



The Recursive Record Semantics of Objects Revisited

G rard Boudol

► **To cite this version:**

G rard Boudol. The Recursive Record Semantics of Objects Revisited. RR-4199, INRIA. 2001. inria-00072423

HAL Id: inria-00072423

<https://hal.inria.fr/inria-00072423>

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destin e au d p t et   la diffusion de documents scientifiques de niveau recherche, publi s ou non,  manant des  tablissements d'enseignement et de recherche fran ais ou  trangers, des laboratoires publics ou priv s.

The Recursive Record Semantics of Objects Revisited

G rard Boudol

N  4199

June 2001

TH ME 1



*Rapport
de recherche*

The Recursive Record Semantics of Objects Revisited

Gérard Boudol

Thème 1 — Réseaux et systèmes
Projets MIMOSA

Rapport de recherche n° 4199 — June 2001 — 44 pages

Abstract: In a call-by-value language, representing objects as recursive records requires using an unsafe fixpoint. We design, for a core language including extensible records, a type system which rules out unsafe recursion and still supports the reconstruction of a principal type. We illustrate the expressive power of this language with respect to object-oriented programming by introducing a sub-language for “mixin-based” programming.

Key-words: fixpoint, types, records, objects, semantics

Work partially supported by the CTI “Objets Migrants: Modélisation et Vérification”, France Télécom R&D, and by the French-Portugese Project “Sémantique des Object Concurrents”.

Sur la Sémantique des Objets comme Enregistrements Récursifs

Résumé : La représentation des objets comme enregistrements récursifs dans un langage en appel par valeur requiert l'utilisation d'un combinateur de point fixe qui peut introduire des erreurs à l'exécution. Nous présentons un typage de la récursion qui offre les avantages du typage à la ML, à savoir: typage implicite avec reconstruction d'un type principal, et garantie qu'un programme typable ne produira pas d'erreur à l'exécution. Nous illustrons le pouvoir expressif du calcul obtenu en définissant un sous-calcul de "mixins", et en donnant quelques exemples exprimés dans ce langage.

Mots-clés : point fixe, types, enregistrements, objets, sémantique

1. Introduction

During the past fifteen years there has been very active research about the formalization of object-oriented programming concepts. One of the main purposes of this research was to design operational models of objects supporting rich type systems, so that one could benefit both from the flexibility of the object-oriented style, and from the safety properties guaranteed by typing. Let us be more precise here: our goal is to have an expressive language – as far as object-oriented constructs are concerned – with a type discipline à la ML [16, 34], i.e. implicit typing with reconstruction of a principal type, ruling out run-time errors. This goal has proven difficult to achieve, and most of the many proposals that were put forward fall short of achieving it – with the exception of OCAML [30, 44], that we will discuss later.

While the meaning of “typing à la ML” should be clear, it is perhaps less easy to see what is meant by “object-oriented”. We do not claim to answer to this question here. Let us just say that, in our view, objects encapsulate a state and react to messages, i.e. method invocations, by updating their state and sending messages, possibly to themselves. Moreover, in our view, object-orientation also involves inheritance, which includes – but should not be limited to, as we shall see – the ability to add and redefine methods. With this informal notion of object-orientation in mind, let us review some of the proposals we alluded to.

An elegant proposal was made by Wand [51], based on his *row variables* [49], consisting of a class-based model, where classes are functions from instance variables and a *self* parameter to extensible records of methods, and objects are fixpoints of instantiated classes, that is, recursive records. In this model invoking the method of an object amounts to selecting the corresponding component of the record representing the object. An operation of record extension is used to provide a simple model of inheritance, à la SMALLTALK: a class B inherits from a class A by adding to it new methods, or redefining (overriding) some of them. Unfortunately, although its elegance and simplicity make it very appealing, Wand’s model is not expressive enough. More specifically, it does not support state changes in objects: one may override a method in an inherited class, but one apparently cannot modify the state of the object during its life-time (see for instance [2] Section 6.7.2). This is because in creating the object, the *self* parameter is bound too early.

Wand’s model is an instance of what is known as the *recursive record semantics* for objects (see [21]), initiated by Cardelli [11]. Based on this idea that an object is the fixpoint of an instantiated class, Cook proposed a more elaborate model [14], where updating the state of an object is possible, by creating new objects, instances of the same class. Then a class is also recursive in Cook’s model, since methods may have to call a *myClass* parameter. This model is operationally quite expressive, but the type theory that it uses is also quite elaborate, and does not fulfil our desires, of reconstruction of a principal type. The same remark actually applies to all the object models that use higher-order types, e.g. [2, 9, 18, 20, 39].

In another approach, due to Kamin [28] and known as the *self-application semantics*, an object is a record of pre-methods, that are functions of the object itself. The object is bound to *self* only when a method is invoked, by applying the pre-method to the object. In this way, the state of the object may dynamically be updated. In this model, which looks indeed operationally satisfactory, an object is not quite a record, since from a typing point of view, we must know that the first parameters (that is, *self*) of all its pre-methods have the same type. In other words, one must have in this approach specific constructs for objects and object types, depending on the type of *self*, thus different from record types. This has been developed in *object calculi*, most notably by Fisher and Mitchell [19, 20, 21] (who call it “the axiomatic approach”) and Abadi and Cardelli [1, 2], but as we already noticed, in calculi that support a rich form of inheritance, and in particular object extension, like [20], the type theory which is used is quite elaborate, and does not support implicit typing.

Object calculi claim to fix the principles for objects, thus providing simple formal models, but they actually take design decisions, about inheritance in particular – a concept which is still a matter of debate in the object-oriented programming community (see [48] for example). As a matter of fact, many of the proposals for an object model, including OCAML, follow this approach of designing a specific calculus, e.g. [2, 4, 9, 20, 44]. However, there could be some benefits from deriving object-oriented concepts from more basic principles: first, their typing could be derived within simple, unquestionable typing systems. Second, they could be better integrated in a standard computational model, in which one could formalize and compare various approaches to objects, and get more flexible object models. Furthermore, we would not have to develop specific theories for reasoning about them.

In this paper we pursue Wand’s approach, aiming at encoding object-oriented concepts by means of extensible records. One may observe that the update operation of object calculi [2, 20] is actually overloaded: it serves both in inheritance, to override methods, and in the dynamic behaviour of an object, to update the state (see [2], Section 5.2). As we have seen, the first usage is not problematic in Wand’s model, whereas the second is. Then an obvious idea is to abandon the “functional update” approach in favor of a rather more natural *imperative update* approach, as in [2, 4, 18, 44]. This means that we are in a language with *references* (following ML’s terminology), where a call-by-value strategy is assumed for evaluation. Now a new problem arises: to build objects as recursive records one must have the ability to build recursive non-functional values, and this, in principle, is not supported in a typed call-by-value language. More specifically, we would like to use $(\text{let rec } x = N \text{ in } M)$, where N may be of a record type. This is evaluated by first computing a value for N , returning a cyclic binding to this value for x , and then computing M . Notice that side effects and creation of new references arising from the evaluation of N are completed before a cyclic binding is returned. This is what we need to install the state of an object before returning its (recursive) record of methods. The resulting object model is similar to what is known as the “cyclic record” encoding, see [2], Sections 18.2.4 and 18.3.4 (see also [17], which includes a discussion about how to compensate for the lack of a general fixpoint).

As remarked by Rémy [43], a recursive record semantics of objects works fine with the `let rec` construct, except that this construct is *unsafe*. Indeed, some languages, like SCHEME or OCAML, provide us with this feature, but, except for defining recursive functions, its semantics is implementation-dependent. More precisely, in computing $(\text{let rec } x = N \text{ in } M)$, it could happen that evaluating N we have to call the value of x , which is not yet computed, thus getting stuck at this point. An example, in the simply-typed call-by-value λ -calculus with recursion, is $(\text{let rec } x = \mathbf{F}(xV) \text{ in } \dots)$ where \mathbf{F} is the combinator $\lambda x \lambda y y$ and V is any typable value. One must then have means to prevent such a run-time error in order to design a “safe” object model from recursive records. We must point out that, although this problem of determining restrictions on recursive definitions to ensure that they define something is not at all a new one, no obvious solution to our specific problem emerges from the literature, since we have to sometimes accept $(\text{let rec } x = (Gx) \text{ in } M)$, especially when G reduces to a “generator” $\lambda \text{self } M$ [13].

The main contribution of this paper is a solution to this problem: first, we extend the core “Reference ML” language, as considered by Wright and Felleisen [52], with `let rec` and operations on records, similar to the ones of Cardelli and Mitchell [12]. We then provide a type system for this language, refining the simple typing by assigning a boolean information, 1 or 0 – that we call the “degree” –, to variables in the typing context. This “degree” is to be interpreted as “certainly safe”, or conversely “possibly unsafe”, for recursion. Typically, a variable occurring within a value is safe for recursion, hence may have degree 1 – this is basically the standard approach, where recursion is “guarded”, like for instance in $(\text{let rec } x = \lambda y N \text{ in } M)$. We also have to introduce degrees in function types, considering types of the form $\theta^d \rightarrow \tau$. Then a function of type $\theta^1 \rightarrow \tau$ is “protective” towards its argument, like for instance $\mathbf{K} = \lambda x \lambda y x$, or more generally $\lambda x V$. For a

$M, N \dots$	$::=$	$V \mid (MN) \mid (\text{let } D \text{ in } M)$	<i>expressions</i>
		$\mid \langle M, \ell = N \rangle \mid (M.\ell) \mid (M \setminus \ell)$	<i>core constructs</i>
			<i>record operations</i>
$V, W \dots$	$::=$	$x \mid \text{ref} \mid ! \mid \text{set} \mid (\text{set } V)$	<i>values</i>
		$\mid \lambda x M \mid () \mid R$	
R	$::=$	$x \mid \diamond \mid \langle R, \ell = V \rangle$	<i>record values</i>
D	$::=$	$x = N \mid \text{rec } x = N$	<i>declarations</i>

Figure 1: Syntax

recursion ($\text{let rec } x = (Gx) \text{ in } M$) to be safe, the function G must be “protective”. Regarding our type system, we prove the standard properties: we show that the evaluation of a typable term either diverges or returns a value, thus avoiding to get stuck in run-time errors, and, adapting a result by Jategaonkar and Mitchell [26], we show that a principal type may be computed for any typable expression. To assess the usefulness of the approach, and to illustrate the expressive power of the model, we introduce a few derived constructs for a *mixin-based* style of programming.

The rest of the paper is organized as follows: a first section introduces the language, from an operational point of view. We characterize in particular the possible outcomes of a computation. Section 3 introduces the type system. In the next one the type safety result is established. This relies upon the fact that substituting a term of appropriate type for a variable preserves the typing; the proof of this fact is more difficult than usual, because we have to deal with degrees. In Section 5 we present the type reconstruction algorithm, and prove the principal type property. In Section 6 we introduce our sub-language for mixin-based programming, and its derived typing. We illustrate the flexibility of the approach by means of a series of examples. Finally we discuss related work, and present some conclusions.

Note. The type system presented in this paper is similar to, but not the same as the one presented in the conference version of this work [6], which appeared to be not expressive enough for the purpose of typing the object-oriented constructs presented below.

2. The Calculus

Assuming that a set \mathcal{X} of variables, ranged over by $x, y, z \dots$, and a set \mathcal{L} of labels are given, the syntax of our core language is given in Figure 1, where $x \in \mathcal{X}$ and $\ell \in \mathcal{L}$. It contains the “Reference ML” calculus of [52] – defining the call-by-value fixpoint combinator Y as $(\text{let rec } y = \lambda f.f \lambda x.yfx \text{ in } y)$, where we use the standard abbreviations, namely $\lambda x_1 \dots x_n.M$ for $\lambda x_1 \dots \lambda x_n.M$ and $MN_1 \dots N_k$ for $(\dots (MN_1) \dots N_k)$, and denoting $:=$ by set . Free (**fv**) and bound (**bv**) variables are defined as usual, and we denote by $\{x \mapsto N\}M$ the capture-free substitution. As usual we write $(M ; N)$ for $(\text{let } x = M \text{ in } N)$ provided that x does not occur in N .

Regarding records, we use the operations of [12], denoting by $\langle M, \ell = N \rangle$ the record M extended with a new field, labelled ℓ , with value N . As in [12], this will only be well-typed if M does not exhibit an ℓ field, whereas the restriction operation, still denoted $(M \setminus \ell)$ and consisting of removing the ℓ field from M , will only be well-typed here if M does contain an ℓ field. This record calculus is equivalent to the one defined by means of pattern matching in [26]: using abstraction on patterns, one may define $(M.\ell)$ and $(M \setminus \ell)$ respectively as $\lambda \langle x, \ell = y \rangle y$ and $\lambda \langle x, \ell = y \rangle x$. Conversely, the expression $\lambda \langle x, \ell = y \rangle M$ for instance may be written $\lambda z (\text{let } x = z \setminus \ell \text{ in } (\text{let } y = z.\ell \text{ in } M))$. The overriding operation is denoted $\langle M, \ell \leftarrow N \rangle$; this is an abbreviation for $\langle (M \setminus \ell), \ell = N \rangle$. We may also define the renaming operation $M[\ell \leftarrow \ell'] = (\text{let } x = M \text{ in } \langle x \setminus \ell, \ell' = x.\ell \rangle)$. We shall write $\langle \ell_1 = M_1, \dots, \ell_n = M_n \rangle$ for the record $\langle \dots \langle \diamond, \ell_1 = M_1 \rangle \dots, \ell_n = M_n \rangle$.

$$\begin{aligned} \mathbf{E} ::= & \square \mid (\mathbf{E}N) \mid (V\mathbf{E}) \mid (\text{let } x = \mathbf{E} \text{ in } M) \mid (\text{let rec } x = \mathbf{E} \text{ in } M) \\ & \mid \langle \mathbf{E}, \ell = N \rangle \mid \langle R, \ell = \mathbf{E} \rangle \mid (\mathbf{E}.\ell) \mid (\mathbf{E} \setminus \ell) \end{aligned}$$

Figure 2: Evaluation Contexts

$$\begin{aligned} (\lambda x M V) &\rightarrow \{x \mapsto V\}M \\ (\text{let } x = V \text{ in } M) &\rightarrow \{x \mapsto V\}M \\ (\text{let rec } x = V \text{ in } M) &\rightarrow \{x \mapsto (\text{let rec } x = V \text{ in } V)\}M \\ (\langle R, \ell = V \rangle.\ell) &\rightarrow V \\ (\langle R, \ell = V \rangle.\ell') &\rightarrow (R.\ell') && \ell' \neq \ell \\ (\langle R, \ell = V \rangle \setminus \ell) &\rightarrow R \\ (\langle R, \ell = V \rangle \setminus \ell') &\rightarrow \langle (R \setminus \ell'), \ell = V \rangle && \ell' \neq \ell \\ M \rightarrow M' &\Rightarrow \mathbf{E}[M] \rightarrow \mathbf{E}[M'] \end{aligned}$$

Figure 3: Local Reduction

$$\begin{aligned} M \rightarrow M' &\Rightarrow [S \mid M] \rightarrow [S \mid M'] \\ [S \mid \mathbf{E}[(\text{ref } V)]] &\rightarrow [u := V ; S \mid \mathbf{E}[u]] && u \text{ fresh, } \text{fv}(V) \cap \text{capt}(\mathbf{E}) = \emptyset \\ [S \mid \mathbf{E}[(!u)]] &\rightarrow [S \mid \mathbf{E}[V]] && S(u) = V, \text{fv}(V) \cap \text{capt}(\mathbf{E}) = \emptyset \\ [S \mid \mathbf{E}[(\text{set } u)V]] &\rightarrow [\{u := V\}S \mid \mathbf{E}[\emptyset]] && \text{fv}(V) \cap \text{capt}(\mathbf{E}) = \emptyset \end{aligned}$$

Figure 4: Global Reduction

Now we specify the semantics of our language, defining an *evaluation relation* $M \rightarrow M'$, that we also call *local* (or functional) *reduction*, which can be performed in *evaluation contexts*. The axioms and rules are given in Figure 2. Here, as it is standard, we use a semantics by substitution for the let construct, and for β_v -reduction. This will simplify the proofs of the technical results, since we do not enter into the details of α -conversion, which is a standard matter. But obviously, an implementation would rather be closer to an abstract machine description, based on environments and closures. In this case, the semantics of $(\text{let rec } x = N \text{ in } M)$ is that of Landin's imperative fixpoint [29]: allocate a new address u , assign to it the value of $(\lambda x N)u$ and return the content of u as the value for x in M .

To describe the semantics of the imperative constructs, given by the rules for *global reduction* in Figure 3, we enrich the language with a denumerable set \mathcal{N} of *names*, or *locations* $u, v, w \dots$, distinct from the variables and the labels. These names are also values. A *configuration* is a pair $[S \mid M]$ of an expression M and a *store* S , that is a mapping from locations to values. We use the following syntax for stores:

$$S ::= \varepsilon \mid u := V ; S$$

The value $S(u)$ of a name in the store, and the partial operation $\{u := V\}S$ of updating the store, are defined in the obvious way. In the side conditions of the rules for global reduction, $\text{capt}(\mathbf{E})$ is the set of variables that are bound in \mathbf{E} by a let rec binder introducing a sub-context. Let us see an example – which will be the standard object-oriented example of a “point”. Assuming that some arithmetical operations are given, we define a “class” of unidimensional points as follows:

$$\begin{aligned} \text{let point} &= \lambda x \lambda \text{self} \langle \text{pos} = \text{ref } x, \\ &\quad \text{move} = \lambda y ((\text{set self.pos})(! \text{self.pos} + y)) \rangle \text{ in } \dots \end{aligned}$$

Within the scope of this definition, we may define a point object, instance of that class, by instantiating the position parameter x to some initial value, and building a recursive record of methods. Let us define the fixpoint operator `fix` as follows:

$$\text{fix} =_{\text{def}} \lambda f(\text{let rec } x = fx \text{ in } x)$$

Then we have for instance, if we let $V = \lambda y((\text{set } x.\text{pos})(!x.\text{pos} + y))$ and $R = \langle \text{pos} = u, \text{move} = V \rangle$:

$$\begin{aligned} [\varepsilon \mid \text{fix}(\text{point } 0)] &\xrightarrow{*} [\varepsilon \mid (\text{let rec } x = \langle \text{pos} = \text{ref } 0, \text{move} = V \rangle \text{ in } x)] \\ &\rightarrow [u := 0; \varepsilon \mid (\text{let rec } x = R \text{ in } x)] \\ &\xrightarrow{*} [u := 0; \varepsilon \mid \langle \text{pos} = u, \text{move} = \{x \mapsto O\}V \rangle] \end{aligned}$$

where $O = (\text{let rec } x = R \text{ in } R)$. One can see that there are two parts in this evaluated object: a state part, which records the (mutable) position of the object, and the (recursive, immutable) record of methods. Now imagine that we want to enhance the point class with a `clear` method that resets the position to the origin. Then we introduce a new class inheriting from `point`:

$$\text{let point}' = \lambda x \lambda \text{self} \langle (\text{point } x)\text{self}, \text{clear} = ((\text{set self.pos})0) \rangle \text{ in } \dots$$

However, we cannot create an object instance of that class. More precisely, the type system will reject an expression like `fix(point' 0)`, and rightly so. Indeed, if we try to compute this expression, we get stuck in $[u := 0; \varepsilon \mid (\text{let rec } x = \mathbf{E}[x] \text{ in } x)]$ where $\mathbf{E} = \langle \text{pos} = u, \text{move} = V, \text{clear} = ((\text{set } \square.\text{pos})0) \rangle$. In the `clear` method, the `self` parameter ought to be protected from being evaluated, and a standard way to do this is to define this method as a “thunk”, `clear = $\lambda y(\text{self.pos} := 0)$` , which may be invoked on an object, $o = \text{fix}(\text{point}' 42)$ for instance, as $o.\text{clear}()$. This is the main technical point of the paper: to create objects instance of some class, we must be able to sometimes accept, sometimes reject terms of the form $(\text{let rec } x = (Gx) \text{ in } N)$, in particular when $G \xrightarrow{*} \lambda \text{self } M$, depending on whether the function (with side effects) G is “protective” towards its argument or not. As we mentioned in the Introduction, unsafe recursion is not tied to record computations: it already shows up in the purely functional fragment of the language.

In the type system we will use a notion of a *pure* expression, which is an expression that can be evaluated without producing any side effect. In particular, evaluating a pure expression does not expand the store, and therefore such expressions have also been called non-expansive expressions. They are given by the following syntax:

$$U ::= V \mid (\text{let } x = U \text{ in } U') \mid (\text{let rec } x = U \text{ in } U') \mid (\text{set } U) \mid \langle U, \ell = U' \rangle \mid (U.\ell) \mid (U \setminus \ell)$$

LEMMA 2.1.

- (i) If U and U' are pure expressions, then $\{x \mapsto U\}U'$ is a pure expression.
- (ii) If M is a pure expression and $M \rightarrow M'$ then M' is a pure expression.

PROOF:

- (i) Immediate.
- (ii) By induction on the proof of $M \rightarrow M'$, using the previous point. \square

In order to establish a type safety result, we need to analyse the possible behaviour of expressions under evaluation: computing an expression may end on a value, or may go forever, but there are other possibilities – in particular an expression may “go wrong” [34] (or “be faulty”, following the terminology of [52]). Our analysis is slightly non-standard here, since we have to deal with open terms. Besides the faulty expressions, we distinguish what we call “global redexes” and “head expressions”, where a variable appears in a position where it has to be evaluated, and where something has to be done with its value.

DEFINITION 2.2. A term M is a global redex if M is $\mathbf{E}[(\text{ref } V)]$, or $\mathbf{E}[(!u)]$, or else $\mathbf{E}[(\text{set } u)V]$ for some value V and location u , with $\text{fv}(V) \cap \text{capt}(\mathbf{E}) = \emptyset$.

