A Record Calculus Based on Symmetric Concatenation

Robert Harper Benjamin Pierce

August 21, 1990
CMU-CS-90-157$_2$

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A number of different formulations of record calculi have been proposed. These may be broadly classified according to whether or not they are based on subsumption and whether they admit the specification of positive or negative information about record type variables. The systems considered by Wand (and their derivatives) may be classified as subsumption-free, negative information calculi. Early systems of bounded quantification are subsumption-based, and admit specification only of positive information. More recently, Cardelli and Mitchell have proposed a subsumption-based system featuring both positive and negative constraints, and, in earlier work, the authors have considered a subsumption-free variant. None of these systems copes well with the "symmetric merge" operation in which two records are considered mergeable only if they have no overlapping fields. To address this question, Cardelli and Mitchell suggest an extension to their subsumption-based calculus to admit mergeability constrains. In this paper we show that a subsumption-free calculus based only on mergeability constraints suffices not only for symmetric merge, but also for a wide class of other record operations. Moreover, the proposed calculus avoids some known difficulties with recursive types associated with positive-information calculi; in particular, the motivating examples for F-bounded quantification may be readily expressed without extending the language. A number of object-oriented features, such as those considered by Wand and Buneman and Ohori, may also be represented naturally in this calculus.

This research was supported in part by the Office of Naval Research and in part by the Defense Advanced Research Projects Agency (DOD), monitored by the Office of Naval Research under Contract N00014-84-K-0415, ARPA Order No. 5404.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ONR, DARPA or the U.S. government.

Keywords: Lambda calculus and related systems, Language theory, Programming, Type structure, Data types and structures, Records.

1 Introduction

Cardelli [2, 3] observed that certain aspects of *inheritance* in object-oriented languages can be understood in terms of inclusion relations among record types in a typed λ-calculus. These inclusions are defined formally as a *subtype* relation: a type $\mathtt{t}$ is a subtype of $\mathtt{t}'$, written $\mathtt{t} \leq \mathtt{t}'$, if any member of $\mathtt{t}$ may safely be used in a context where a member of $\mathtt{t}'$ is expected. The fact that the type of an expression may always be promoted to a supertype is captured by the rule of *subsumption*:

$$\frac{\mathtt{G} \vdash \mathtt{e} \in \mathtt{t} \qquad \mathtt{G} \vdash \mathtt{t} \leq \mathtt{t}'}{\mathtt{G} \vdash \mathtt{e} \in \mathtt{t}'} \qquad \text{(SUBSUMPTION)}$$

Cardelli and Wegner [6] extended this idea to a powerful second-order type system combining Cardelli's ordering on types with type quantification [9, 22], using techniques developed by Mitchell [16]. Wand [24, 25] analyzed the concept of record inclusions in the context of ML type inference and introduced the notion of "row variables," which allow types to be given to terms involving a natural *record extension* operator. This work was refined by Jategaonkar and Mitchell [12, 13] and Stansifer [23].

Rémy [19] introduced the notion of *positive and negative information* about record fields and the intuition that increasing either positive or negative information — specifying that fields are either definitely present or definitely absent — gives more refined types. This intuition, formalized as an appropriate extension to the kind system, plus the restriction that the set of field labels is finite, enabled him to use ordinary unification as in ML [8] to do type inference for programs involving extensible records. Both Wand [27] and Rémy [20, 21] later extended this system to infinite label sets.

More recently, Cardelli and Mitchell [4, 5] discovered an elegant calculus of primitive record operations combining bounded quantification with positive and negative information about fields and generalizing Cardelli's original subtype ordering on fixed-length records. In this system, the preorder on types is used to encode both positive and negative information. For example, record extensions like $\mathtt{e}|\mathtt{l}{=}\mathtt{e}'$ ("$\mathtt{e}$ extended with value $\mathtt{e}'$ at label $\mathtt{l}$") are only well formed when the field being added is not already present; to prevent run time type errors, the typing rule for | must ensure that this is the case. Cardelli and Mitchell express this constraint in terms of the preorder by requiring that "$\mathtt{e}$ has some type $\mathtt{r}$ that is a subtype of a type lacking $\mathtt{l}$." The *restriction* operator \ is used to increase negative information; for example, if $\mathtt{r}$ is the record type $\{\mathtt{l}_1{:}\mathtt{t}_1\}$, then $\mathtt{r}\backslash\mathtt{l}_2$ is a subtype of $\mathtt{r}$.

In an earlier report [10], we set out to represent positive and negative information about record types as directly as possible, using explicit constraints rather than encoding them with a preorder structure on types. For example, this system expresses the well-typedness constraint on $\mathtt{e}|\mathtt{l}{=}\mathtt{e}'$ as "$\mathtt{e}$ has type $\mathtt{r}$ and $\mathtt{r}$ lacks $\mathtt{l}$," where the judgement form "$\mathtt{r}$ lacks $\mathtt{l}$" is axiomatized explicitly. The result is a somewhat simpler but more verbose system with no preorder on types, where genericity over record types arises solely from *constrained quantification* of type variables: a quantified type "$\forall \mathtt{a}$ lacks $\mathtt{L}^-$ has $\mathtt{L}^+{:}\mathtt{T}^+.\,\mathtt{t}$" can be instantiated with any record type $\mathtt{r}$, so long as $\mathtt{r}$'s set of fields is disjoint from $\mathtt{L}^-$ and includes each $\mathtt{l}_i{:}\mathtt{t}_i$ in $\mathtt{L}^+{:}\mathtt{T}^+$. This line of development essentially amounts to reverse-engineering Cardelli and Mitchell's system back in the direction of Rémy's. In fact, during the early stages of their work Cardelli and Mitchell independently developed a similar system by extending the kind system of the polymorphic λ-calculus along the lines suggested by Rémy.

Other useful ways of manipulating record types are provided by the *merge* or *concatenate* operator ||, which combines the fields of two existing records. There are several different forms of the merge operator, distinguished by what happens when the records $\mathtt{e}_1$ and $\mathtt{e}_2$ in a merge expression $\mathtt{e}_1 \,||\, \mathtt{e}_2$ have one or more fields in common. The *asymmetric merge* operator gives preference to the values from $\mathtt{e}_2$. The *symmetric merge* operator disallows merge expressions where the fields of $\mathtt{e}_1$ and $\mathtt{e}_2$ are not guaranteed to be disjoint. The *recursive merge* computes the values of common fields by recursively merging their values in $\mathtt{e}_1$ and $\mathtt{e}_2$. Of these, the symmetric variant seems to us to be the most useful, since it makes the finest distinctions. This is the version we assume in the present paper, except where explicitly indicated.

A restricted form of merging can be defined directly. For example, $\{\mathtt{l}_1{:}\mathtt{t}_1, \mathtt{l}_2{:}\mathtt{t}_2\} \,||\, \{\mathtt{l}_3{:}\mathtt{t}_3, \mathtt{l}_4{:}\mathtt{t}_4\}$ can be rewritten simply as $\{\mathtt{l}_1{:}\mathtt{t}_1, \mathtt{l}_2{:}\mathtt{t}_2, \mathtt{l}_3{:}\mathtt{t}_3, \mathtt{l}_4{:}\mathtt{t}_4\}$. But in general — when the types of $\mathtt{e}_1$ and $\mathtt{e}_2$ may be variables, for example — typechecking with || requires a substantial increase in the complexity of the type

system. To ensure that $\lambda x{:}a.\ \texttt{lmbda}\ y : b.(x||y)$ is well typed, we need to guarantee that the type variables a and b are never instantiated to types with overlapping fields. This condition cannot be stated in terms of the previous forms of positive and negative information ("has" and "lacks" constraints), but must be provided explicitly by annotating type variables and quantifiers with *compatibility constraints*. Once this information is provided for type variables, we can define what it means for two arbitrary types r_1 and r_2 to be compatible, written $r_1 \# r_2$. The merge expression $e_1 \,||\, e_2$ is then well typed when $e_1 \in r_1$, $e_2 \in r_2$, and $r_1 \# r_2$. The type of this expression is the *merge type* $r_1 \,||\, r_2$.

Wand [27] studied type inference for an extension of ML with an asymmetric merge operator and showed how to encode class definitions similar to those found in object-oriented programming languages in this calculus. The asymmetric merge operation has the advantage that compatibility constraints are not necessary: any two records may be merged, with the rightmost one overriding. On the other hand, the use of symmetric merge allows the type checker to detect inadvertent clashes of labels, which can be useful in practice. Both systems have the property that the type checker must keep track of which component of a merge has a given field, which leads to problems in type reconstruction.

Ohori and Buneman studied type inference in for database programming languages [18], and for object-oriented programming [17]. In their work on database programming languages they consider a type system with records and sets, with operations (such as the relational join) chosen to model typical database applications. To support ML-like type inference for this language, they introduced the notion of a *conditional type scheme*, in which a type variable may be constrained to range over records possessing certain components. These constraints are similar to the "has" constraints discussed above. (They also used another form of constraint relevant only to the relational join and projection operations.) In their work on object-oriented programming, they consider a form of record update operation in which a value of a given field may be overridden. Once again, their notion of conditional type scheme plays a central role. They also considered extensions to support object-oriented programming constructs similar to those suggested by Wand. In order to support inheritance, they introduced an *ad hoc* form of subsumption in connection with "self." This introduced an additional form of condition on conditional type schemes; they exhibited a sound type inference algorithm for this extension.

Cardelli and Mitchell [5] sketched several increasingly ambitious formulations of recursive record concatenation as extensions to their calculus of operations on records. The most powerful of these requires the notion of "constrained, multiply bounded quantification," where a finite set of type variables is bound simultaneously, with each constrained to be a subtype of some given type and certain subsets constrained to be compatible. For example, a function that takes two compatible records x_1 and x_2, where x_1 has at least a field l_1 of type Int and x_2 has at least a field l_2 of type Int, and returns the result of merging x_1 and x_2, can be written (altering their concrete syntax slightly)

$$\forall(a_1 \le \{l_1 : \texttt{Int}\},\ a_2 \le \{l_2 : \texttt{Int}\},\ a_1 \# a_2).\ \lambda x_1{:}a_1.\ \lambda x_2{:}a_2.\ x_1 || x_2.$$

In the present paper, we study a record calculus $\lambda^{||}$ based on a much more straightforward formulation of constrained quantification. In $\lambda^{||}$, each record type variable in a context G has a list of *compatibility assumptions* R, where the elements of R are record types and a#R asserts that $a \# r_i$ for each $r_i \in R$. The constrained type abstraction operator $\Lambda a \# R.\ e$ adds the assumption a#R to the context used to typecheck e. A type application e[r] must check that r satisfies all of the constraints on the quantifier. For example, the function mentioned in the previous paragraph can be defined in $\lambda^{||}$ as follows:

$$\begin{array}{l}\forall a_1 \# l_1{:}\texttt{Int}.\ \forall a_2 \# a_1, l_2{:}\texttt{Int}.\\ \quad \lambda x_1{:}a_1 || l_1{:}\texttt{Int}.\ \lambda x_2{:}a_2 || l_2{:}\texttt{Int}.\\ \quad\quad x_1 \,||\, x_2.\end{array}$$

This example underscores an important point: the form of constraint used in $\lambda^{||}$ can only be used to express *negative* information about record type variables. The function above takes two type variables, each of which *lacks* the appropriate field. To form the types expected for the records x_1 and x_2 on the two λ-abstractions, the missing fields are merged back into a_1 and a_2. This kind of transformation from mixed positive and negative constraints on quantifiers to pure negative constraints can be carried out mechanically.

This has a very similar flavor to Wand's treatment of merging with row variables [27]. In fact, we adopt the same point of view as Wand with regard to subsumption: rather than introduce a preorder on types that includes record extension as a special case, we prefer to use a form of quantification to capture the

possible extensions of a record type and use type application to choose the appropriate extension for a given context. However, in contrast to Wand, we are dealing with an explicitly-typed, second-order calculus with a symmetric, rather than asymmetric, merge operator. This leads to a somewhat different overall flavor, as we shall illustrate below.

Our central claim in this paper is that the straightforward formulation of constrained quantification embodied in $\lambda^{||}$ may be viewed as *primary*, in the sense that most of the examples motivating row variables, bounded quantification, and Cardelli and Mitchell's bounded quantification with positive and negative information can be expressed in a calculus based on this form of constrained quantifier, with no need for additional mechanisms like subsumption. Similarly, in Section 3 we sketch an extension $\lambda^{||\mu}$ of $\lambda^{||}$ with recursive types and show that the examples motivating the extension from systems with bounded quantification and recursive types to systems with F-bounded quantification [1] can be expressed directly in $\lambda^{||\mu}$.

In Section 2 we define the syntax and typing rules of $\lambda^{||}$ and sketch a proof of the decidability of typechecking. Section 3 illustrates the expressiveness of the system through a number of examples. Section 4 describes some promising avenues for further research. Section 5 offers concluding remarks. A complete listing of the typing rules for $\lambda^{||}$ appears in Appendix A; complete rules for the algorithmic formulation used to prove decidability are given in Appendix B.

2 Definition and Properties of $\lambda^{||}$

2.1 Syntax

This section introduces the notational conventions used in the rest of the paper and defines the concrete syntax of $\lambda^{||}$.

The metavariables `q`, `r`, `s` and `t` range over types (`q`, `r`, and `s` are used when the type in question is expected to be a record type); `p` ranges over primitive types; `a` and `b` range over record type variables; `R` and `S` range over constraint lists; `e` and `f` range over terms; `x` ranges over variables; `l` ranges over field labels.

Types:

`t`	::=	`p`	primitive
	|	`a`	record type variable
	|	`Empty`	empty record
	|	`l:t`	single-field record
	|	`r\l`	restriction
	|	$\texttt{r}_1 \,||\, \texttt{r}_2$	merge
	|	$\texttt{t}_1 \rightarrow \texttt{t}_2$	function space
	|	`∀a#R. t`	constrained type quantification

The record types are those built up from record type variables, `Empty`, and single-field records by applications of merge and restriction. Ordinary type quantification is omitted from this presentation, but could be added by considering general type variables and an associated quantifier.

Constraint lists:

`R`	::=	$\diamond$	empty constraint list
	|	`r,R`	nonempty constraint list

Note that we make no provision here for including "bare labels" in constraint lists. Formally, expressions like `∀a#l. t` are taken as abbreviations for `∀a#(l:s). t`, where `s` is some dummy type. (See Section 3 for examples of this idiom.)

It is sometimes convenient to consider a list as a finite set and write, for example, $\texttt{r} \in \texttt{R}$.

Terms:

e	::=	x	variable
		λx:t. e	abstraction
		$e_1 e_2$	application
		empty	empty record
		l=e	single-field record
		e\l	restriction
		$e_1 \mid\mid e_2$	merge
		e.l	selection
		Λa#R. e	constrained type abstraction
		e[t]	constrained type application

Free and bound variables are defined in the usual way, with the proviso that in the case of type quantification and abstraction, the variable a is *not* considered bound in the constraint list R. Terms and types are identified up to renaming of bound variables. The notation [t/a]t′ denotes capture-avoiding substitution of t for free occurrences of a in t′; similarly, [e/x]e′ denotes capture-avoiding substitution of e for free occurrences of x in e′.

The metavariable T ranges over type contexts — finite sequences of declarations of the form a#R with no type variable declared twice. The metavariable G ranges over term contexts — finite sequences of declarations of the form x:t with no variable mentioned twice.

If l is a label and r is a well-formed record type, then r_l is defined to be the type associated with label l in r, if any. More precisely, the partial function r_l is defined by induction on the structure of r as follows. (We write r_l↑ for "r_l is undefined" and r_l↓ for "r_l is defined.")

(l:t)_l	=	t	
(r\l′)_l	=	r_l	when $l \neq l'$
$(r_1 \mid\mid r_2)$_l	=	r_1_l	if r_1_l↓ and r_2_l↑
$(r_1 \mid\mid r_2)$_l	=	r_2_l	if r_2_l↓ and r_1_l↑
r_l↑			otherwise.

2.2 Typing Rules

$\lambda^{\mid\mid}$ is a formal system for deriving judgements of the following forms:

Well-formed type context:	T ok
Well-formed term context:	T ⊢ G ok
Well-formed type:	T ⊢ t type
Well-formed record type:	T ⊢ r record
Well-formed constraint list:	T ⊢ R ok
Constraint list satisfaction:	T ⊢ r # R
Compatible types:	T ⊢ r_1 # r_2
Equivalent types:	$t_1 \sim t_2$
Equivalent constraint sets:	$R_1 \sim R_2$
Well-formed term:	T ; G ⊢ e ∈ t

The complete set of rules for $\lambda^{\mid\mid}$ appears in Appendix A. A representative selection of the rules appears in Figures 2 to 5. The rules are summarized in the remainder of this section.

The rules for context formation are the expected ones (see Appendix A). A type context T, a#R is well-formed if a is not already declared in T and R is well-formed in T. A term context G, x:t is well-formed in T if x is not already declared in G and if t is a well-formed type in T.

A constraint list R is well formed in T iff each component type expression r in R is a well-formed record type in T.

