi_i)_I \\ \theta_i \doteq a : \alpha_i; \varphi_i \quad i \in I \end{cases}} \qquad \textsc{Mut}_{a \triangleright f}$$

$$\frac{a : \tau; \rho \doteq b : \sigma; \theta}{\exists\, \varphi.\ \bigwedge \begin{cases} \rho \doteq b : \sigma; \varphi \\ \theta \doteq a : \tau; \varphi \end{cases}} \qquad \textsc{Mut}_{a \triangleright b}$$

$$\frac{\partial\,(\tau) \doteq a : \sigma; \rho}{\exists\, \alpha.\ \bigwedge \begin{cases} \alpha \doteq \sigma \doteq \tau \\ \rho \doteq \partial\,(\alpha) \end{cases}} \qquad \textsc{Mut}_{\partial \triangleright b}$$

$$\frac{\partial\,(\tau) \doteq f\,(\rho_i)_I}{\exists\, (\alpha_i)_I \cdot \bigwedge \begin{cases} \tau \doteq f\,(\alpha_i)_I \\ \rho_i \doteq \partial\,(\alpha_i) \quad i \in I \end{cases}} \qquad \textsc{Mut}_{f \triangleright \partial}$$

For all other pairs of terms (χ, ξ), if they have identical top symbols, they are decomposable, that is

$$\frac{\chi \doteq \xi}{\bigwedge_I (\chi_{/i} \doteq \xi_{/i})} \qquad \textsc{Decompose}$$

otherwise they produce a collision

$$\frac{\chi \doteq \xi}{\bigwedge_I (\chi_{/i} \doteq \xi_{/i})} \qquad \textsc{Collision}$$

All mutation, decomposition and collision rules can be generalized to rules where the premise is a multi-equation rather than an equation: for any mutation rule

$$\frac{\chi \doteq \xi}{Q}$$

we build the generalized mutation rule:

$$\frac{e \doteq \chi \doteq \xi}{e \doteq X \wedge Q}$$

The fusion of multi-equations is:

$$\frac{\pi \doteq e \wedge \pi \doteq e'}{\pi \doteq e \doteq e'} \qquad \textsc{Fuse}$$

Applying the generalized mutation and the fusion in any order always terminates on small unificands. Unificands that cannot be reduced are necessarily in *canonical* forms, that is, completely decomposed and fused.

The last step does the occur check on canonical unificands while instantiating the equations by partial solutions. On canonical unificands Q, we say that the multi-equation e' is directly inner the multi-equation e if there is at least a variable term of e' that appears in a non variable term of e. We note $\lessdot_Q$ its transitive closure. The occur check is the rule

$$\text{if } e \lessdot_Q e, \qquad \frac{Q}{\bot} \qquad \textsc{Occur}$$

Otherwise, we can apply the rule:

$$\text{if } e \notin Q, \qquad \frac{e \wedge Q}{e \wedge \hat{e}(Q)} \qquad \textsc{Replace}$$

where $\hat{e}$ is the trivial solution of e that sends all variable terms of e to the non variable term if it exists, or to any variable term otherwise. The Replace rule is completed by the elimination of useless existentials

$$\text{if } \pi \notin e \cap Q, \qquad \frac{\exists\, \pi.\, (\pi \doteq e \wedge Q)}{e \wedge Q} \qquad \textsc{Restrict}$$

The succession of the three steps either fails or ends with a system $\exists W.Q$ where all multi-equations are independent. A principal solution of the system is $\hat{Q}$, that is, the composition, in any order, of the trivial solutions of its multi-equations. It is defined up to a renaming of variables in W.

The last step may be reduced to the occur check, and the equations in the unificand need not be instantiated by rule Replace, since the canonical unificand itself is a good and compact representation of a principal unifier.

Although it is described in a more general framework, the algorithm is very close to the one of Martelli-Montanari for empty theories [MM82], some of the collisions have been replaced by mutations in a way that copies the axioms of the theory. This is a property of syntactic theories [Kir86, KK89]. Proving the correctness of the algorithm is reduced to proving the syntacticness of the theory and the termination of the second step. Proving the termination is standard, but proving that the theory is syntactic is the difficult part.

The second step may not be restricted to small terms. In this case the generalized mutation and decomposition rules need to include the minimum of generalization so that there is enough sharing to ensure the termination.

