A Record Calculus Based on Symmetric Concatenation

Robert Harper Benjamin Pierce
Carnegie Mellon University

Abstract

Type systems for operations on extensible records form a foundation for statically typed languages addressing some aspects of object oriented programming and database applications. A number of primitive operations have been proposed: extending a record with a new field, overwriting an existing field, removing a field, and various kinds of concatenation. We show here that a record calculus based on a symmetric concatenation operator, where two records may be concatenated only if they have no overlapping fields, also captures the types of many other useful primitive record operations. "Mergeability constraints" are expressed directly using explicit annotations on type variables and constrained second-order type quantification instead of a rule of subsumption; we argue that the resulting system is more straightforward than subsumption-based alternatives.

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*:

$$\frac{\mathtt{G} \vdash \mathtt{e} \in \mathtt{t} \qquad \mathtt{G} \vdash \mathtt{t} \leq \mathtt{t}'}{\mathtt{G} \vdash \mathtt{e} \in \mathtt{t}'}$$

Cardelli and Wegner [6] extended this idea to a powerful second-order type system combining Cardelli's ordering on record types with type quantification [9, 22] using techniques developed by Mitchell [16]. Wand [24, 25] analyzed the concept of record polymorphism 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 [13, 14] 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 extension 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 $\backslash$ 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 [11], we set out to represent positive and negative information as directly as possible, using explicit constraints on record type variables rather than encoding constraints in a preorder structure on types. For example, this system expresses the well-typedness constraint on $\mathtt{e|l{=}e'}$ 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 rule of subsumption, where genericity over record types arises solely from *constrained quantification* of type variables: a quantified type "$\forall\mathtt{a}$ lacks $\mathtt{L^-}$ has $\mathtt{L^+{:}T^+.t}$" can be instantiated with any record type $\mathtt{r}$ such that $\mathtt{r}$'s set of fields is disjoint from $\mathtt{L^-}$ and includes each $\mathtt{l_i{:}t_i}$ in $\mathtt{L^+{:}T^+}$. This line of development essentially amounts to reverse-engineering Cardelli and Mitchell's system back toward 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 records are provided by the *merge* or *concatenate* operator $||$, which combines the fields of two existing records. There are several forms of the merge operator, distinguished by what happens when the records $\mathtt{e_1}$ and $\mathtt{e_2}$ in a merge expression $\mathtt{e_1||e_2}$ 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, $\mathtt{\{l_1{:}t_1,\ l_2{:}t_2\}\ ||\ \{l_3{:}t_3,\ l_4{:}t_4\}}$ can be rewritten simply as $\mathtt{\{l_1{:}t_1,\ l_2{:}t_2,\ l_3{:}t_3,\ l_4{:}t_4\}}$. But in general — when the types of $\mathtt{e_1}$ and $\mathtt{e_2}$ may involve variables — typechecking with $||$ requires a substantial increase in the complexity of the type system. To ensure that $\lambda\mathtt{x{:}a}.\ \lambda\mathtt{y{:}b}.\ \mathtt{(x||y)}$ is well typed, we need to guarantee that the type variables $\mathtt{a}$ and $\mathtt{b}$ 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 $\mathtt{r_1}$ and $\mathtt{r_2}$ to be compatible, written $\mathtt{r_1} \# \mathtt{r_2}$. The merge expression $\mathtt{e_1||e_2}$ is then well typed when $\mathtt{e_1} \in \mathtt{r_1}$, $\mathtt{e_2} \in \mathtt{r_2}$, and $\mathtt{r_1} \# \mathtt{r_2}$. The type of this expression is the *merge type* $\mathtt{r_1||r_2}$.

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, leading to problems in type reconstruction.

Ohori and Buneman studied type inference for object-oriented programming [17] and for database programming languages [18] — another promising application area for record type systems. In their work on database programming languages they consider a type system including records and sets, with operations such as "relational join" chosen to support database applications. To perform ML-like type inference for this language, they introduce 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 pertinant 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 consider extensions to support object-oriented programming constructs similar to those suggested by Wand. In order to support inheritance, they introduce an *ad hoc* form of subsumption in connection with "self" and an additional form of condition on conditional type schemes.