DEFINITION 2.3. A term M is a head expression if $M = \mathbf{H}[x]$ with $x \notin \text{capt}(\mathbf{H})$, where the \mathbf{H} contexts are given as follows:

$$\mathbf{H} ::= \mathbf{E}[(\square V)] \mid \mathbf{E}[(!\square)] \mid \mathbf{E}[(\text{set } \square)] \mid \mathbf{E}[(\square.\ell)] \mid \mathbf{E}[(\square \setminus \ell)]$$

DEFINITION 2.4. A term M is faulty if it contains a sub-expression of one of the following forms:

- (i) (VN) , where V is either a location, or $()$, or a record value;
- (ii) $(\text{let rec } x = \mathbf{H}[x] \text{ in } M)$ with $x \notin \text{capt}(\mathbf{H})$
- (iii) $(\text{let rec } x = \mathbf{E}[N] \text{ in } M)$ where N is either $(\text{ref } V)$ or $((\text{set } u)V)$ with $x \in \text{fv}(V)$;
- (iv) $(!V)$ or $(\text{set } V)$ where V is neither a variable nor a location;
- (v) $\langle V, \ell = N \rangle$ where V is not a record value;
- (vi) $(V.\ell)$ or $(V \setminus \ell)$, where V is neither a variable, nor a non-empty record-value.

Then our first result is:

PROPOSITION 2.5. For any expression M , either M reduces, i.e. $M \rightarrow M'$ for some M' , or M is a head expression, or a faulty expression, or a global redex, or a value.

PROOF: by induction on the structure of M . The case where M is a value is trivial.

1. In the case of an application (MN) , we use the induction hypothesis for M :

1.1. if M reduces into M' then (MN) reduces into $(M'N)$.

1.2. If M is a head expression, then (MN) is also a head expression.

1.3. If M is faulty, then (MN) is faulty too.

1.4. If M is a global redex, then the same holds for (MN) .

1.5. If M is a value V , we examine the possible cases for V . If V is neither a variable, nor a functional value (that is, V is either a location, or $()$, or a record value), then (VN) is a faulty expression. If V is a variable or a functional value, we use the induction hypothesis for N . If N reduces, or is a head expression, or is faulty, or else is a global redex, then the same holds for (VN) . If N is a value W , then we examine the possible cases for V : if V is a variable, then (VW) is a head expression. If $V = \text{ref}$ then (VW) is a global redex. If $V = !$ then either W is not a variable, nor a location, in which case (VW) is faulty, or (VW) is a global redex (if W is a location) or a head expression (if W is a variable). If $V = \text{set}$ then (VW) is a value (which may be faulty if W is neither a variable nor a location, and is also a head expression if W is a variable). If $V = (\text{set } V')$ then (VW) is a head expression if V' is a variable, or a global redex if V' is a location, and a faulty expression otherwise. Finally if V is an abstraction $\lambda x M'$, then (VW) reduces to $\{x \mapsto W\}M'$.

2. In the case of a let expression $(\text{let } x = N \text{ in } M)$, we use the induction hypothesis for N :

2.1-4. if N reduces, or is a head expression, or is faulty, or else is a global redex, then the same holds for $(\text{let } x = N \text{ in } M)$.

2.5. If N is a value V , then $(\text{let } x = N \text{ in } M)$ reduces to $\{x \mapsto V\}M$.

3. In the case of a let expression $(\text{let rec } x = N \text{ in } M)$, we use the induction hypothesis for N :

3.1. if N reduces, then $(\text{let rec } x = N \text{ in } M)$ reduces too.

3.2. If N is a head expression $\mathbf{H}[z]$, there are two cases: if $z = x$ then $(\text{let rec } x = N \text{ in } M)$ is faulty, and a head expression otherwise.

3.3. If N is faulty, then $(\text{let rec } x = N \text{ in } M)$ is faulty.

3.4. If N is a global redex $\mathbf{E}[N']$, then either there exist V and $x \in \text{fv}(V)$ such that $N' = (\text{ref } V)$ or $N' = ((\text{set } u)V)$, in which cases $(\text{let rec } x = N \text{ in } M)$ is faulty, or $(\text{let rec } x = N \text{ in } M)$ is a global redex.

3.5. If N is a value V , then $(\text{let rec } x = N \text{ in } M)$ reduces to $\{x \mapsto (\text{let rec } x = V \text{ in } V)\}M$.

All the other cases, that is $\langle M, \ell = N \rangle$, $(M.\ell)$, $(M \setminus \ell)$, are similar to the one of the application (MN) (and in fact simpler for the latter two). \square

Let us say that a configuration $[S \mid M]$ is *closed* if M is a closed expression, that is $\text{fv}(M) = \emptyset$, $S = u := V ; S'$ implies that V is closed, and the locations occurring in M are in $\text{dom}(S)$. It is easy to see that if $[S \mid M] \rightarrow [S' \mid N]$ and $[S \mid M]$ is closed, then $[S' \mid N]$ is closed too.

COROLLARY 2.6. *For any closed configuration $[S \mid M]$, if its evaluation terminates on $[S' \mid N]$, that is $[S \mid M] \xrightarrow{*} [S' \mid N]$, and $[S' \mid N]$ is irreducible, then N is either faulty or a value.*

3. The Type System

The aim in using a type system is to prevent run-time errors – and also to provide some interesting information about expressions. Then we have to design such a system in a way that rules out faulty expressions. Apart from the case of `let rec` expressions, that is (ii) and (iii) of Definition 2.4, we already know how to do that: use functional types $\theta \rightarrow \tau$ for expressions that have to be applied, reference types $\tau \text{ ref}$ for expressions that have to be de-referenced or assigned to, and record types, with row variables [41, 49], $\langle \rho, \ell_1 : \tau_1, \dots, \ell_k : \tau_k \rangle$ for expressions that are subject to selection, or extension, or restriction. Since we want to be able to define recursive expressions of any type, there is no specific type construct associated with recursion. However, to exclude unsafe recursion, we will use “decorated types”, where the decorations are boolean values 0 or 1 (with $0 \leq 1$), also called degrees, to which we must add, in order to obtain principal types, degree variables $p, q \dots$. We denote by $a, b, c \dots \in \mathcal{D}$ these *degrees*, either constant or variable.

Following Milner [34], we use a polymorphic `let` construct. This is crucial for defining classes that may be inherited in various ways, and instantiated into objects, and to type generic functions like $\lambda x(x.\ell)$ or $\lambda x \lambda y \langle x, \ell \leftarrow y \rangle$. Then we will use *type schemes*. As in [26], we do not allow the same label to occur several times in a given record type – but our treatment of row variables is quite different from the one of [26]. Therefore, in quantifying on a row variable, we must take into account the context in which it occurs, by means of the set L of labels that it must not contain. We call such a finite set of labels an *annotation*⁽¹⁾, and we have a simple “annotating” system to ensure that (record) types are well-formed. Given a set $\mathcal{T}y\mathcal{V}ar$ of type variables, the syntax of types and type schemes is:

$$\begin{aligned} \tau, \theta \dots & ::= \text{unit} \mid t \mid (\theta^a \rightarrow \tau) \mid \tau \text{ ref} \mid \rho \\ \rho & ::= t \mid \diamond \mid \langle \rho, \ell : \tau \rangle \\ \sigma, \zeta \dots & ::= \tau \mid (\forall C.\sigma) \end{aligned}$$

where t is any type variable, a is any degree, and $C = t_1 :: L_1, \dots, t_n :: L_n$. The quantification $(\forall t :: L.\sigma)$ is similar to the bounded quantification $(\forall t <: \diamond \setminus L.\sigma)$ of [12]. As for record expressions, we shall denote a record type $\langle \dots \langle \diamond, \ell_1 : \tau_1 \rangle \dots, \ell_k : \tau_k \rangle$ by $\langle \ell_1 : \tau_1 \dots, \ell_k : \tau_k \rangle$, and similarly $\langle t, \ell_1 : \tau_1 \dots, \ell_k : \tau_k \rangle$ stands for $\langle \cdot \cdot \langle t, \ell_1 : \tau_1 \rangle \dots, \ell_k : \tau_k \rangle$. Notice that we do not distinguish row variables from type variables – the former will generally have a non-empty annotation. We shall denote by $=_{\mathcal{T}}$ the least congruence on type schemes containing α -conversion of type variables bound by quantification, and satisfying the following law:

$$\langle \langle \rho, \ell : \tau \rangle, \ell' : \tau' \rangle = \langle \langle \rho, \ell' : \tau' \rangle, \ell : \tau \rangle$$

The annotation of type variables is not the only constraint we have to take into account in the type system: we also have to deal with constraints on degrees, that take the form of a set of inequalities $p \leq \alpha$ where p is a degree variable and α is a *degree expression*, built from degrees by using the

⁽¹⁾ we borrow this terminology from Fisher's thesis [19].

$$\begin{array}{c}
\frac{}{t :: L, C \vdash t :: L'} \quad L' \subseteq L \qquad \frac{}{C \vdash \text{unit} :: \emptyset} \qquad \frac{C \vdash \theta :: \emptyset \quad C \vdash \tau :: \emptyset}{C \vdash (\theta^d \rightarrow \tau) :: \emptyset} \qquad \frac{C \vdash \tau :: \emptyset}{C \vdash \tau \text{ ref} :: \emptyset} \\
\\
\frac{}{C \vdash \diamond :: L} \qquad \frac{C \vdash \rho :: L \cup \{\ell\} \quad C \vdash \tau :: \emptyset}{C \vdash \langle \rho, \ell : \tau \rangle :: L} \quad \ell \notin L \qquad \frac{t :: L, C \vdash \sigma :: \emptyset}{C \vdash (\forall t :: L. \sigma) :: \emptyset} \quad t \notin C \\
\\
\frac{}{C \vdash 0 \leq \alpha} \qquad \frac{}{C \vdash \alpha \leq 1} \qquad \frac{}{C \vdash \alpha \leq \alpha} \qquad \frac{C \vdash \alpha \leq \kappa \quad C \vdash \kappa \leq \beta}{C \vdash \alpha \leq \beta} \\
\\
\frac{}{p \leq \alpha, C \vdash p \leq \alpha} \qquad \frac{}{C \vdash (\alpha \wedge \beta) \leq \alpha} \qquad \frac{}{C \vdash (\alpha \wedge \beta) \leq \beta} \qquad \frac{C \vdash \kappa \leq \alpha \quad C \vdash \kappa \leq \beta}{C \vdash \kappa \leq (\alpha \wedge \beta)}
\end{array}$$

Figure 5: Constraints: Annotations and Inequalities

conjunction (that is, the meet) operation \wedge . We denote by $\alpha, \beta, \kappa \dots \in \mathcal{DExp}$ these expressions. The meet operation is used to represent conjunction of constraints; namely, $p \leq \alpha \wedge \beta$ is equivalent to $p \leq \alpha \ \& \ p \leq \beta$. Notice that constraints on degrees of this kind are obviously satisfiable, e.g. assigning uniformly 0 to the degree variables. In order to give a simple form to typing rules, we group the two kinds of constraints into a single component, called a *constraint*, still denoted by C . This is a pair of a mapping C_{typ} from a finite set $\text{dom}(C_{\text{typ}})$ of type variables into annotations (finite subsets of \mathcal{L}), and of a mapping C_{deg} from a finite set $\text{dom}(C_{\text{deg}})$ of degree variables into degree expressions. As usual, we write $t :: L, C$ for the constraint C updated by the assignment of annotation L to t , and similarly for $p \leq \alpha, C$. This notation is extended to C, C' in the obvious way. The constraint system, given in Figure 5, allows us to infer judgements of the form $C \vdash \sigma :: L$, which can be read “ σ is well-formed under the constraint C and is not a record type containing one of the labels in L ”, and of the form $C \vdash \alpha \leq \beta$, meaning that this inequality is a consequence of C . As one can see, the only constraints on types are that in $\langle \rho, \ell : \tau \rangle$, the type ρ must not contain the label ℓ , and that one may only quantify over a type variable which does not appear in the context, which we denote by $t \notin C$. Given a set A of assertions of the form $\sigma :: L$ or $\alpha \leq \beta$, we denote by $C \vdash A$ the fact that all the assertions of A are provable from the constraint C .

As usual, we need the notion of an instance of a type scheme, obtained by substituting not only types for type variables, but also degrees for degree variables. Then a type and degree substitution S is a mapping from type variables to types, and from degree variables to degrees (not degree expressions), which is the identity, except for a finite set $\text{dom}(S)$ of variables. We write $S = \{t_i \mapsto \tau_i \mid i \in I\} \cup \{p_j \mapsto a_j \mid j \in J\}$ if $S(t_i) = \tau_i$, $S(p_j) = a_j$, and S is the identity otherwise. If X is a set of (type and degree) variables, and S is a substitution, then $S \upharpoonright X$ is the substitution that coincides with S on X , and is the identity otherwise. We denote by $S(\sigma)$ the result of applying the (capture-free) substitution S to the type scheme σ , and similarly for $S(\alpha)$. The composition $S'S$ of substitutions is defined as follows:

$$S'S(t) = \begin{cases} S'(S(t)) & \text{if } t \in \text{dom}(S) \\ t & \text{otherwise} \end{cases}$$

Notice that $\text{dom}(S'S) = \text{dom}(S)$. In most cases, we need to ensure that applying a substitution to a type scheme results in a well-formed type. Given two constraints C_0 and C_1 , we then define $\text{Sub}(C_0, C_1)$ as follows:

$$S \in \text{Sub}(C_0, C_1) \quad \Leftrightarrow_{\text{def}} \quad \text{dom}(S) \subseteq \text{dom}(C_0) \ \& \ C_1 \vdash S(C_0)$$

where $S(C) = \{S(t) :: L \mid t :: L \in C\} \cup \{S(p) \leq S(\alpha) \mid p \leq \alpha \in C\}$. Then it is easy to see, by induction on σ , that the following holds:

LEMMA 3.1. *If $C_0 \vdash \sigma :: L$ and $S \in \text{Sub}(C_0, C_1)$ then $C_1 \vdash S(\sigma) :: L$.*

Then for instance the standard relation of being a *generic instance* (see [16]) is relative to some constraint: we write $C \vdash \sigma \succeq \sigma'$ if $\sigma = (\forall C'. \tau)$ and $\sigma' = (\forall C''. S(\tau))$ for some $S \in \text{Sub}(C', C \cup C'')$.

The typing judgements have the form $C ; \Gamma \vdash M : \tau$, where C is a constraint, τ is a type⁽²⁾, and Γ is a *typing context*. This maps a finite set of variables not only to type schemes, but also to degree expressions. The idea is that with a variable x we associate an assumption about the fact that it will or will not occur in a dangerous – w.r.t. recursion – position. This assumption is the *degree* of the variable in the context – 0 standing for “dangerous”, i.e. potentially unsafe. We also need to type locations, and therefore a context is a mapping from a finite set of variables to pairs (σ, α) , written σ^α , and from a finite set of locations to types. We shall write $\Gamma_{\text{typ}}(x) = \sigma$ and $\Gamma_{\text{deg}}(x) = \alpha$ if $\Gamma(x) = \sigma^\alpha$, and similarly $\Gamma_{\text{typ}}(u) = \Gamma(u)$. In what follows we denote by Ψ a type assignment, and by δ an assignment of degree expressions to variables. These are the two ingredients composing a typing context, e.g. $\Psi = \Gamma_{\text{typ}}$ and $\delta = \Gamma_{\text{deg}}$. We use the following predicate and operations on the typing contexts:

- (i) let X be a set of variables. Then $C \vdash \Delta \leq \Gamma$ on X if and only if $\Gamma_{\text{typ}} = \Delta_{\text{typ}}$ and $C \vdash \Delta_{\text{deg}}(x) \leq \Gamma_{\text{deg}}(x)$ for all $x \in X$;
- (ii) let δ be a mapping from variables to degree expressions. Then we define the context Γ^δ as follows: $(\Gamma^\delta)_{\text{typ}} = \Gamma_{\text{typ}}$ and $(\Gamma^\delta)_{\text{deg}}(x) = \delta(x)$.

For instance $\Gamma = \Psi^\delta$ if $\Psi = \Gamma_{\text{typ}}$ and $\delta = \Gamma_{\text{deg}}$. We use in particular this last notation when δ is $\lambda x(\text{if } x \in \text{fv}(M) \text{ then } \alpha \text{ else } 1)$, which is abbreviated into α_M . As usual, we let $\delta \wedge \gamma$ denote the function defined pointwise, by $(\delta \wedge \gamma)(x) = \delta(x) \wedge \gamma(x)$. We also abusively write 1 for $\lambda x.1$, and similarly for 0. In the typing axioms, we have to check that the types involved in the judgement are acceptable. This is done by means of a proof system for judgements of the form $C \vdash \Psi$, which is actually trivial:

$$\frac{}{C \vdash \emptyset} \qquad \frac{C \vdash \sigma :: \emptyset \quad C \vdash \Psi}{C \vdash x : \sigma, \Psi}$$

Now let us comment on some of the rules that are presented in Figures 6 and 7. The first one is a “degree weakening” rule, stating that “optimistic” assumptions, assigning for instance degree 1 to some variables, can always be safely downgraded. The intuition about degrees is that a variable is safe, and therefore may be assigned degree 1, basically when it occurs within a value, that is guarded by a λ -abstraction, and more generally when a value for it is not required during the computation. This explains the typing axiom $C ; x : \sigma^0, \Psi^1 \vdash x : \tau$: a value for x must be fetched, say, from the environment, to evaluate the expression x , and therefore x has degree 0. On the other hand, any other variable is not concerned with the evaluation of x , hence may be assigned degree 1. More generally, in all typing axioms we make such “optimistic” degree assumptions, which can always be weakened, regarding the variables that do not occur in the typed expression.

In the rule for abstraction of x , we assume that the degree of x does not contain the \wedge operation, but this is not a restriction, since we may always add a fresh constraint $p \leq \alpha$ and use the weakening rule. The rule for abstraction promotes the typing context to a definitely safe one (Γ^1), since all the variables occurring in the abstraction value are protected from being evaluated by the λ .

⁽²⁾ to simplify the presentation we do not include the usual rules of instantiation and generalization (see [16]), but they would easily be shown to be admissible if judgements $C ; \Gamma \vdash M : \sigma$ were allowed, and therefore we will use them in the examples.

$$\begin{array}{c}
\frac{C; \Gamma \vdash M : \tau \quad C \vdash \Delta \leq \Gamma \text{ on } \text{fv}(M)}{C; \Delta \vdash M : \tau} \quad \frac{C \vdash x : \sigma, \Psi \quad C \vdash \sigma \succeq \tau}{C; x : \sigma^0, \Psi^1 \vdash x : \tau} \\
\frac{C; x : \theta^a, \Gamma \vdash M : \tau}{C; \Gamma^1 \vdash \lambda x M : (\theta^a \rightarrow \tau)} \quad \frac{C; \Gamma \vdash M : \theta^a \rightarrow \tau \quad C; \Gamma \vdash N : \theta}{C; \Gamma^{0_M \wedge \delta} \vdash (MN) : \tau} \quad (1) \\
\frac{C', C; \Gamma \vdash N : \theta \quad C; x : (\forall C'. \theta)^\alpha, \Gamma \vdash M : \tau}{C; \Gamma^\delta \vdash (\text{let } x = N \text{ in } M) : \tau} \quad (2) \\
\frac{C', C; x : \theta^1, \Gamma \vdash N : \theta \quad C; x : (\forall C'. \theta)^\alpha, \Gamma \vdash M : \tau}{C; \Gamma^\delta \vdash (\text{let rec } x = N \text{ in } M) : \tau} \quad (2)
\end{array}$$

(1) where

$$\delta(x) = \begin{cases} a & \text{if } N = x \notin \text{fv}(M) \\ 1 & \text{if } x \notin \text{fv}(N) \\ a \wedge \Gamma_{\text{deg}}(x) & \text{otherwise} \end{cases}$$

(2) where $t \in \text{dom}(C') \Rightarrow t \notin \text{dom}(C)$ and C' is empty if N is not pure, and

$$\delta(y) = \begin{cases} \alpha \wedge \Gamma_{\text{deg}}(y) & \text{if } y \in \text{fv}(N) \text{ and } M \text{ is not pure} \\ \Gamma_{\text{deg}}(y) & \text{otherwise} \end{cases}$$

Figure 6: The Type System (Functional Fragment)

Conversely, the variables occurring in the function part of an application are potentially dangerous, like for instance x in $(\lambda y. xy)V$. Then they are all downgraded to having the degree 0. Regarding the argument, we must be more careful: applying a function of type $\theta^1 \rightarrow \tau$ that does not put its argument in danger, like $\lambda f \lambda x (fx)$ for instance (see below), we may decide that its free variables are protected. However, this is only true if they are protected in the argument itself. Then applying a function of type $\theta^a \rightarrow \tau$ places the argument in a position where the variables have a degree which is, at best, a or the degree they have in the argument. This is where we use the \wedge operation⁽³⁾. Nevertheless, we are able to make a special case when the argument is a variable (whose degree is 0): anticipating that the function is an abstraction $\lambda y N$, thus typed using an assumption about y , we may consider that its degree is the one of y , instead of $a \wedge 0 = 0$ (see the Lemma 4.5 below). This is important for our purpose, where we should not always reject $(\text{let rec } x = Gx \text{ in } M)$. Let us see an example. Since

$$\frac{\begin{array}{c} \vdots \\ \hline C \vdash \alpha \leq \beta \quad C \vdash \alpha \leq 1 \end{array}}{C \vdash \alpha \leq \beta \wedge 1}$$

we have, for the term $\lambda f \lambda x (fx)$, the typing shown in Figure 8 where $C = t :: \emptyset, t' :: \emptyset, q \leq p$ (omitting the proof of $C \vdash f : (t^p \rightarrow t'), x : t$). In the examples that follow, we shall often omit the use of the degree weakening rule, when it amounts to use obvious inequalities like $\alpha \leq \alpha \wedge 1$. To see why we need the meet operation, the reader may try to type $f(gx)$, where the degree of x depends on the nature of both f and g . In the Introduction, we said that we may call “protective” a function whose

⁽³⁾ For simplicity, our presentation of the type system is “additive”, in the sense that the components of a binary construct share the same typing context. However, we may have to use the degree weakening rule, hence the meet, to keep to this pattern.