The rules for formation of primitive types and function types are as usual. A quantified type ∀a#R. t is well-formed in T if R is a well-formed constraint list in T, and t is a well-formed type in T, r#R (rule WFT-ALL). Every record type is a typerule WFT-REC).

The formation rules for record types are summarized in Figure 1. A record type variable is a record type in T provided that it is declared in T (rule WFR-VAR). The empty record is always well-formed (rule

$$\frac{\mathtt{T_1, a\#R, T_2\ ok}}{\mathtt{T_1, a\#R, T_2 \vdash a\ record}} \qquad \text{(WFR-VAR)}$$

$$\frac{\mathtt{T \vdash t\ type}}{\mathtt{T \vdash 1{:}t\ record}} \qquad \text{(WFR-BASE)}$$

$$\frac{\mathtt{T \vdash r\ record} \qquad \mathtt{r\_1\downarrow}}{\mathtt{T \vdash r\backslash 1\ record}} \qquad \text{(WFR-RESTR)}$$

$$\frac{\mathtt{T\ ok}}{\mathtt{T \vdash Empty\ record}} \qquad \text{(WFR-EMPTY)}$$

$$\frac{\mathtt{T \vdash r_1 \# r_2}}{\mathtt{T \vdash r_1 \,||\, r_2\ record}} \qquad \text{(WFR-MERGE)}$$

Figure 1: Selected well-formedness rules for types

$$\frac{\mathtt{T;G \vdash e \in t} \qquad \mathtt{T \vdash t'\ type} \qquad \mathtt{t \sim t'}}{\mathtt{T;G \vdash e \in t'}} \qquad \text{(WTT-EQ)}$$

$$\frac{\mathtt{T, a\#R;\Gamma \vdash e \in t}}{\mathtt{T;G \vdash \Lambda a\#R.e \in \forall a\#R.\ t}} \qquad \text{(WTT-ALL-I)}$$

$$\frac{\mathtt{T;G \vdash e \in \forall a\#R.\ t} \qquad \mathtt{T \vdash r \# R}}{\mathtt{T;G \vdash e[r] \in [r/a]t}} \qquad \text{(WTT-ALL-E)}$$

Figure 2: Selected typing rules

WFR-EMPTY). The single-field record `1:t` is well-formed in `T` if `t` is a well-formed type in `T` (rule WFR-BASE). The restriction `r\1` is well formed in `T` if `r` is well formed in `T` and `r_1` is defined (rule WFR-RESTR). This means, in particular, that no expression of the form `a\1` where `a` is a variable is well formed, since `a_1` is never defined. This reflects the fact that compatibility constraints on a variable `a` are negative in character and cannot be used to postulate that all instances of `a` *have* any particular fields, only that they *lack* certain fields. This also implies that if `r\1` is a well-formed record expression, then the restriction may be eliminated (see below for a discussion of type equivalence). Finally, a merge $\mathtt{r_1}||\mathtt{r_2}$ is well-formed in `T` if $\mathtt{r_1}$ and $\mathtt{r_2}$ are well-formed in `T`, and, morover, $\mathtt{r_1}$ and $\mathtt{r_2}$ are compatible in `T`. Informally, this means that $\mathtt{r_1}$ lacks whatever fields $\mathtt{r_2}$ has and vice versa, so that the merge is well-defined. It is important to realize that the compatibility relation must be axiomatized relative to a context since, in the presence of constrained quantification, it is not possible to determine from the syntactic form of record expressions alone whether they are compatible. (See below for further discussion of the compatibility relation.)

The typing rules for $\lambda^{||}$ terms are organized along standard lines. (See Figure 2.) Typing is invariant under type equality (rule WTT-EQ). The rules for variables, lambda abstractions, and application are standard. The empty record is always well-formed. A single-field record `1=e` has type `1:t` in `T` if `e` has type `t` in `T`. The restriction `e\1` has type `r\1` in `T` provided that `e` has type `r` in `T` and `r_1`$\downarrow$: we may restrict only on a field that `e` actually possesses. The merge $\mathtt{e_1}||\mathtt{e_2}$ has type $\mathtt{r_1}||\mathtt{r_2}$ in `T` provided that $\mathtt{e_1}$ has type $\mathtt{r_1}$ in `T` and $\mathtt{e_2}$ has type $\mathtt{r_2}$ in `T` and $\mathtt{r_1}$ and $\mathtt{r_2}$ are compatible in `T`. In other words, we may not merge two records unless they are non-overlapping. To achieve the effect of overriding, one must restrict on the fields to be overridden before forming the merge. The selection `e.1` has type `r_1` in `T` if `e` has type `r` in `T` and `r_1`$\downarrow$. By the definition of `r_1`, the type of `e.1` is unique if it well-formed.

The abstraction `Λa#R. e` has type `∀a#R. t` provided that `e` has type `t` in `T, a#R`, which also entails that `R` is a well-formed constraint set in `T` (rule WTT-ALL-I). It is important to realize that the constraint list

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{s} \qquad \mathtt{r} \sim \mathtt{r}' \qquad \mathtt{s} \sim \mathtt{s}'}{\mathtt{T} \vdash \mathtt{r}' \# \mathtt{s}'} \qquad \text{(CMP-EQ)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{s}}{\mathtt{T} \vdash \mathtt{s} \# \mathtt{r}} \qquad \text{(CMP-SYMM)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{l{:}t} \qquad \mathtt{T} \vdash \mathtt{t}' \text{ type}}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{l{:}t}'} \qquad \text{(CMP-BASE)}$$

$$\frac{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2 \text{ ok} \qquad \mathtt{r}_i \in \mathtt{R}}{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2 \vdash \mathtt{a} \# \mathtt{r}_i} \qquad \text{(CMP-TVAR)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \# (\mathtt{s1}||\mathtt{s2})}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{si}} \qquad \text{(CMP-MERGE-E)}$$

$$\frac{\mathtt{T} \vdash \mathtt{s1} \# \mathtt{s2} \qquad \mathtt{T} \vdash \mathtt{r} \# \mathtt{s1} \qquad \mathtt{T} \vdash \mathtt{r} \# \mathtt{s2}}{\mathtt{T} \vdash \mathtt{r} \# (\mathtt{s1}||\mathtt{s2})} \qquad \text{(CMP-MERGE-I)}$$

$$\frac{\mathtt{l} \neq \mathtt{l}' \qquad \mathtt{T} \vdash \mathtt{l{:}t} \text{ record} \qquad \mathtt{T} \vdash \mathtt{l}'\mathtt{:t}' \text{ record}}{\mathtt{T} \vdash \mathtt{l{:}t} \# \mathtt{l}'\mathtt{:t}'} \qquad \text{(CMP-BASE/BASE)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \text{ record}}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{Empty}} \qquad \text{(CMP-EMPTY)}$$

Figure 3: Selected compatibility rules

R cannot be replaced by a single record type r, for two related reasons. First, it is necessary to postulate that a variable be compatible with a number of different record types. For example, if we are to merge a variable a with the base records l:t and l′:t′, then a must be constrained to be compatible with both of these records, which is to say that all instances of a must not contain l or l′ fields. Second, the constraint list R cannot be collapsed into a single record type consisting of the merge of the component record types r of R because the records in R need not be mutually compatible. In particular, we may wish to postulate that a variable a is compatible with two other variables b and c so that the merges a||b and a||c are both well-formed. This in no way entails that b and c must be compatible, and it would be a serious restriction to require that they be so.

The type application e[r] is well formed in T if e has type ∀a#R. t in T, r is a record in T, and r is compatible with each record type in R (relative to T). In other words, r must satisfy the constraints associated with the quantifier in order for the type application to be sensible. When this is the case, the type of e[r] is as expected, namely [r/a]t (rule WTT-ALL-E). It will turn out that the formulation of the system ensures that if r satisfies the constraints in R relative to T, then r is a record type in T. To support general parametric polymorphism, we would have to extend the system with a separate form of quantifier that quantifies over all types. We omit this extension for the sake of simplicity.

We now turn to the definition of compatibility. Informally, T ⊢ r # s holds iff every instance r′ and s′ of r and s obtained by substituting record types for type variables consistently with the constraints in T is such that whatever fields r′ has, s′ lacks, and conversely, whatever fields s′ has, r′ lacks. (The notion of "consistently with the constraints in T" itself makes use of the notion of compatibility, but for terms with fewer type variables.) This description raises several important points. First, the definition of compatibility is relative to a type context T. Second, what matters about r′ and s′ is the singleton records that occur within them, and, moreover, it is only the *labels*, not the types of these singletons that matter for compatibility. In other words l:t is compatible with l′:t′ provided only that l is different from l′, and conversely l:t is incompatible with l:t′ even if t and t′ are identical. Third, the definition of compatibility should respect type equivalence: equal types (as we will see below) have the same fields.

$$\mathtt{r}||\mathtt{Empty} \sim \mathtt{r} \qquad \text{(Eq-Merge-Unit)}$$

$$\mathtt{r}_1 \,||\, (\mathtt{r}_2||\mathtt{r}_3) \sim (\mathtt{r}_1||\mathtt{r}_2) \,||\, \mathtt{r}_3 \qquad \text{(Eq-Merge-Assoc)}$$

$$\mathtt{r}_1 \,||\, \mathtt{r}_2 \sim \mathtt{r}_2 \,||\, \mathtt{r}_1 \qquad \text{(Eq-Merge-Comm)}$$

$$\mathtt{r}\backslash\mathtt{l}\backslash\mathtt{l}' \sim \mathtt{r}\backslash\mathtt{l}'\backslash\mathtt{l} \qquad \text{(Eq-Restr/Restr')}$$

$$(\mathtt{l}{:}\mathtt{t})\backslash\mathtt{l} \sim \mathtt{Empty} \qquad \text{(Eq-Base/Restr)}$$

$$\frac{\mathtt{r}_1 \,.\, \mathtt{l}\downarrow}{(\mathtt{r}_1||\mathtt{r}_2)\backslash\mathtt{l} \sim (\mathtt{r}_1\backslash\mathtt{l}||\mathtt{r}_2)} \qquad \text{(Eq-Merge/Restr)}$$

$$\frac{\mathtt{R} \sim \mathtt{R}' \qquad \mathtt{t} \sim \mathtt{t}'}{\forall \mathtt{a}\#\mathtt{R}.\ \mathtt{t} \sim \forall \mathtt{a}\#\mathtt{R}'.\ \mathtt{t}'} \qquad \text{(Eq-Cong-All)}$$

Figure 4: Selected type equivalence rules

$$\mathtt{r}, (\mathtt{r}', \mathtt{R}) \sim \mathtt{r}', (\mathtt{r}, \mathtt{R}) \qquad \text{(Ceq-Swap)}$$

$$\mathtt{Empty}, \mathtt{R} \sim \mathtt{R} \qquad \text{(Ceq-Empty)}$$

$$(\mathtt{r}_1||\mathtt{r}_2), \mathtt{R} \sim \mathtt{r}_1, (\mathtt{r}_2, \mathtt{R}) \qquad \text{(Ceq-Merge)}$$

$$\mathtt{r}, (\mathtt{r}, \mathtt{R}) \sim \mathtt{r}, \mathtt{R} \qquad \text{(Ceq-Dupl)}$$

$$\mathtt{l}{:}\mathtt{t}, \mathtt{R} \sim \mathtt{l}{:}\mathtt{t}', \mathtt{R} \qquad \text{(Ceq-Base)}$$

Figure 5: Selected constraint list equivalence rules

These observations lead to the following formalization of the compatibility relation (see Figure 3). Compatibility is symmetric (rule Cmp-Symm) and respects type equivalence (rule Cmp-Eq). If a record is compatible with a single-field record $\mathtt{l}{:}\mathtt{t}$, then it is compatible with every single-field record $\mathtt{l}{:}\mathtt{t}'$ (rule Cmp-Base). A type variable is compatible with the records in its constraint list (rule Cmp-Tvar). A record is compatible with a merge iff it is compatible with each of the components of the merge (rules Cmp-Merge-I and Cmp-Merge-E). Two single-field records are compatible if they have different labels (rule Cmp-Base/Base). Every record is compatible with the empty record (rule Cmp-Empty). Finally, a record type satisfies a constraint list iff it is compatible with every element of the list.

It is worth noting that there are no rules for compatibility of restrictions. It will be possible to prove that every well-formed record is equivalent to a unique restriction-free record; hence compatibility of restrictions is covered by the general rule of invariance of compatibility under equivalence (rule Cmp-Eq).

Finally, we consider equivalence of types and constraint sets. (A selection of the equivalence rules appears in Figures 4 and 5.) We begin with type equivalence. The relation $\mathtt{t} \sim \mathtt{t}'$ is an equivalence relation and is a congruence with respect to the type-forming operations. The axioms for record type equivalence are as follows. Restrictions commute with one another (rule Eq-Restr/Restr') (in a well-formed expression $\mathtt{l}$ will be distinct from $\mathtt{l}'$, but there is no reason to insist on that in the rule.) Restriction of a singleton $\mathtt{l}{:}\mathtt{t}$ on

`l` results in the empty record type (rule EQ-BASE-RESTR). Restriction distributes to the appropriate (given EQ-SYMM and EQ-CONG-RESTR) component of a merge (rule EQ-MERGE-RESTR). Merge is associative (rule EQ-MERGE-ASSOC), `Empty` is a unit of merge (rule EQ-MERGE-UNIT), and merge is commutative (rule EQ-MERGE-COMM). This last rule is a reflection of the fact that in a well-formed merge there can be no overlapping fields.

The presence of the constrained quantifier requires that we say something about equivalence of constraint lists (rule EQ-CONG-ALL). Here the guiding principle is that equivalent constraint sets should be satisfied by the same record types. In addition to being an equivalence relation and compatible with the "cons" operation on constraint lists, we require that order and multiplicity be irrelevant (rules CEQ-SWAP and CEQ-DUPL) so that constraint lists are essentially finite sets. We may also "flatten" merges (rule CEQ-MERGE), eliminate `Empty` (rule CEQ-EMPTY), and ignore the types of single-field records (rule CEQ-BASE). As we will see below, these equivalences are sufficient to ensure that constraint sets may be normalized into the form $\mathtt{l}_1, ..., \mathtt{l}_n, \mathtt{a}_1, ..., \mathtt{a}_k$ where the $\mathtt{l}_i$'s are labels and the $\mathtt{a}_i$'s are record type variables. This is consistent with the intuition that the relevant properties of a record, for the purposes of consistency checking, are its atomic components. Note that it will not do to simply *define* constraint sets this way: substitutions of record types for record type variables can disturb this form by instantiating variables to complex record types. The constraint set equivalence rules are needed to restore this simple form.

It should be stressed that there is considerable leeway in the choice of type and constraint list equivalence. The choices made above are guided by the intended interpretation of record types, and by algorithmic considerations (we impose enough equivalences to achieve well-behaved normal forms.) Since we postulate the invariance of typing under type equivalence, the class of well-typed terms increases as more types are identified. It is therefore desirable to impose whatever equivalences may be consistently added to the system, constrained only by informal (at this stage) considerations about models and implementations of the calculus.

This completes our summary of $\lambda^{||}$.

2.3 Properties

In this section we establish the decidability of type checking for $\lambda^{||}$.

First, we prove that equivalence of well-formed types and constraint lists is decidable. The main complication is due to the associative and commutative axioms for record type equality and the permutation and duplicate-elimination axioms for constraint lists. However, these may be handled by applying Huet's method of "confluence modulo an equivalence relation" [11]. The idea is to segregate the "proper" reduction axioms from the "pure" equational axioms and to show that the full equivalence relation may be characterizes as "pure" equivalence of normal forms. A byproduct of this development is a useful characterization of normal forms.

Second, we consider the decidability of the compatibility relation for well-formed, restriction-free record types. The main complication here arises from the rules of symmetry, invariance under change of single-field record type, and respect for equivalence. The algorithm is presented as an "almost syntax-directed" set of inference rules that are readily shown to be decidable, and that may be proved equivalent (in a suitable sense) to the full compatibility relation. The algorithm relies on the characterization of normal forms mentioned above, and is therefore sensitive to the definition of equivalence of record types and constraint sets.

Finally, we give a type synthesis algorithm that computes, given a context and a term, a representative of the equivalence class of possible types for that term. This algorithm proceeds along standard lines, making use of the algorithms for checking equivalence and compatibility.

2.3.1 Technical Lemmas

Lemma 2.3.1.1: A derivation of the well-formedness of a type, record type, or constraint list includes subderivations of the well-formedness of all of its subphrases.

Lemma 2.3.1.2:

1. $\mathtt{a} \backslash \mathtt{l}$ is never well formed for any variable $\mathtt{a}$, regardless of context.

2. `Empty\l` is never well-formed.

3. `r\l\l'` is well-formed only if $\mathtt{l} \neq \mathtt{l}'$ and `r` is not a single-field record.

4. `l:t\l'` is well-formed only if $\mathtt{l} = \mathtt{l}'$.

Lemma 2.3.1.3: If `r_l` is defined and $\mathtt{r} \sim \mathtt{s}$, then `s_l` is defined and $\mathtt{r\_l} \sim \mathtt{s\_l}$.