Cardelli and Mitchell sketched several increasingly ambitious formulations of recursive record concatenation as extensions to their calculus of operations on records [5]. 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 $\mathtt{x_1}$ and $\mathtt{x_2}$, where $\mathtt{x_1}$ has at least a field $\mathtt{l_1}$ of type $\mathtt{Int}$ and $\mathtt{x_2}$ has at least a field $\mathtt{l_2}$ of type $\mathtt{Int}$, and returns the result of merging $\mathtt{x_1}$ and $\mathtt{x_2}$, can be written

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

(altering their concrete syntax slightly).

In the present paper, we study a record calculus $\lambda^{||}$ based on a 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 Λa#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 above can be defined in $\lambda^{||}$ as follows:

$\Lambda a_1\#l_1{:}Int, l_2{:}Int.\ \Lambda a_2\#a_1, l_1{:}Int, l_2{:}Int.$
$\lambda x_1{:}a_1||l_1{:}Int.\ \lambda x_2{:}a_2||l_2{:}Int.\ x_1 \,||\, x_2.$

Similarly, a function that accepts any record not already possessing an l field and adds the field l=5 can be written:

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

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

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

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

These examples underscore 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.

The use of constrained type variables here 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 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.

In Section 2, we define the syntax and typing rules of $\lambda^{||}$ and briefly sketch a proof of the decidability of type-checking. Section 3 illustrates the expressiveness of the system by translating an object-oriented programming example from Wand and showing how to encode one of the motivating examples for F-bounded quantification in $\lambda^{||}$. Section 4 offers concluding remarks. A complete listing of the typing rules for $\lambda^{||}$ appears in Appendix A. The full version of the paper [10] includes more detailed proofs, several additional examples, and a discussion of avenues for future research.

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

2.1 Syntax

This section introduces some notational conventions and defines the concrete syntax of $\lambda^{||}$.

The metavariable t and u range over types; r, and s range over record types; p ranges over primitive types; a and b range over record type variables; R and S range over finite sequences of record types; e and f range over terms; x ranges over variables; l ranges over field labels.

Types:

t	::=	p	primitive
	|	$t_1 \rightarrow t_2$	function space
	|	∀a#R. t	constrained quantification
	|	r	record type

Record types:

r	::=	a	record type variable
	|	Empty	empty record
	|	l:t	single-field record
	|	r\l	restriction
	|	$r_1 \,||\, r_2$	merge

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.

Terms:

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

Free and bound variables are defined in the usual way; in the case of type quantification and abstraction, the

$$\frac{\mathtt{T_1, a\#R, T_2}\ \text{ok}}{\mathtt{T_1, a\#R, T_2} \vdash \mathtt{a}\ \text{record}}$$

$$\frac{\mathtt{T} \vdash \mathtt{t}\ \text{type}}{\mathtt{T} \vdash \mathtt{l{:}t}\ \text{record}}$$

$$\frac{\mathtt{T} \vdash \mathtt{r}\ \text{record} \qquad \mathtt{r\_l}\downarrow}{\mathtt{T} \vdash \mathtt{r\backslash l}\ \text{record}}$$

$$\frac{\mathtt{T}\ \text{ok}}{\mathtt{T} \vdash \mathtt{Empty}\ \text{record}}$$

$$\frac{\mathtt{T} \vdash \mathtt{r_1} \# \mathtt{r_2}}{\mathtt{T} \vdash \mathtt{r_1} \,||\, \mathtt{r_2}\ \text{record}}$$

Figure 1: Selected formation rules for record types

$$\frac{\mathtt{T\,;G} \vdash \mathtt{e} \in \mathtt{t}}{\mathtt{T\,;G} \vdash \mathtt{l{=}e} \in \mathtt{l{:}t}}$$

$$\frac{\mathtt{T\,;G} \vdash \mathtt{e} \in \mathtt{r} \qquad \mathtt{r\_l}\downarrow}{\mathtt{T\,;G} \vdash \mathtt{e\backslash l} \in \mathtt{r\backslash l}}$$

$$\frac{\mathtt{T} \vdash \mathtt{G}\ \text{ok}}{\mathtt{T\,;G} \vdash \mathtt{empty} \in \mathtt{Empty}}$$