$$\begin{array}{c}
\frac{C \vdash u : \tau, \Psi}{C ; u : \tau, \Psi^1 \vdash u : \tau \text{ ref}} \quad \frac{C \vdash \Psi \quad C \vdash \tau :: \emptyset}{C ; \Psi^1 \vdash \text{ref} : \tau^0 \rightarrow \tau \text{ ref}} \quad \frac{C \vdash \Psi \quad C \vdash \tau :: \emptyset}{C ; \Psi^1 \vdash ! : (\tau \text{ ref})^0 \rightarrow \tau} \\
\frac{C \vdash \Psi \quad C \vdash \tau :: \emptyset}{C ; \Psi^1 \vdash \text{set} : (\tau \text{ ref})^0 \rightarrow \tau^0 \rightarrow \text{unit}} \quad \frac{C \vdash \Psi}{C ; \Psi^1 \vdash () : \text{unit}} \\
\frac{C \vdash \Psi}{C ; \Psi^1 \vdash \diamond : \diamond} \quad \frac{C ; \Gamma \vdash M : \rho \quad C ; \Gamma \vdash N : \tau \quad C \vdash \rho :: \{\ell\}}{C ; \Gamma \vdash \langle M, \ell = N \rangle : \langle \rho, \ell : \tau \rangle} \\
\frac{C ; \Gamma \vdash M : \langle \rho, \ell : \tau \rangle}{C ; \Gamma \vdash (M.\ell) : \tau} \quad \frac{C ; \Gamma \vdash M : \langle \rho, \ell : \tau \rangle}{C ; \Gamma \vdash (M \setminus \ell) : \rho} \\
\frac{C ; \Gamma \vdash \varepsilon}{C ; \Gamma \vdash \varepsilon} \quad \frac{C ; u : \tau, \Gamma \vdash V : \tau \quad C ; u : \tau, \Gamma \vdash S}{C ; u : \tau, \Gamma \vdash u := V ; S} \quad \frac{C ; \Gamma \vdash S \quad C ; \Gamma \vdash M : \tau}{C ; \Gamma \vdash [S \mid M] : \tau}
\end{array}$$

Figure 7: The Type System (Continued)

$$\begin{array}{c}
\vdots \quad \vdots \\
\frac{C ; f : (t^p \rightarrow t')^0, x : t^1 \vdash f : (t^p \rightarrow t')}{C ; f : (t^p \rightarrow t')^0, x : t^0 \vdash f : (t^p \rightarrow t')} \quad \frac{C ; f : (t^p \rightarrow t')^1, x : t^0 \vdash x : t}{C ; f : (t^p \rightarrow t')^0, x : t^0 \vdash x : t} \\
\frac{C ; f : (t^p \rightarrow t')^{0 \wedge 1}, x : t^{1 \wedge p} \vdash (fx) : t'}{C ; f : (t^p \rightarrow t')^0, x : t^q \vdash (fx) : t'} \\
\frac{C ; f : (t^p \rightarrow t')^1 \vdash \lambda x (fx) : t^q \rightarrow t'}{C ; \vdash \lambda f \lambda x (fx) : (t^p \rightarrow t')^1 \rightarrow t^q \rightarrow t'}
\end{array}$$

Figure 8: Typing the Application

type is of the form $\theta^1 \rightarrow \tau$. Since we are using degree variables, this has to be slightly extended: we say that M is *protective* if it has a type $\theta^a \rightarrow \tau$ in a context $C ; \Gamma$ such that $C \not\vdash a \leq 0$. For instance, this is the case with $C ; \vdash \mathbf{K} : \tau^p \rightarrow \theta^q \rightarrow \tau$ where $\mathbf{K} = \lambda x \lambda y x$ (and $C \vdash \tau :: \emptyset$ and $C \vdash \theta :: \emptyset$). This is a more general typing than $C ; \vdash \mathbf{K} : \tau^1 \rightarrow \theta^q \rightarrow \tau$.

One may notice that in the rule for the let construct, we use the “value polymorphism” approach of SML [35] (also proposed by Wright [53]), to solve the difficulties in polymorphic typing due to imperative features. The rule for the let `rec` construct is the only one involving a real – i.e. possibly unsatisfiable – constraint on degrees, namely $1 \leq \alpha$. It is exemplified by the following typing of the fixpoint combinator, where $\Gamma = f : (t^1 \rightarrow t)^0, x : t^0$ and $\Delta = f : (t^1 \rightarrow t)^0, x : t^1$:

$$\begin{array}{c}
\vdots \quad \vdots \\
\frac{t :: \emptyset ; \Gamma \vdash f : (t^1 \rightarrow t) \quad t :: \emptyset ; \Gamma \vdash x : t}{t :: \emptyset ; \Delta \vdash fx : t} \quad \frac{\vdots}{t :: \emptyset ; \Gamma \vdash x : t} \\
\frac{t :: \emptyset ; f : (t^1 \rightarrow t)^0 \vdash (\text{let rec } x = fx \text{ in } x) : t}{t :: \emptyset ; \vdash \text{fix} : (t^1 \rightarrow t)^0 \rightarrow t}
\end{array}$$

Notice that, as in ML, (let $\text{rec } f = \lambda x N$ in M) is always allowed, provided that M and N have appropriate typings. For instance, the call-by-value fixpoint combinator

$$Y = (\text{let rec } y = \lambda f.f(\lambda x.(yf)x) \text{ in } y)$$

has the following typing:

$$t :: \emptyset, t' :: \emptyset, p \leq q; \vdash Y : ((t^p \rightarrow t')^r \rightarrow t^q \rightarrow t')^0 \rightarrow t^q \rightarrow t'$$

An example of a program from the “functional ML” fragment of the language that is rejected by our type system – as any other faulty expression, as we shall see –, is (let $\text{rec } x = \mathbf{F}(xV)$ in x) where \mathbf{F} is the combinator $\lambda z \lambda y y$. Observe that this expression would be typable – provided that V has some type – with a “standard” typing rule for let rec expressions, that is omitting the constraint on the degree of the recursive variable, but that it is an irreducible expression which is not a value. Conversely, types are, as usual, only approximations, and some expressions are rejected that actually do not cause any trouble. For instance, assigning degree 0 to variables that appear in a function position, that is at the left of an application, is sometimes overly pessimistic, as in (let $\text{rec } x = (\mathbf{F}x)V$ in \dots) where $x \notin \text{fv}(V)$ for instance.

The functional core of the language concentrates all the subtleties of the use of degrees – the rest of the type system is quite trivial, and in particular there is not much choice in the typing of the record constructs: as we said, the extension operation $\langle M, \ell = N \rangle$ is strict, and thus requires that M is a record not containing the label ℓ , and similarly the strict restriction operation $(M \setminus \ell)$ requires ℓ to be present in M . Then there is no ambiguity in typing these operations. Admittedly, having to check the well-formedness of record types complicates the system, in a way that has nothing to do with the problem of typing safe recursion, but we think that solving this problem is only worth if this has some application, and that an interesting one is in modelling object-oriented constructions.

4. Type Safety

In this section we prove our first technical result, asserting that typable programs cannot “go wrong”, that is they do not entail any run-time error. A first step towards this property is the following:

LEMMA 4.1. *The faulty expressions are not typable.*

PROOF: the type system is compositional, and therefore a term is typable only if all its subterms are typable. Then it is enough to check that the “basic” faulty expressions, as defined by the clauses (i-vi) of the Definition 2.4 are not typable. The cases of (i) and (iv-vi) are immediate. Regarding (ii), one first notices that if $C; \Gamma \vdash M : \tau$ where M is either (xV) , or $(!x)$, or $(\text{set } x)$, or else $(x.\ell)$ or $(x \setminus \ell)$, then $C \vdash \Gamma_{\text{deg}}(x) \leq 0$. Moreover, one can prove by induction on \mathbf{E} the following fact: if M is not a variable, and $C'; \Delta \vdash M : \theta$ implies $C' \vdash \Delta_{\text{deg}}(x) \leq 0$, and $C; \Gamma \vdash \mathbf{E}[M] : \tau$ where $x \notin \text{capt}(\mathbf{E})$, then $C \vdash \Gamma_{\text{deg}}(x) \leq 0$. Therefore $C; \Gamma \vdash \mathbf{H}[x] : \tau$ implies $C \vdash \Gamma_{\text{deg}}(x) \leq 0$ if $x \notin \text{capt}(\mathbf{H})$. Then (let $\text{rec } x = \mathbf{H}[x]$ in M) is not typable, since this would imply $C \vdash 1 \leq 0$, which is impossible, for C is consistent. The proof is similar in the case (iii). \square

Then we have the standard “type preservation” – or “subject reduction” – property. To establish this property, we need some preliminary results. First we observe that if $C; \Gamma \vdash M : \tau$ is provable, then $C \vdash \Gamma_{\text{typ}}$ and $C \vdash \tau :: \emptyset$. The following weakening property is standard:

LEMMA (WEAKENING) 4.2.

- (i) If $C; \Gamma \vdash M : \tau$ then for any x and σ such that $C \vdash \sigma :: \emptyset$ the judgement $C; \Gamma, x : \sigma^\alpha \vdash M : \tau$ is provable, with a proof having the same structure as the one of $C; \Gamma \vdash M : \tau$.
- (ii) If $C; \Gamma \vdash M : \tau$ then for any u and θ such that $C \vdash \theta :: \emptyset$ the judgement $C; \Gamma, u : \theta \vdash M : \tau$ is provable, with a proof having the same structure as the one of $C; \Gamma \vdash M : \tau$.
- (iii) If $C; \Gamma \vdash M : \tau$ and C' is a constraint such that $\text{dom}(C') \cap \text{dom}(C) = \emptyset$, then $C', C; \Gamma \vdash M : \tau$ is provable, with a proof having the same structure as the one of $C; \Gamma \vdash M : \tau$.

We omit the similar statements regarding the judgements $C ; \Gamma \vdash S$ and $C ; \Gamma \vdash [S \mid M] : \tau$. Then, given a substitution $S \in \text{Sub}(C_0, C_1)$, a typing context Γ and a type τ such that $C_0 \vdash \Gamma_{\text{typ}}$ and $C_0 \vdash \tau :: \emptyset$, we denote by $S(\Gamma \vdash M : \tau)$ the statement $\Delta \vdash M : S(\tau)$ where Δ is given by $\Delta(u) = S(\Gamma(u))$ and $\Delta(x) = S(\sigma)^{S(\alpha)}$ if $\Gamma(x) = \sigma^\alpha$. We have the standard result that typing is compatible with type substitution (see [16]):

LEMMA 4.3. *If $C ; \Gamma \vdash M : \tau$ and $S \in \text{Sub}(C, C')$ then $C' ; S(\Gamma \vdash M : \tau)$ is provable, with a proof having the same structure as the one of $C ; \Gamma \vdash M : \tau$.*

PROOF: by induction on the inference of $C ; \Gamma \vdash M : \tau$, straightforward. \square

LEMMA 4.4.

- (i) *If $C ; x : \sigma^\alpha, \Gamma \vdash M : \tau$ and $x \notin \text{fv}(M)$ then $C ; \Gamma \vdash M : \tau$ is provable, with a proof having the same structure as the one of $C ; x : \sigma^\alpha, \Gamma \vdash M : \tau$.*
- (ii) *If $C ; u : \theta, \Gamma \vdash M : \tau$ and u does not occur in M then $C ; \Gamma \vdash M : \tau$ is provable, with a proof having the same structure as the one of $C ; u : \theta, \Gamma \vdash M : \tau$.*
- (iii) *If $C ; u : \theta, \Gamma \vdash S$ and u does not occur in S then $C ; \Gamma \vdash S$ is provable, with a proof having the same structure as the one of $C ; u : \theta, \Gamma \vdash S$.*
- (iv) *If $C ; u : \theta, \Gamma \vdash [S \mid M] : \tau$ and u does not occur in S nor in M then $C ; \Gamma \vdash [S \mid M] : \tau$ is provable, with a proof having the same structure as the one of $C ; u : \theta, \Gamma \vdash [S \mid M] : \tau$.*
- (v) *If $t :: L, C ; \Gamma \vdash M : \tau$ and t does not occur free in Γ or τ then $C ; \Gamma \vdash M : \tau$ is provable, with a proof having the same structure as the one of $t :: L, C ; \Gamma \vdash M : \tau$.*

The proof, by induction on the inference of the typing judgements, is trivial. As usual, a crucial property to show is a “substitution lemma”, relating typing and substitution. As a special case, we first show:

LEMMA 4.5. $C ; x : \sigma^\alpha, \Gamma \vdash M : \tau \ \& \ y \notin \text{dom}(\Gamma) \Rightarrow C ; y : \sigma^\alpha, \Gamma \vdash \{y \mapsto x\}M : \tau$

PROOF: by induction on the inference of $C ; x : \sigma^\alpha, \Gamma \vdash M : \tau$, trivial. \square

To state our crucial proposition, let us denote by $\Gamma \asymp \Delta$ the fact that Γ and Δ assign the same types, but possibly different degrees, to the same variables and locations, that is $\Gamma_{\text{typ}} = \Delta_{\text{typ}}$.

PROPOSITION 4.6. *If C' is a type constraint (i.e. $C' = C'_{\text{typ}}$) such that $\text{dom}(C') \cap \text{dom}(C_{\text{typ}}) = \emptyset$, and N is pure, then the following rule is admissible:*

$$\frac{C', C ; \Gamma \vdash N : \theta \quad C ; x : (\forall C'. \theta)^\alpha, \Delta \vdash M : \tau}{C ; \Gamma^{\gamma \wedge \delta} \vdash \{x \mapsto N\}M : \tau} \Gamma \asymp \Delta$$

where $\gamma(y) = \alpha \wedge \Gamma_{\text{deg}}(y)$ if M is not pure and $y \in \text{fv}(N)$, and $\gamma(y) = \Gamma_{\text{deg}}(y)$ otherwise, and $\delta(y) = \Delta_{\text{deg}}(y)$ if $y \in \text{fv}(M)$, and $\delta(y) = 1$ otherwise.

PROOF: first we observe that, if $x \notin \text{fv}(M)$, then $\{x \mapsto N\}M = M$ and $C ; \Delta \vdash M : \tau$ by the Lemma 4.4, and the proposition is easily established in this case, by using the degree weakening rule. Therefore we assume $x \in \text{fv}(M)$ for the rest of the proof.

We proceed by induction on the inference of $C ; x : (\forall C'. \theta)^\alpha, \Delta \vdash M : \tau$, and by case on the last rule used to infer this sequent. This rule can only be either the degree weakening rule, or a rule depending on the structure of M . In the first case, we simply use the induction hypothesis, and the degree weakening rule to conclude. Then we assume for the rest of the proof that the last rule is not the degree weakening rule; that is, we proceed by induction on the structure of M .

- If $M = x$, we have $C \vdash (\forall C'. \theta) \succeq \tau$, that is $\tau = S(\theta)$ for some $S \in \text{Sub}(C', C)$, hence also $S \in \text{Sub}(C'', C)$ where $C'' = C', C$. By the Lemma 4.3 we have $C ; \Gamma \vdash N : \tau$, since S is the identity

for the variables occurring in Γ , which are included into $\text{dom}(C)$. This concludes the proof in this case, where $\gamma = \Gamma_{\text{deg}}$.

- If $M = \lambda z M'$ then we have $C ; x : (\forall C'. \theta)^\beta, z : \tau_0^a, \Delta' \vdash M' : \tau_1$ for some β , with $\alpha = 1$, $\Delta = \Delta'^1$ and $\tau = \tau_0^a \rightarrow \tau_1$. We may assume that $z \notin \text{dom}(\Gamma)$, hence $z \notin \text{fv}(N)$. Then by the Lemma 4.2 we also have $C', C ; z : \tau_0^a, \Gamma \vdash N : \theta$, and therefore $C ; (z : \tau_0^a, \Gamma)^\xi \vdash \{x \mapsto N\} M' : \tau_1$ by induction hypothesis, for some ξ such that $\xi(z) = a$. Then $C ; z : \tau_0^a, \Gamma^\xi \vdash \{x \mapsto N\} M' : \tau_1$, and therefore $C ; \Gamma^1 \vdash \{x \mapsto N\} M : \tau$ by the typing rule for abstraction, and we conclude using the degree weakening rule.

- If $M = (M_0 M_1)$ then there exist a, τ_0 and Σ such that $C ; \Sigma \vdash M_0 : \tau_0^a \rightarrow \tau$ and $C ; \Sigma \vdash M_1 : \tau_0$, with $x : (\forall C'. \theta)^\alpha, \Delta = \Sigma^{0_{M_0} \wedge \xi}$ where

$$\xi(y) = \begin{cases} a & \text{if } M_1 = y \notin \text{fv}(M_0) \\ 1 & \text{if } y \notin \text{fv}(M_1) \\ a \wedge \Sigma_{\text{deg}}(y) & \text{otherwise} \end{cases}$$

It must be that $\Sigma = x : (\forall C'. \theta)^\beta, \Theta$, and obviously $\Gamma \asymp \Theta$, therefore, by induction hypothesis, $C ; \Gamma^{\gamma_0 \wedge \delta_0} \vdash M'_0 : \tau_0^a \rightarrow \tau$ where $M'_0 = \{x \mapsto N\} M_0$ and

$$\gamma_0(y) = \begin{cases} \alpha \wedge \Gamma_{\text{deg}}(y) & \text{if } M_0 \text{ is not pure and } y \in \text{fv}(N) \\ \Gamma_{\text{deg}}(y) & \text{otherwise} \end{cases}$$

and $\delta_0(y) = \Theta_{\text{deg}}(y)$ if $y \in \text{fv}(M_0)$, and $\delta_0(y) = 1$ otherwise. We notice that, since M is not pure, we have

$$\gamma(y) = \begin{cases} \alpha \wedge \Gamma_{\text{deg}}(y) & \text{if } y \in \text{fv}(N) \\ \Gamma_{\text{deg}}(y) & \text{otherwise} \end{cases}$$

and obviously $C \vdash \delta = 0_{M_0} \wedge \xi$ on $\text{fv}(M)$. Let $M'_1 = \{x \mapsto N\} M_1$. We examine the various cases.

1. If M_1 is a variable z which does not occur in $\text{fv}(M_0)$, then we have

$$\xi(y) = \begin{cases} a & \text{if } y = z \\ 1 & \text{otherwise} \end{cases}$$

1.1. If $z = x$, we have $M'_1 = N$ and $M'_0 = M_0$, and also $C \vdash \alpha = \xi(x) = a$. By the Lemma 4.2 and the degree weakening rule, we have $C ; \Delta \vdash M_0 : \tau_0^a \rightarrow \tau$, and therefore there exists Π (with $C \vdash \Pi \leq \Gamma$ and $C \vdash \Pi \leq \Delta$) such that $C ; \Pi \vdash M_0 : \tau_0^a \rightarrow \tau$ and $C ; \Pi \vdash N : \tau_0$, by the degree weakening rule. Then

$$C ; \Gamma^{0_{M_0} \wedge \xi'} \vdash \{x \mapsto N\} M : \tau$$

by the rule for application, where

$$\xi'(y) = \begin{cases} a & \text{if } N = y \notin \text{fv}(M_0) \\ 1 & \text{if } y \notin \text{fv}(N) \\ a \wedge \Gamma_{\text{deg}}(y) & \text{otherwise} \end{cases}$$

It is easy to see that $C \vdash \gamma \leq \xi'$ on $\text{fv}(N)$, and therefore we conclude $C ; \Gamma^{\gamma \wedge \delta} \vdash \{x \mapsto N\} M : \tau$ by the degree weakening rule.

1.2. If $z \neq x$, we have $M'_1 = z$ and $C \vdash \alpha = 0$ since $x \in \text{fv}(M_0)$, and therefore

$$C \vdash \gamma(y) = \begin{cases} 0 & \text{if } y \in \text{fv}(N) \\ \Gamma_{\text{deg}}(y) & \text{otherwise} \end{cases}$$

Since $C ; \Gamma^{\gamma_0 \wedge \delta_0} \vdash M'_0 : \tau_0^a \rightarrow \tau$, we have, by the rule for application

$$C ; \Gamma^{0_{M'_0} \wedge \xi'} \vdash \{x \mapsto N\}M : \tau$$

where

$$\xi'(y) = \begin{cases} a & \text{if } y = z \notin \text{fv}(N) \\ 1 & \text{if } y \neq z \\ a \wedge \Gamma_{\text{deg}}(y) & \text{if } y = z \in \text{fv}(N) \end{cases}$$

Since $\gamma(y) = 0$ for $y \in \text{fv}(N)$ and $C \vdash \delta(y) = \Delta_{\text{deg}}(y) \leq 0$ for $y \in \text{fv}(M_0)$, we have $C \vdash \gamma \wedge \delta \leq 0_{M'_0}$ on $\text{fv}(M'_0)$, and $C \vdash \gamma(z) \leq \xi'(z)$ if $z \in \text{fv}(N)$. If $z \notin \text{fv}(N)$, we have

$$C \vdash \delta(z) = \Delta_{\text{deg}}(z) \leq \xi(z) = a = \xi'(z)$$

This shows that $C \vdash \gamma \wedge \delta \leq 0_{M'_0} \wedge \xi'$ on $\text{fv}(\{x \mapsto N\}M)$, and we conclude using the degree weakening rule.