Lemma 2.3.1.4: Assume that $\mathtt{T} \vdash \mathtt{r} \# \mathtt{s}$. Then it is not the case that both `r_l` and `s_l` are defined.

Corollary 2.3.1.5: If $\mathtt{r}||\mathtt{s}$ is well formed, then `r_l` defined implies `s_l` undefined, and vice versa.

Lemma 2.3.1.6: $\vdash \mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2$ ok implies $\mathtt{T}_1 \vdash \mathtt{R}$ ok as a subderivation.

Lemma 2.3.1.7: $\mathtt{T} \vdash \mathtt{r} \# \mathtt{s}$ implies $\mathtt{T} \vdash \mathtt{r}$ record and $\mathtt{T} \vdash \mathtt{s}$ record.

Lemma 2.3.1.8: If $\mathtt{T} \vdash \mathtt{r} \# \mathtt{R}$ and $\mathtt{r} \sim \mathtt{r}'$ and $\mathtt{R} \sim \mathtt{R}'$, then $\mathtt{T} \vdash \mathtt{r}' \# \mathtt{R}'$.

Lemma 2.3.1.9: Let `J` stand for an arbitrary judgement. If $\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2 \vdash \mathtt{J}$, and $\mathtt{T}_1 \vdash \mathtt{r}\#\mathtt{R}$, then $\mathtt{T}_1, [\mathtt{r}/\mathtt{a}]\mathtt{T}_2 \vdash [\mathtt{r}/\mathtt{a}]\mathtt{J}$.

2.3.2 Confluence of reduction modulo an equivalence relation

Our proof of the confluence of type and constraint list normalization uses a technique due to Huet [11]. In this section we briefly recapitulate Huet's main theorem.

Definition 2.3.2.1: The metavariables `M`, `N`, `P`, `Q`, and `U` here range over an unspecified set of phrases (in the next section they will be either types or constraint lists).

Definition 2.3.2.2: $\mathcal{V}(\mathtt{M})$ is the set of variables appearing free in a phrase `M`.

Definition 2.3.2.3: A phrase `M` is linear iff all of its variable occurrences are distinct.

Definition 2.3.2.4: A rewrite rule is a pair of terms.

Definition 2.3.2.5: A critical pair $\langle \mathtt{P}, \mathtt{Q} \rangle$ for two rewriting rules $\langle \mathtt{M}_1, \mathtt{N}_1 \rangle$ and $\langle \mathtt{M}_2, \mathtt{N}_2 \rangle$ is a most general common instance of the two rules, found by superposing $\mathtt{M}_2$ on a non-variable subphrase `U` of $\mathtt{M}_1$. `P` is the result of rewriting the subphrase `U` of the common instance according to the rule $\langle \mathtt{M}_2, \mathtt{N}_2 \rangle$. `Q` is the result of rewriting the whole common instance according to the rule $\langle \mathtt{M}_1, \mathtt{N}_1 \rangle$.

Definition 2.3.2.6: An equational theory $(\mathcal{R}, \mathcal{E})$ consists of two sets of rewriting rules: a "reduction part" $\mathcal{R}$ and an "equivalence part" $\mathcal{E}$.

Definition 2.3.2.7: Let $(\mathcal{R}, \mathcal{E})$ be an equational theory. The one-step reduction relation induced by $\mathcal{R}$ is denoted by $\twoheadrightarrow^1_{\mathcal{R}}$, and the one-step reduction relation induced by $\mathcal{E}$ is denoted by $\approx^1_{\mathcal{E}}$. The reflexive, transitive closure of $\twoheadrightarrow^1_{\mathcal{R}}$ is denoted by $\twoheadrightarrow_{\mathcal{R}}$, and the reflexive, symmetric, and transitive closure of $\approx^1_{\mathcal{E}}$ is denoted by $\approx_{\mathcal{E}}$.

The subscripts are omitted when they are clear from context.

Definition 2.3.2.8: The set of critical pairs of a reduction relation $\mathcal{R}$ is

$$\{\langle \mathtt{P}, \mathtt{Q} \rangle \mid \langle \mathtt{P}, \mathtt{Q} \rangle \text{ is a critical pair for } \langle \mathtt{M}_1, \mathtt{M}_2 \rangle, \langle \mathtt{N}_1, \mathtt{N}_2 \rangle \in \mathcal{R}\}.$$

Definition 2.3.2.9: Let $\langle \mathcal{R}, \mathcal{E} \rangle$ be an equational theory. The set of critical pairs of $\mathcal{E}/\mathcal{R}$ is

$$\begin{array}{ll} & \{\langle \mathtt{P}, \mathtt{Q} \rangle \mid \langle \mathtt{P}, \mathtt{Q} \rangle \text{ is a critical pair for } \langle \mathtt{M}_1, \mathtt{M}_2 \rangle \in \mathcal{R} \text{ and } \langle \mathtt{N}_1, \mathtt{N}_2 \rangle \in (\mathcal{E} \cup \mathcal{E}^{-1})\} \\ \cup & \{\langle \mathtt{P}, \mathtt{Q} \rangle \mid \langle \mathtt{P}, \mathtt{Q} \rangle \text{ is a critical pair for } \langle \mathtt{M}_1, \mathtt{M}_2 \rangle \in (\mathcal{E} \cup \mathcal{E}^{-1}) \text{ and } \langle \mathtt{N}_1, \mathtt{N}_2 \rangle \in \mathcal{R}\}, \end{array}$$

where

$$\mathcal{E}^{-1} = \{\langle \mathtt{N}, \mathtt{M} \rangle \mid \langle \mathtt{M}, \mathtt{N} \rangle \in \mathcal{E}\}.$$

Definition 2.3.2.10: The set of critical pairs of a theory $\langle \mathcal{R}, \mathcal{E} \rangle$ is the union of the set of critical pairs of $\mathcal{R}$ and the set of critical pairs of $\mathcal{E}/\mathcal{R}$.

Definition 2.3.2.11: $\mathtt{M}^*$ denotes an arbitrary $\mathcal{R}$-normal form of $\mathtt{M}$.

Proposition 2.3.2.12: (Huet's Theorem 3.3) Let $\langle \mathcal{R}, \mathcal{E} \rangle$ be an equational theory such that

1. for all $\langle \mathtt{M}\ ,\ \mathtt{N} \rangle \in \mathcal{R}$, $\mathcal{V}(\mathtt{N}) \subseteq \mathcal{V}(\mathtt{M})$ and $\mathtt{M}$ is linear;
2. for all $\langle \mathtt{M}\ ,\ \mathtt{N} \rangle \in \mathcal{E}$, $\mathcal{V}(\mathtt{M}) = \mathcal{V}(\mathtt{N})$;
3. $\rightarrow^1 \circ \approx$ is noetherian, where $\approx$ is the reflexive, symmetric, transitive closure of $\approx^1$.

The theory $\langle \mathcal{R}, \mathcal{E} \rangle$ is confluent iff for all its critical pairs $\langle \mathtt{P}, \mathtt{Q} \rangle$ we have $\mathtt{P}^* \approx \mathtt{Q}^*$. When $\langle \mathcal{R}, \mathcal{E} \rangle$ is confluent, we say that "$\mathcal{R}$ is confluent modulo $\mathcal{E}$." Then $\mathtt{M} =_{\mathcal{R} \cup \mathcal{E}} \mathtt{N}$ iff $\mathtt{M}^* \approx \mathtt{N}^*$, where $=_{\mathcal{R} \cup \mathcal{E}}$ is the equivalence relation induced by $\mathcal{R}$ and $\mathcal{E}$.

2.3.3 Equivalence of Types and Constraint Lists

We now apply Huet's theory to the confluence of a type normalization procedure for $\lambda^{||}$.

Definition 2.3.3.1: For the algorithmic version of the system, constraint lists are extended with "bare labels":

R	::=	◇	empty constraint list
		| r,R	constraint list beginning with a type
		| l,R	constraint list beginning with a bare label

We extend the relation $\mathtt{T} \vdash \mathtt{r} \# \mathtt{R}$ to constraint lists by defining $\mathtt{T} \vdash \mathtt{r} \# \mathtt{l},\mathtt{R}$ to mean $\mathtt{T} \vdash \mathtt{r} \# \mathtt{R}$ and $\mathtt{T} \vdash \mathtt{r} \# \mathtt{l}{:}\mathtt{t}$ for any type $\mathtt{t}$.

Definition 2.3.3.2: In this section, it is convenient to be able to refer to phrases from certain related syntactic categories uniformly:

1. The metavariable α ranges over atoms: variables, base records, and bare labels.
2. The metavariables ρ, σ, and τ range over types and bare labels.
3. The metavariable ϕ ranges over type variables and bare labels.
4. The metavariable $\mathtt{C}$ ranges over types and constraint lists.

Definition 2.3.3.3: Let $\twoheadrightarrow^1$ denote the following reduction relation on record types and constraint lists, compatibly extended to all type expressions:

$$\texttt{Empty} \,||\, \texttt{r} \twoheadrightarrow^1 \texttt{r} \qquad \text{(A-RED-EMPTY-1)}$$

$$\texttt{r} \,||\, \texttt{Empty} \twoheadrightarrow^1 \texttt{r} \qquad \text{(A-RED-EMPTY-2)}$$

$$\texttt{l:t}\backslash\texttt{l} \twoheadrightarrow^1 \texttt{Empty} \qquad \text{(A-RED-RESTR/BASE)}$$

$$(\texttt{r} \,||\, \texttt{s})\backslash\texttt{l} \twoheadrightarrow^1 (\texttt{r}\backslash\texttt{l}) \,||\, \texttt{s} \quad \text{if } \texttt{r\_l}\downarrow \qquad \text{(A-RED-RESTR/MERGE-1)}$$

$$(\texttt{r} \,||\, \texttt{s})\backslash\texttt{l} \twoheadrightarrow^1 \texttt{r} \,||\, (\texttt{s}\backslash\texttt{l}) \quad \text{if } \texttt{s\_l}\downarrow \qquad \text{(A-RED-RESTR/MERGE-2)}$$

$$\rho||\sigma,\texttt{R} \twoheadrightarrow^1 \rho,\sigma,\texttt{R} \qquad \text{(A-CRED-MERGE)}$$

$$\texttt{Empty},\texttt{R} \twoheadrightarrow^1 \texttt{R} \qquad \text{(A-CRED-EMPTY)}$$

$$\texttt{l:t},\texttt{R} \twoheadrightarrow^1 \texttt{l},\texttt{R} \qquad \text{(A-CRED-BASE)}$$

Definition 2.3.3.4: Let $\approx^1$ denote the following reduction relation on record types and constraint lists, compatibly extended to all type expressions:

$$\texttt{r}\backslash\texttt{l}\backslash\texttt{l}' \approx^1 \texttt{r}\backslash\texttt{l}'\backslash\texttt{l} \qquad \text{(A-EQ-RESTR/RESTR')}$$

$$\texttt{r} \,||\, \texttt{s} \approx^1 \texttt{s} \,||\, \texttt{r} \qquad \text{(A-EQ-COMM)}$$

$$\texttt{r} \,||\, (\texttt{s} \,||\, \texttt{t}) \approx^1 (\texttt{r} \,||\, \texttt{s}) \,||\, \texttt{t} \qquad \text{(A-EQ-ASSOC)}$$

$$\phi,\phi,\texttt{R} \approx^1 \phi,\texttt{R} \qquad \text{(A-CEQ-DUPL)}$$

$$\rho,\sigma,\texttt{R} \approx^1 \sigma,\rho,\texttt{R} \qquad \text{(A-CEQ-COMM)}$$

Let $\approx$ denote the reflexive, symmetric, transitive closure of $\approx^1$, that is, the equivalence relation generated by $\approx^1$.

Let $\equiv$ denote the reflexive, symmetric, transitive closure of $\twoheadrightarrow^1 \cup \approx^1$, that is, the equivalence relation generated by $\twoheadrightarrow^1$ and $\approx^1$.

Theorem 2.3.3.5:

1. If $\texttt{T} \vdash \texttt{r}$ record and $\texttt{r} \twoheadrightarrow^1 \texttt{r}'$, then $\texttt{T} \vdash \texttt{r}'$ record.
2. If $\texttt{T} \vdash \texttt{t}$ type and $\texttt{t} \twoheadrightarrow^1 \texttt{t}'$, then $\texttt{T} \vdash \texttt{t}'$ type.
3. If $\texttt{T} \vdash \texttt{R}$ ok, and $\texttt{R} \twoheadrightarrow^1 \texttt{R}'$, then $\texttt{T} \vdash \texttt{R}'$ ok.

Proof: By induction on derivations, making use of the technical lemmas in Section 2.3.1. □

Definition 2.3.3.6: The "outer size" of a type r is defined as follows:

$$\begin{array}{lcl}
\text{outer-size}(\mathtt{a}) & = & 1 \\
\text{outer-size}(\mathtt{Empty}) & = & 1 \\
\text{outer-size}(\mathtt{l{:}t}) & = & 1 \\
\text{outer-size}(\mathtt{r}\backslash\mathtt{l}) & = & \text{outer-size}(\mathtt{r}) + 1 \\
\text{outer-size}(\mathtt{r}_1||\mathtt{r}_2) & = & \text{outer-size}(\mathtt{r}_1) + \text{outer-size}(\mathtt{r}_2) + 1 \\
\text{outer-size}(\mathtt{p}) & = & 0 \\
\text{outer-size}(\mathtt{t}_1 \rightarrow \mathtt{t}_2) & = & 0 \\
\text{outer-size}(\forall \mathtt{a}\#\mathtt{R}.\ \mathtt{t}) & = & 0.
\end{array}$$

Definition 2.3.3.7: The rank of a constraint list is defined as follows:

$$\begin{array}{lcl}
\text{rank}(\diamond) & = & 0 \\
\text{rank}(\mathtt{l}, \mathtt{R}) & = & \text{rank}(\mathtt{R}) \\
\text{rank}(\mathtt{r}, \mathtt{R}) & = & \text{rank}(\mathtt{r}) + \text{rank}(\mathtt{R}).
\end{array}$$

The rank of a type is the sum

$$\text{rank}(\mathtt{t}) = \text{restr-count}(\mathtt{t}) + \text{base-count}(\mathtt{t}) + \text{empty-count}(\mathtt{t}) + \text{merge-count}(\mathtt{t}),$$

where

- restr-count(t) is the sum, over all occurrences of the form $\mathtt{s}\backslash\mathtt{l}$ in t, of outer-size(s),
- base-count(t) is the number of base record phrases $\mathtt{l{:}t'}$ in t,
- empty-count(t) is the number of occurrences of **Empty** in t, and
- merge-count(t) is the number of occurrences of || in t.

Lemma 2.3.3.8: The relation $\twoheadrightarrow^1 \circ \approx$ is Noetherian when restricted to well-formed expressions. That is, there are no infinite reduction sequences of the form

$$\cdot \twoheadrightarrow^1 \cdot \approx \cdot \twoheadrightarrow^1 \cdot \approx \cdot \twoheadrightarrow^1 \cdots$$

where each of the terms in the sequence is well-formed.

Proof: Applying any of the $\twoheadrightarrow^1$ rules to a type or constraint list strictly reduces its rank, while applying any of the $\approx^1$ rules in either direction leaves the rank unchanged. □

Lemma 2.3.3.9:

1. If $\mathtt{T} \vdash (\mathtt{r}\backslash\mathtt{l} \,||\, \mathtt{s})\backslash\mathtt{l}'$ record and $\mathtt{T} \vdash (\mathtt{r}||\mathtt{s})\backslash\mathtt{l}\backslash\mathtt{l}'$ record, then $((\mathtt{r}\backslash\mathtt{l} \,||\, \mathtt{s})\backslash\mathtt{l}')^* \approx ((\mathtt{r}||\mathtt{s})\backslash\mathtt{l}'\backslash\mathtt{l})^*$.
2. If $\mathtt{T} \vdash \mathtt{a}, \mathtt{b}, (\mathtt{a}||\mathtt{b}), \mathtt{R}$ ok and $\mathtt{T} \vdash (\mathtt{a}||\mathtt{b}), \mathtt{R}$ ok, then $(\mathtt{a}, \mathtt{b}, (\mathtt{a}||\mathtt{b}), \mathtt{R})^* \approx ((\mathtt{a}||\mathtt{b}), \mathtt{R})^*$.
3. If $\mathtt{T} \vdash \mathtt{a}, \mathtt{b}, \mathtt{R}$ ok and $\mathtt{T} \vdash (\mathtt{a}||\mathtt{b}), (\mathtt{a}||\mathtt{b}), \mathtt{R}$ ok, then $(\mathtt{a}, \mathtt{b}, \mathtt{R})^* \approx ((\mathtt{a}||\mathtt{b}), (\mathtt{a}||\mathtt{b}), \mathtt{R})^*$.
4. If $\mathtt{T} \vdash \tau, \rho, \sigma, \mathtt{R}$ ok and $\mathtt{T} \vdash (\rho||\sigma), \tau, \mathtt{R}$ ok, then $(\tau, \rho, \sigma, \mathtt{R})^* \approx ((\rho||\sigma), \tau, \mathtt{R})^*$.