B A simpler set of typing rules for the projective calculus

The judgements are of the form $H, K \vdash M : \rho$, where H and K are row assertions. The typing rules, called (S) are:

$$\frac{x : \partial\,\tau \in H \setminus K}{H, K \vdash x : \partial\,\tau} \qquad \frac{x : \rho \in K}{H, K \vdash x : \rho} \qquad \textsc{Var}$$

$$\frac{H, K[x : \rho] \vdash M : \theta}{H, K \vdash \lambda\, x.\, M : \rho \Rightarrow \theta} \qquad \textsc{Fun}$$

$$\frac{H, K \vdash M : \theta \Rightarrow \rho \qquad H, K \vdash N : \theta}{H, K \vdash M\ N : \rho} \qquad \textsc{App}$$

$$\frac{HK, \emptyset \vdash M : \rho}{H, K \vdash [M] : \partial\,[\rho]} \qquad \textsc{Elevate}$$

$$\frac{\begin{array}{c} H, K \vdash N : \partial\,(\sigma) \\ H, K \vdash M : \partial\,[a : \tau\ ;\ \theta] \end{array}}{H, K \vdash M[a = N] : \partial\,[a : \sigma\ ;\ \theta]} \qquad \textsc{Modify}$$

$$\frac{H, K \vdash [M] : \partial\,[a : \tau\ ;\ \theta]}{H, K \vdash M/a : \partial\,\tau} \qquad \textsc{Project}$$

$$\frac{H, K \vdash M : \theta \qquad \theta =_E \rho}{H, K \vdash M : \rho} \qquad \textsc{Equal}$$

C Type inference

The above set of rules is completed with:

$$\frac{\begin{array}{c} H, K \vdash M : \rho \\ H, K[x : \forall\ (\mathcal{V}(\rho) \setminus \mathcal{V}(HK)) \cdot \rho] \vdash N : \theta \end{array}}{H, K \vdash \mathit{let}\ x = M\ \mathit{in}\ N : \theta} \qquad \textsc{Let}$$

The rules are not exactly those of ML. The two rules Modify and Project can be treated as application of constants. The rule equal, due to an extended type equality, does not add any difficulty, provided that the theory is regular and has a decidable and unitary unification algorithm [Rém90]. The only difference with ML (extended with equations on types) is the mark in the context. However, the position of the mark is rigid, and the type inference algorithms of ML very easily extends to the system S. We describe the algorithm in terms of unificands. The substitution lemma (that extends to PML) allows to consider type inference problems as unificands, written $H, K \triangleright M : \rho$, whose solutions are the substitutions μ such that $\mu(H), \mu(K) \vdash M : \mu(\rho)$ is a valid judgement. We give below equivalence transformations of these unificands.

Case Var: If $x : \partial\,\tau$ is in $H \setminus K$, and μ is a renaming of variables of $\mathcal{V}(\tau)$ outside of σ, then

$$\frac{H, K \triangleright x : \sigma}{\exists\, \mathcal{V}(\mu(\tau)).\, \sigma = \mu(\tau)} \qquad \text{T-Var}$$

If $x : \rho$ is in K, and is a renaming of variables of $\mathcal{V}(\rho)$ outside of θ, then

$$\frac{H, K \triangleright x : \theta}{\exists\, \mathcal{V}(\mu(\rho)).\, \theta = \mu(\rho)} \qquad \text{R-Var}$$

If x is not in HK, then $H, K \triangleright x : \alpha$ is not solvable.

Case App:

$$\frac{H, K \triangleright M N : \rho}{\exists\, \psi.\, H, K \triangleright M : \psi \wedge H, K \triangleright N : \psi \Rightarrow \rho} \qquad \textsc{App}$$

Case Fun:

$$\frac{H, K \triangleright \lambda\, x.\, M : \rho}{\exists\, \varphi\psi.\, H, K[x : \varphi] \triangleright M : \psi \wedge \rho = \varphi \Rightarrow \psi} \qquad \textsc{Fun}$$

Case Let: If β is outside of HK and $\exists\, W.Q$ is a solvable independent unificand equivalent to $H, K \triangleright M : \beta$, then

$$\frac{H, K \triangleright \mathit{let}\ x = M\ \mathit{in}\ N : \tau}{\exists\, W.\, H, K[x : \hat{Q}(\beta)] \triangleright N : \tau} \qquad \textsc{Let}$$

If $H, K \triangleright M : \beta$ is not solvable, then neither is $H, K \triangleright \mathit{let}\ x = M\ \mathit{in}\ N : \alpha$.

Case Elevate:

$$\frac{H, K \triangleright [M] : \rho}{\exists\, \alpha.\, HK, \emptyset \triangleright M : \alpha \wedge \partial\,\alpha \doteq \rho} \qquad \textsc{Elevate}$$

The above rules applied in any order either fail or reduce any type inference problem to a unification problem.