2. Otherwise, M_1 is not a variable, or if it is a variable, this variable occurs free in M_0 . Since

$$C \vdash \alpha = \begin{cases} 0 & \text{if } x \in \text{fv}(M_0) \\ \xi(x) & \text{otherwise} \end{cases}$$

we have

$$C \vdash \alpha = \begin{cases} 0 & \text{if } x \in \text{fv}(M_0) \\ a \wedge \beta & \text{otherwise} \end{cases}$$

since $x \in \text{fv}(M)$. This shows that $C \vdash \alpha \leq \beta$. By induction hypothesis $C ; \Gamma^{\gamma_1 \wedge \delta_1} \vdash M'_1 : \tau_0$ where

$$\gamma_1(y) = \begin{cases} \beta \wedge \Gamma_{\text{deg}}(y) & \text{if } M_1 \text{ is not pure and } y \in \text{fv}(N) \\ \Gamma_{\text{deg}}(y) & \text{otherwise} \end{cases}$$

and

$$\delta_1(y) = \begin{cases} \Theta_{\text{deg}}(y) & \text{if } y \in \text{fv}(M_1) \\ 1 & \text{otherwise} \end{cases}$$

Let $\pi = \gamma_0 \wedge \delta_0 \wedge \gamma_1 \wedge \delta_1$, so that $C ; \Gamma^\pi \vdash M'_0 : \tau_0^a \rightarrow \tau$ and $C ; \Gamma^\pi \vdash M'_1 : \tau_0$. Then by the rule for application, we have

$$C ; \Gamma^{0_{M'_0} \wedge \xi'} \vdash \{x \mapsto N\}M : \tau$$

where, since M'_1 cannot be a variable which does not occur in M'_0 :

$$\xi'(y) = \begin{cases} a \wedge \pi(y) & \text{if } y \in \text{fv}(M'_1) \\ 1 & \text{otherwise} \end{cases}$$

We conclude the proof by showing that $C \vdash (\gamma \wedge \delta)(y) \leq (0_{M'_0} \wedge \xi')(y)$ for $y \in \text{fv}(M'_0 M'_1)$. This is true for $y \in \text{fv}(M_0)$, since $C \vdash \delta(y) \leq 0$ in this case. If $y \in \text{fv}(M_1) - (\text{fv}(N) \cup \text{fv}(M_0))$, we have $y \neq x$ and

$$C \vdash \delta(y) = \Delta_{\text{deg}}(y) = \xi(y) = a \wedge \Sigma_{\text{deg}}(y)$$

We also have $\xi'(y) = a \wedge \pi(y)$, and

$$\begin{aligned} \gamma_0(y) &= \gamma_1(y) = \Gamma_{\text{deg}}(y) = \gamma(y) \\ \delta_0(y) &= 1 \\ \delta_1(y) &= \Theta_{\text{deg}}(y) = \Sigma_{\text{deg}}(y) \end{aligned}$$

and therefore

$$C \vdash \xi'(y) = a \wedge \gamma(y) \wedge \Sigma_{\text{deg}}(y)$$

whence $C \vdash (\gamma \wedge \delta)(y) = \xi'(y) = (0_{M'_0} \wedge \xi')(y)$ for $y \in \text{fv}(M_1) - (\text{fv}(N) \cup \text{fv}(M_0))$. Finally let $y \in \text{fv}(N) - \text{fv}(M)$. If $x \in \text{fv}(M_0)$ then $\alpha = 0$, and since M is not pure, we have $C \vdash \gamma(y) = 0$, and we are done in this case. Otherwise, $x \in \text{fv}(M_1)$, hence

$$C \vdash \alpha = \xi(x) = a \wedge \Sigma_{\text{deg}}(x) = a \wedge \beta$$

and therefore $C \vdash \gamma(y) = a \wedge \beta \wedge \Gamma_{\text{deg}}(y)$. Then $C \vdash \gamma(y) \leq \xi'(y)$ in this case, since

$$\begin{aligned} \gamma_0(y) &\geq \alpha \wedge \Gamma_{\text{deg}}(y) = a \wedge \beta \wedge \Gamma_{\text{deg}}(y) \\ \delta_0(y) &= \delta_1(y) = 1 \\ \gamma_1(y) &\geq \beta \wedge \Gamma_{\text{deg}}(y) \end{aligned}$$

• If $M = (\text{let } z = M_0 \text{ in } M_1)$ then $x : (\forall C'.\theta)^\alpha, \Delta = \Sigma^\xi$ with $C'', C ; \Sigma \vdash M_0 : \theta'$, where the constraint C'' is empty if M_0 is not pure, and $C ; z : (\forall C''.\theta')^\beta, \Sigma \vdash M_1 : \tau$ and

$$\xi(y) = \begin{cases} \beta \wedge \Sigma_{\text{deg}}(y) & \text{if } M_1 \text{ is not pure and } y \in \text{fv}(M_0) \\ \Sigma_{\text{deg}}(y) & \text{otherwise} \end{cases}$$

We may assume that $z \notin \text{dom}(\Gamma)$ (and $z \neq x$), hence also $z \notin \text{fv}(N)$, so that

$$C', C ; z : (\forall C''.\theta')^\beta, \Gamma \vdash N : \theta$$

by the Lemma 4.2, and also $C'', C', C ; \Gamma \vdash N : \theta$ by the same lemma. We have $\Sigma = x : (\forall C'.\theta)^\alpha, \Theta$. Then, by induction hypothesis

$$C'', C ; \Gamma^{\gamma_0 \wedge \delta_0} \vdash \{x \mapsto N\} M_0 : \theta'$$

where

$$\gamma_0(y) = \begin{cases} \alpha' \wedge \Gamma_{\text{deg}}(y) & \text{if } M_0 \text{ is not pure and } y \in \text{fv}(N) \\ \Gamma_{\text{deg}}(y) & \text{otherwise} \end{cases}$$

and

$$\delta_0(y) = \begin{cases} \Theta_{\text{deg}}(y) & \text{if } y \in \text{fv}(M_0) \\ 1 & \text{otherwise} \end{cases}$$

Similarly,

$$C ; z : (\forall C''.\theta')^\beta, \Gamma^{\gamma_1 \wedge \delta_1} \vdash \{x \mapsto N\} M_1 : \tau$$

with

$$\gamma_1(y) = \begin{cases} \alpha' \wedge \Gamma_{\text{deg}}(y) & \text{if } M_1 \text{ is not pure and } y \in \text{fv}(N) \\ \Gamma_{\text{deg}}(y) & \text{otherwise} \end{cases}$$

and

$$\delta_1(y) = \begin{cases} \Theta_{\text{deg}}(y) & \text{if } y \in \text{fv}(M_1) \\ 1 & \text{otherwise} \end{cases}$$

Notice that $C \vdash \delta \leq \delta_0 \wedge \delta_1$, since $\xi \leq \Theta_{\text{deg}}$. Let $\pi = \gamma_0 \wedge \delta_0 \wedge \gamma_1 \wedge \delta_1$. By the Lemma 2.1(i), if M'_0 is not pure, then M_0 is not pure. Then by the degree weakening rule, and the rule for the let construct, we then have $C ; \Gamma^{\xi'} \vdash \{x \mapsto N\} M : \tau$ where, denoting $\{x \mapsto N\} M_i$ by M'_i :

$$\xi'(y) = \begin{cases} \beta \wedge \pi(y) & \text{if } M'_1 \text{ is not pure and } y \in \text{fv}(M'_0) \\ \pi(y) & \text{otherwise} \end{cases}$$

Now we show that $C \vdash \gamma \wedge \delta \leq \xi'$. We notice that $C \vdash \alpha \leq \alpha'$ since

$$\alpha = \begin{cases} \beta \wedge \alpha' & \text{if } M_1 \text{ is not pure and } x \in \text{fv}(M_0) \\ \alpha' & \text{otherwise} \end{cases}$$

and therefore we have $C \vdash \gamma \leq \gamma_0 \wedge \gamma_1$, and therefore $C \vdash \gamma \wedge \delta \leq \pi$. Then assume that M'_1 is not a pure expression, and let $y \in \text{fv}(M'_0)$. By the Lemma 2.1(i), M_1 is not pure. If $y \in \text{fv}(M_0) - \{x\}$ then

$$\delta(y) = \Delta_{\text{deg}}(y) = \xi(y) = \beta \wedge \Sigma_{\text{deg}}(y)$$

and therefore $C \vdash \delta(y) \leq \beta$. Otherwise, that is if $x \in \text{fv}(M_0)$ and $y \in \text{fv}(N)$, we have $\alpha = \beta \wedge \alpha'$, and therefore (since M is not pure, for M_1 is not a pure expression) $C \vdash \gamma(y) \leq \beta$. This concludes the proof in this case.

- The case of $(\text{let rec } y = M_0 \text{ in } M_1)$ is similar, and all the other ones are easier. \square

To establish the “subject reduction” property, we also need a “replacement lemma” (see [52]) and a form of weakening involving generic instantiation of type schemes.

LEMMA 4.7. *Let M be an expression which is not pure. If $C ; \Gamma \vdash \mathbf{E}[M] : \tau$ is provable, with a sub-proof of $C' ; \Delta \vdash M : \theta$ at the occurrence \mathbf{E} of M , let N be such that $C' ; \Delta \vdash N : \theta$. Then $C ; \Gamma \vdash \mathbf{E}[N] : \tau$.*

PROOF: by induction on the inference of $C ; \Gamma \vdash \mathbf{E}[M] : \tau$. In the case where $\mathbf{E} = (V \mathbf{E}')$ and $\mathbf{E}'[N]$ is a variable which does not occur in V , we have to use the degree weakening rule, since $\mathbf{E}'[M]$ cannot be a variable. \square

LEMMA 4.8. *If $C ; x : \zeta^\alpha, \Gamma \vdash M : \tau$ and $C \vdash \sigma \succeq \zeta$ then $C ; x : \sigma^\alpha, \Gamma \vdash M : \tau$.*

PROOF: this is a standard result, see [16]. \square

PROPOSITION (TYPE PRESERVATION) 4.9.

(i) $C ; \Gamma \vdash M : \tau$ & $M \xrightarrow{*} N \Rightarrow C ; \Gamma \vdash N : \tau$,

(ii) if $C ; \Gamma \vdash [S \mid M] : \tau$ with $u \in \text{dom}(\Gamma) \Rightarrow u \in \text{dom}(S)$ and $[S \mid M] \xrightarrow{*} [S' \mid N]$ then $C ; \Gamma \vdash [S \mid N] : \tau$.

PROOF:

(i) It is enough to prove this for $M \rightarrow N$. We proceed by induction on the proof of $M \rightarrow N$, and then by induction on the inference of $C ; \Gamma \vdash M : \tau$, and by case on the last rule used to infer this sequent. The case where this rule is the degree weakening is trivial, and is omitted.

- If $M = (\lambda x M')V$ and $N = \{x \mapsto V\}M'$, then the proof of $C ; \Gamma \vdash M : \tau$ may be reduced, by converting successive uses of the degree weakening rule into one, to:

$$\frac{\frac{\frac{\vdots}{C ; x : \theta^a, \Delta \vdash M' : \tau}}{C ; \Delta^1 \vdash \lambda x M' : \theta^a \rightarrow \tau}}{C ; \Gamma \vdash \lambda x M' : \theta^a \rightarrow \tau} \quad \frac{\vdots}{C ; \Gamma \vdash V : \theta}}{C ; \Gamma^0_{\lambda x M' \wedge \xi} \vdash M : \tau}$$

with $C \vdash \Gamma \leq \Delta^1$ on $\text{fv}(\lambda x M')$ and

$$\xi(y) = \begin{cases} a & \text{if } V = y \notin \text{fv}(M) \\ 1 & \text{if } y \notin \text{fv}(V) \\ a \wedge \Gamma_{\text{deg}}(y) & \text{otherwise} \end{cases}$$

If V is a variable z which does not occur free in $\lambda x M'$, then $\Gamma = z : \sigma^\alpha, \Gamma'$ with $C \vdash \sigma \succeq \theta$ and $C \vdash \alpha \leq 0$. Then we also have $\Delta = z : \sigma^\beta, \Delta'$, hence $C ; x : \theta^\alpha, \Delta' \vdash M' : \tau$ by the Lemma 4.4. Therefore $C ; z : \theta^\alpha, \Delta' \vdash N : \tau$ by the Lemma 4.5, hence $C ; z : \sigma^\alpha, \Delta' \vdash N : \tau$ by the Lemma 4.8. Since in this case $\xi = a_z$, it is easy to see that

$$C \vdash \Gamma^{0_{\lambda x M'} \wedge \xi} \leq z : \sigma^\alpha, \Delta'$$

on $\text{fv}(N)$, and we conclude using the degree weakening rule in this case. If V is not a variable, or is a variable which occurs free in $\lambda x M'$, then

$$\xi(y) = \begin{cases} a \wedge \Gamma_{\text{deg}}(y) & \text{if } y \in \text{fv}(V) \\ 1 & \text{otherwise} \end{cases}$$

By the Proposition 4.6 we have $C ; \Gamma^{\gamma \wedge \delta} \vdash N : \tau$ with

$$\gamma(y) = \begin{cases} a \wedge \Gamma_{\text{deg}}(y) & \text{if } M' \text{ is not pure and } y \in \text{fv}(V) \\ \Gamma_{\text{deg}}(y) & \text{otherwise} \end{cases}$$

and $\delta = \Delta_{\text{deg}}$. Then $C \vdash \gamma \geq \xi$ on $\text{fv}(V)$, hence $C \vdash \gamma \wedge \delta \geq 0_{\lambda x M'} \wedge \xi$ on $\text{fv}(N)$, and we conclude $C ; \Gamma^{0_{\lambda x M'} \wedge \xi} \vdash N : \tau$ by the degree weakening rule.

• The case where $M = (\text{let } x = V \text{ in } M')$ and $N = \{x \mapsto V\}M'$ is an immediate consequence of the Proposition 4.6. If $M = (\text{let rec } x = V \text{ in } M')$ and $N = \{x \mapsto (\text{let rec } x = V \text{ in } V)\}M'$, we have

$$\frac{\frac{\vdots}{C', C ; x : \theta^1, \Gamma \vdash V : \theta} \quad \frac{\vdots}{C ; x : (\forall C'. \theta)^\alpha, \Gamma \vdash M' : \tau}}{C ; \Gamma^\xi \vdash (\text{let rec } x = V \text{ in } M') : \tau}$$

where

$$\xi(y) = \begin{cases} \alpha \wedge \Gamma_{\text{deg}}(y) & \text{if } M' \text{ is not pure and } y \in \text{fv}(V) \\ \Gamma_{\text{deg}}(y) & \text{otherwise} \end{cases}$$

and therefore

$$\frac{\frac{\vdots}{C', C ; x : \theta^1, \Gamma \vdash V : \theta} \quad \frac{\vdots}{C', C ; x : \theta^1, \Gamma \vdash V : \theta}}{C', C ; \Gamma \vdash (\text{let rec } x = V \text{ in } V) : \theta}$$

by the rule for the `let rec` construct. Then $C ; \Gamma^{\gamma \wedge \delta} \vdash N : \tau$ by the Proposition 4.6, where $\gamma = \xi$ and $\delta = \Gamma_{\text{deg}}$. Since $C \vdash \gamma \leq \delta$, we have $C \vdash \gamma \wedge \delta = \xi$, and we conclude in this case using the degree weakening rule.

All the other cases are straightforward. For the reductions $\mathbf{E}[M] \rightarrow \mathbf{E}[M']$ with $M \rightarrow M'$, we proceed by induction on \mathbf{E} , using the induction hypothesis. In the case where $\mathbf{E} = (V\mathbf{E}')$ and $\mathbf{E}'[M']$ is a variable which does not occur in V , we have to use the degree weakening rule, since $\mathbf{E}'[M]$ cannot be a variable. In the case where $\mathbf{E} = (\text{let } x = \mathbf{E}' \text{ in } N)$ or $\mathbf{E} = (\text{let rec } x = \mathbf{E}' \text{ in } N)$, and $\mathbf{E}'[M]$ is pure, we use the Lemma 2.1(ii).

(ii) Again it is enough to prove this for $[S \mid M] \rightarrow [S' \mid N]$. We proceed by induction on the proof of this reduction. We have $C ; \Gamma \vdash S$ and $C ; \Gamma \vdash M : \tau$. If $M \rightarrow N$, then we use the previous point. If $M = \mathbf{E}[(\text{ref } V)]$ with $S' = (u := V ; S)$, $N = \mathbf{E}[u]$ and u does not occur in S or $\mathbf{E}[V]$,

then we may assume, by the Lemma 4.4, that $u \notin \text{dom}(\Gamma)$. In the proof of $C ; \Gamma \vdash M : \tau$ there is a sub-proof regarding $(\text{ref } V)$, of the form

$$\frac{\frac{\frac{\vdots}{C \vdash \Psi}}{C ; \Psi^1 \vdash \text{ref} : \theta^0 \rightarrow \theta \text{ref}} \quad \frac{\vdots}{C ; \Delta \vdash V : \theta}}{C ; \Delta \vdash \text{ref} : \theta^0 \rightarrow \theta \text{ref}}}{C ; \Delta^{0v} \vdash (\text{ref } V) : \theta \text{ref}}}{C ; \Theta \vdash (\text{ref } V) : \theta \text{ref}}$$

with $C \vdash \Delta \leq \Psi^1$ and $C \vdash \Theta \leq \Delta^{0v}$. Since $u \notin \text{dom}(\Gamma)$, by weakening there is a proof of $C ; u : \theta, \Gamma \vdash M : \tau$ having the same structure as the proof of $C ; \Gamma \vdash M : \tau$. In particular, we have a sub-proof of $C ; u : \theta, \Theta \vdash (\text{ref } V) : \theta \text{ref}$ with $C ; u : \theta, \Delta \vdash V : \theta$. Since obviously $C \vdash \Theta \leq \Delta$ we also have $C ; u : \theta, \Theta \vdash V : \theta$. Then $C ; u : \theta, \Gamma \vdash \mathbf{E}[u] : \tau$ by the Lemma 4.7. The other cases of global reduction are similar. \square

Combining this result with the Corollary 2.6 and the Lemma 4.1, we get:

THEOREM (TYPE SAFETY) 4.10. *For any closed configuration $[S \mid M]$, if it is typable, i.e. $C ; \Gamma \vdash [S \mid M] : \tau$ for some C, Γ and τ , and if its evaluation terminates on $[S' \mid N]$, then N is a value (and not a faulty expression) of type τ .*

Notice that an expression of the form $(\text{set } V)$ where V is neither a variable nor a location is both a value and a faulty expression, which is then rejected by the type system.

5. Type Reconstruction

In this section we prove that the standard result about typability in ML, namely that one can compute a principal type for any typable program, extends to our language. Regarding the record calculus, this was established by Jategaonkar and Mitchell [26], though with a different view of row variables. Then the main novelty here consists in showing that a typable program has a (computable) type which is the most general one also for what regards the degrees. Nevertheless, since our approach is slightly different from that of [26], we present the type reconstruction algorithm in full details.

The data for the type reconstruction algorithm are a constraint C , a type assignment Ψ and a term M , and the algorithm, if it does not fail, yields a type τ , a degree assignment γ , a constraint C' and a substitution $S \in \text{Sub}(C, C')$ such that $C' ; S(\Psi)^\gamma \vdash M : \tau$ is a valid typing. As usual, type reconstruction involves solving *verification conditions*. Here we have not only to solve type equations, by means of unification, but also degree inequalities. Then our verification conditions are of the form $(Q. A ; E)$ where

- Q is a sequence of existential quantifications $\exists t$ over type variables;
- A is a set of annotation assertions $\sigma :: L$ and of degree inequalities of the form $a \leq \alpha$;
- E is a set of type equations $\tau = \theta$ (to be solved), and of degree equations, of the form $a = b$ or $\alpha = 1$.

Notice that the degree equations that we have to deal with are quite trivial to solve (recall that a and b are either constants, 0 or 1, or variables, but do not involve the \wedge operation). We shall only deal with verification conditions $(Q. A ; E)$ satisfying the requirement that the type variables occurring in the equations of E also occur in the annotation assertions of A . We use existential

quantification to deal with the “fresh variables” that the algorithm may introduce (see [27]). Let us denote by $\text{fv}(E)$, and similarly $\text{var}(Q)$, the set of (type and degree) variables occurring in E (resp. in Q).

DEFINITION 5.1. *A solution of the verification condition $(Q, A; E)$ is a pair (C, S) such that there exists S' with $S = S' \upharpoonright (\text{fv}(E) - \text{var}(Q))$ and*

- (i) $C \vdash S'(A)$;
- (ii) $\tau = \tau' \in E \Rightarrow C \vdash S'(\tau) :: \emptyset, C \vdash S'(\tau') :: \emptyset$ and $S'(\tau) =_{\mathcal{T}} S'(\tau')$;
- (iii) $\alpha = \beta \in E \Rightarrow \vdash S'(\alpha) = S'(\beta)$.

A solution (C, S) is more general than (C', S') if there exists a substitution $S'' \in \text{Sub}(C, C')$ such that $S' = S''S$.

As one can see, if (C, S) is a solution of $(Q, A; E)$, then S is the identity for the variables that do not occur free in this verification condition. As usual (see [32, 27]), solving $(Q, A; E)$ consists in transforming it into a “solved form”:

DEFINITION 5.2. *A verification condition $(Q, A; E)$ is a solved form if A is a constraint in the sense of the type system, that is $A = \{t_h :: L_h \mid h \in H\} \cup \{p_k \leq \alpha_k \mid k \in K\}$, and $E = \{t'_i = \tau_i \mid i \in I\} \cup \{q_j = a_j \mid j \in J\}$ with:*

- (i) *the type variables t'_i only occur in E as the left members of the equations $t'_i = \tau_i$;*
- (ii) *the degree variables q_j only occur in A and E as the left members of the equations $q_j = a_j$;*
- (iii) $\{t'_i \mid i \in I\} \subseteq \{t_h \mid h \in H\}$ and $A \vdash \tau_i : A(t'_i)$ for all i .

We shall use the notation $(Q, C; S)$ for solved forms. Such a solved form has an obvious solution, namely (C, S) where $S = \{t'_i \mapsto \tau_i \mid i \in I\} \cup \{q_j \mapsto a_j \mid j \in J\}$. We can check that $S \in \text{Sub}(C_0, C)$ where $C_0 = \{t'_i :: \emptyset \mid i \in I\}$, using the following

LEMMA 5.3. $C \vdash \sigma :: L \ \& \ L' \subseteq L \Rightarrow C \vdash \sigma :: L'$

(the proof, by induction on the inference of $C \vdash \sigma :: L$, is trivial). It is easy to see that the “canonical solution” S – as just described – of a solved form $(Q, C; S)$ is also a most general solution, because any other solution S' is such that $S' = S''S$ where $S''(t) = S'(t)$ except for $t \in \{t'_i \mid i \in I\}$ and similarly for $S'(p)$. To show that S'' is indeed an acceptable substitution, we need to use the following:

LEMMA 5.4. *If $C \vdash S(\tau) :: L$ then there exists C' such that $\text{dom}(C') = \text{fv}(\tau)$ with $C' \vdash \tau :: L$ and $C \vdash S(t) :: C'(t)$ for any $t \in \text{fv}(\tau)$.*

(again, the proof is straightforward). Now we define the transformations of verification conditions. First, we have a set of transformations to decompose – if this does not fail – a set of annotation assertions A into a constraint C . These transformations $A \triangleright A'$ are described in Figure 9, where we do not include the rule that decomposing annotation assertions holds up to α -conversion of type schemes. In this figure, $C \setminus t$ denotes the restriction of C to variables different from t , that is $C \setminus t = C \upharpoonright (\text{dom}(C) - \{t\})$. In the last rule, \triangleright^* is the reflexive and transitive closure of \triangleright , given by

$$\frac{}{A \triangleright^* A} \qquad \frac{A \triangleright A'}{A \triangleright^* A'} \qquad \frac{A \triangleright^* A'' \quad A'' \triangleright^* A'}{A \triangleright^* A'}$$

The following lemma asserts the correctness of this decomposition process.