Lemma 2.3.3.10: For every critical pair $\langle \mathtt{C}_1, \mathtt{C}_2 \rangle$ of $\langle \twoheadrightarrow^1, \approx^1 \rangle$, we have $\mathtt{C}_1{}^* \approx \mathtt{C}_2{}^*$.

Proof: By a tedious verification, using Lemma 2.3.3.9 for the interesting cases. □

Theorem 2.3.3.11: The restriction of $\twoheadrightarrow^1$ to well-typed terms is confluent modulo $\approx$.

Corollary 2.3.3.12:

1. If $\mathtt{r}$ and $\mathtt{s}$ are well-formed records, then $\mathtt{r} \equiv \mathtt{s}$ iff there exist $\mathtt{r}'$ and $\mathtt{s}'$ in normal form such that $\mathtt{r} \twoheadrightarrow \mathtt{r}'$ and $\mathtt{s} \twoheadrightarrow \mathtt{s}'$ and $\mathtt{r}' \approx \mathtt{s}'$.

2. If $\mathtt{t}$ and $\mathtt{u}$ are well-formed types, then $\mathtt{t} \equiv \mathtt{u}$ iff there exist $\mathtt{t}'$ and $\mathtt{u}'$ such that $\mathtt{t} \twoheadrightarrow \mathtt{t}'$ and $\mathtt{u} \twoheadrightarrow \mathtt{u}'$ and $\mathtt{t}' \approx \mathtt{u}'$.

3. If $\mathtt{R}$ and $\mathtt{S}$ are well-formed constraint lists, then $\mathtt{R} \approx \mathtt{S}$ iff there exist $\mathtt{R}'$ and $\mathtt{S}'$ such that $\mathtt{R} \twoheadrightarrow \mathtt{R}'$ and $\mathtt{S} \twoheadrightarrow \mathtt{S}'$ and $\mathtt{R}' \approx \mathtt{S}'$.

Corollary 2.3.3.13: Normal forms are unique up to $\approx$.

If $\mathtt{r}$ is a well-formed record, then $\mathtt{r}^*$ denotes one of its normal forms computed by applying the $\twoheadrightarrow$ rules in some canonical order; similarly for types and constraint lists.

Theorem 2.3.3.14:

1. Let $\mathtt{r}$ be a well-formed record type in normal form. Then $\mathtt{r}$ has the form

$$\mathtt{a}_1 \,||\, \mathtt{a}_2 \,||\, \ldots \,||\, \mathtt{a}_n \,||\, \mathtt{l}_1{:}\mathtt{t}_1 \,||\, \ldots \,||\, \mathtt{l}_k{:}\mathtt{t}_k,$$

   where the $\mathtt{a}_i$'s are distinct variables, the $\mathtt{l}_j$'s are distinct labels, the $\mathtt{t}_j$'s are in normal form, and n and k are greater than or equal to 0 (when both are 0, the normal form is `Empty`). By "has the form ..." we mean that $\mathtt{r}$ has the above form up to associativity and commutativity of $||$.

2. Let $\mathtt{R}$ be a well-formed constraint list. Then $\mathtt{R}$ has the form

$$\mathtt{a}_1, \ldots, \mathtt{a}_n, \mathtt{l}_1, \ldots, \mathtt{l}_k$$

   where the $\mathtt{a}_i$'s are all distinct, the $\mathtt{l}_j$'s are all distinct, and n and k are greater than or equal to 0. By "has the form ..." we mean that $\mathtt{R}$ is a list of variables and labels in some order.

Note that as a consequence, a well-formed normal form is restriction-free. This implies that every well-formed record expression reduces to a well-formed restriction-free record expression.

Lemma 2.3.3.15: The rule

$$\mathtt{r}, \mathtt{r}, \mathtt{R} \equiv \mathtt{r}, \mathtt{R} \qquad \text{(A-CEQ-FULL-DUPL)}$$

is admissible when $\mathtt{r}, \mathtt{r}, \mathtt{R}$ and $\mathtt{r}, \mathtt{R}$ are well formed.

Proof sketch: By construction. Consider a reduction of $\mathtt{r}, \mathtt{R}$ to normal form. By Theorem 2.3.3.14, $(\mathtt{r}, \mathtt{R})^*$ consists only of type variables and labels. We may therefore use rules A-CEQ-DUPL and A-CEQ-MERGE to duplicate the sublist of type variables and labels that come from $\mathtt{r}$. By replaying the reduction from $\mathtt{r}, \mathtt{R}$ to $(\mathtt{r}, \mathtt{R})^*$ twice in reverse, we obtain $\mathtt{r}, \mathtt{r}, \mathtt{R} \equiv (\mathtt{r}, \mathtt{R})^*$. The result follows by transitivity of $\equiv$. □

Theorem 2.3.3.16: The relations $\sim$ and $\equiv$ coincide on well-typed terms.

Proof: It is easy to see that every $\twoheadrightarrow^1$ and $\approx^1$ axiom is a valid equivalence, and conversely (using Lemma 2.3.3.15 in the case of EQ-DUPL), that every equivalence axiom holds in $\equiv$. □

Corollary 2.3.3.17: The relation $\sim$ is decidable for well-formed expressions.

Proof: To decide whether or not $\mathtt{r} \sim \mathtt{s}$, given that $\mathtt{r}$ and $\mathtt{s}$ are well-formed, reduce each to their normal forms and test whether or not $\mathtt{r}^* \sim \mathtt{s}^*$. Given the characterization of normal forms given above, this reduces to re-grouping the atomic record types in some standard order. Similar methods apply to constraint sets and types. □

$$\mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} \mathtt{Empty} \qquad \text{(A-CMP-EMPTY-1)}$$

$$\mathtt{T} \vdash \mathtt{Empty} \#^{\Rightarrow} \mathtt{r} \qquad \text{(A-CMP-EMPTY-2)}$$

$$\frac{\mathtt{T} \vdash \mathtt{s}_1 \#^{\Rightarrow} \mathtt{r} \qquad \mathtt{T} \vdash \mathtt{s}_2 \#^{\Rightarrow} \mathtt{r}}{\mathtt{T} \vdash (\mathtt{s}_1 \parallel \mathtt{s}_2) \#^{\Rightarrow} \mathtt{r}} \qquad \text{(A-CMP-MERGE-I-1)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} \mathtt{s}_1 \qquad \mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} \mathtt{s}_2}{\mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} (\mathtt{s}_1 \parallel \mathtt{s}_2)} \qquad \text{(A-CMP-MERGE-I-2)}$$

$$\frac{\mathtt{l} \neq \mathtt{l}'}{\mathtt{T} \vdash \mathtt{l}:\mathtt{t} \#^{\Rightarrow} \mathtt{l}':\mathtt{t}'} \qquad \text{(A-CMP-BASE/BASE)}$$

$$\frac{\mathtt{a} \in \mathtt{S}^*}{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2, \mathtt{b}\#\mathtt{S}, \mathtt{T}_3 \vdash \mathtt{a} \#^{\Rightarrow} \mathtt{b}} \qquad \text{(A-CMP-TVAR/TVAR-1)}$$

$$\frac{\mathtt{a} \in \mathtt{S}^*}{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2, \mathtt{b}\#\mathtt{S}, \mathtt{T}_3 \vdash \mathtt{b} \#^{\Rightarrow} \mathtt{a}} \qquad \text{(A-CMP-TVAR/TVAR-2)}$$

$$\frac{\mathtt{l} \in \mathtt{R}^*}{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2 \vdash \mathtt{a} \#^{\Rightarrow} \mathtt{l}:\mathtt{t}} \qquad \text{(A-CMP-BASE/TVAR-1)}$$

$$\frac{\mathtt{l} \in \mathtt{R}^*}{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2 \vdash \mathtt{l}:\mathtt{t} \#^{\Rightarrow} \mathtt{a}} \qquad \text{(A-CMP-BASE/TVAR-2)}$$

Figure 6: Algorithmic compatibility rules

2.3.4 Compatibility Checking

The compatibility checking algorithm is presented as a collection of inference rules for deriving judgements of the form $\mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} \mathtt{s}$ where $\mathtt{r}$ and $\mathtt{s}$ are restriction-free, well-formed record types. The rules appear in Figure 6. To prove that these rules define an algorithm for compatibility checking, we show that they are sound and complete with respect to the declarative formulation, and that we may effectively decide whether or not a derivation exists in accordance with these rules.

The soundness of the algorithm is stated by the following theorem:

Theorem 2.3.4.1:

1. If $\mathtt{T} \vdash \mathtt{r}$ record and $\mathtt{T} \vdash \mathtt{r}'$ record and $\mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} \mathtt{r}'$, then $\mathtt{T} \vdash \mathtt{r} \# \mathtt{r}'$.
2. If $\mathtt{T} \vdash \mathtt{r}$ record and $\mathtt{T} \vdash \mathtt{R}$ ok and $\mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} \mathtt{R}$, then $\mathtt{T} \vdash \mathtt{r} \# \mathtt{R}$.

The proof relies on the following facts:

Lemma 2.3.4.2: If $\mathtt{T} \vdash \mathtt{R}$ ok and $\mathtt{R} \twoheadrightarrow^1 \mathtt{R}'$ and $\mathtt{T} \vdash \mathtt{r} \# \mathtt{R}'$, then $\mathtt{T} \vdash \mathtt{r} \# \mathtt{R}$.

Lemma 2.3.4.3: If $\vdash \mathtt{T}$ ok and $\mathtt{T} = \mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2, \mathtt{b}\#\mathtt{S}, \mathtt{T}_3$ and $\mathtt{a} \in \mathtt{S}^*$, then $\mathtt{T} \vdash \mathtt{a} \# \mathtt{b}$.

Lemma 2.3.4.4: If $\vdash \mathtt{T}$ ok and $\mathtt{T} = \mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2$ and $\mathtt{l} \in \mathtt{R}^*$ and $\mathtt{T} \vdash \mathtt{l}:\mathtt{t}$ record, then $\mathtt{T} \vdash \mathtt{a} \# \mathtt{l}:\mathtt{t}$.

The completeness of the algorithm is stated by the following theorem:

Theorem 2.3.4.5:

1. If $\mathtt{T} \vdash \mathtt{r} \# \mathtt{s}$, then $\mathtt{T} \vdash \mathtt{r}^* \#^{\Rightarrow} \mathtt{s}^*$.
2. If $\mathtt{T} \vdash \mathtt{r} \# \mathtt{R}$, then $\mathtt{T} \vdash \mathtt{r}^* \#^{\Rightarrow} \mathtt{R}^*$.

The proof relies on a number of preliminary results.

Lemma 2.3.4.6: The relation computed by the algorithm is symmetric on normal forms:

1. Suppose that $\mathtt{r}$ and $\mathtt{s}$ are well-formed record types in normal form and that $\mathtt{r} \approx \mathtt{r}'$ and $\mathtt{s} \approx \mathtt{s}'$. Then $\mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} \mathtt{s}$ iff $\mathtt{T} \vdash \mathtt{r}' \#^{\Rightarrow} \mathtt{s}'$.

2. Suppose that $\mathtt{r}$ and $\mathtt{s}$ are well-formed record types in normal form. If $\mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} \mathtt{s}$, then $\mathtt{T} \vdash \mathtt{s} \#^{\Rightarrow} \mathtt{r}$.

Proof:

1. Since $\mathtt{r}$ and $\mathtt{s}$ are in normal form, $\mathtt{r}'$ and $\mathtt{s}'$ must also be, and $\mathtt{r}'$ and $\mathtt{s}'$ must be a commutative, associative re-grouping of $\mathtt{r}$ and $\mathtt{s}$.

2. A simple induction on derivations. Box

Lemma 2.3.4.7: Suppose that $\mathtt{r}$ and $\mathtt{t}$ are well-formed normal forms. Then $\mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} \mathtt{l{:}t}$ implies $\mathtt{T} \vdash \mathtt{r} \# \mathtt{l{:}t'}$ for any well-formed type $\mathtt{t}'$.

Proof: Consider the form of $\mathtt{r}$, and note that the property holds for atomic record types. □

Lemma 2.3.4.8: If $\mathtt{T} \vdash \mathtt{s}_1 || \mathtt{s}_2$ record and $\mathtt{T} \vdash \mathtt{r}^* \#^{\Rightarrow} \mathtt{s}_1^*$ and $\mathtt{T} \vdash \mathtt{r}^* \#^{\Rightarrow} \mathtt{s}_2^*$, then $\mathtt{T} \vdash \mathtt{r}^* \#^{\Rightarrow} (\mathtt{s}_1 || \mathtt{s}_2)^*$.

Proof: By a straightforward induction on a reduction sequence from $\mathtt{s}_1 || \mathtt{s}_2$ to $(\mathtt{s1}||\mathtt{s2})^*$. □

Definition 2.3.4.9: Let atoms($\mathtt{r}^*$) be

1. the empty set if $\mathtt{r}^*$ is **Empty**,

2. otherwise, the result of replacing each atom of the form $\mathtt{l{:}t}$ by the bare label $\mathtt{l}$ in the set of atoms of the normal form of $\mathtt{r}$.

Lemma 2.3.4.10: If $\mathtt{T} \vdash \mathtt{R}$ ok and $\mathtt{r} \in \mathtt{R}$, then atoms($\mathtt{r}^*$) $\subseteq \mathtt{R}^*$.

Proof: If $\mathtt{r}^*$ is **Empty**, the result is immediate. So assume $\mathtt{r}^*$ is not **Empty**. Since the relation $\twoheadrightarrow$ is confluent modulo $\approx$, we may focus on any reduction path from $\mathtt{R}$ to a $\twoheadrightarrow$-normal form $\mathtt{R}^{*\prime}$ such that $\mathtt{R}^{*\prime} \approx \mathtt{R}^*$. Begin by reducing $\mathtt{r}$ to $\mathtt{r}^*$ in $\mathtt{R}$, giving $\mathtt{R}'$. Then flatten $\mathtt{r}^*$ in $\mathtt{R}'$ to its constituent atoms, giving $\mathtt{R}''$, and perform all the base reductions $\mathtt{l{:}t} \twoheadrightarrow \mathtt{l}$ in the residuals of $\mathtt{r}$ in $\mathtt{R}''$, giving $\mathtt{R}^{*\prime}$. Observe that atoms($\mathtt{r}^*$) $\subseteq \mathtt{R}^{*\prime}$. Finally, rewrite $\mathtt{R}^{*\prime}$ to $\mathtt{R}^*$ using the axioms for $\approx$, noting that no bare label or type variable is dropped from $\mathtt{R}^{*\prime}$ (considered as a set) during any of these reductions. □

Lemma 2.3.4.11: Let $\mathtt{T} = \mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2$, and suppose that $\vdash \mathtt{T}$ ok and $\mathtt{r} \in \mathtt{R}$ (so that $\mathtt{T} \vdash \mathtt{a} \# \mathtt{r}$.) Then $\mathtt{T} \vdash \mathtt{a} \#^{\Rightarrow} \mathtt{r}^*$.

Proof: By induction on the form of $\mathtt{r}^*$, using Lemma 2.3.4.10. □

It remains to show that the relation axiomatized by the rules of Figure 6 is decidable. This follows from two observations. First, the rules are "almost syntax-directed" in the sense that there is a pair of rules for each form of restriction-free record type, one the symmetric version of the other. Second, in the cases where two rules apply to a given pair of types, it is immaterial which rule is applied. To see this we have to consider four combinations. If both $\mathtt{r}$ and $\mathtt{s}$ are **Empty**, either rule yields the same result (success). If one of $\mathtt{r}$ and $\mathtt{s}$ is **Empty** and the other is a merge, then no matter how we proceed the result is the same (success). In the case that both $\mathtt{r}$ and $\mathtt{s}$ are merges, we have no choice but to apply A-CMP-MERGE-I-1 and A-CMP-MERGE-I-2 to break down both merges into their components; the order in which we do this has no effect on the result. We may therefore check compatibility by decomposing merges in an arbitrary order until we reach atoms, then apply the appropriate rules. If any derivation exists, this procedure will construct it, and we can effectively decide whether any rule applies.

$$\mathtt{T_1, a\#R, T_2 \vdash a \Rightarrow record} \qquad \text{(A-WFR-VAR)}$$

$$\frac{\mathtt{T \vdash r_1 \Rightarrow record} \quad \mathtt{T \vdash r_2 \Rightarrow record} \quad \mathtt{T \vdash r_1 \#^{\Rightarrow} r_2}}{\mathtt{T \vdash r_1 \,||\, r_2 \Rightarrow record}} \qquad \text{(A-WFR-MERGE)}$$

$$\frac{\mathtt{T;G \vdash e_1 \Rightarrow r_1} \quad \mathtt{T;G \vdash e_2 \Rightarrow r_2} \quad \mathtt{T \vdash r_1^* \#^{\Rightarrow} r_2^*}}{\mathtt{T;G \vdash e_1 \,||\, e_2 \Rightarrow r_1 \,||\, r_2}} \qquad \text{(A-WTT-MERGE)}$$

$$\frac{\mathtt{T;G \vdash e \Rightarrow \forall a\#R.\, t} \quad \mathtt{T \vdash r \Rightarrow record} \quad \mathtt{T \vdash r^* \#^{\Rightarrow} R^*}}{\mathtt{T;G \vdash e[r] \Rightarrow [r/a]t}} \qquad \text{(A-WTT-ALL-E)}$$

Figure 7: Selected type synthesis rules

2.3.5 Type Synthesis

The type checking algorithm for $\lambda^{||}$ is given in terms of a type synthesis procedure that constructs a "canonical" type for a given expression in a given context. This procedure is described by a formal system for deriving judgements of the form $\mathtt{T ; G \vdash e \Rightarrow t}$, together with a number of auxiliary judgements of a similar form. As with the compatibility checker, we show that this formal system defines a type checking algorithm by proving that it is sound and complete with respect to the definition of $\lambda^{||}$, and that we may effectively decide whether or not a derivation exists. A representative set of rules from the definition of the type synthesis algorithm is given in Figure 7. The complete set of rules appears in Appendix B.