LEMMA 5.5. $C \vdash A$ if and only if there exists C' such that $A \triangleright^* C'$ and $C \vdash C'$.

$$t :: L_0, t :: L_1, A \triangleright t :: L_0 \cup L_1, A \quad (1)$$

$$\text{unit} :: \emptyset, A \triangleright A \quad (2)$$

$$(\theta^a \rightarrow \tau) :: \emptyset, A \triangleright \theta :: \emptyset, \tau :: \emptyset, A \quad (3)$$

$$\tau \text{ ref} :: \emptyset, A \triangleright \tau :: \emptyset, A \quad (4)$$

$$\diamond :: L, A \triangleright A \quad (5)$$

$$\langle \rho, \ell : \tau \rangle :: L, A \triangleright \rho :: (L \cup \{\ell\}), \tau :: \emptyset, A \quad (\ell \notin L) \quad (6)$$

$$\frac{\sigma :: \emptyset \triangleright^* C}{(\forall t :: L.\sigma) :: \emptyset, A \triangleright C \setminus t, A} \quad C(t) \subseteq L, t \notin A \quad (7)$$

Figure 9: Decomposition of Annotation Assertions

PROOF: for the \Leftarrow direction, we prove that $A \triangleright^* A'$ and $C \vdash A'$ implies $C \vdash A$, by induction on the definition of $A \triangleright^* A'$. Conversely, we show that $C \vdash \sigma :: L$ implies $\sigma :: L \triangleright^* C'$ for some $C' \subseteq C$ (which means that $\text{dom}(C') \subseteq \text{dom}(C)$ and $C'(t) \subseteq C(t)$ for any t). We conclude using the fact that if $C \vdash A$ and $C' \vdash A'$ then $C \cup C' \vdash A, A'$ where $C \cup C'$ is the constraint given by $\text{dom}(C \cup C') = \text{dom}(C) \cup \text{dom}(C')$, and $(C \cup C')(t) = C(t) \cup C'(t)$, where by convention $C(t) = \emptyset$ if $t \notin \text{dom}(C)$. \square

LEMMA 5.6. *For any substitution S , if $A \triangleright A'$ and $C \vdash S(A)$ then $C \vdash S(A')$.*

The proof, by induction on $A \triangleright A'$, is easy (one uses the fact that if $C \vdash \sigma :: L$ and $C \vdash \sigma :: L'$ then $C \vdash \sigma :: L \cup L'$).

Now we show how to decompose verification conditions – we still use the notation \triangleright . As usual (see [27]), the intention is that this decomposition either fails, meaning that the verification condition has no solution, or terminate on a solved form, while preserving the set of solutions. The decomposition relation, including decomposition of annotation assertions and of type equations, is described in Figure 10, where we use some loose notations (such as $\tau = \tau', E$ for $\{\tau = \tau'\} \cup E$) and a notion of “fresh” variable, meaning “not occurring in the current context”. In the clauses (14) and (15) we use the symbols \mathbf{e} and \mathbf{x} to denote respectively either a type or degree expression, and either a type or degree variable. In the last rule S denotes a set of equations satisfying the clause (i) of Definition 5.2.

PROPOSITION 5.7.

(i) *The decomposition \triangleright terminates.*

(ii) *If $(Q. A ; E) \triangleright (Q'. A' ; E')$ then the variables occurring free in $(Q'. A' ; E')$ also occur in $(Q. A ; E)$, and $(Q. A ; E)$ and $(Q'. A' ; E')$ have the same solutions.*

(iii) *If $(Q. A ; E)$ is irreducible with respect to \triangleright , then it has a solution if and only if it is a solved form.*

PROOF: for any type scheme σ , let $\#(\sigma)$ be the number of logical symbols occurring in σ . Then for instance $\#(\forall t :: L.\sigma) = \#(\sigma) + 1$, and so on. For any set $A = \{\sigma_h :: L_h \mid h \in H\} \cup \{a_k \leq \alpha_k \mid k \in K\}$ of annotation assertions and degree inequations, we define its size $|A|$ to be (m, n) where $m = \sum\{\#(\sigma_h) \mid h \in H\}$ and n is the number of elements of H . Let \leq be the lexicographic ordering on tuples of integers (of a given length). Then we have:

$$A \triangleright A' \Rightarrow |A| > |A'|$$

$$\frac{A \triangleright A'}{(\mathcal{Q}.A; E) \triangleright (\mathcal{Q}.A'; E)} \quad (8)$$

$$(\mathcal{Q}. p \leq \alpha, p \leq \beta, A; E) \triangleright (\mathcal{Q}. p \leq \alpha \wedge \beta, A; E) \quad (9)$$

$$(\mathcal{Q}. 0 \leq \alpha, A; E) \triangleright (\mathcal{Q}. A; E) \quad (10)$$

$$(\mathcal{Q}. 1 \leq \alpha, A; E) \triangleright (\mathcal{Q}. A; \alpha = 1, E) \quad (11)$$

$$(\mathcal{Q}. A; (\alpha \wedge \beta) = 1, E) \triangleright (\mathcal{Q}. A; \alpha = 1, \beta = 1, E) \quad (12)$$

$$(\mathcal{Q}. A; q = a, E) \triangleright (\mathcal{Q}. \{q \mapsto a\}A; q = a, \{q \mapsto a\}E) \quad (13)$$

where $q \in (\text{fv}(A) \cup \text{fv}(E)) - \{d\}$

$$(\mathcal{Q}. A; \mathbf{e} = \mathbf{e}, E) \triangleright (\mathcal{Q}. A; E) \quad (14)$$

$$(\mathcal{Q}. A; \mathbf{e} = \mathbf{x}, E) \triangleright (\mathcal{Q}. A; \mathbf{x} = \mathbf{e}, E) \quad (\mathbf{e} \notin \mathcal{V}ar, \mathbf{x} \in \mathcal{V}ar) \quad (15)$$

$$(\mathcal{Q}. A; \theta_0^a \rightarrow \tau_0 = \theta_1^b \rightarrow \tau_1, E) \triangleright (\mathcal{Q}. A; E', E) \quad (16)$$

where $E' = \{\theta_0 = \theta_1, \tau_0 = \tau_1, a = b\}$

$$(\mathcal{Q}. A; \tau_0 \text{ ref} = \tau_1 \text{ ref}, E) \triangleright (\mathcal{Q}. A; \tau_0 = \tau_1, E) \quad (17)$$

$$(\mathcal{Q}. A; \langle \rho_0, \ell : \tau_0 \rangle = \langle \rho_1, \ell : \tau_1 \rangle, E) \triangleright (\mathcal{Q}. A; \rho_0 = \rho_1, \tau_0 = \tau_1, E) \quad (18)$$

$$(\mathcal{Q}. A; \langle \rho_0, \ell_0 : \tau_0 \rangle = \langle \rho_1, \ell_1 : \tau_1 \rangle, E) \triangleright (\mathcal{Q}. \exists r. A'; E', E) \quad (19)$$

where $A' = r :: \{\ell_0, \ell_1\}, A$

$E' = \{\rho_0 = \langle r, \ell_1 : \tau_1 \rangle, \rho_1 = \langle r, \ell_0 : \tau_0 \rangle\}$

if $\ell_0 \neq \ell_1$, where r is fresh.

$$(\mathcal{Q}. A; t = \tau, E) \triangleright (\mathcal{Q}. A; t = \tau, \{t \mapsto \tau\}E) \quad t \in \text{fv}(E) - \text{fv}(\tau) \quad (20)$$

$$\frac{\{\tau :: C(t) \mid t = \tau \in S\}, C \triangleright^* C'}{(\mathcal{Q}. C; S) \triangleright (\mathcal{Q}. C'; S)} \quad \{t = \tau \mid t = \tau \in S \ \& \ C \not\vdash \tau :: C(t)\} \neq \emptyset \quad (21)$$

Figure 10: Decomposition of Verification Conditions

This is easy to see, by induction on the definition of $A \triangleright A'$: the rules (1,2,5) strictly decrease the number n of annotation assertions, and do not increase the sum m of the size of the type schemes, while the rules (3,4,6) – and also (2,5) – strictly decrease the sum m of the size of the type schemes. Finally for the rule (7) we have $|\sigma :: \emptyset| \geq |C|$ by induction, and therefore $|(\forall t :: L.\sigma) :: \emptyset, A| > |C \setminus t, A|$.

Now, for any type equation $\tau = \tau'$, let $\#(\tau = \tau') = \#(\tau) + \#(\tau')$, and similarly $\#(\alpha = \beta) = \#(\alpha) + \#(\beta) = \#(\alpha \leq \beta)$ where the size of a degree expression is the number of symbols occurring in it. Then we define the size $|(\mathcal{Q}. A; E)|_k$ of a verification condition with respect to some integer k as follows:

$$|(\mathcal{Q}.A; E)|_k = (n, n', n_k, \dots, n_0, m, l, |A|)$$

where:

- n is the number of degree inequalities in A , that is the number of elements of K ,
- n' is the number of type variables which do not occur only once as the left hand side of some equation of E , but may occur in A (in particular this number is strictly positive if E contains an

equation $\tau = t$, or $t = \tau$, E' and t occurs elsewhere), plus the number of degree variables which do not occur only once as the left hand side of some equation of E ,

- n_i is the number of inequations of size i in A , plus the number of equations of size i in E ,
- m is the number of equations in E of the form $\mathbf{e} = \mathbf{x}$ with $\mathbf{e} \notin \mathcal{V}ar$ and $\mathbf{x} \in \mathcal{V}ar$, and
- l is 0 if (A, E) is not a (C, S) , and the number of equations $t = \tau$ of S such that $C \not\vdash \tau :: C(t)$ otherwise.

It is easy to see that

$$(\mathcal{Q}. A ; E) \triangleright (\mathcal{Q}'. A' ; E') \Rightarrow \max\{\#(\mathbf{e} = \mathbf{e}') \mid \mathbf{e} = \mathbf{e}' \in E\} \geq \max\{\#(\mathbf{e} = \mathbf{e}') \mid \mathbf{e} = \mathbf{e}' \in E'\}$$

Now let k be such that $k \geq \max\{\#(\mathbf{e} = \mathbf{e}') \mid \mathbf{e} = \mathbf{e}' \in E\}$. We prove that

$$(\mathcal{Q}. A ; E) \triangleright (\mathcal{Q}'. A' ; E') \Rightarrow |(\mathcal{Q}. A ; E)|_k > |(\mathcal{Q}'. A' ; E')|_k$$

by induction on the definition of \triangleright . The rule (8) strictly decreases the size of the set of annotation assertions, as we have seen, while not affecting the other components. The rules (9), (10) and (11) strictly decrease n , that is the number of degree inequalities in A . The rules (12,14,16,17,18,19) strictly decrease some n_i (notice that (19) does not decrease the sum of the size of the type equations). The rule (15) strictly decreases the number m of equations of the form $\tau = t$ (and possibly also n), and the rules (13) and (20) strictly decreases n' . Finally let us see that (21) strictly decreases l : since for any equation $t = \tau$ of S the variable t does not occur in the right hand side of an equation of S , decomposing $\{\tau :: C(t) \mid t = \tau \in S\}$, C into C' cannot add any new assertion about t , that is $C'(t) = C(t)$. By the Lemma 5.5, we then have $C' \vdash \tau :: C'(t)$ for any $t = \tau \in S$. This concludes the proof of the point (i) of the Proposition.

The point (ii) is almost trivial. Let us just see the case of the rule (19). Let $S \in \mathcal{S}ub(C_0, C_1)$ be a solution of $(\mathcal{Q}. A ; \langle \rho_0, \ell_0 : \tau_0 \rangle = \langle \rho_1, \ell_1 : \tau_1 \rangle, E)$, and let C'_0 and S' be as in the Definition 5.1. Then $r \notin \text{dom}(C'_0)$ since r is fresh. Let $C''_0 = r :: \{\ell_0, \ell_1\}, C'_0$. Since $S' \langle \rho_0, \ell_0 : \tau_0 \rangle =_{\mathcal{T}} S' \langle \rho_1, \ell_1 : \tau_1 \rangle$ we have $S'(\rho_0) =_{\mathcal{T}} \langle \rho, \ell_1 : S'(\tau_1) \rangle$ and $S'(\rho_1) =_{\mathcal{T}} \langle \rho, \ell_0 : S'(\tau_0) \rangle$. Let $S'' = S' \cup \{r \mapsto \rho\}$. Since $C_1 \vdash S' \langle \rho_0, \ell_0 : \tau_0 \rangle :: \emptyset$ and $C_1 \vdash S' \langle \rho_1, \ell_1 : \tau_1 \rangle :: \emptyset$, we have $C_1 \vdash \rho :: \{\ell_0, \ell_1\}$, and therefore $S'' \in \mathcal{S}ub(C''_0, C_1)$, and $C_1 \vdash S''(A')$ where $A' = r :: \{\ell_0, \ell_1\}, A$. It is easy to see that the other conditions for $S \in \mathcal{S}ub(C_0, C_1)$ to be a solution of $(\mathcal{Q}. \exists r. A' ; E', E)$ are met, where $E' = \{\rho_0 = \langle r, \ell_1 : \tau_1 \rangle, \rho_1 = \langle r, \ell_0 : \tau_0 \rangle\}$. Conversely, if $S \in \mathcal{S}ub(C_0, C_1)$ is a solution of $(\mathcal{Q}. \exists r. A' ; E', E)$, we have to check that $C_1 \vdash S' \langle \rho_0, \ell_0 : \tau_0 \rangle :: \emptyset$ (again using S' as given in Definition 5.1), and similarly for $\langle \rho_1, \ell_1 : \tau_1 \rangle$. We have

$$\begin{aligned} S' \langle \rho_0, \ell_0 : \tau_0 \rangle &=_{\mathcal{T}} \langle S'(\rho_0), \ell_0 : S'(\tau_0) \rangle \\ &=_{\mathcal{T}} \langle \langle S'(r), \ell_1 : S'(\tau_1) \rangle, \ell_0 : S'(\tau_0) \rangle \end{aligned}$$

Since $C_1 \vdash S'(\tau_i) :: \emptyset$ and $C_1 \vdash S'(r) :: \{\ell_0, \ell_1\}$, we easily conclude.

Regarding the last point, we have seen that solved forms are solvable. Now if $(\mathcal{Q}. A ; E)$ is irreducible with respect to \triangleright , but is not a solved form, then a case analysis shows that $(\mathcal{Q}. A ; E)$ has no solution. This holds in particular if E contains $0 = 1$ or $1 = 0$. \square

An immediate corollary is:

COROLLARY 5.8. *If a verification condition is solvable, then it has a most general solution.*

We denote by $\mathcal{S}ol(\mathcal{Q}. A ; E)$ the most general solution of the verification condition $(\mathcal{Q}. A ; E)$, if it exists (in which case it is unique, up to a permutation of the variables).

Now, coming back to the issue of type inference, we describe the algorithm for type reconstruction as a function $\text{Type}(C, \Psi, M) = (\tau, \gamma, (C', S))$, with the idea that $S \in \mathcal{S}ub(C, C')$ and $C' ; S(\Psi)^\gamma \vdash M : \tau$. However, this will be true only if $C \vdash \Psi$. We could report a failure otherwise, but we shall actually use the algorithm only in this case. We assume here that M does not contain

any location, although the algorithm could easily be extended to cover this case too, and that Ψ assigns some type to each free variable of M (we should otherwise report a failure). The function Type is defined up to α -conversion performed in Ψ and M , and the type and degree variables introduced by the algorithm are implicitly assumed to be fresh. In the definition of the algorithm, we use the notation $S \setminus t$ for the substitution that coincides with S , except for $(S \setminus t)(t) = t$. We also abusively write $S \setminus C$ for the substitution that coincides with S , except on $\text{dom}(C)$, where it is the identity. We denote by $C \upharpoonright \Psi$ the constraint defined as follows: $(C \upharpoonright \Psi)_{\text{deg}} = C_{\text{deg}}$ and $(C \upharpoonright \Psi)_{\text{typ}} = C_{\text{typ}} \upharpoonright \text{fv}(\Psi)$. When we write $(C', S) = \text{Sol}(\mathcal{Q}. A; E)$ in the definition of $\text{Type}(\Psi, M)$, we mean that the algorithm reports a failure (treated as an exception) in the case where $\text{Sol}(\mathcal{Q}. A; E)$ does not exist, and similarly when $A \triangleright^* C'$ is used.

- If $\Psi(x) = (\forall C_0. \tau)$ with $\text{dom}(C_0) \cap \text{dom}(C) = \emptyset$,
then $\text{Type}(C, \Psi, x) = (\tau, 0_x, (C \cup C_0, \text{id}))$.
- If $\text{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Psi, M) = (\tau, \gamma, (C', S))$
then $\text{Type}(C, \Psi, \lambda x M) = (S(t)^p \rightarrow \tau, 1, (\{p \leq \gamma(x)\} \cup C', S \setminus t))$.
- If $\text{Type}(C, \Psi, M) = (\tau_0, \gamma_0, (C_0, S_0))$
and $\text{Type}(C'_0, S_0(\Psi), N) = (\tau_1, \gamma_1, (C_1, S_1))$ where $C'_0 = C_0 \upharpoonright S_0(\Psi)$
and $(C', S) = \text{Sol}(\{t :: \emptyset\} \cup C_1; \{S_1(\tau_0) = \tau_1^p \rightarrow t\})$
then $\text{Type}(C, \Psi, (MN)) = (S(t), 0_M \wedge S(\delta), (C', SS_1 S_0))$ where

$$\delta(x) = \begin{cases} p & \text{if } N = x \notin \text{fv}(M) \\ 1 & \text{if } x \notin \text{fv}(N) \\ p \wedge (S_1(\gamma_0) \wedge \gamma_1)(x) & \text{otherwise} \end{cases}$$

- If $\text{Type}(C, \Psi, N) = (\tau_0, \gamma_0, (C_0, S_0))$
and C_2 is $C_0 \setminus \text{fv}(S_0(\Psi))$, if N is pure, and $C_2 = \emptyset$ otherwise
and $\text{Type}(C_0 - C_2, \{x : (\forall C_2. \tau_0)\} \cup S_0(\Psi), M) = (\tau_1, \gamma_1, (C_1, S_1))$
then $\text{Type}(C, \Psi, (\text{let } x = N \text{ in } M)) = (\tau_1, \delta, (C_1, S_1 S_0))$ where

$$\delta(y) = \begin{cases} \gamma_1(x) \wedge (S_1(\gamma_0) \wedge \gamma_1)(y) & \text{if } y \in \text{fv}(N) \text{ and } M \text{ is not pure} \\ (S_1(\gamma_0) \wedge \gamma_1)(y) & \text{otherwise} \end{cases}$$

- If $\text{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Psi, N) = (\tau_0, \gamma_0, (C_0, S_0))$
and $(C_1, S_1) = \text{Sol}(C_0; \{S_0(t) = \tau_0, \gamma_0(x) = 1\})$
and C_2 is $C_1 \setminus \text{fv}(S_1 S_0(\Psi))$, if N is pure, and $C_2 = \emptyset$ otherwise
and $\text{Type}(C_1 - C_2, \{x : (\forall C_2, S_1 S_0(t))\} \cup S_1 S_0(\Psi), M) = (\tau, \gamma_1, (C', S))$
then $\text{Type}(C, \Psi, (\text{let rec } x = N \text{ in } M)) = (\tau, \delta, (C', SS_1 S_0))$ where

$$\delta(y) = \begin{cases} \gamma_1(x) \wedge (SS_1(\gamma_0) \wedge \gamma_1)(y) & \text{if } y \in \text{fv}(N) \text{ and } M \text{ is not pure} \\ (SS_1(\gamma_0) \wedge \gamma_1)(y) & \text{otherwise} \end{cases}$$

- $\text{Type}(C, \Psi, \text{ref}) = (t^0 \rightarrow t \text{ ref}, 1, (\{t :: \emptyset\} \cup C, \text{id}))$
and $\text{Type}(C, \Psi, !)$ $= ((t \text{ ref})^0 \rightarrow t, 1, (\{t :: \emptyset\} \cup C, \text{id}))$
and $\text{Type}(C, \Psi, \text{set}) = ((t \text{ ref})^0 \rightarrow t^0 \rightarrow \text{unit}, 1, (\{t :: \emptyset\} \cup C, \text{id}))$
and $\text{Type}(C, \Psi, \text{()}) = (\text{unit}, 1, (C, \text{id}))$
and $\text{Type}(C, \Psi, \text{()}) = (\text{()}, 1, (C, \text{id}))$.

- If $\text{Type}(C, \Psi, M) = (\tau_0, \gamma_0, (C_0, S_0))$
and $\text{Type}(C'_0, S_0(\Psi), N) = (\tau_1, \gamma_1, (C_1, S_1))$ where $C'_0 = C_0 \upharpoonright S_0(\Psi)$
and $\{\tau_0 :: \{\ell\}\} \cup C_1 \triangleright^* C'$
then $\text{Type}(C, \Psi, \langle M, \ell = N \rangle) = (\langle \tau_0, \ell : \tau_1 \rangle, S_1(\gamma_0) \wedge \gamma_1, (C', S_1 S_0))$.
- If $\text{Type}(C, \Psi, M) = (\tau, \gamma, (C_0, S_0))$
and $(C', S) = \text{Sol}(\exists r. \{r :: \{\ell\}, t :: \emptyset\} \cup C_0; \{\tau = \langle r, \ell : t \rangle\})$
then $\text{Type}(C, \Psi, (M.\ell)) = (S(t), \gamma, (C', SS_0))$.
- If $\text{Type}(C, \Psi, M) = (\tau, \gamma, (C_0, S_0))$
and $(C', S) = \text{Sol}(\exists t. \{r :: \{\ell\}, t :: \emptyset\} \cup C_0; \{\tau = \langle r, \ell : t \rangle\})$
then $\text{Type}(C, \Psi, (M \setminus \ell)) = (S(r), \gamma, (C', SS_0))$.

To establish the correctness of the algorithm, we first prove a preliminary lemma.

LEMMA 5.9. *If $C \vdash \Psi$ and $\text{Type}(C, \Psi, M) = (\tau, \gamma, (C', S))$ then $S \in \text{Sub}(C \upharpoonright \Psi, C')$ and $C' \vdash \tau :: \emptyset$.*

PROOF: by induction on M . Let us examine some cases:

- If $M = x$ then $\Psi(x) = (\forall C_0.\tau)$ with $\text{dom}(C_0) \cap \text{dom}(C) = \emptyset$, $C' = C \cup C_0$ and $S = \text{id}$. It is easy to see that $C \vdash (\forall C_0.\tau) :: \emptyset$ implies $C_0, C \vdash \tau :: \emptyset$. Since $C \subseteq C'$, the fact that $\text{id} \in \text{Sub}(C \upharpoonright \Psi, C')$ is obvious.
- If $M = \lambda x N$, then $C' = \{p \leq \delta(x)\} \cup C_0$, $S = S_0 \setminus t$ and $\tau = S_0(t)^p \rightarrow \theta$ where $(\theta, \delta, (C_0, S_0)) = \text{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Psi, N)$. By induction hypothesis $S_0 \in \text{Sub}(\{t :: \emptyset\} \cup C \upharpoonright \Psi, C_0)$, hence obviously $S \in \text{Sub}(C \upharpoonright \Psi, C')$ and $C_0 \vdash S_0(t) :: \emptyset$. Since $C_0 \vdash \theta :: \emptyset$ by induction hypothesis, we have $C_0 \vdash \tau :: \emptyset$, hence also $C' \vdash \tau :: \emptyset$.
- If $M = (M'N)$ then $S = S' S_1 S_0$ and $\tau = S'(t)$ with $\text{Type}(C, \Psi, M') = (\tau_0, \gamma_0, (C_0, S_0))$, $\text{Type}(C'_0, S_0(\Psi), N) = (\tau_1, \gamma_1, (C_1, S_1))$ where $C'_0 = C_0 \upharpoonright S_0(\Psi)$ and

$$(C', S') = \text{Sol}(\{t :: \emptyset\} \cup C_1; \{S_1(\tau_0) = \tau_1^p \rightarrow t\})$$

By induction hypothesis $S_0 \in \text{Sub}(C \upharpoonright \Psi, C_0)$, hence also $S_0 \in \text{Sub}(C \upharpoonright \Psi, C'_0)$ and $C'_0 \vdash S_0(\Psi)$. Then by induction hypothesis $S_1 \in \text{Sub}(C'_0, C_1)$, therefore (by the Lemma 3.1) $S_1 S_0 \in \text{Sub}(C \upharpoonright \Psi, C_1)$. We clearly have $S' \in \text{Sub}(\{t :: \emptyset\} \cup C_1, C')$, hence also $S' \setminus t \in \text{Sub}(C_1, C')$, whence $S \in \text{Sub}(C \upharpoonright \Psi, C')$ (for t is fresh, and therefore $t \notin \text{fv}(S_1 S_0(C))$) and $C' \vdash S'(t) :: \emptyset$, that is $C' \vdash \tau :: \emptyset$.

- If $M = (\text{let rec } x = N \text{ in } M')$ then $S = S' S_1 S_0$ with $\text{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Psi, N) = (\tau_0, \gamma_0, (C_0, S_0))$, $(C_1, S_1) = \text{Sol}(C_0; \{S_0(t) = \tau_0, \gamma_0(x) = 1\})$, $\text{Type}(C_1 - C_2, \Phi, M') = (\tau, \gamma_1, (C', S'))$ where $\Phi = \{x : (\forall C_2. S_1 S_0(t))\} \cup S_1 S_0(\Psi)$ and C_2 is $C_0 \setminus \text{fv}(S_1 S_0(\Psi))$ if N is pure, and $C_2 = \emptyset$ otherwise. By induction hypothesis, $S_0 \in \text{Sub}(\{t :: \emptyset\} \cup (C \upharpoonright \Psi), C_0)$, and $S_1 \in \text{Sub}(C_0, C_1)$, therefore $S_1 S_0 \in \text{Sub}(\{t :: \emptyset\} \cup (C \upharpoonright \Psi), C_1)$, hence $C_1 \vdash \{x : S_1 S_0(t)\} \cup S_1 S_0(\Psi)$, and therefore $C_1 - C_2 \vdash \Phi$. By induction hypothesis $C' \vdash \tau :: \emptyset$, and $S' \in \text{Sub}((C_1 - C_2) \upharpoonright \Phi, C')$. We have $C_1 \vdash S_1 S_0(t) :: \emptyset$ (see the clause (ii) of Definition 5.1), hence $\text{fv}(S_1 S_0(t)) \subseteq \text{dom}(C_1)$. By definition of C_2 , we then have $(C_1 - C_2) \upharpoonright \Phi = C_1 \upharpoonright S_1 S_0(\Psi)$, and therefore $S \in \text{Sub}(C \upharpoonright \Psi, C')$ since obviously $S_1 S_0 \in \text{Sub}(C \upharpoonright \Psi, C_1 \upharpoonright S_1 S_0(\Psi))$. \square

One may observe from this proof that, if we start the algorithm with (C, Ψ, M) such that $C \vdash \Psi$, then all the recursive calls to Type operate on arguments (C', Ψ', M') such that $C' \vdash \Psi'$.

PROPOSITION (SOUNDNESS) 5.10.

If $\text{Type}(C, \Psi, M) = (\tau, \gamma, (C', S))$ with $C \vdash \Psi$ then $C' ; S(\Psi)^\gamma \vdash M : \tau$.

PROOF: by induction on M . We examine only some cases.

- If $M = x$, we have $\Psi(x) = (\forall C_0.\tau)$ with $\text{dom}(C_0) \cap \text{dom}(C) = \emptyset$, $\gamma = 0_x$, $C' = C \cup C_0$, and $S = \text{id}$. By the previous Lemma we have $C' \vdash (\forall C_0.\tau) \succeq \tau$, therefore $C' ; S(\Psi)^\gamma \vdash x : \tau$.

- If $M = \lambda x N$ then $\tau = S'(t)^p \rightarrow \theta$, $\gamma = 1$ and $C' = \{p \leq \delta(x)\} \cup C_0$ where $(\theta, \delta, (C_0, S')) = \text{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Psi, N)$ and $S = S' \setminus t$. Since t is fresh, and in particular $t \notin \text{fv}(\Psi)$, we have $S(\Psi) = S'(\Psi)$. By induction hypothesis $C_0 ; x : S'(t)^{\delta(x)}, S(\Psi)^\delta \vdash N : \theta$, hence, by the Lemma 4.2(iii), $p \leq \delta(x), C_0 ; x : S'(t)^{\delta(x)}, S(\Psi)^\delta \vdash N : \theta$, whence $C' ; S(\Psi)^\delta \vdash M : \tau$ by the degree weakening rule, and the typing rule for abstraction.
- If $M = (M'N)$ then $\tau = S'(t)$, $\gamma = 0_{M'} \wedge S'(\delta)$ and $S = S'S_1S_0$ with $\text{Type}(C, \Psi, M') = (\tau_0, \gamma_0, (C_0, S_0))$, $\text{Type}(C'_0, S_0(\Psi), N) = (\tau_1, \gamma_1, (C_1, S_1))$ where $C'_0 = C_0 \upharpoonright S_0(\Psi)$,

$$(C', S') = \text{Sol}(\{t :: \emptyset\} \cup C_1 ; \{S_1(\tau_0) = \tau_1^p \rightarrow t\})$$

and

$$\delta(x) = \begin{cases} p & \text{if } N = x \notin \text{fv}(M') \\ 1 & \text{if } x \notin \text{fv}(N) \\ p \wedge (S_1(\gamma_0) \wedge \gamma_1)(x) & \text{otherwise} \end{cases}$$

By induction hypothesis, $C'_0 ; S_0(\Psi)^{\gamma_0} \vdash M' : \tau_0$, and by the Lemma 5.9, $C'_0 \vdash S_0(\Psi)$. Then by induction hypothesis $C_1 ; S_1S_0(\Psi)^{\gamma_1} \vdash N : \tau_1$. We have $C_1 ; S_1S_0(\Psi)^{S_1(\gamma_0)} \vdash M' : S_1(\tau_0)$ by the Lemmas 5.9 and 4.3, and since $S' \in \text{Sub}(C_1, C')$ we have, by the Lemma 4.3 again,

$$C' ; S(\Psi)^{S'S_1(\gamma_0)} \vdash M' : S'S_1(\tau_0)$$

and

$$C' ; S(\Psi)^{S'(\gamma_1)} \vdash N : S'(\tau_1)$$

Since $S'S_1(\tau_0) = S'(\tau_1)^{S'(p)} \rightarrow \tau$, we conclude using the degree weakening rule, and the typing rule for application.

- If $M = (\text{let rec } x = N \text{ in } M')$ then $S = S'S_1S_0$ and

$$\gamma(y) = \begin{cases} \gamma_1(x) \wedge (S'S_1(\gamma_0) \wedge \gamma_1)(y) & \text{if } y \in \text{fv}(N) \text{ and } M' \text{ is not pure} \\ (S'S_1(\gamma_0) \wedge \gamma_1)(y) & \text{otherwise} \end{cases}$$

with $\text{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Psi, N) = (\tau_0, \gamma_0, (C_0, S_0))$, $(C_1, S_1) = \text{Sol}(C_0 ; \{S_0(t) = \tau_0, \gamma_0(x) = 1\})$ and $\text{Type}(C_1 - C_2, \Phi, M') = (\tau, \gamma_1, (C', S'))$ where $\Phi = \{x : (\forall C_2. S_1S_0(t))\} \cup S_1S_0(\Psi)$ and C_2 is $C_1 \setminus \text{fv}(S_1S_0(\Psi))$, if N is pure, and $C_2 = \emptyset$ otherwise. Then $C_0 ; (x : S_0(t), S_0(\Psi))^{\gamma_0} \vdash N : \tau_0$, by induction hypothesis, and since $S_1 \in \text{Sub}(C_0, C_1)$ with $S_1S_0(t) = S_1(\tau_0) = \theta$ and $S_1(\gamma_0(x)) = 1$, we have $C_1 ; x : \theta^1, S_1S_0(\Psi)^{S_1(\gamma_0)} \vdash N : \theta$. By the Lemma 5.9 we have $C_1 - C_2 \vdash \Phi$, and therefore $S' \in \text{Sub}((C_1 - C_2) \upharpoonright \Phi, C')$ by the Lemma 5.9 again. In particular, S' is the identity on $\text{dom}(C_2)$, and therefore we have

$$C' ; x : (\forall C_2. S'(\theta))^{\gamma_1(x)}, S(\Psi)^{\gamma_1} \vdash M' : \tau$$

by induction hypothesis. Since obviously $S' \in \text{Sub}(C_1 \upharpoonright \Phi', C' \cup C_2)$ where $\Phi' = \{x : \theta\} \cup S_1S_0(\Psi)$, we have

$$C_2, C' ; x : S'(\theta)^1, S(\Psi)^{S'(S_1(\gamma_0))} \vdash N : S'(\theta)$$

by the Lemmas 4.4 and 4.3 (and possibly the Lemma 4.2 if $\text{dom}(C') \cap \text{dom}(C_2) \neq \emptyset$). We conclude using the degree weakening rule, and the typing rule for the let rec construct. \square

PROPOSITION (COMPLETENESS) 5.11. *Let Ψ and M be such that $C' ; S(\Psi)^\delta \vdash M : \tau$ for some type τ , degree assignment δ and substitution $S \in \text{Sub}(C, C')$ with $C \vdash \Psi$ and $C \upharpoonright \Psi = C$. Then $\text{Type}(C, \Psi, M) = (\theta, \gamma, (C_0, S_0))$, and there exists a substitution $S' \in \text{Sub}(C_0, C')$ such that $S = S'S_0$, $\tau = S'(\theta)$ and $C' \vdash \delta \leq S'(\gamma)$.*

PROOF: by induction on the inference of $C' ; \mathsf{S}(\Psi)^\delta \vdash M : \tau$, and by case on the last rule used to infer this sequent. The case where this rule is degree weakening is trivial, and is omitted. Then the proof actually proceeds by induction on M . We only examine some cases.

- If $M = x$ then $\mathsf{S}(\Psi)^\delta = x : \sigma^0, \Gamma$ with $C' \vdash \sigma \succeq \tau$, that is $\sigma = (\forall C_1. \tau_1)$ with $\tau = \mathsf{S}_1(\tau_1)$ for $\mathsf{S}_1 \in \mathsf{Sub}(C_1, C')$. We may assume that $\text{dom}(C_1) \cap (\text{dom}(C) \cup \text{dom}(C')) = \emptyset$, and therefore $\Psi(x) = (\forall C_1. \tau_0)$ with $\tau_1 = \mathsf{S}(\tau_0)$. Then $\text{Type}(C, \Psi, M) = (\tau_0, 0_x, (C \cup C_1, \text{id}))$. We may let $S' = \mathsf{S}_1 \cup \mathsf{S}$ in this case.

- If $M = \lambda x N$ then $\tau = \tau_0^d \rightarrow \tau_1$ and $\delta = 1$ with $C' ; x : \tau_0^d, \mathsf{S}(\Psi)^{\delta'} \vdash N : \tau_1$ for some δ' . If we let $\Phi = \{x : t\} \cup \Psi$ where t is fresh, and $\mathsf{S}_1 = \{t \mapsto \tau_0\} \cup \mathsf{S}$, then by induction hypothesis $\text{Type}(\{t :: \emptyset\} \cup C, \Phi, N) = (\theta, \gamma, (C_0, \mathsf{S}_0))$ and there exists $S'' \in \mathsf{Sub}(C_0, C')$ such that $\mathsf{S}_1 = S''\mathsf{S}_0$, $\tau_1 = S''(\theta)$ and $C' \vdash \{x \mapsto d\} \cup \delta' \leq S''(\gamma)$. In particular $\tau_0 = S''(\mathsf{S}_0(t))$ and $C' \vdash d \leq S''(\gamma(x))$. We have

$$\text{Type}(C, \Psi, M) = (\mathsf{S}_0(t)^p \rightarrow \theta, 1, (\{p \leq \gamma(x)\} \cup C_0, \mathsf{S}_0 \setminus t))$$

We let $S' = \{p \mapsto d\} \cup S''$ in this case.

- If $M = (M'N)$ then $C' ; \mathsf{S}(\Psi)^{\delta'} \vdash M' : \theta^a \rightarrow \tau$ and $C' ; \mathsf{S}(\Psi)^{\delta'} \vdash N : \theta$ with $\delta = 0_{M'} \wedge \delta''$ where

$$\delta''(x) = \begin{cases} a & \text{if } N = x \notin \text{fv}(M) \\ 1 & \text{if } x \notin \text{fv}(N) \\ a \wedge \delta'(x) & \text{otherwise} \end{cases}$$

Then by induction hypothesis $\text{Type}(C, \Psi, M') = (\tau_0, \gamma_0, (C_0, \mathsf{S}_0))$ and there exists $S'_0 \in \mathsf{Sub}(C_0, C')$ such that $\mathsf{S} = S'_0\mathsf{S}_0$, $\theta^a \rightarrow \tau = S'_0(\tau_0)$ and $C' \vdash \delta' \leq S'_0(\gamma_0)$. By the Lemma 5.9 $C_0 \vdash \mathsf{S}_0(C)$, and therefore by induction hypothesis $\text{Type}(C'_0, \mathsf{S}_0(\Psi), N) = (\tau_1, \gamma_1, (C_1, \mathsf{S}_1))$ where $C'_0 = C_0 \upharpoonright \mathsf{S}_0(\Psi)$, and there exists $S'_1 \in \mathsf{Sub}(C_1, C')$ such that $\mathsf{S}'_0 = S'_1\mathsf{S}_1$, $\theta = S'_1(\tau_1)$ and $C' \vdash \delta' \leq S'_1(\gamma_1)$. Then the verification condition $(\{t :: \emptyset\} \cup C_1 ; \{\mathsf{S}_1(\tau_0) = \tau_1^p \rightarrow t\})$ has a solution. Let (C'', S'') be its most general solution, so that $\{t \mapsto \tau\} \cup S'_1 = S''S'_1$. Then

$$\text{Type}(C, \Psi, (M'N)) = (S''(t), 0_{M'} \wedge S''(\gamma), (C'', S''S'_1\mathsf{S}_0))$$

where

$$\gamma(x) = \begin{cases} p & \text{if } N = x \notin \text{fv}(M) \\ 1 & \text{if } x \notin \text{fv}(N) \\ p \wedge (\mathsf{S}_1(\gamma_0) \wedge \gamma_1)(x) & \text{otherwise} \end{cases}$$

We have $\mathsf{S} = S''S'_1\mathsf{S}_1\mathsf{S}_0$, $S''S''(t) = \tau$ and $S''S''(p) = a$, and $C' \vdash \delta \leq S''(0_{M'} \wedge S''(\gamma))$, as it is easy to check.

- If $M = (\text{let rec } x = N \text{ in } M')$ then $C'', C' ; x : \theta^1, \mathsf{S}(\Psi)^{\delta'} \vdash N : \theta$ and $C' ; x : (\forall C'' . \theta)^\alpha, \mathsf{S}(\Psi)^{\delta'} \vdash M' : \tau$, where $t \in \text{dom}(C'') \Rightarrow t \notin \text{dom}(C')$ and C'' is empty if N is not pure, and

$$\delta(y) = \begin{cases} \alpha \wedge \delta'(y) & \text{if } y \in \text{fv}(N) \text{ and } M \text{ is not pure} \\ \delta'(y) & \text{otherwise} \end{cases}$$

By induction hypothesis $\text{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Psi, N) = (\tau_0, \gamma_0, (C_0, \mathsf{S}_0))$ and there exists $S'_0 \in \mathsf{Sub}(C_0, C')$ such that $\{t \mapsto \theta\} \cup \mathsf{S} = S'_0\mathsf{S}_0$, $\theta = S'_0(\tau_0)$, $C' \vdash 1 \leq S'_0(\gamma_0(x))$ and $C' \vdash \delta' \leq S'_0(\gamma_0)$. Then the verification condition $(C_0 ; \{\mathsf{S}_0(t) = \tau_0, \gamma_0(x) = 1\})$ has a solution. Let (C_1, S_1) be its most general solution, so that $\mathsf{S}'_0 = S'_1\mathsf{S}_1$, and let C_2 be $C_1 \setminus \text{fv}(\mathsf{S}_1\mathsf{S}_0(\Psi))$, if N is pure, and $C_2 = \emptyset$ otherwise. We have $C' \vdash (\forall C_2. \mathsf{S}_1\mathsf{S}_0(t)) \succeq (\forall C'' . \theta)$, hence $C' ; x : (\forall C_2. \mathsf{S}_1\mathsf{S}_0(t))^\alpha, \mathsf{S}(\Psi)^{\delta'} \vdash M' : \tau$ by the Lemma 4.8. Then by induction hypothesis $\text{Type}(C_1 - C_2, \{x : (\forall C_2. \mathsf{S}_1\mathsf{S}_0(t))\} \cup \mathsf{S}_1\mathsf{S}_0(\Psi), M) =$

$(\tau_1, \gamma_1, (C', S'))$ and there exists S'' such that $S'_1 = S''S'$, $\tau = S''(\tau_1)$, $C' \vdash \alpha \leq S''(\gamma_1(x))$ and $C' \vdash \delta' \leq S''(\gamma_1)$. Finally we have $\text{Type}(C, \Psi, M) = (\tau_1, \gamma, (C', S'S_1S_0))$ where

$$\gamma(y) = \begin{cases} \gamma_1(x) \wedge (S'S_1(\gamma_0) \wedge \gamma_1)(y) & \text{if } y \in \text{fv}(N) \text{ and } M \text{ is not pure} \\ (S'S_1(\gamma_0) \wedge \gamma_1)(y) & \text{otherwise} \end{cases}$$

It is easy to check that $C' \vdash \delta \leq S''(\gamma)$. \square

Finally, combining the soundness and completeness properties, we get our second main result:

THEOREM (TYPE RECONSTRUCTION) 5.12. *If there is no C such that $C \vdash \Psi$, or if $C \vdash \Psi$ and $\text{Type}(C, \Psi, M)$ fails, then for any degree assignment δ , the expression M is not typable in the context $C; \Psi^\delta$. If Ψ is closed, that is $\text{fv}(\Psi) = \emptyset$, and $C; \Psi^\delta \vdash M : \tau$ for some C , δ and τ , then $\text{Type}(\emptyset, \Psi, M)$ succeeds, returning $(\theta, \gamma, (C_0, \text{id}))$, so that $C_0; \Psi^\gamma \vdash M : \theta$, and there exists $S \in \text{Sub}(C_0, C)$ such that $\tau = S(\theta)$ and $C \vdash \delta \leq S(\gamma)$.*

One may observe that, thanks to the Lemma 5.5, there is a C such that $C \vdash \Psi$ if and only if there exists C such that $\mathcal{C}(\Psi) \triangleright^* C$ where $\mathcal{C}(\Psi)$ is the set of annotation assertions to validate in order for Ψ to be acceptable, that is $\mathcal{C}(x_1 : \sigma_1, \dots, x_n : \sigma_n) = \{\sigma_1 :: \emptyset, \dots, \sigma_n :: \emptyset\}$.