The judgements derived by the algorithm are:

$$\mathtt{T \vdash R \Rightarrow ok} \qquad \mathtt{T \vdash t \Rightarrow type} \qquad \mathtt{T \vdash r \Rightarrow record} \qquad \mathtt{T;G \vdash e \Rightarrow t.}$$

Note that in rules A-WTT-MERGE and A-WTT-ALL-E, the record types and constraint lists are normalized before checking compatibility.

The soundness and completness of the algorithm is stated by the following theorems:

Theorem 2.3.5.1: (Soundness)

1. If $\vdash$ T ok and $\mathtt{T \vdash r \Rightarrow}$ record, then $\mathtt{T \vdash r}$ record.
2. If $\vdash$ T ok and $\mathtt{T \vdash t \Rightarrow}$ type, then $\mathtt{T \vdash t}$ type.
3. If $\vdash$ T ok and $\mathtt{T \vdash R \Rightarrow}$ ok, then $\mathtt{T \vdash R}$ ok.
4. If $\mathtt{T \vdash G}$ ok and $\mathtt{T;G \vdash e \Rightarrow t}$, then $\mathtt{T;G \vdash e \in t}$.

Theorem 2.3.5.2: (Completeness)

1. If $\mathtt{T \vdash r}$ record, then $\mathtt{T \vdash r \Rightarrow}$ record.
2. If $\mathtt{T \vdash t}$ type, then $\mathtt{T \vdash t \Rightarrow}$ type.
3. If $\mathtt{T \vdash R}$ ok, then $\mathtt{T \vdash R \Rightarrow}$ ok.
4. If $\mathtt{T \vdash e \in t}$, then $\mathtt{T \vdash e \Rightarrow t'}$ for some $\mathtt{t'}$ such that $\mathtt{t' \sim t}$.

Both may be proved by induction on derivations.

For decidability, we have only to note that the relevant instances of compatibility and conversion checking are decidable, and that the rules are syntax-directed.

3 Examples

In this section we develop some examples illustrating the expressiveness of $\lambda^{||}$. Section 3.1 presents some short examples and compares them to their formulations in related record calculi. Section 3.2 shows how Wand's model of a simple object-oriented programming language with method inheritance and a notion of `self` may be emulated in $\lambda^{||}$. Section 3.3 shows that in the presence of recursive types, quantifiers carrying negative constraints are sufficient to satisfactorily express the examples motivating the extension of bounded quantification to F-bounded quantification. Section 3.4 shows how some other primitive record operations may be added to the language.

3.1 Simple Record Manipulation

A function that accepts any record not already possessing an `l` field and adds the field `l=5` can be written in $\lambda^{||}$ as:

$$\begin{array}{l}\Lambda a\#l.\\ \quad \lambda x{:}a.\\ \quad\quad x \,||\, l{=}5\\ \in \forall a\#l.\ a \rightarrow (a \,||\, l{:}\mathtt{Int})\end{array}$$

A similar function that accepts any record *with* an `l` field and overrides it with the field `l=5` can also be expressed:

$$\begin{array}{l}\Lambda b.\ \Lambda a\#l.\\ \quad \lambda x{:}(a||l{:}b).\\ \quad\quad (x\backslash l) \,||\, l{=}5\\ \in \mathtt{All}\ b.\forall a\#l.\ (a||l{:}b) \rightarrow (a\backslash l) \,||\, l{:}\mathtt{Int}\end{array}$$

(Strictly speaking, this function can only accept a record whose `l` field has some *record* type, because for simplicity we have omitted ordinary type quantification. The empty constraint list on the first quantifier is dropped, by convention, to reduce clutter, but it should not be confused with the usual unconstrained quantifier.)

Unlike the calculi of Rémy, Wand, and Cardelli and Mitchell, $\lambda^{||}$ is unable to express the function that takes any record whatsoever and gives it an `l` field with value 5.

Functions that perform "deep updates" of fields within fields of records, preserving all the type information about the unmodified fields, can be expressed in $\lambda^{||}$. For example:

$$\begin{array}{l}\forall a\#l_a.\\ \quad \forall b\#l_b.\\ \quad\quad \lambda x{:}(b \,||\, l_b{:}(a||l_a{:}\mathtt{Bool})).\\ \quad\quad\quad (x\backslash l_b) \,||\, l_b = ((x.l_b)\backslash l_a \,||\, l_a = (\mathtt{not}\ x.l_b.l_a))\end{array}$$

Cardelli and Mitchell's formulation of this function, which relies on *extraction types* like `a.l`, is somewhat shorter.

3.2 Object-Oriented Programming

Wand [24, 25, 26, 27] has shown how a simple form of object-oriented programming may be modeled in an extension of ML [14, 15] with record concatenation and recursive types. An analogous construction can be made in $\lambda^{||}$.

We assume the existence of a number of extensions of the basic type system, including polymorphism (quantification over arbitrary types), fixed point combinators, recursive types, and base types such as `int` and `bool` (and associated operations). Of these, only recursive types present any serious problems. Recursive

types are denoted μ a. t(a), where t(a) is an arbitrary type expression, possibly involving the variable a. Let $\lambda^{||\mu}$ stand for the extension of $\lambda^{||}$ with recursive types. Type equivalence for $\lambda^{||\mu}$ is defined to be the greatest fixed point of a monotone operator Φ on binary relations on types determined by the inference rules for type equivalence given in Appendix A, augmented by the axiom μ a. t(a) $\sim$ t(μ a. t(a)). This characterization of type equality captures the informal notion that recursively-defined types stand for their "infinite unrollings" as regular trees. In the examples we make use of the following useful characterization of type equivalence: to show that s $\sim$ t, it suffices to assume that s $\sim$ t and prove that s $\Phi(\sim)$ t. Although this method is useful in the examples, we have not studied the decidability of this notion of type equivalence, and hence the decidability of type checking for $\lambda^{||\mu}$ remains open.

Following Wand, we consider first a very simple notion of class that does not include a notion of **self** or provide for inheritance. In this setting classes are functions from a parameter type, representing instance-specific values, to some record type, representing the methods. Specifically,

class x:t **methods** l_1=e_1, ..., l_n=e_n

stands for the expression

λx:t. (l_1=e_1 || ... || l_n = e_n)

which has type

t $\rightarrow$ (l_1:t_1 || ... || l_n:t_n)

provided that t is a type, x:t $\vdash$ $e_i \in t_i$, and all the l_i's are distinct. Let C be such a class. If e is a term of type t, then C(e) is a record of type l_1:t_1 || ... || l_n:t_n, so that C(e).l_i selects the method associated with the instance of C determined by e. Following Wand we write

new C e

for the application C(e).

Wand's list-processing example may be readily translated into this setting as follows. First, we introduce some abbreviations:

```
cellPlus[r,s,t]  =  r || null:Bool || car:s || cdr:t
cell[s,t]        =  cellPlus[Empty,s,t]
listPlus[r,s]    =  μa. cellPlus[r,s,a]
list[s]          =  listPlus[Empty,s].
```

Note that

```
listPlus[r,s]  ~  cellPlus[r,s,listPlus[s]]
               ~  r || null:Bool || car:s || cdr:listPlus[r,s]
```

and that

```
list[s]  ~  cell[s,list[s]]
         ~  null:Bool || car:s || cdr:list[s].
```

These equations will be important for type checking the examples below.

Define

```
nil  =  Λs,t. λx:1. (null=true || car=error[s] || cdr =error[t])
     ∈  ∀s,t. 1 → cell[s,t].
```

Note that nil is parameterized by the types of the cells. We assume a special constant error $\in \forall$t. t assigning an "error element" to each type. (There are other possible solutions. In a richer calculus, evaluating nil.car could raise an exception, for example.)

Define

```
cons  =  Λs,t. λ(h,t):s×t. (null=false || car=h || cdr=t)
      ∈  ∀s,t. (s×t) → cell[s,t].
```

Now consider

```
map  =  Λs,t. Λr#null,car,cdr. λf:s→t.
          fix_u λmap':u. λl:listPlus[r,s].
            if l.null
                then new nil[t,list[t]]()
                else new cons[t][list[t]](f l.car)(map' l.cdr)
```

where $\mathtt{u} = \mathtt{listPlus[r,s]{\rightarrow}list[t]}$.

It is easy to see that map has type

$$\forall \mathtt{s,t}.\ \forall \mathtt{r\#null,car,cdr}.\ (\mathtt{s{\rightarrow}t}) \rightarrow \mathtt{listPlus[r,s]} \rightarrow \mathtt{list[s]}$$

under some obvious assumptions about typing conditionals and taking account of the equations between recursive types noted above.

Extending the example to a simple, self-less form of multiple inheritance presents no particular difficulties:

$$\mathtt{class\ x{:}t\ inherits\ f_1,\ldots,f_k\ methods\ l_1{=}e_1,\ldots,l_n{=}e_n\ end}$$

stands for

$$\lambda \mathtt{x{:}t}.\ (\mathtt{f_1} \parallel \ldots \parallel \mathtt{f_k} \parallel \mathtt{l_1{=}e_1} \parallel \ldots \parallel \mathtt{l_n{=}e_n})$$

which has type

$$\mathtt{t} \rightarrow (\mathtt{F_1} \parallel \ldots \parallel \mathtt{F_k} \parallel \mathtt{l_1{:}t_1} \parallel \ldots \parallel \mathtt{l_n{:}t_n})$$

provided that $\mathtt{t}$ is well formed, $\mathtt{x{:}t} \vdash \mathtt{f_j} \in \mathtt{F_j}$, $\mathtt{x{:}t} \vdash \mathtt{e_i} \in \mathtt{t_i}$, and the following compatibility conditions are satisfied:

1. all $\mathtt{l_i}$'s are distinct,
2. $\mathtt{F_i} \# \mathtt{F_j}$ whenever $i \neq j$,
3. $\mathtt{F_i} \# \mathtt{l_j}$ for each $\mathtt{F_i}$ and each label $\mathtt{l_j}$.

These conditions are not enforced in Wand's system since he prefers the asymmetric form of merge in which the rightmost occurrence of a field in a record type determines the value assigned to that field. We prefer instead that the type checker ensure that no clashes are possible, and insist that the programmer explicitly eliminate the field that he or she wishes to override using the restriction operation.

The extension to model "**self**" presents some interesting problems for typing. We begin by considering a form of **self** that does not behave correctly in the presence of inheritance and then consider how to combine **self** and inheritance. The simple-minded approach to **self** proceeds by defining

$$\mathtt{class\ x{:}s\ methods\ l_1{=}e_1,\ldots,l_n{=}e_n\ end}$$

to be

$$\lambda \mathtt{x{:}s}.\ \lambda \mathtt{self{:}t}.\ (\mathtt{l_1{=}e_1} \parallel \ldots \parallel \mathtt{l_n{=}e_n})$$

which has type $\mathtt{s{\rightarrow}t{\rightarrow}t}$, where $\mathtt{t} = (\mathtt{l_1{:}t_1} \parallel \ldots \parallel \mathtt{l_n{:}t_n})$, provided that $\mathtt{t}$ is a type and $\mathtt{x{:}t, self{:}t} \vdash \mathtt{e_i} : \mathtt{t_i}$.

Class instantiation is performed by taking a fixed point, as described in Wand's paper. If **C** is a class of the form just described, then

$$\mathtt{new\ C\ e}$$

is defined to be

$$\mathtt{fix_t\ (C(e))},$$

where **C** has type $\mathtt{s{\rightarrow}t{\rightarrow}t}$ and **e** has type **s**.

This simple-minded approach interacts badly with inheritance in two ways. First, the type assigned to **self** is required to coincide with the type of the body of the class definition. This forces us to define at class definition time all methods that are used by any method of the class. This precludes the possibility of "mixing in" the missing methods at a later stage, a significant feature of object-oriented programming. Second, the type assigned to **self** does not take account of the fact that the class may occur as the superclass of some subclass which may define not only the "missing" methods, but also some additional methods not relevant to the definition of the superclass. In short, **self** should refer to the entire subclass, and not just the superclass or even just the subclass of the superclass required to achieve "closure."

This means that a class definition must have a "floating" notion of **self** that can be instantiated at any subclass consistent with the requirements imposed on **self** and with the methods that are given for **self** as part of the class definition. Following Wand, we take

$$\mathtt{class\ x{:}t\ inherits\ f_1,\ldots,f_k\ methods\ l_1{=}e_1,\ldots,l_n{=}e_n\ end}$$

to stand for (roughly)

$$\lambda \mathtt{x}{:}\mathtt{t}.\ \mathtt{fix}_{\mathtt{v}}\ (\lambda \mathtt{s}{:}\mathtt{v}.\ (\mathtt{f}_1(\mathtt{s})\ ||\ \ldots\ ||\ \mathtt{f}_2(\mathtt{s})\ ||\ \mathtt{l}_1{=}\mathtt{e}_1\ ||\ \ldots\ ||\ \mathtt{l}_n{=}\mathtt{e}_n))$$

where $\mathtt{v}$ is a suitable type expression. This ensures that the $\mathtt{f}_i$'s and the new class share a common sense of `self`.

Since our system is explicitly-typed, we must take explicit account of the variability of the type of `self` by introducing suitable constrained type abstractions. The main idea is to assign to abstract with respect to a record type variable $\mathtt{a}$ representing an arbitrary "extension" to the type of record comprising the object, and to assign to `self` a type of the form $\mathtt{a}\ ||\ \mathtt{l}_1{:}\mathtt{t}_1\ ||\ \ldots\ ||\ \mathtt{l}_n{:}\mathtt{t}_n$, where the $\mathtt{l}_i{:}\mathtt{t}_i$'s include both the types of the methods actually defined by the class and the types of the methods assumed by the new methods. By choosing the type $\mathtt{a}$ appropriately we may instantiate `self` to be any superclass of the class, as required for inheritance.

A class definition of the form

$$\mathtt{class\ x{:}t\ methods\ l_1{=}e_1, \ldots, l_n{=}e_n\ end}$$

is represented by a λ-term of the form

$$\begin{array}{l}\lambda \mathtt{x}{:}\mathtt{t}.\\ \quad \Lambda \mathtt{a}\#\mathtt{l}_1, \ldots, \mathtt{l}_n.\\ \quad\quad \lambda \mathtt{s}{:}(\mathtt{a}\ ||\ \mathtt{l}_1{:}\mathtt{t}_1\ ||\ \ldots\ ||\ \mathtt{l}_n{:}\mathtt{t}_n).\\ \quad\quad\quad (\mathtt{l}_{i_1}{=}\mathtt{e}_{i_1}\ ||\ \ldots\ ||\ \mathtt{l}_{i_r}{=}\mathtt{e}_{i_r}),\end{array}$$

where $\{i_1, \ldots, i_r\} \subseteq 1 \ldots n$. This expression has type

$$\begin{array}{l}\mathtt{t} \rightarrow \forall \mathtt{a}\#\mathtt{l}_1, \ldots, \mathtt{l}_n.\\ \quad (\mathtt{a}\ ||\ \mathtt{l}_1{:}\mathtt{t}_1\ ||\ \ldots\ ||\ \mathtt{l}_n{:}\mathtt{t}_n)\\ \quad\quad \rightarrow (\mathtt{l}_{i_1}{:}\mathtt{t}_{i_1}\ ||\ \ldots\ ||\ \mathtt{l}_{i_r}{:}\mathtt{t}_{i_r})\end{array}$$

provided that $\mathtt{t}$ type and $\mathtt{x}{:}\mathtt{t}$, $\mathtt{a}\#\mathtt{l}_1, \ldots, \mathtt{l}_n$ and $\mathtt{s}{:}\mathtt{a}||\mathtt{l}_1 : \mathtt{t}_1||\ldots||\mathtt{l}_n : \mathtt{t}_n \vdash \mathtt{e}_j \in \mathtt{t}_j$.