6. Object-Oriented Programming

In this section we illustrate the expressive power of our calculus, as regards object-orientation, both from an operational and from a typing point of view. To this end, we will introduce a few derived constructs. Our approach to object-orientation is *mixin-based*. The notion of a “mixin” has been introduced in object-oriented programming languages of the 80’s, mainly in languages based on LISP. In [7], it has been advocated as the “building block” for inheritance, and has recently received some attention, see for instance [3, 5, 22]. Roughly speaking, a mixin is a class definition parameterized over its superclass (there is some similarity with the parameterized classes of EIFFEL [33] and the “virtual classes” of BETA [31]). Let us introduce some informal terminology:

- an *object* is the fixpoint ($\text{fix } \text{Gen}$) of a generator (*cf.* [13]).
- a *generator* is a function of a “self” parameter, returning a record of fields and methods. Then a typical generator value is thus

$$\lambda s \langle \dots \text{fields} \dots \text{methods} \dots \rangle$$

like for instance the empty generator $\lambda s \diamond$, and an object is therefore a recursive record, as in [11, 14, 46, 51]. Notice that for a generator to be able to generate some object, the type of self (that is, s) should be unifiable with the type of the record returned by the generator, and moreover the generator must be a protective function.

- a *field* is like any ordinary field in a record, either mutable or not. In an object generator, a field is not supposed to contain the self parameter.
- a *method* is a field in a record, whose value is a “thunk”, that is a function of a dummy parameter (just to freeze evaluation at this point). A method explicitly depends on two parameters, **super** and **self**, representing respectively the current object as a member of the superclass, and as a member of the current class, as far as the ownership of methods is concerned (these parameters are not keywords; rather, they are bound names, subject to α -conversion). We use a special syntax for invoking a method ℓ of an object, different from selection, namely $(M \Leftarrow \ell)$ (just to unfreeze evaluation).
- a *mixin* is a function mapping generators to generators, usually by extending and/or modifying the record returned by its argument. A typical mixin value is thus $\lambda g \lambda s \langle \dots \rangle$, and inheritance is basically mixin composition, that is $\lambda g M(M'g)$ if M inherits M' . In the methods introduced by a

$M, N \dots$	$::= \dots \mid (M \Leftarrow \ell)$	<i>method invocation</i>
$V, W \dots$	$::= \dots \mid \text{new} \mid \text{mixin}(T)$	
T	$::=$	<i>mixins</i>
	$\mid \text{var } \ell = N \mid \text{cst } \ell = N$	<i>field definition</i>
	$\mid \text{meth } \ell(y, x) = M \mid \text{meth } \ell(y, x) \leftarrow M$	<i>method definition/override</i>
	$\mid \text{without } \ell \mid \text{rename } \ell \text{ as } \ell'$	<i>restriction, renaming</i>
	$\mid \text{inherit } M \mid (T, T')$	<i>inheritance</i>

Figure 11: Mixin Constructs

mixin, the **super** parameter is to be interpreted as $(g\ s)$, the “generic object” (where **self** is not yet bound) of the superclass, while the **self** parameter is obviously interpreted as s .

- a *class* is a function taking as argument a series of “instance parameters” and returning a mixin⁽⁴⁾. A typical class value is thus

$$\lambda x_1 \dots x_n. \lambda g \lambda s \langle \dots \text{fields} \dots \text{methods} \dots \rangle$$

An object instance of such a class C is the fixpoint of the generator obtained by applying the class to initial values of the instance parameters, usually determining the initial state of the object, and to the empty generator, that is:

$$\text{new}(C\ N_1 \dots N_n) = \text{fix}(C\ N_1 \dots N_n (\lambda s \diamond))$$

If the class, or more appropriately the parameterized mixin C has no instance, because the type of **self** is not unifiable with the type of the record returned by the generator, we say that the class is *abstract*. To inherit a class, one has to “extract” from it the mixin it returns. This is usually done by applying the class to formal parameters, i.e. $(C\ y_1 \dots y_n)$, which are instance parameters of the subclass, but one may more generally inherit the class as $(C\ N_1 \dots N_n)$.

Having thus informally described our model for objects and inheritance, we now introduce some corresponding syntax to define mixins and objects, extending the language as indicated in Figure 11. We could also add a notation for field override, but this can actually be written (**without** ℓ , **var** $\ell = N'$) or (**without** ℓ , **cst** $\ell = N'$). We use the same notation for method override than for overriding a field in a record, although these are not exactly the same operations. The interpretation of the extended language into the core language is given in Figure 12, by means of a translation $\llbracket \cdot \rrbracket$ which is the identity as regards the core constructs. In this figure, we use $\lambda.M$ to denote a thunk, that is an abstraction over a variable which does not occur in M . In the translation of mixins, the variables g , s and z are supposed to be fresh. Then one can see that, in particular, a field has no knowledge of the **self** parameter, since s does not occur in $\llbracket N \rrbracket$, whereas the **super** and **self** parameters of a method are instantiated into s and z , with $z = gs$, respectively. We have adopted a sophisticated interpretation of (T, T') , involving η -expansion, to avoid using this construct with arguments of inappropriate type (for the same reason we have introduced a dummy declaration **let** $z = gs$ in the interpretation of **inherit** M). However, one should understand (T, T') simply as $\lambda g (\llbracket T' \rrbracket (\llbracket T \rrbracket g))$, that is $\llbracket T \rrbracket \circ \llbracket T' \rrbracket$.

Regarding the derived typing of the extended language, we first observe that, since it is a thunk, a method in a mixin always has a type of the form $(t^p \rightarrow \tau)$ (with no constraint on p), and that

⁽⁴⁾ this notion of a class is slightly non-standard, since there are some mixins that one would not normally call “classes”. Indeed the name “mixin” is sometimes used only for classes that are not intended to be instantiated into objects.

$$\begin{aligned}
\llbracket M \Leftarrow \ell \rrbracket &= (\llbracket M \rrbracket.\ell)\langle \rangle \\
\llbracket \text{new} \rrbracket &= \lambda m(\text{fix}(m \lambda s \langle \rangle)) \\
\llbracket \text{mixin}(T) \rrbracket &= \llbracket T \rrbracket \\
\llbracket \text{var } \ell = N \rrbracket &= \lambda g \lambda s \langle gs, \ell = \text{ref } \llbracket N \rrbracket \rangle \\
\llbracket \text{cst } \ell = N \rrbracket &= \lambda g \lambda s \langle gs, \ell = N \rangle \\
\llbracket \text{meth } \ell(y, x) = M \rrbracket &= \lambda g \lambda s (\text{let } z = gs \text{ in } \langle z, \ell = \lambda.((\lambda y \lambda x \llbracket M \rrbracket)z s) \rangle) \\
\llbracket \text{meth } \ell(y, x) \leftarrow M \rrbracket &= \lambda g \lambda s (\text{let } z = gs \text{ in } \langle z \setminus \ell, \ell = \lambda.((\lambda y \lambda x \llbracket M \rrbracket)z s) \rangle) \\
\llbracket \text{without } \ell \rrbracket &= \lambda g \lambda s ((gs) \setminus \ell) \\
\llbracket \text{rename } \ell \text{ as } \ell' \rrbracket &= \lambda g \lambda s (\text{let } z = gs \text{ in } \langle z \setminus \ell, \ell' = z.\ell \rangle) \\
\llbracket \text{inherit } M \rrbracket &= \lambda g \lambda s (\text{let } z = gs \text{ in } \llbracket M \rrbracket gs) \\
\llbracket T, T' \rrbracket &= \lambda g \lambda s (\llbracket T' \rrbracket (\lambda s (\llbracket T \rrbracket (\lambda s (gs)) s)) s)
\end{aligned}$$

Figure 12: Interpretation

when invoking this method, the type variable t will be instantiated into $\text{unit}^{(5)}$. Then we may introduce the notation $\langle \rho, \text{meth } \ell : \tau \rangle$, standing for $\langle \rho, \ell : \text{unit}^p \rightarrow \tau \rangle$, and one can see that a derived typing for method invocation is:

$$\frac{C ; \Gamma \vdash \llbracket M \rrbracket : \langle \rho, \text{meth } \ell : \tau \rangle}{C ; \Gamma^{0_M} \vdash (M \Leftarrow \ell) : \tau}$$

Regarding the combinator `new` that is intended to instantiate a class, we have

$$\frac{}{C ; \Gamma \vdash \text{new} : ((\theta^p \rightarrow \langle \rangle)^q \rightarrow \tau^1 \rightarrow \tau)^0 \rightarrow \tau}$$

The derived typing of the `mixin` constructs is given in Figure 13, which we now comment. A first observation is that the type of a `mixin` is generally quite big. Although this is clearly very important from a pragmatic point of view, we shall not in this paper attempt to introduce meaningful abbreviations regarding these types – except for the type of methods –, since our purpose is mainly to experiment with a preliminary language design.

A `mixin` has a type of the form $(\theta^a \rightarrow \tau_0)^b \rightarrow \theta^c \rightarrow \tau_1$ where $(\theta^a \rightarrow \tau_0)$ is the type of the generator argument (associated with the superclass), so that θ is the type of `self` and τ_0 is the type of the `super` parameter, and τ_1 is the type of the value returned by the `mixin`. In the examples that we shall examine, θ and τ_1 are “open” record types, that is $\langle t, \ell_1 : \vartheta_1, \dots, \ell_n : \vartheta_n \rangle$. In most models of typed objects, the latter usually is a “fixed” record type, that is $\langle \ell'_1 : \vartheta'_1, \dots, \ell'_n : \vartheta'_n \rangle$, taken as representing the type, or the interface of the class. In our model, a class – being a `mixin` – depends on a superclass whose value is only fixed at object creation time, and this explains the “open” type. We must also point out that, in most models of typed objects, except the one of Wand [51] that we follow (see also [18]), the type of `self` is supposed to be a subtype of the type of the class. In particular, `self` is generally supposed to support all the methods offered by the class. Since we are inferring types of variables from their usage in expressions, this will generally not be the case here, and therefore the type of `self` is a useful information to know about the type of a `mixin`. Notice that for a class to have some instance, one must be able to solve the equation $\theta = \tau_1$, as required in the type of `new`. We could at compile time declare the class to be abstract if this equation has no solution. On the other hand, except for pathological uses of `inherit M`, a `mixin` is always a protective function of `self` (provided that the super-generator is protective too, which is obviously the case of

⁽⁵⁾ in the implementation, the dummy parameter of a method is forced to have type `unit`, and therefore any method has a type of the form $(\text{unit}^p \rightarrow \tau)$.

$$\begin{array}{c}
\frac{C; \Gamma \vdash \llbracket N \rrbracket : \tau \quad C \vdash \rho :: \{\ell\} \quad C \vdash c \leq a}{C; \Gamma^1 \vdash \text{var } \ell = N : (\theta^a \rightarrow \rho)^b \rightarrow \theta^c \rightarrow \langle \rho, \ell : \tau \text{ ref} \rangle} \\
\frac{C; \Gamma \vdash \llbracket N \rrbracket : \tau \quad C \vdash \rho :: \{\ell\} \quad C \vdash c \leq a}{C; \Gamma^1 \vdash \text{cst } \ell = N : (\theta^a \rightarrow \rho)^b \rightarrow \theta^c \rightarrow \langle \rho, \ell : \tau \rangle} \\
\frac{C; y : \rho^{a'}, x : \theta^{b'}, \Gamma \vdash \llbracket M \rrbracket : \tau \quad C \vdash \rho :: \{\ell\} \quad C \vdash c \leq a}{C; \Gamma^1 \vdash \text{meth } \ell(y, x) = M : (\theta^a \rightarrow \rho)^b \rightarrow \theta^c \rightarrow \langle \rho, \text{meth } \ell : \tau \rangle} \\
\frac{C; y : \langle \rho, \ell : \tau' \rangle^{a'}, x : \theta^{b'}, \Gamma \vdash \llbracket M \rrbracket : \tau \quad C \vdash \rho :: \{\ell\} \quad C \vdash c \leq a}{C; \Gamma^1 \vdash \text{meth } \ell(y, x) \leftarrow M : (\theta^a \rightarrow \langle \rho, \ell : \tau' \rangle)^b \rightarrow \theta^c \rightarrow \langle \rho, \text{meth } \ell : \tau \rangle} \\
\frac{C \vdash \rho :: \{\ell\}}{C; \Gamma^1 \vdash \text{without } \ell : (\theta^a \rightarrow \langle \rho, \ell : \tau \rangle)^b \rightarrow \theta^c \rightarrow \rho} \\
\frac{C \vdash \rho :: \{\ell, \ell'\}}{C; \Gamma^1 \vdash \text{rename } \ell \text{ as } \ell' : (\theta^a \rightarrow \langle \rho, \ell : \tau \rangle)^b \rightarrow \theta^c \rightarrow \langle \rho, \ell' : \tau \rangle} \\
\frac{C; \Gamma \vdash \llbracket M \rrbracket : (\theta^a \rightarrow \tau_0)^b \rightarrow \theta^c \rightarrow \tau_1 \quad C \vdash c' \leq a \wedge c}{C; \Gamma^1 \vdash \text{inherit } M : (\theta^a \rightarrow \tau_0)^{b'} \rightarrow \theta^{c'} \rightarrow \tau_1} \\
\frac{C; \Gamma \vdash \llbracket T \rrbracket : (\theta^a \rightarrow \tau_0)^b \rightarrow \tau_1^c \rightarrow \tau_2 \quad C; \Gamma \vdash \llbracket T' \rrbracket : (\tau_1^{a'} \rightarrow \tau_2)^{b'} \rightarrow \tau_3^{c'} \rightarrow \tau_4 \quad C \vdash c'' \leq c', a' \leq c, a \leq a''}{C; \Gamma^1 \vdash (T, T') : (\theta^{a''} \rightarrow \tau_0)^{b''} \rightarrow \tau_3^{c''} \rightarrow \tau_4}
\end{array}$$

Figure 13: Typing the Mixins

the “universal” generator $\lambda s \diamond$), since the self parameter is only used in method bodies, which are values. Therefore we normally do not end up with an unsafe recursion when trying to create an object instance of a class.

Now let us see examples illustrating the use of these derived constructs. Obviously, we do not claim that the examples we propose are interesting programs by themselves; they are just meant to give an idea of the flexibility of the approach. They are all variations around the standard example in discussing models of objects, namely the “point”. We start by rewriting in our syntax the class of points that we gave in Section 2 – using the standard notation for assignment, that is $M := N$ instead of $\text{set } M N$:

```

let point = λx mixin (
    var pos = x,
    meth move(z, s) = λd(s.pos := !s.pos + d)
)
in ...

```

One can see that, up to some β_v -conversions, the translation of the mixin defining the points is as follows:

$$\lambda x \lambda g \lambda s \langle gs, \text{pos} = \text{ref } x, \\ \text{move} = \lambda. \lambda d (s.\text{pos} := !s.\text{pos} + d) \rangle$$

which is very similar to our definition of the point class in Section 2, except for the g parameter, and the record gs that this mixin extends. Then, again up to some β_v -conversions, the expression $(\text{point } 0) \lambda s \langle \rangle$ evaluates into

$$\lambda s \langle \text{pos} = u, \\ \text{move} = \lambda. \lambda d (s.\text{pos} := !s.\text{pos} + d) \rangle$$

where u is a location whose value in the store is 0, and therefore the object $\text{fix}((\text{point } 0) \lambda s \langle \rangle)$, that is $\text{new}(\text{point } 0)$, is the recursive record

$$(\text{let rec } s = \langle \text{pos} = u, \text{move} = \lambda. \lambda d (s.\text{pos} := !s.\text{pos} + d) \rangle \text{ in } s)$$

which could be optimized into $(\text{let rec } s = \langle \text{pos} = u, \text{move} = \lambda. \lambda d (u := !u + d) \rangle \text{ in } s)$. Regarding the typing, one can see that, abbreviating $(\forall t :: \emptyset. \sigma)$ into $(\forall t. \sigma)$, the (polymorphic) type of **point** is – assuming that $+$ is of type $\text{int}^0 \rightarrow \text{int}^0 \rightarrow \text{int}$:

$$r' \leq r; \vdash \text{point} : \forall t_0. \forall t_1 :: \{\text{pos}\}. \forall t_2 :: \{\text{pos}, \text{move}\}. t_0^p \rightarrow \\ (\rho^r \rightarrow t_2)^q \rightarrow \\ \rho^{r'} \rightarrow \\ \langle t_2, \text{pos} : t_0 \text{ ref}, \text{meth move} : \text{int}^0 \rightarrow \text{unit} \rangle$$

where ρ is the type of **self**, namely $\rho = \langle t_1, \text{pos} : \text{int ref} \rangle$, and therefore we have:

$$\vdash (\text{point } 0) \lambda s \langle \rangle : \langle t_1, \text{pos} : \text{int ref} \rangle^{r'} \rightarrow \\ \langle \text{pos} : \text{int ref}, \text{meth move} : \text{int}^0 \rightarrow \text{unit} \rangle$$

We finally get the expected type for the point object:

$$\vdash \text{new}(\text{point } 0) : \langle \text{pos} : \text{int ref}, \text{meth move} : \text{int}^0 \rightarrow \text{unit} \rangle$$

It is worth noting (*cf.* [51]) that the type of **self** only retains what is required of the current object from its usage in the definition of the class, namely the presence of a **pos** field with an **int ref** type. In particular, the type $\langle t_1, \text{pos} : \text{int ref} \rangle$ of **self** is not a subtype of the “interface” of the **point** class, that we may represent with $\langle \text{pos} : t_0 \text{ ref}, \text{meth move} : \text{int}^0 \rightarrow \text{unit} \rangle$ – this is the type of $\lambda x \lambda s (\text{point } x (\lambda s \langle \rangle) s)$. This allows us to make use of some non-standard forms of code reuse, like for instance *inheritance by restriction*:

$$\text{immobilePoint} = \lambda x \text{mixin} (\\ \text{inherit}(\text{point } x), \\ \text{without move} \\)$$

Here we inherit a point object, with a formal initial position x , and we decide to introduce a new kind of points, which we cannot move. The translation of this class is, again with some optimisation:

$$\lambda x \lambda g \lambda s \langle gs, \text{pos} = \text{ref } x \rangle$$

Then we can create an object instance of that class, because the type of `self`, which is still $\langle t_1, \text{pos} : \text{intref} \rangle$, may be unified with the type of the record returned by the class. This particular example may not look especially interesting – we shall see later another example of the use of restriction. Some similar examples of excluding methods, like for instance building a class of stacks from a given class of dequeues, were given long ago by Snyder [46, 47]. Clearly, allowing such a reuse mechanism reinforces the fact that “inheritance is not subtyping” [14].

The fact that the type of `self` in the `point` class does not contain any reference to the `move` method can be exploited in another way: we may redefine this method with a type which is unrelated (except for what it requires of `self`) to the one it has in the `point` class. For instance, one may decide to modify the `move` method, so that it now takes one further argument, which is a unit of measure. Then we write, assuming a function f giving the conversion factor into the default unit:

$$\begin{aligned} \text{uPoint} = \lambda x \text{ mixin} (\\ & \text{inherit}(\text{point } x), \\ & \text{meth move}(z, s) \leftarrow \lambda d \lambda y (s.\text{pos} := !s.\text{pos} + d * (f y)) \\ &) \end{aligned}$$

The type of `self` in this class is the same as in `point`, whereas the (abbreviated) type of `move` is now $\text{int}^0 \rightarrow \vartheta \rightarrow \text{unit}$ where ϑ is the type of measure units. This is an example of *inheritance by method override* (or *redefinition*). Usually, in this kind of inheritance, the redefined method is required to have the same type as the overridden method, or a subtype of it. Let us see now an example of the use of the `super` parameter in methods. We decide to create a new kind of points, similar to the previous one, but using the unit of measure as an instance parameter, fixing the scale for its movements:

$$\begin{aligned} \text{scaledPoint} = \lambda x \lambda y \text{ mixin} (\\ & \text{inherit}(\text{uPoint } (x * (f y))), \\ & \text{meth move}(z, s) \leftarrow \lambda d (z \leftarrow \text{move } d y) \\ &) \end{aligned}$$

Here we inherit the `uPoint` class with an argument which is not just an instance parameter, as in the previous examples, but a compound value. Notice that while we used, in methods, the record selection syntax $M.\ell$ when accessing the field of an object, we must obviously use the derived construct $M \leftarrow \ell$ to invoke a method of an object – namely, in this example, invoking the previous version of the `move` method, attached to the superclass. The next example uses the very common form of *inheritance by extension*, consisting in adding new fields or methods. A method to fix a new position of a point is introduced:

$$\begin{aligned} \text{resetablePoint} = \lambda x \text{ mixin} (\\ & \text{inherit}(\text{point } x), \\ & \text{meth reset}(z, s) = \lambda x (s.\text{pos} := x) \\ &) \end{aligned}$$

We can now give another example of inheritance by restriction, similar to the one of stacks from dequeues in [46]: we may decide to restrict the use of the `reset` method to put the point back to its

initial position, and to remove access to this method:

```
clearablePoint = λx mixin (
    inherit(resettablePoint x),
    meth clear(z, s) = (z ← reset)x,
    without reset
)
```

A feature which we might wish to have is the ability to perform some given procedure at each object creation from a class. For instance, we may wish to have a counter incremented, or some warning printed each time an object is created. The simplest way to do that is to prefix the definition of the class with the appropriate procedure, as follows:

```
let iPoint = let n = ref 0
    in let init = λx (n := !n + 1) ; print_string "new point number " ; print_int !n ;
        print_string " at " ; print_int x ; print_newline();
    in λx (init x) ; (point x)
in ...
```

We could also incorporate the initialisation procedure as a method in the class, thus gaining the ability to invoke other methods. In this case, to create an object we would use the function:

```
newlnit = λm (let o = new m in o ← init ; o \init)
```

All the previously introduced classes are typable, and objects may be created from them. Up to now, the form of inheritance that we have used is quite standard in that we always specified the inherited superclass – some kind of point –, while adding, modifying or removing some of its ingredients. However, mixins are really means to transform classes, getting new ones by “applying” the mixin to various superclasses, thus *reusing the transformation*. Here is a more mixin-oriented example: coloring an object. We define the coloring mixin as follows:

```
let coloring = λc mixin (
    var color = c,
    meth paint(z, s) = λy (s.color := y)
)
in ...
```

Then, in the scope of this declaration, we can use *inheritance by composition* to get a colored version of a variety of classes, for instance:

```
colorPoint = λx λc mixin (
    inherit(point x),
    inherit(coloring c)
)

colorRectangle = λx λy λc mixin (
    inherit(coloring c),
    inherit(rectangle x y)
)
```