To build a subclass of such a class `C`, we proceed as follows. Let us suppose that we intend to build a class `D`, and instantiate it at some value $\mathtt{v}$. Therefore we must at least define the methods assumed in the definition of `C`, and perhaps some additional methods particular to `D`. Thus we will want to use a definition of the form

$$\mathtt{class\ x{:}t\ inherits\ new\ C\ e\ methods\ l_{j_1}{=}e_{j_1}\ ||\ \ldots\ ||\ l_{j_q}{=}e_{j_q}\ ||\ k_1{=}d_1\ ||\ \ldots\ ||\ k_p{=}d_p\ end}$$

where $\mathtt{l}_{i_1}, \ldots, \mathtt{l}_{i_r}, \mathtt{l}_{j_1}, \ldots, \mathtt{l}_{j_q} = \mathtt{l}_1 \ldots \mathtt{l}_n$ (so that the $\mathtt{l}_j$'s supply the missing methods and the $\mathtt{k}_i$'s are new methods associated with `D`). This corresponds to the following lambda term:

$$\begin{array}{l}\lambda \mathtt{x}{:}\mathtt{t}.\ \Lambda \mathtt{a}\#\mathtt{l}_1, \ldots, \mathtt{l}_n, \mathtt{k}_1, \ldots, \mathtt{k}_p.\\ \quad \lambda \mathtt{self}{:}(\mathtt{a}||\mathtt{l}_1{:}\mathtt{t}_1||\ldots||\mathtt{l}_n{:}\mathtt{t}_n||\mathtt{k}_1{:}\mathtt{s}_1||\ldots||\mathtt{k}_p{:}\mathtt{s}_p).\\ \quad\quad \mathtt{C}(\mathtt{e})[\mathtt{a}||\mathtt{k}_1{:}\mathtt{s}_1||\ldots||\mathtt{k}_p{:}\mathtt{s}_p](\mathtt{self})\\ \quad\quad\quad ||\ \mathtt{l}_{j_1}{=}\mathtt{e}_{j_1}\ ||\ \ldots\ ||\ \mathtt{l}_{j_q}{=}\mathtt{e}_{j_q}\ ||\ \mathtt{k}_1{=}\mathtt{d}_1\ ||\ \ldots\ ||\ \mathtt{k}_p{=}\mathtt{d}_p\end{array}$$

which has type

$$\begin{array}{l}\mathtt{t} \rightarrow \forall \mathtt{a}\#\mathtt{l}_1, \ldots, \mathtt{l}_n, \mathtt{k}_1, \ldots, \mathtt{k}_p.\\ \quad\quad (\mathtt{a}||\mathtt{l}_1{:}\mathtt{t}_1||\ldots||\mathtt{l}_n{:}\mathtt{t}_n||\mathtt{k}_1{:}\mathtt{s}_1||\ldots||\mathtt{k}_p{:}\mathtt{s}_p)\\ \quad\quad \rightarrow (\mathtt{l}_1{:}\mathtt{t}_1||\ldots||\mathtt{l}_n{:}\mathtt{t}_n||\mathtt{k}_1{:}\mathtt{s}_1||\ldots||\mathtt{k}_p{:}\mathtt{s}_p)\end{array}$$

so that $\lambda \mathtt{x}{:}\mathtt{t}.\ \mathtt{fix}(\mathtt{D}(\mathtt{x})[\mathtt{Empty}]) \in \mathtt{t} \rightarrow \mathtt{l}_1{:}\mathtt{t}_1\ ||\ \ldots\ ||\ \mathtt{k}_p{:}\mathtt{s}_p$ (assuming that the $\mathtt{l}_j$'s supplied the missing methods).

Here is a concrete example, taken from Wand. Define the class A to be

$$\begin{array}{l}\lambda \mathtt{x}{:}\mathtt{int}.\\ \quad \Lambda \mathtt{a}\#\mathtt{sum}, \mathtt{n}.\\ \quad\quad \lambda \mathtt{self}{:}(\mathtt{a}||\mathtt{sum}{:}\mathtt{int}||\mathtt{n}{:}\mathtt{int}).\\ \quad\quad\quad \mathtt{sum}{=}(\mathtt{x} + \mathtt{self.n}).\end{array}$$

with type

$$\mathtt{int} \rightarrow \forall \mathtt{a}\#\mathtt{sum}, \mathtt{n}.\ (\mathtt{a}||\mathtt{sum}{:}\mathtt{int}||\mathtt{n}{:}\mathtt{int}){\rightarrow}(\mathtt{sum}{:}\mathtt{int}).$$

This class cannot be instantiated since it lacks a method for `n`. Define the class `B` to be a subclass of `A` that provides a method for `n`:

```
λy:int.
  Λa#sum,n.
    λself:(a||sum:int||n:int).
      A(5)[a](self) || n=y
```

with type

$$\texttt{int} \rightarrow \forall \texttt{a\#sum,n}.\ (\texttt{a||sum:int||n:int}) \rightarrow (\texttt{sum:int||n:int}).$$

Then $\texttt{B(3)[Empty]} \in \texttt{sum:int||n:int} \rightarrow \texttt{sum:int||n:int}$, so that $\texttt{fix(B(3)[Empty]).sum} = 8$.

We may also define a class C that extends A with a method for n and a method for m, as follows:

```
λy:int.
  Λa#sum,n,m.
    λself:(a||sum:int||n:int||m:int).
      A(5)[a||m:int](self) || n=y || m=10
```

so that fix(C(3)[Empty]) makes sense, and is an object whose sum and n fields agree with B, and which also has an m field with value 10.

It is worth noting that the translations of the object-oriented programming idioms into our calculus are to be understood only informally since the target expression involves information not apparent in the source expression. Moreover, it seems reasonable to impose additional typing constraints on classes beyond those that would arise out of their representation in $\lambda^{||\mu}$. For example, it seems sensible to insist that the type ascribed to self be consistent with the type of the body of the class. This suggests that in a more complete development of these ideas, special syntax for class operations would be needed.

One major difference between this model of object-oriented programming and Wand's is that Wand's record concatenation operator is asymmetric while that of $\lambda^{||\mu}$ is symmetric. This difference represents an important methodological committment in the models of object oriented programming that arise from the two type systems.

Any language with multiple inheritance must address the issue of what happens when a class definition inherits the same method from two or more superclasses. One solution, adopted in many existing languages, is to choose one of the superclasses — say, the one mentioned furthest to the right in the subclass' definition — as the "principal superclass," resolving all conflicts in favor of this one. Wand's model implements this kind of conflict resolution scheme.

Another possibility, preferred by some designers of strongly typed object-oriented languages, is simply to disallow class definitions where the same method is inherited ambiguously from more than one superclass. When describing a class whose superclasses have overlapping sets of method names, the programmer must explicitly specify which superclass each ambigous method comes from. Our model can be used to implement this kind of conflict resolution strategy: the typing rules for the merge operator ensure that each method comes from exactly one superclass; the restriction operator is used to hide all but one of the inherited instances of a method.

3.3 F-Bounded Quantification

The ABEL group at HP-Labs has argued convincingly that "bounded quantification does not provide the same degree of flexibility in the presence of recursive types as it does for non-recursive types." [1, 7]. They propose an extended notion, called *F-bounded quantification*, where, rather than just pure bounded quantifiers of the form $\forall \texttt{a} \leq \texttt{r}.\ \texttt{t}$, they allow quantifiers of the form $\forall \texttt{a} \leq \texttt{F(a)}.\ \texttt{t}$, where F is a function from types to types.

In this section, we show that an analog of F-bounded quantification is *already* available in $\lambda^{||\mu}$.

In one class of situations, involving recursive record types where the recursion variable appears in negative positions, Canning et al. show that the pure type system of Cardelli and Wegner [6] does not allow functions to be applied to a variety of values for which they make semantic sense. For example, define the type

$$\texttt{PartialOrder} = \mu \texttt{po}.\ \{\texttt{leq} : \texttt{po} \rightarrow \texttt{Bool}\}$$

and assume we are given a function for computing the minimum of two values of any "subclass" of PartialOrder:

$$\texttt{min} \in \forall \texttt{a} \leq \texttt{PartialOrder}.\ \texttt{a}\rightarrow\texttt{a}\rightarrow\texttt{a}.$$

One of the types that we would like to be able to pass to min is

$$\texttt{Number} = \mu\,\texttt{num}.\ \{\texttt{leq} : \texttt{num}\rightarrow\texttt{Bool},\ \texttt{otherstuff} : \texttt{t}\}.$$

But by the usual rule for subtyping on recursive types,

$$\frac{\texttt{G}, \texttt{a} \leq \texttt{b} \vdash \texttt{s} \leq \texttt{t} \qquad \begin{array}{l}\texttt{a free only in s}\\ \texttt{b free only in t}\end{array}}{\texttt{G} \vdash \mu\texttt{a}.\ \texttt{s} \leq \mu\texttt{b}.\ \texttt{t}} \qquad \text{(LEQ-REC)}$$

it is not the case that Number $\leq$ PartialOrder.

F-bounded quantification can be used to redefine min so that it can be applied to elements of Number as well as PartialOrder. Define a type function

$$\texttt{FPartialOrder(t)} = \{\texttt{leq} : \texttt{t}\rightarrow\texttt{Bool}\}$$

and check that Number $\leq$ FPartialOrder(Number). Now write

$$\begin{array}{rcl}\texttt{min} & = & \Lambda\texttt{a} \leq \texttt{FPartialOrder(a)}.\\ & & \quad \lambda\texttt{x:a}.\ \lambda\texttt{y:a}.\\ & & \quad\quad \texttt{if x.leq(y) then x else y}\\ & \in & \forall\texttt{a} \leq \texttt{FPartialOrder(a)}.\ \texttt{a}\rightarrow\texttt{a}\rightarrow\texttt{a}.\end{array}$$

[*What if the instance of* a *is supposed to involve* b*?*]

To express min in $\lambda^{||}$ so that it can be applied to elements of Number, we write:

$$\begin{array}{l}\texttt{PartialOrder} = \mu\,\texttt{po}.\ \texttt{leq} : \texttt{po}\rightarrow\texttt{Bool}\\ \texttt{min} \in \forall\texttt{a\#leq}.\ \mu\,\texttt{b}.\ (\texttt{a} \,||\, \texttt{leq:b}\rightarrow\texttt{Bool})\\ \qquad\qquad \rightarrow \mu\,\texttt{b}.\ (\texttt{a} \,||\, \texttt{leq:b}\rightarrow\texttt{Bool})\\ \qquad\qquad \rightarrow \mu\,\texttt{b}.\ (\texttt{a} \,||\, \texttt{leq:b}\rightarrow\texttt{Bool})\\ \texttt{Number} = \mu\,\texttt{num}.\ (\texttt{leq:num}\rightarrow\texttt{Bool} \,||\, \texttt{otherstuff:t})\\ \texttt{five} \in \texttt{Number}.\end{array}$$

To type the application

$$\texttt{min five five}$$

we need to restrict away the leq field from Number:

$$\texttt{min [Number\leq] five five}.$$

The type application is well formed because Number\leq certainly lacks leq. To apply this term to five, we need to know that

$$\texttt{Number} \sim \mu\,\texttt{b}.\ (\texttt{Number\leq}) \,||\, \texttt{leq:b}\rightarrow\texttt{Bool},$$

that is

$$\texttt{Number} \sim \mu\,\texttt{b}.\ (\texttt{otherstuff:[Number/num]t}) \,||\, \texttt{leq:b}\rightarrow\texttt{Bool},$$

which may be readily verified using the proof technique suggested above. Specifically, let Number′ be the right-hand side of the above equation, and assume that Number $\sim$ Number′. To show that this is consistent, it suffices to unroll Number and Number′, and show that the corresponding fields of the resulting record types are equivalent. But this follows immediately from the assumption and the reflexivity of equivalence. The reasoning for the second application is identical.

The other class of situations where bounded quantification appears to be inadequate are those involving positive occurrences of the recursion variable in a recursively defined record type. Here again, pure bounded quantification prevents functions from being applied to a variety of values for which they make sense.

For example, define a recursive record type

$$\texttt{Movable} = \mu\texttt{mv}.\{\texttt{move} : \texttt{Real}\rightarrow\texttt{Real}\rightarrow\texttt{mv}\}$$

and a function

$$\texttt{translate} = \lambda\texttt{x:Movable}.\ \texttt{x.move}\ 1.0\ 1.0.$$

Since **translate** is intended to operate on values possessing any "subsclass" of "class" **Movable**, we want to show that it has type $\forall \mathtt{a} \leq \mathtt{Movable}.\ \mathtt{a} \to \mathtt{a}$. But it does not; the best that can be said for it is that it has type $\forall \mathtt{a} \leq \mathtt{Movable}.\ \mathtt{a} \to \mathtt{Movable}$, which is no better than $\mathtt{Movable} \to \mathtt{Movable}$. So if

$$\mathtt{Point} = \mu\,\mathtt{pnt}.\ \{\mathtt{x{:}void{\to}Real},\ \mathtt{y{:}void{\to}Real},\ \mathtt{move{:}Real{\to}Real{\to}pnt}\}$$

and

$$\mathtt{mypoint} \in \mathtt{Point},$$

then the application

$$\mathtt{translate\ mypoint}$$

will have type **Movable**, not **Point**.

Using F-bounded quantification, a **translate** function that will map elements of **Point** back into **Point** can be written as follows. Define the type function

$$\mathtt{FMovable(t)} = \{\mathtt{move} : \mathtt{Real} \to \mathtt{Real} \to \mathtt{t}\}.$$

and write

$$\begin{aligned}\mathtt{translate} \;=\; & \Lambda \mathtt{a} \leq \mathtt{FMovable(a)}.\\ & \lambda \mathtt{x{:}a}.\\ & \quad \mathtt{x.move}\ 1.0\ 1.0.\end{aligned}$$

Then since $\mathtt{Point} \leq \mathtt{FMovable(Point)}$, the application **translate** [Point] **mypoint** is well typed and has result type **Point**.

To express **translate** in $\lambda^{||}$ so that it can be applied to **mypoint**, we write:

$$\begin{aligned}\mathtt{translate} \;=\; & \Lambda \mathtt{a\#move}.\\ & \lambda \mathtt{x{:}}(\mu\,\mathtt{b}.\ \mathtt{a}||(\mathtt{move} : \mathtt{Real{\to}Real{\to}b})).\\ & \quad \mathtt{x.move}\ 1.01.0\\ \in\; & \forall \mathtt{a\#move}.\ (\mu\,\mathtt{b}.\ \mathtt{a}||(\mathtt{move} : \mathtt{Real{\to}Real{\to}b})) \to (\mu\,\mathtt{b}.\ \mathtt{a}||(\mathtt{move} : \mathtt{Real{\to}Real{\to}b})).\end{aligned}$$

Again, we apply **translate** to **mypoint** by first restricting away the **move** field of its type:

$$\mathtt{translate\ [Point \backslash move]\ mypoint}.$$

3.4 Language Extensions

This section shows how some other primitive record operations may be added to $\lambda^{||}$.

The "consistent update" operator $\mathtt{e}$ upd $\mathtt{l{=}e'}$ takes a record $\mathtt{e}$ with an $\mathtt{l}$ and replaces the value of this field with $\mathtt{e'}$, which must have exactly the same type as the present contents of the field. This operation cannot be expressed directly in $\lambda^{||}$, but it can be provided easily by adding a new typing rule:

$$\frac{\mathtt{T;G} \vdash \mathtt{e} \in \mathtt{r} \qquad \mathtt{r\_l}\downarrow \qquad \mathtt{T;G} \vdash \mathtt{e'} \in \mathtt{t} \qquad \mathtt{t} \sim \mathtt{r\_l}}{\mathtt{T;G} \vdash \mathtt{e\ upd\ l{=}e'} \in \mathtt{r}} \qquad \text{(WTT-UPD)}$$

Similarly, field renaming can be provided by adding two new rules:

$$\frac{\mathtt{T} \vdash \mathtt{r\ record} \qquad \mathtt{r\_l_1}\downarrow \qquad \mathtt{r\_l_2}\uparrow}{\mathtt{T;G} \vdash \mathtt{r}[\mathtt{l_1{\to}l_2}]\ \mathtt{record}} \qquad \text{(WFT-RENAME)}$$

$$\frac{\mathtt{r\_l_1}\downarrow \qquad \mathtt{r\_l_2}\uparrow}{\mathtt{T;G} \vdash \mathtt{r}[\mathtt{l_1{\to}l_2}] \sim (\mathtt{r} \backslash \mathtt{l_1})\ ||\ \mathtt{l_2{:}r\_l}} \qquad \text{(EQ-RENAME)}$$

At the level of values, syntactic sugar suffices for the field renaming operation:

$$\mathtt{e}[\mathtt{l_1{\to}l_2}] = \mathtt{e} \backslash \mathtt{l_1}\ ||\ \mathtt{l_1{=}(e.l_1)}.$$

The identity function on all records with *only* an $\mathtt{l}$ field requires ordinary type quantification:

$$\Lambda \mathtt{a}.\ \Lambda \mathtt{b\#(l{:}a)}.\ \lambda \mathtt{x{:}b}.\ \mathtt{x.l}.$$

4 Future Work

The research described here suggests a number of intriguing paths for future work. Among those that we would like to pursue are:

Record kinds: At one stage in the development of $\lambda^{||}$, we attempted to classify record types into *kinds* according to the record types with which they can safely be merged. Rather than "$\mathtt{r} \# \mathtt{R}$," we wrote "$\mathtt{r} : \mathrm{rec}\#\mathtt{R}$," meaning "$\mathtt{r}$ has the kind describing records compatible with the types in $\mathtt{R}$."

This reorganization has the advantage of allowing record type quantification to be unified cleanly with ordinary type quantification. Also, a form of subsumption — already implicit in $\lambda^{||}$ in the fact that a record type abstraction may be applied to a record type that happens to satisfy *more* constraints than those required by the quantifier — can be made explicit in this reformulation by introducing a "weaker than" ordering on record kinds.

In the end, the formulation used in this paper appeared to have a more more tractable proof theory, so we chose to stick with it for this stage of our investigation. But the elegance of the formulation in terms of kinds remains attractive.

Alternative primitives: The definition of $\lambda^{||}$ is strongly committed to a particular choice of primitives: symmetric merge and "checked" restriction, where a record can only be restricted at a field that it definitely possesses. Substituting a different formulation of either of these operators would require making some changes to the definition of the system, and probably major changes to the algorithms for typechecking. Nevertheless, we believe that the methodology used to develop $\lambda^{||}$ could be effectively applied to similar systems with different — perhaps more useful — sets of primitives.

Type reconstruction: The prenex fragment of $\lambda^{||}$, in which quantifiers are restricted to appear only outermost in type phrases, can be viewed as an explicitly typed version of Wand's calculus with row variables, with the difference that the merge operator is symmetric and the restriction operator is checked. From this analogy, we believe that the problem of finding a well-typed term whose erasure is the same as a given untyped term in this fragment (type reconstruction) may be decidable, though a type reconstruction algorithm would almost certainly have exponential complexity because, like Wand's, it would be inferring not principal types but finite sets of principal types.

Recursive types: In order to fully justify the examples inspired by F-bounded quantification in Section 3.3, the decidability of typechecking for $\lambda^{||\mu}$ needs to be established. Typechecking for polymorphic λ-calculi in the presence of μ-types is beginning to be understood, but the complex equivalence relation already defined on types appears to complicate matters substantially.

Notation for object-oriented programming: From the examples inspired by Wand in Section 3.2, it appears that a type system like this one could be used as the basis of a fairly expressive object-oriented programming language. It would be interesting to try to formalize the informal "class" notation used in these examples and describe a rigorous translation into $\lambda^{||\mu}$.

5 Conclusions

The decision to annotate quantifiers with purely positive information, with purely negative information, or with a mixture of positive and negative information is an important point of variation among calculi of record operations. Ordinary bounded quantification [6], F-bounded quantification [1], and the systems of Ohori and Buneman [17, 18] are positive-information systems. Cardelli and Mitchell's calculus [4, 5] and our earlier "symmetric system" [10] are mixed positive and negative. Wand's system of row variables [27] and $\lambda^{||}$ are pure negative-information systems.

The differences among these classes of systems are particularly clear in the presence of recursive types. In positive-information systems, one seems to need something like F-bounded quantification: in the case of our "symmetric system" this would be a something like a recursive "has" constraint. In the negative setting one does not need an analogue of F-bounded quantification, since one can explicitly quantify over

the "rest" of the fields in a record type. This leads to the observation that, in mixed systems like Cardelli and Mitchell's, the negative-information fragment can be used to directly express the examples motivating F-bounded quantification. Indeed, the construction given in Section 3.3 can be carried out almost verbatim in Cardelli and Mitchell's calculus.

6 Acknowledgements

We are grateful for productive discussions with Luca Cardelli and Didier Rémy.

A Complete Typing Rules

A.1 Judgement Forms

Well-formed type context:	$\mathtt{T}$ ok
Well-formed term context:	$\mathtt{T} \vdash \mathtt{G}$ ok
Well-formed type:	$\mathtt{T} \vdash \mathtt{t}$ type
Well-formed record type:	$\mathtt{T} \vdash \mathtt{r}$ record
Well-formed constraint list:	$\mathtt{T} \vdash \mathtt{R}$ ok
Constraint list satisfaction:	$\mathtt{T} \vdash \mathtt{r} \# \mathtt{R}$
Compatible types:	$\mathtt{T} \vdash \mathtt{r}_1 \# \mathtt{r}_2$
Equivalent types:	$\mathtt{t}_1 \sim \mathtt{t}_2$
Equivalent constraint sets:	$\mathtt{R}_1 \sim \mathtt{R}_2$
Well-formed term:	$\mathtt{T}\,;\mathtt{G} \vdash \mathtt{e} \in \mathtt{t}$

A.2 Well-formed type contexts

$$\diamond \text{ ok} \qquad \text{(WFTC-EMPTY)}$$

$$\frac{\mathtt{T} \vdash \mathtt{R} \text{ ok}}{\mathtt{T}, \mathtt{a}\#\mathtt{R} \text{ ok}} \qquad \text{(WFTC-TVAR)}$$

A.3 Well-formed constraint lists

$$\frac{\mathtt{T} \text{ ok}}{\mathtt{T} \vdash \diamond \text{ ok}} \qquad \text{(WFCL-NIL)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \text{ record} \qquad \mathtt{T} \vdash \mathtt{R} \text{ ok}}{\mathtt{T} \vdash \mathtt{r}, \mathtt{R} \text{ ok}} \qquad \text{(WFCL-CONS)}$$

A.4 Well-formed term contexts

$$\frac{\mathtt{T} \text{ ok}}{\mathtt{T} \vdash \diamond \text{ ok}} \qquad \text{(WFC-EMPTY)}$$

$$\frac{\mathtt{T} \vdash \mathtt{G} \text{ ok} \qquad \mathtt{T} \vdash \mathtt{t} \text{ type}}{\mathtt{T} \vdash \mathtt{G}, \mathtt{x}{:}\mathtt{t} \text{ ok}} \qquad \text{(WFC-VAR)}$$

A.5 Well-formed types:

$$\frac{\mathtt{T}\ \mathrm{ok}}{\mathtt{T} \vdash \mathtt{p}\ \mathrm{type}} \qquad \text{(WFT-PRIM)}$$

$$\frac{\mathtt{T} \vdash \mathtt{t}_1\ \mathrm{type} \qquad \mathtt{T} \vdash \mathtt{t}_2\ \mathrm{type}}{\mathtt{T} \vdash \mathtt{t}_1 \rightarrow \mathtt{t}_2\ \mathrm{type}} \qquad \text{(WFT-ARROW)}$$

$$\frac{\mathtt{T}, \mathtt{a}\#\mathtt{R} \vdash \mathtt{t}\ \mathrm{type}}{\mathtt{T} \vdash \forall \mathtt{a}\#\mathtt{R}.\ \mathtt{t}\ \ \mathrm{type}} \qquad \text{(WFT-ALL)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r}\ \mathrm{record}}{\mathtt{T} \vdash \mathtt{r}\ \mathrm{type}} \qquad \text{(WFT-REC)}$$

A.6 Well-formed record types

$$\frac{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2\ \mathrm{ok}}{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2 \vdash \mathtt{a}\ \mathrm{record}} \qquad \text{(WFR-VAR)}$$

$$\frac{\mathtt{T} \vdash \mathtt{t}\ \mathrm{type}}{\mathtt{T} \vdash \mathtt{l}{:}\mathtt{t}\ \mathrm{record}} \qquad \text{(WFR-BASE)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r}\ \mathrm{record} \qquad \mathtt{r}\_\mathtt{l}\downarrow}{\mathtt{T} \vdash \mathtt{r}\backslash\mathtt{l}\ \mathrm{record}} \qquad \text{(WFR-RESTR)}$$

$$\frac{\mathtt{T}\ \mathrm{ok}}{\mathtt{T} \vdash \mathbf{Empty}\ \mathrm{record}} \qquad \text{(WFR-EMPTY)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r}_1 \# \mathtt{r}_2}{\mathtt{T} \vdash \mathtt{r}_1 \,||\, \mathtt{r}_2\ \mathrm{record}} \qquad \text{(WFR-MERGE)}$$

A.7 Constraint list satisfaction

$$\frac{\mathtt{T} \vdash \mathtt{r}\ \mathrm{record}}{\mathtt{T} \vdash \mathtt{r} \# \diamond} \qquad \text{(CMP-LIST-NIL)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{r}_i \qquad \mathtt{T} \vdash \mathtt{r} \# \mathtt{R}}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{r}_i, \mathtt{R}} \qquad \text{(CMP-LIST-CONS)}$$

A.8 Compatibility

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{s} \qquad \mathtt{r} \sim \mathtt{r}' \qquad \mathtt{s} \sim \mathtt{s}'}{\mathtt{T} \vdash \mathtt{r}' \# \mathtt{s}'} \qquad \text{(CMP-EQ)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{s}}{\mathtt{T} \vdash \mathtt{s} \# \mathtt{r}} \qquad \text{(CMP-SYMM)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{l}{:}\mathtt{t} \qquad \mathtt{T} \vdash \mathtt{t}'\ \mathrm{type}}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{l}{:}\mathtt{t}'} \qquad \text{(CMP-BASE)}$$

$$\frac{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2\ \mathrm{ok} \qquad \mathtt{r}_i \in \mathtt{R}}{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2 \vdash \mathtt{a} \# \mathtt{r}_i} \qquad \text{(CMP-TVAR)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \# (\mathtt{s}_1 || \mathtt{s}_2)}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{s}_i} \qquad \text{(CMP-MERGE-E)}$$

$$\frac{\mathtt{T} \vdash \mathtt{s}_1 \# \mathtt{s}_2 \qquad \mathtt{T} \vdash \mathtt{r} \# \mathtt{s}_1 \qquad \mathtt{T} \vdash \mathtt{r} \# \mathtt{s}_2}{\mathtt{T} \vdash \mathtt{r} \# (\mathtt{s}_1 || \mathtt{s}_2)} \quad \text{(CMP-MERGE-I)}$$

$$\frac{\mathtt{l} \neq \mathtt{l}' \qquad \mathtt{T} \vdash \mathtt{l}{:}\mathtt{t}\ \text{record} \qquad \mathtt{T} \vdash \mathtt{l}'{:}\mathtt{t}'\ \text{record}}{\mathtt{T} \vdash \mathtt{l}{:}\mathtt{t} \# \mathtt{l}'{:}\mathtt{t}'} \quad \text{(CMP-BASE/BASE)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r}\ \text{record}}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{Empty}} \quad \text{(CMP-EMPTY)}$$

A.9 Constraint list equivalence

$$\mathtt{R} \sim \mathtt{R} \quad \text{(CEQ-REFL)}$$

$$\frac{\mathtt{R} \sim \mathtt{R}'}{\mathtt{R}' \sim \mathtt{R}} \quad \text{(CEQ-SYMM)}$$

$$\frac{\mathtt{R} \sim \mathtt{R}' \qquad \mathtt{R}' \sim \mathtt{R}''}{\mathtt{R} \sim \mathtt{R}''} \quad \text{(CEQ-TRANS)}$$

$$\frac{\mathtt{R} \sim \mathtt{R}' \qquad \mathtt{r} \sim \mathtt{r}'}{\mathtt{r},\mathtt{R} \sim \mathtt{r}',\mathtt{R}'} \quad \text{(CEQ-INNER)}$$

$$\mathtt{r},(\mathtt{r}',\mathtt{R}) \sim \mathtt{r}',(\mathtt{r},\mathtt{R}) \quad \text{(CEQ-SWAP)}$$

$$\mathtt{Empty},\mathtt{R} \sim \mathtt{R} \quad \text{(CEQ-EMPTY)}$$

$$(\mathtt{r}_1||\mathtt{r}_2),\mathtt{R} \sim \mathtt{r}_1,(\mathtt{r}_2,\mathtt{R}) \quad \text{(CEQ-MERGE)}$$

$$\mathtt{r},(\mathtt{r},\mathtt{R}) \sim \mathtt{r},\mathtt{R} \quad \text{(CEQ-DUPL)}$$

$$\mathtt{l}{:}\mathtt{t},\mathtt{R} \sim \mathtt{l}{:}\mathtt{t}',\mathtt{R} \quad \text{(CEQ-BASE)}$$

A.10 Type equivalence

$$\mathtt{r}||\mathtt{Empty} \sim \mathtt{r} \quad \text{(EQ-MERGE-UNIT)}$$

$$\mathtt{r}_1 \,||\, (\mathtt{r}_2||\mathtt{r}_3) \sim (\mathtt{r}_1||\mathtt{r}_2) \,||\, \mathtt{r}_3 \quad \text{(EQ-MERGE-ASSOC)}$$

$$\mathtt{r}_1 \,||\, \mathtt{r}_2 \sim \mathtt{r}_2 \,||\, \mathtt{r}_1 \quad \text{(EQ-MERGE-COMM)}$$

$$\mathtt{r}\backslash\mathtt{l}\backslash\mathtt{l}' \sim \mathtt{r}\backslash\mathtt{l}'\backslash\mathtt{l} \quad \text{(EQ-RESTR/RESTR')}$$

$$(\mathtt{l}{:}\mathtt{t})\backslash\mathtt{l} \sim \mathtt{Empty} \quad \text{(EQ-BASE/RESTR)}$$

$$\frac{\mathtt{r}_1 \,.\, \mathtt{l}\downarrow}{(\mathtt{r}_1||\mathtt{r}_2)\backslash\mathtt{l} \sim (\mathtt{r}_1\backslash\mathtt{l}||\mathtt{r}_2)} \quad \text{(EQ-MERGE/RESTR)}$$

A.11 Type equivalence – congruence rules:

$$\mathtt{t} \sim \mathtt{t} \qquad \text{(Eq-Refl)}$$

$$\frac{\mathtt{t}' \sim \mathtt{t}}{\mathtt{t} \sim \mathtt{t}'} \qquad \text{(Eq-Symm)}$$

$$\frac{\mathtt{t} \sim \mathtt{t}' \qquad \mathtt{t}' \sim \mathtt{t}''}{\mathtt{t} \sim \mathtt{t}''} \qquad \text{(Eq-Trans)}$$

$$\frac{\mathtt{t}_1 \sim \mathtt{t}_1' \qquad \mathtt{t}_2 \sim \mathtt{t}_2'}{\mathtt{t}_1 \to \mathtt{t}_2 \sim \mathtt{t}_1' \to \mathtt{t}_2'} \qquad \text{(Eq-Cong-Arrow)}$$

$$\frac{\mathtt{R} \sim \mathtt{R}' \qquad \mathtt{t} \sim \mathtt{t}'}{\forall \mathtt{a}\#\mathtt{R}.\ \mathtt{t} \sim \forall \mathtt{a}\#\mathtt{R}'.\ \mathtt{t}'} \qquad \text{(Eq-Cong-All)}$$

$$\frac{\mathtt{t} \sim \mathtt{t}'}{\mathtt{l}{:}\mathtt{t} \sim \mathtt{l}{:}\mathtt{t}'} \qquad \text{(Eq-Cong-Base)}$$

$$\frac{\mathtt{r} \sim \mathtt{r}}{\mathtt{r}\backslash\mathtt{l} \sim \mathtt{r}'\backslash\mathtt{l}} \qquad \text{(Eq-Cong-Restr)}$$

$$\frac{\mathtt{r}_1 \sim \mathtt{r}_1' \qquad \mathtt{r}_2 \sim \mathtt{r}_2'}{\mathtt{r}_1 \,||\, \mathtt{r}_2 \sim \mathtt{r}_1' \,||\, \mathtt{r}_2'} \qquad \text{(Eq-Cong-Merge)}$$