The `colorPoint` for instance is typable, because we may unify the types that the `self` parameter has in the `point` and `coloring` classes, which are respectively $\langle t_1, \text{pos} : \text{int ref} \rangle$ and $\langle t'_1, \text{color} : t'_0 \text{ ref} \rangle$. Notice that the polymorphism associated with the `let`-construct, and more specifically the polymorphism offered by Wand's row variables is crucial here for the inheritance mechanism to work properly, where one usually employs some form of subtyping (see [10, 21]). This kind of polymorphism allows us in particular to reuse the `coloring` mixin in various contexts. Obviously, the type system would reject a composition of mixins making incompatible requirements about the type of `self`, like for instance having a common field with different (i.e. non-unifiable) types. Although we do not use a subtype relation, it is obviously possible to write explicit coercion functions, like for instance:

$$\text{colorless} = \lambda x(x \setminus \text{color} \setminus \text{paint})$$

Notice that $(\text{colorless } x)$ is the “same” object as x , in the sense that they share the same state, apart from what regards the color. Alternatively, one can use pattern-matching (see [26]) to “extract a sub-object”:

$$\text{asPoint} = \lambda x \langle \text{pos} = x.\text{pos}, \text{move} = x.\text{move} \rangle$$

To get a less structural view of subtyping, we could also attach this function as a method of the `point` class, writing it as: `meth asPoint(z, s) = ⟨pos = s.pos, move = s.move⟩`. Again, $(\text{asPoint } x)$ is, as a point, the “same” object as x . We cannot use a similar technique to make a copy of a point, since it is not possible to break the (recursive) binding of `self` in the `move` method. We should instead use another function, namely:

$$\text{copyAsPoint} = \lambda x \text{new}(\text{point}(!x.\text{pos}))$$

Since a mixin is a class transformer, it may introduce some new ingredients – fields or methods – while relying on the fact that some other ingredients will be provided by the superclass – these could be called “virtual”. This shows up in the type of `self` (or `super`), and therefore the type system will reject any attempt to create an object instance of such an *abstract class*. An example is:

$$\text{resetPos} = \text{mixin}(\text{meth reset}(z, s) = \lambda x(s.\text{pos} := x))$$

This mixin can be reused (inherited), but not instantiated, since typing (new resetPos) would involve solving the equation $\langle t_1, \text{pos} : t_0 \text{ ref} \rangle = \langle \text{meth reset} : t_0 \rightarrow \text{unit} \rangle$, which is impossible. This mixin is intended to be used in combination with another one that introduces a `pos` field. To illustrate the use of the renaming facility, we elaborate on the last example: we redefine the `reset` method so that it updates not only the position, but also the color of an object (of some superclass), while keeping the possibility of updating the position only.

$$\begin{aligned} \text{resetPosColor} = \text{mixin} (\\ & \text{rename reset as resetPos,} \\ & \text{meth reset}(z, s) = \lambda x \lambda c(z \leftarrow \text{resetPos } x ; z \leftarrow \text{paint } c) \\ &) \end{aligned}$$

Notice that it would be wrong – that is, the type system would complain – using overriding to define the `reset` method here, because after renaming it into `resetPos`, it is no longer present. The next kind of point illustrates a form of *multiple inheritance*:

$$\begin{aligned} \text{richPoint} = \lambda x \lambda c \text{mixin} (\\ & \text{inherit}(\text{resettablePoint } x), \\ & \text{inherit}(\text{coloring } c), \\ & \text{inherit resetPosColor} \\ & \dots) \end{aligned}$$

We have omitted some parentheses here, and it should indeed be possible to prove that inheritance by composition is associative, with respect to some observational semantics. The order in which the components are introduced in the inheritance chain is also sometimes irrelevant: we could commute `inherit(resettablePoint x)` and `inherit(coloring c)` in the example without affecting the result, but the type system rejects inconsistencies – definition of methods (or fields) already present or required – that would arise if we had written one of these two ingredients after inheriting `resetPosColor`. Then multiple inheritance is restricted here to a *linear* pattern where the ingredients are introduced from left to right, and can be used or overridden subsequently, but not re-introduced. Typically, inheriting twice the same class, either directly or indirectly, is forbidden, if this class introduces some new field or method.

An issue that we have not discussed is that of the *visibility* of the ingredients of a class. In our previous examples, one can always “externally” update the state of a point p , that is, its position, simply by executing a statement `p .pos := ...`. However, in some cases we would like to forbid such a manipulation (see [21] for some examples). In many object-oriented programming languages, the visibility of the fields, and of some of the methods of a class is, implicitly or explicitly, restricted to inheriting classes and/or to objects instances of the same class. In this paper we do not investigate this issue, leaving this for further work, but we merely notice that one may always use the `let` construct to achieve state encapsulation, as in

```
secretPoint = λx let pos = ref x
               in mixin (
                   meth move( $z, s$ ) = λd(pos := !pos + d)
                   meth draw( $z, s$ ) = ...
                   )
```

for instance. This approach, where the fields are strictly private to objects – but may be given access to by means of methods – is the one advocated by Snyder in [46, 47]. We could indeed have adopted a strict interpretation of fields (this can actually be done simply by removing the related constructs), but we think in some cases it may be worth having more opportunities for inheritance.

A feature we would like to add to our language is the ability to return, or send `self`. For instance, we may wish to attach the `colorless` function as a method of the `coloring` mixin, writing it as

```
meth colorless( $z, s$ ) = s\color\paint
```

Another typical example is a method for *cloning* an object, that we could define using recursive classes, as in [14]:

```
let rec clonablePoint = λx mixin (
    inherit(point  $x$ ),
    meth clone( $z, s$ ) = new(clonablePoint(!s.pos))
    )
```

However, to be able to type this class, and instances of the `coloring` mixin extended with the `colorless` method, we should extend the type language with recursive record types $\mu t.\rho$. This is also necessary

for typing a “subject/observer” interaction, where a subject notify itself to an observer, like in:

```

let subject = λo mixin (meth notify(z, s) = λy.y o s)
in let window = λxλo mixin (
    inherit(point x),
    inherit(subject o),
    meth move(z, s) ← λd(z ← move d ; s ← notify(λo.o ← moved)),
    meth draw(z, s) = ...
)
in let manager = mixin (meth moved(z, s) = λw.w ← draw)
in let m = new manager
in new (window 42 m)

```

which, assuming that the code for the `draw` method is of type `unit`, has a type `Window` satisfying the equation:

$$\begin{aligned}
 \text{Window} &= \langle \text{pos} : \text{int ref} \\
 &\quad \text{meth move} : \text{int} \rightarrow \text{unit} \\
 &\quad \text{meth notify} : \text{Manager} \rightarrow \text{Window} \rightarrow \text{unit} \\
 &\quad \text{meth draw} : \text{unit} \rangle \\
 \text{Manager} &= \langle \text{meth moved} : \text{Window} \rightarrow \text{unit} \rangle
 \end{aligned}$$

Notice that one could modify the `notify` method, so that it transmits a restricted view of the current object, like for instance:

$$\text{meth notify}(z, s) \leftarrow \lambda y.z \leftarrow \text{notify } y \text{ o } (s \setminus \text{pos} \setminus \text{move})$$

Sending objects, including `self`, to other objects is a natural way of programming in a higher-order, functional language, and therefore adding recursive record types is a natural extension to consider, which does not interfere with the typing of safe recursion. It would be interesting to see how our model, thus extended with recursive types, supports programming of various patterns that have emerged from object-oriented programming practice (*cf.* [23]).

7. Related Work

The issue of imposing restrictions on recursive definitions to ensure that they define something is not at all a new one: many examples of notions of “contractive” or “guarded” recursive definitions may be found, for instance in formal language theory (Greibach’s normal form of context-free grammars), process calculi or co-inductive types theories. In a language like ML, recursion is usually restricted to the form $(\text{let rec } f = \lambda x N \text{ in } M)$, but as we discussed in the introduction, this does not suit our purpose. This has been recently generalized to deal with recursive modules by Harper & al. [15], who use a “valuability” predicate, drawing upon Moggi’s existence predicate. However, this again does not suit our purpose: for one thing, we wish to accept recursive expressions with evaluation that yields some computational effects. Moreover, in order to create objects instance of a class, we need to accept expressions like $(\text{let rec } x = (Gx) \text{ in } M)$, as we have pointed out, and this – hence in particular our fixpoint combinator $\text{fix} = \lambda f(\text{let rec } x = fx \text{ in } x)$ – is rejected by the valuability system of [15]. As far as I can see, no obvious solution to this specific problem emerges from the literature.

Regarding the modelling of objects, the literature is quite rich – we have mentioned a bit of it in the introduction. We did not formally compare our model to other ones, but it should be clear that,

as far as objects (not mixins) are concerned, what we propose is very close to Reddy’s denotational semantics of classes and objects [40], and to the cyclic record semantics (see [2], with the difference that we do not have to use assignment to model method override). Reddy did not deal with types however (nor with operational semantics), while this is our main concern. The mixin-based constructs we proposed are very close to the ones introduced by Bracha in the design of JIGSAW [8]; they are here integrated, by means of a formal interpretation, into an implicitly typed language. As we said in the introduction, most of the proposed models that include the main features of object-oriented programming use higher-order type theories, for which type reconstruction is not available. Palsberg has explored the type inference problem for fragments of Abadi and Cardelli’s object calculi [37, 38], but these calculi do not have the principal type property. As a matter of fact, with the exception of OCAML⁽⁶⁾, none of the object models we have cited supports principal type inference à la ML.

The OCAML’s model [44] integrates a class-based object layer into the ML language. The operational semantics is self-application, that is, the current object is substituted for `self` in the body of the method which is invoked. However, methods are not functions of `self` (but nonetheless regarded as values), as in Abadi and Cardelli’s calculi for instance: `self` is a “special variable” that occurs free in the methods, as in the recursive record semantics of objects. As a consequence, typing is similar to the one of Wand’s model, with specific type constructions for classes and objects, though class types are not arrow types, but depend on the type of `self` given by the typing context. This is not exactly the model of the OCAML language [30] however, in which a class body begins with a statement `object(s)`, which is a binding for a `self` parameter. This is needed if one has to program with nested classes for instance. The semantics of this specific binder is not formally described; a class body is a value, but when an instance object is created, the scope of this binder is opened to evaluation (more precisely, the state part of the object is evaluated, but not the method part).

Apart from the fact that OCAML is, as a programming language, obviously much more elaborate than the preliminary design we proposed, there are some differences, and also strong similarities between the two models. Indeed, we do not claim our object model is by itself original: as we said, it is very close to the one of JIGSAW [8], and was also strongly inspired by the one of OCAML. For instance, although in OCAML classes are not first class, and there is no explicit “mixin” facility, the inheritance mechanism is very close to the one we have adopted – with a difference regarding how to use `super`, however. Also, we may use a restriction operation over classes, which is absent from OCAML. Another difference is that we do not require fields to be private to objects. In this paper, we have favoured an object encoding approach, rather than an object calculus approach – which is the one of OCAML –, with the idea that this, in particular, should provide us with firm foundations for the typing of object-oriented constructs. One can see for instance that in OCAML, although the type system is very close to Wand’s one, the typing of some constructions is not as general as it could be. For instance, method overriding is invariant, as far as types are concerned, and in the typing of classes, the type of `self` is required to extend the record type of methods, and this prohibits reusing classes by restricting their set of methods.

Encoding objects as recursive records obviously relies on some record calculus. One could in principle use standard, “fixed” records, as in [17] for instance, but this is not quite compatible with the idea of reusing code, since in inheriting from a class, one would have to explicitly include the method list of the superclass (as it is done also in [2]). Various calculi of records that are “extensible” in some sense have been studied, starting with Wand’s one [49] (some initial difficulties with principal typing were later solved, see [26, 36, 41, 51]). For instance, various forms of record concatenation, symmetric or asymmetric, have been considered, see [25, 42, 50]. In this paper we have chosen to use, mainly for a simplicity reason, a strict version of Cardelli and Mitchell’s calculus [12], with a

⁽⁶⁾ and also obviously Wand’s model, which is however not expressive enough. The OML language of Reppy and Riecke [45] also supports principal typing, but not inheritance.

simpler typing however, where we express negative information by means of “annotations” rather than using subtyping and bounded quantification. Our record calculus is equivalent to the one of Jategaonkar and Mitchell [26], who use a form of pattern-matching. There are some limitations with our choice: for instance, we only allow a limited form of (linear) multiple inheritance, but multiple inheritance is well-known to be difficult to manage (see [47] for instance). Also, one must be aware of the names of methods of the superclass when inheriting from it, in order to decide whether to use extension or overriding. However, the type system warns the designer of the heir class about unintentional conflicts.

8. Conclusion

In this paper we have adapted and extended Wand’s typed model of classes and objects to an imperative setting, where the state of an object is a collection of mutable values. Our main achievement is the design of a type system which only accepts safe `let rec` declarations, while retaining the ability to construct a principal type for a typable term. The type reconstruction algorithm, as well as an interpreter of the language presented in this paper (including the `mixin` constructs), have been implemented by Pascal Zimmer. The first experiments he has made confirm that, since the equations on degree expressions that we have to handle are very easy to solve, the task of building a principal type is not more complicated than in the standard case.

We believe that our type system does not impose any new restriction on the underlying language, where recursion is limited to `(let rec x = N in M)` where N is a value: it should not be difficult to show that a term of this language is typable, without using degrees, if and only if it is typable, with the “same” type, in our system, thus showing that our typing is a conservative extension of the usual one, if we forget about degrees. We also believe that our system can be extended to include more types, since only the core functional fragment is concerned with the technicalities arising from the degrees. Another issue to investigate is whether our approach may be applied to other situations where programming with recursive non-functional values could help.

Acknowledgments

The implementation done by Pascal Zimmer was of invaluable help to me. In particular, it allowed me to realise that previous versions of the type system were not powerful enough to accept the object-oriented constructs I had in mind, and to type-check the examples presented in the paper.

References

- [1] M. ABADI, *Baby Modula-3 and a theory of objects*, J. of Functional Programming Vol. 4, No. 2 (1994) 249-283.
- [2] M. ABADI, L. CARDELLI, *A Theory of Objects*, Springer-Verlag (1996).
- [3] D. ANCONA, E. ZUCCA, *A theory of mixin modules: basic and derived operators*, Math. Struct. in Comput. Sci. Vol. 8 (1998) 401-446.
- [4] V. BONO, A. PATEL, V. SHMATIKOV, J. MITCHELL, *A core calculus of classes and objects*, MFPS’99, Electronic Notes in Comput. Sci. Vol. 20 (1999).
- [5] V. BONO, A. PATEL, V. SHMATIKOV, J. MITCHELL, *A core calculus of classes and mixins*, ECOOP’99, Lecture Notes in Comput. Sci. 1628 (1999) 43-66.
- [6] G. BOUDOL, *The recursive record semantics of objects revisited (extended abstract)*, ESOP 2001, Lecture Notes in Comput. Sci. 2028 (2001) 269-283.

- [7] G. BRACHA, W. COOK, *Mixin-based inheritance*, ECOOP/OOPSLA'90 (1990) 303-311.
- [8] G. BRACHA, *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*, PhD Thesis, The University of Utah (1992).
- [9] K. BRUCE, *Safe type checking in a statically-typed object-oriented programming language*, POPL'93 (1993) 285-298.
- [10] K. BRUCE, L. PETERSEN, A. FIECH, *Subtyping is not a good "match" for object-oriented languages*, ECOOP'97, Lecture Notes in Comput. Sci. 1241 (1997) 104-127.
- [11] L. CARDELLI, *A semantics of multiple inheritance*, Semantics of Data Types, Lecture Notes in Comput. Sci. 173 (1984) 51-67. Also published in Information and Computation, Vol. 76 (1988).
- [12] L. CARDELLI, J.C. MITCHELL, *Operations on records*, in [24], 295-350.
- [13] W. COOK, J. PALSBERG, *A denotational semantics of inheritance and its correctness*, OOPSLA'89, ACM SIGPLAN Notices Vol. 24 No. 10 (1989) 433-443.
- [14] W. COOK, W. HILL, P. CANNING, *Inheritance is not subtyping*, in [24], 497-517.
- [15] K. CRARY, R. HARPER, S. PURI, *What is a recursive module?*, PLDI'99 (1999) 50-63.
- [16] L. DAMAS, R. MILNER, *Principal type-schemes for functional programs*, POPL'82 (1982) 207-212.
- [17] J. EIFRIG, S. SMITH, V. TRIFONOV, A. ZWARICO, *An interpretation of typed OOP in a language with state*, LISP and Symbolic Computation Vol. 8 (1995) 357-397.
- [18] J. EIFRIG, S. SMITH, V. TRIFONOV, *Sound polymorphic type inference for objects*, OOPSLA'95, ACM SIGPLAN Notices Vol. 30 No. 10 (1995) 169-184.
- [19] K. FISHER, *Types Systems for Object-Oriented Programming Languages*, PhD Thesis, Stanford University (1996).
- [20] K. FISHER, F. HONSELL, J. MITCHELL, *A lambda calculus of objects and method specialization*, LICS'93 (1993) 26-38.
- [21] K. FISHER, J. MITCHELL, *The development of type systems for object-oriented languages*, Theory and Practice of Object Systems Vol. 1, No. 3 (1995) 189-220.
- [22] M. FLATT, S. KRISHNAMURTHI, M. FELLEISEN, *Classes and Mixins*, POPL'98 (1998) 171-183.
- [23] E. GAMMA, R. HELM, R. JOHNSON, J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1994).
- [24] C. GUNTER, J. MITCHELL (Eds.), *Theoretical Aspects of Object-Oriented Programming*, The MIT Press (1994).
- [25] R. HARPER, B. PIERCE, *A record calculus based on symmetric concatenation*, POPL'91 (1991) 131-142.
- [26] L.A. JATEGAONKAR, J. MITCHELL, *Type inference with extended pattern matching and subtypes*, Fundamenta Informaticae Vol. 19 (1993) 127-166.

- [27] J.-P. JOUANNAUD, C. KIRCHNER, *Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification*, in Computational Logic, Essays in Honor of A. Robinson (J.-L. Lassez, G. Plotkin, Eds.), The MIT Press (1991) 257-321.
- [28] S. KAMIN, *Inheritance in SMALLTALK-80: a denotational definition*, POPL'88 (1988) 80-87.
- [29] P.J. LANDIN, *The mechanical evaluation of expressions*, Computer Journal Vol. 6 (1964) 308-320.
- [30] X. LEROY, D. DOLIGEZ, J. GARRIGUE, D. RÉMY, J. VOUILLON, *The Objective Caml System*, release 3.00, Documentation and user's manual, available at <http://caml.inria.fr> (2000).
- [31] O.L. MADSEN, B. MØLLER PEDERSEN, *Virtual Classes: A powerful mechanism in object-oriented programming*, OOPSLA'89, ACM SIGPLAN Notices Vol. 24 No. 10 (1989) 397-406.
- [32] A. MARTELLI, U. MONTANARI, *An efficient unification algorithm*, ACM TOPLAS Vol. 4, No. 2 (1982) 258-282.
- [33] B. MEYER, *Genericity versus inheritance*, OOPSLA'86, ACM SIGPLAN Notices Vol. 21 No. 11 (1986) 391-405.
- [34] R. MILNER, *A theory of type polymorphism in programming*, J. of Computer and System Sciences Vol. 17 (1978) 348-375.
- [35] R. MILNER, M. TOFTE, R. HARPER, D. MACQUEEN, *The definition of Standard ML (Revised)*, The MIT Press (1997).
- [36] A. OHORI, P. BUNEMAN, *Static type inference for parametric classes*, in [24], 121-147.
- [37] J. PALSBERG, *Efficient inference of object types*, Information and Computation Vol. 123, No. 2 (1995) 198-209.
- [38] J. PALSBERG, T. JIM, *Type inference with simple selftypes is NP-complete*, Nordic Journal of Computing Vol. 4, No. 3 (1997) 259-286.
- [39] B.C. PIERCE, D. TURNER, *Simple type-theoretic foundations for object-oriented programming*, J. of Functional Programming Vol. 4 No. 2 (1994) 207-247.
- [40] U.S. REDDY, *Objects as closures: abstract semantics of object-oriented languages*, ACM Symp. on LISP and Functional Programming (1988) 289-297.
- [41] D. RÉMY, *Type inference for records in a natural extension of ML*, in [24], 67-95.
- [42] D. RÉMY, *Typing record concatenation for free*, in [24], 351-372.
- [43] D. RÉMY, *Programming with ML-ART: an extension to ML with abstract and record types*, TACS'94, Lecture Notes in Comput. Sci. 789 (1994) 321-346.
- [44] D. RÉMY, J. VOUILLON, *Objective ML: an effective object-oriented extension of ML*, Theory and practice of Objects Systems, Vol. 4, No. 1 (1998) 27-50.
- [45] J. REPPY, J. RIECKE, *Simple objects for Standard ML*, PLDI'96 (1996) 171-180.
- [46] A. SNYDER, *CommonObjects: an overview*, ACM SIGPLAN Notices Vol. 21 No. 10 (1986) 19-28.

- [47] A. SNYDER, *Encapsulation and inheritance in object-oriented programming languages*, OOP-SLA'86, ACM SIGPLAN Notices Vol. 21 No. 11 (1986) 38-45.
- [48] A. TAIVALSAARI, *On the notion of inheritance*, ACM Computing Surveys Vol. 28 No. 3 (1996) 438-479.
- [49] M. WAND, *Complete type inference for simple objects*, LICS'87 (1987) 37-44.
- [50] M. WAND, *Type inference for record concatenation and multiple inheritance*, Information and Computation Vol. 93, No. 1 (1991) 1-15.
- [51] M. WAND, *Type inference for objects with instance variables and inheritance*, in [24], 97-120.
- [52] A. WRIGHT, M. FELLEISEN, *A syntactic approach to type soundness*, Information and Computation Vol. 115 No. 1 (1994) 38-94.
- [53] A. WRIGHT, *Simple imperative polymorphism*, LISP and Symbolic Computation Vol. 8 (1995) 343-355.



Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)

Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
<http://www.inria.fr>
ISSN 0249-6399