A.12 Well-typed terms

$$\frac{\mathtt{T};\mathtt{G} \vdash \mathtt{e} \in \mathtt{t} \qquad \mathtt{T} \vdash \mathtt{t}' \text{ type} \qquad \mathtt{t} \sim \mathtt{t}'}{\mathtt{T};\mathtt{G} \vdash \mathtt{e} \in \mathtt{t}'} \qquad \text{(Wtt-Eq)}$$

$$\frac{\mathtt{T} \vdash \mathtt{G}_1, \mathtt{x}{:}\mathtt{t}, \mathtt{G}_2 \text{ ok}}{\mathtt{T};\mathtt{G}_1, \mathtt{x}{:}\mathtt{t}, \mathtt{G}_2 \vdash \mathtt{x} \in \mathtt{t}} \qquad \text{(Wtt-Var)}$$

$$\frac{\mathtt{T};\mathtt{G}, \mathtt{x}{:}\mathtt{t} \vdash \mathtt{e} \in \mathtt{t}'}{\mathtt{T};\mathtt{G} \vdash \lambda\mathtt{x}{:}\mathtt{t}.\ \mathtt{e} \in \mathtt{t} \to \mathtt{t}'} \qquad \text{(Wtt-Arrow-I)}$$

$$\frac{\mathtt{T};\mathtt{G} \vdash \mathtt{e}_1 \in \mathtt{t} \to \mathtt{t}' \qquad \mathtt{T};\mathtt{G} \vdash \mathtt{e}_2 \in \mathtt{t}}{\mathtt{T};\mathtt{G} \vdash \mathtt{e}_1\ \mathtt{e}_2 \in \mathtt{t}'} \qquad \text{(Wtt-Arrow-E)}$$

$$\frac{\mathtt{T};\mathtt{G} \vdash \mathtt{e} \in \mathtt{t}}{\mathtt{T};\mathtt{G} \vdash \mathtt{l}{=}\mathtt{e} \in \mathtt{l}{:}\mathtt{t}} \qquad \text{(Wtt-Base)}$$

$$\frac{\mathtt{T};\mathtt{G} \vdash \mathtt{e} \in \mathtt{r} \qquad \mathtt{r\_l}\downarrow}{\mathtt{T};\mathtt{G} \vdash \mathtt{e}\backslash\mathtt{l} \in \mathtt{r}\backslash\mathtt{l}} \qquad \text{(Wtt-Restr)}$$

$$\frac{\mathtt{T} \vdash \mathtt{G} \text{ ok}}{\mathtt{T};\mathtt{G} \vdash \mathtt{empty} \in \mathtt{Empty}} \qquad \text{(Wtt-Empty)}$$

$$\frac{\mathtt{T};\mathtt{G} \vdash \mathtt{e}_1 \in \mathtt{r}_1 \qquad \mathtt{T};\mathtt{G} \vdash \mathtt{e}_2 \in \mathtt{r}_2 \qquad \mathtt{T} \vdash \mathtt{r}_1 \,\#\, \mathtt{r}_2}{\mathtt{T};\mathtt{G} \vdash \mathtt{e}_1 || \mathtt{e}_2 \in \mathtt{r}_1 || \mathtt{r}_2} \qquad \text{(Wtt-Merge)}$$

$$\frac{\mathtt{T};\mathtt{G} \vdash \mathtt{e} \in \mathtt{r} \qquad \mathtt{r\_l}\downarrow}{\mathtt{T};\mathtt{G} \vdash \mathtt{e.l} \in \mathtt{r\_l}} \qquad \text{(Wtt-Select)}$$

$$\frac{\mathtt{T}, \mathtt{a}\#\mathtt{R};\mathtt{G} \vdash \mathtt{e} \in \mathtt{t}}{\mathtt{T};\mathtt{G} \vdash \Lambda\mathtt{a}\#\mathtt{R}.\mathtt{e} \in \forall\mathtt{a}\#\mathtt{R}.\ \mathtt{t}} \qquad \text{(Wtt-All-I)}$$

$$\frac{\mathtt{T};\mathtt{G} \vdash \mathtt{e} \in \forall\mathtt{a}\#\mathtt{R}.\ \mathtt{t} \qquad \mathtt{T} \vdash \mathtt{r}\,\#\,\mathtt{R}}{\mathtt{T};\mathtt{G} \vdash \mathtt{e}[\mathtt{r}] \in [\mathtt{r}/\mathtt{a}]\mathtt{t}} \qquad \text{(Wtt-All-E)}$$

B Complete Typechecking Rules (Algorithmic Version)

B.1 Judgement forms

Well-formed type:	$\mathtt{T} \vdash \mathtt{t} \Rightarrow \text{type}$
Well-formed record type:	$\mathtt{T} \vdash \mathtt{r} \Rightarrow \text{record}$
Well-formed constraint list:	$\mathtt{T} \vdash \mathtt{R} \Rightarrow \text{ok}$
Constraint list satisfaction:	$\mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} \mathtt{R}$
Compatible types:	$\mathtt{T} \vdash \mathtt{r}_1 \#^{\Rightarrow} \mathtt{r}_2$
Well-formed term:	$\mathtt{T}\,;\mathtt{G} \vdash \mathtt{e} \Rightarrow \mathtt{t}$

B.2 Type and constraint list normalization

$$\mathtt{Empty} \,||\, \mathtt{r} \twoheadrightarrow^1 \mathtt{r} \qquad \text{(A-Red-Empty-1)}$$

$$\mathtt{r} \,||\, \mathtt{Empty} \twoheadrightarrow^1 \mathtt{r} \qquad \text{(A-Red-Empty-2)}$$

$$\mathtt{l{:}t} \backslash \mathtt{l} \twoheadrightarrow^1 \mathtt{Empty} \qquad \text{(A-Red-Restr/Base)}$$

$$(\mathtt{r} \,||\, \mathtt{s}) \backslash \mathtt{l} \twoheadrightarrow^1 (\mathtt{r} \backslash \mathtt{l}) \,||\, \mathtt{s} \quad \text{if } \mathtt{r\_l}\downarrow \qquad \text{(A-Red-Restr/Merge-1)}$$

$$(\mathtt{r} \,||\, \mathtt{s}) \backslash \mathtt{l} \twoheadrightarrow^1 \mathtt{r} \,||\, (\mathtt{s} \backslash \mathtt{l}) \quad \text{if } \mathtt{s\_l}\downarrow \qquad \text{(A-Red-Restr/Merge-2)}$$

$$\rho || \sigma, \mathtt{R} \twoheadrightarrow^1 \rho, \sigma, \mathtt{R} \qquad \text{(A-Cred-Merge)}$$

$$\mathtt{Empty}, \mathtt{R} \twoheadrightarrow^1 \mathtt{R} \qquad \text{(A-Cred-Empty)}$$

$$\mathtt{l{:}t}, \mathtt{R} \twoheadrightarrow^1 \mathtt{l}, \mathtt{R} \qquad \text{(A-Cred-Base)}$$

$$\mathtt{r} \backslash \mathtt{l} \backslash \mathtt{l}' \approx^1 \mathtt{r} \backslash \mathtt{l}' \backslash \mathtt{l} \qquad \text{(A-Eq-Restr/Restr')}$$

$$\mathtt{r} \,||\, \mathtt{s} \approx^1 \mathtt{s} \,||\, \mathtt{r} \qquad \text{(A-Eq-Comm)}$$

$$\mathtt{r} \,||\, (\mathtt{s} \,||\, \mathtt{t}) \approx^1 (\mathtt{r} \,||\, \mathtt{s}) \,||\, \mathtt{t} \qquad \text{(A-Eq-Assoc)}$$

$$\phi, \phi, \mathtt{R} \approx^1 \phi, \mathtt{R} \qquad \text{(A-Ceq-Dupl)}$$

$$\rho, \sigma, \mathtt{R} \approx^1 \sigma, \rho, \mathtt{R} \qquad \text{(A-Ceq-Comm)}$$

B.3 Well-formed constraint lists

$$\frac{\mathtt{T} \Rightarrow \text{ok}}{\mathtt{T} \vdash \diamond \Rightarrow \text{ok}} \qquad \text{(A-Wfcl-Nil)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \Rightarrow \text{record} \qquad \mathtt{T} \vdash \mathtt{R} \Rightarrow \text{ok}}{\mathtt{T} \vdash \mathtt{r}, \mathtt{R} \Rightarrow \text{ok}} \qquad \text{(A-Wfcl-Cons)}$$

B.4 Well-formed types

$$\mathtt{T} \vdash \mathtt{p} \Rightarrow \text{type} \qquad \text{(A-WFT-PRIM)}$$

$$\frac{\mathtt{T} \vdash \mathtt{t}_1 \Rightarrow \text{type} \quad \mathtt{T} \vdash \mathtt{t}_2 \Rightarrow \text{type}}{\mathtt{T} \vdash \mathtt{t}_1 \rightarrow \mathtt{t}_2 \Rightarrow \text{type}} \qquad \text{(A-WFT-ARROW)}$$

$$\frac{\mathtt{T}, \mathtt{a}\#\mathtt{R} \vdash \mathtt{t} \Rightarrow \text{type} \quad \mathtt{T} \vdash \mathtt{R} \Rightarrow \text{ok}}{\mathtt{T} \vdash \forall \mathtt{a}\#\mathtt{R}.\, \mathtt{t} \Rightarrow \text{type}} \qquad \text{(A-WFT-ALL)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \Rightarrow \text{record}}{\mathtt{T} \vdash \mathtt{r} \Rightarrow \text{type}} \qquad \text{(A-WFT-REC)}$$

B.5 Well-formed record types

$$\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2 \vdash \mathtt{a} \Rightarrow \text{record} \qquad \text{(A-WFR-VAR)}$$

$$\frac{\mathtt{T} \vdash \mathtt{t} \Rightarrow \text{type}}{\mathtt{T} \vdash \mathtt{l}{:}\mathtt{t} \Rightarrow \text{record}} \qquad \text{(A-WFR-BASE)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \Rightarrow \text{record} \quad \mathtt{r}.\mathtt{l}\downarrow}{\mathtt{T} \vdash \mathtt{r}\backslash\mathtt{l} \Rightarrow \text{record}} \qquad \text{(A-WFR-RESTR)}$$

$$\mathtt{T} \vdash \mathtt{Empty} \Rightarrow \text{record} \qquad \text{(A-WFR-EMPTY)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r}_1 \Rightarrow \text{record} \quad \mathtt{T} \vdash \mathtt{r}_2 \Rightarrow \text{record} \quad \mathtt{T} \vdash \mathtt{r}_1 \#\!\!\Rightarrow \mathtt{r}_2}{\mathtt{T} \vdash \mathtt{r}_1 \,||\, \mathtt{r}_2 \Rightarrow \text{record}} \qquad \text{(A-WFR-MERGE)}$$

B.6 Constraint list satisfaction

$$\frac{\mathtt{T} \vdash \mathtt{r} \Rightarrow \text{record}}{\mathtt{T} \vdash \mathtt{r} \#\!\!\Rightarrow \diamond} \qquad \text{(A-CMP-LIST-NIL)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \#\!\!\Rightarrow \mathtt{r}_i \quad \mathtt{T} \vdash \mathtt{r} \#\!\!\Rightarrow \mathtt{R}}{\mathtt{T} \vdash \mathtt{r} \#\!\!\Rightarrow \mathtt{r}_i, \mathtt{R}} \qquad \text{(A-CMP-LIST-CONS)}$$

B.7 Compatibility

$$\mathtt{T} \vdash \mathtt{r} \#\!\!\Rightarrow \mathtt{Empty} \qquad \text{(A-CMP-EMPTY-1)}$$

$$\mathtt{T} \vdash \mathtt{Empty} \#\!\!\Rightarrow \mathtt{r} \qquad \text{(A-CMP-EMPTY-2)}$$

$$\frac{\mathtt{T} \vdash \mathtt{s}_1 \#\!\!\Rightarrow \mathtt{r} \quad \mathtt{T} \vdash \mathtt{s}_2 \#\!\!\Rightarrow \mathtt{r}}{\mathtt{T} \vdash (\mathtt{s}_1 \,||\, \mathtt{s}_2) \#\!\!\Rightarrow \mathtt{r}} \qquad \text{(A-CMP-MERGE-I-1)}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \#\!\!\Rightarrow \mathtt{s}_1 \quad \mathtt{T} \vdash \mathtt{r} \#\!\!\Rightarrow \mathtt{s}_2}{\mathtt{T} \vdash \mathtt{r} \#\!\!\Rightarrow (\mathtt{s}_1 \,||\, \mathtt{s}_2)} \qquad \text{(A-CMP-MERGE-I-2)}$$

$$\frac{\mathtt{l} \neq \mathtt{l}'}{\mathtt{T} \vdash \mathtt{l}{:}\mathtt{t} \#\!\!\Rightarrow \mathtt{l}'{:}\mathtt{t}'} \qquad \text{(A-CMP-BASE/BASE)}$$

$$\frac{\mathtt{a} \in \mathtt{S}^*}{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2, \mathtt{b}\#\mathtt{S}, \mathtt{T}_3 \vdash \mathtt{a} \#\!\!\Rightarrow \mathtt{b}} \qquad \text{(A-CMP-TVAR/TVAR-1)}$$

$$\frac{a \in S^*}{T_1, a\#R, T_2, b\#S, T_3 \vdash b \#^\Rightarrow a} \quad \text{(A-Cmp-Tvar/Tvar-2)}$$

$$\frac{l \in R^*}{T_1, a\#R, T_2 \vdash a \#^\Rightarrow l{:}t} \quad \text{(A-Cmp-Base/Tvar-1)}$$

$$\frac{l \in R^*}{T_1, a\#R, T_2 \vdash l{:}t \#^\Rightarrow a} \quad \text{(A-Cmp-Base/Tvar-2)}$$

B.8 Well-typed terms

$$T\,;\,G_1, x{:}t, G_2 \vdash x \Rightarrow t \quad \text{(A-Wtt-Var)}$$

$$\frac{T \vdash t \Rightarrow \text{type} \qquad T\,;\,G, x{:}t \vdash e \Rightarrow t'}{T\,;\,G \vdash \lambda x{:}t.\ e \Rightarrow t \to t'} \quad \text{(A-Wtt-Arrow-I)}$$

$$\frac{T\,;\,G \vdash e_1 \Rightarrow t \to t' \qquad T\,;\,G \vdash e_2 \Rightarrow t'' \qquad t'' \equiv t}{T\,;\,G \vdash e_1\ e_2 \Rightarrow t'} \quad \text{(A-Wtt-Arrow-E)}$$

$$\frac{T\,;\,G \vdash e \Rightarrow t}{T\,;\,G \vdash l{=}e \Rightarrow l{:}t} \quad \text{(A-Wtt-Base)}$$

$$\frac{T\,;\,G \vdash e \Rightarrow r \qquad r.l\!\downarrow}{T\,;\,G \vdash e\backslash l \Rightarrow r\backslash l} \quad \text{(A-Wtt-Restr)}$$

$$T\,;\,G \vdash \mathbf{empty} \Rightarrow \mathbf{Empty} \quad \text{(A-Wtt-Empty)}$$

$$\frac{T\,;\,G \vdash e_1 \Rightarrow r_1 \qquad T\,;\,G \vdash e_2 \Rightarrow r_2 \qquad T \vdash r_1^* \#^\Rightarrow r_2^*}{T\,;\,G \vdash e_1 \,||\, e_2 \Rightarrow r_1 \,||\, r_2} \quad \text{(A-Wtt-Merge)}$$

$$\frac{T\,;\,G \vdash e \Rightarrow r \qquad r.l\!\downarrow}{T\,;\,G \vdash e.l \Rightarrow r.l} \quad \text{(A-Wtt-Select)}$$

$$\frac{T \vdash R\ \mathbf{ok} \qquad T, a\#R\,;\,G \vdash e \Rightarrow t}{T\,;\,G \vdash \Lambda a\#R.\ e \Rightarrow \forall a\#R.\ t} \quad \text{(A-Wtt-All-I)}$$

$$\frac{T\,;\,G \vdash e \Rightarrow \forall a\#R.\ t \qquad T \vdash r \Rightarrow \text{record} \qquad T \vdash r^* \#^\Rightarrow R^*}{T\,;\,G \vdash e[r] \Rightarrow [r/a]t} \quad \text{(A-Wtt-All-E)}$$

References

[1] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[2] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984.

[3] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[4] Luca Cardelli and John Mitchell. Operations on records. In *Proceedings of Fifth International Conference on Mathematical Foundations of Programming Language Semantics*, Tulane University, New Orleans, March 1989. To appear.

[5] Luca Cardelli and John C. Mitchell. Operations on records. Research report 48, Digital Equipment Corporation, Systems Research Center, August 1989.

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[7] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, January 1990.

[8] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM SIGPLAN/SIGACT, 1982.

[9] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[10] Robert W. Harper and Benjamin C. Pierce. Extensible records without subsumption. Technical Report CMU-CS-90-102, School of Computer Science, Carnegie Melon University, Feburary 1990.

[11] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery*, 27(4):797–821, October 1980.

[12] Lalita A. Jategaonkar. ML with extended pattern matching and subtypes. Master's thesis, MIT, August 1989.

[13] Lalita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

[14] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[15] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[16] John C. Mitchell. Coercion and type inference (summary). In *Proc. 11th ACM Symp. on Principles of Programming Languages*, pages 175–185, January 1984.

[17] Atsushi Ohori and Peter Buneman. Static type inference for parametric classes. In *OOPSLA '89: Object-Oriented Programming Systems, Languages, and Applications, Conference Proceedings*, pages 445–456, October 1989.

[18] Atsushi Ohori and Peter Buneman. Type inference in a database programming language. In *1988 ACM Conference on Lisp and Functional Programming*, pages 174–183, Snowbird, Utah, July 1989. Revised version, September, 1988.

[19] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin*, pages 242–249. ACM, January 1989.

[20] Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels.* PhD thesis, Université Paris VII, 1990.

[21] Didier Rémy. Typechecking records in a natural extension of ML. Submitted to TOPLAS, June 1990.

[22] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[23] Ryan Stansifer. Type inference with subtypes. In *Proceedings of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 88–97, San Diego, CA, January 1988.

[24] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.

[25] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

[26] Mitchell Wand. Type inference for objects with instance variables and inheritance. Technical Report NU-CCS-89-2, College of Computer Science, Northeastern University, 1989.

[27] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92–97, Pacific Grove, CA, June 1989.