$$\frac{\begin{array}{c}\mathtt{T\,;G} \vdash \mathtt{e_1} \in \mathtt{r_1}\\ \mathtt{T\,;G} \vdash \mathtt{e_2} \in \mathtt{r_2}\\ \mathtt{T} \vdash \mathtt{r_1} \# \mathtt{r_2}\end{array}}{\mathtt{T\,;G} \vdash \mathtt{e_1||e_2} \in \mathtt{r_1||r_2}}$$

$$\frac{\mathtt{T\,;G} \vdash \mathtt{e} \in \mathtt{r} \qquad \mathtt{r\_l}\downarrow}{\mathtt{T\,;G} \vdash \mathtt{e.l} \in \mathtt{r\_l}}$$

$$\frac{\mathtt{T, a\#R\,;G} \vdash \mathtt{e} \in \mathtt{t}}{\mathtt{T\,;G} \vdash \Lambda\mathtt{a\#R.e} \in \forall\mathtt{a\#R.\ t}}$$

$$\frac{\mathtt{T\,;G} \vdash \mathtt{e} \in \forall\mathtt{a\#R.\ t} \qquad \mathtt{T} \vdash \mathtt{r} \# \mathtt{R}}{\mathtt{T\,;G} \vdash \mathtt{e[r]} \in \mathtt{[r/a]t}}$$

Figure 2: Selected typing rules

variable a is not considered bound in the constraint list R. Terms and types are identified up to renaming of bound variables. The notation $[\mathtt{t}/\mathtt{a}]\mathtt{t}'$ denotes capture-avoiding substitution of t for free occurrences of a in $\mathtt{t}'$; similarly, $[\mathtt{e}/\mathtt{x}]\mathtt{e}'$ denotes capture-avoiding substitution of e for free occurrences of x in $\mathtt{e}'$.

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. We write $\mathtt{r\_l}\uparrow$ for "r_l is undefined" and $\mathtt{r\_l}\downarrow$ for "r_l is defined."

2.2 Typing Rules

The $\lambda^{||}$ calculus is defined by a collection of inference rules for deriving typing and formation judgements, compatibility judgements, and equivalence judgements. A representative selection of the rules of $\lambda^{||}$ appears in Figures 1–5; the complete set appears in Appendix A. For the most part the formulation of $\lambda^{||}$ proceeds along standard lines; we discuss here only those aspects that are particular to handling extensible records.

The formation rules for record types are summarized in Figure 1. The two most interesting cases are the rules for restriction and merge. The restriction r\l is well formed in T if r is well formed in T and r_l is defined. In particular, no expression of the form a\l, where a is a variable, is ever well formed, since a_l 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. It also implies that if r\l is a well-formed record expression, then the restriction may be eliminated (see the discussion of type equivalence below). A merge $\mathtt{r_1||r_2}$ 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 (see below).

A selection of the typing rules for terms appears in Figure 2. The empty record is always well-formed. A single-field record l=e has type l:t in T if e has type t in T. The restriction e\l has type r\l in T provided that e has type r in T and $\mathtt{r\_l}\downarrow$: we may restrict only on a field that e actually possesses. The merge $\mathtt{e_1||e_2}$ has type $\mathtt{r_1||r_2}$ 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, it is necessary to restrict on the fields to be overridden before forming the merge. The selection e.l has type r_l in T if e has type r in T and $\mathtt{r\_l}\downarrow$. By the definition of r_l, the type of e.l is unique (up to equivalence) if it well-formed.

The abstraction Λa#R. e has type $\forall$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. It is important to realize that the constraint list 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,

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{s} \qquad \mathtt{r} \sim \mathtt{r'} \qquad \mathtt{s} \sim \mathtt{s'}}{\mathtt{T} \vdash \mathtt{r'} \# \mathtt{s'}}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{s}}{\mathtt{T} \vdash \mathtt{s} \# \mathtt{r}}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{l{:}t} \qquad \mathtt{T} \vdash \mathtt{t'} \text{ type}}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{l{:}t'}}$$

$$\frac{\mathtt{T_1, a\#R, T_2} \text{ ok} \qquad \mathtt{r}_i \in \mathtt{R}}{\mathtt{T_1, a\#R, T_2} \vdash \mathtt{a} \# \mathtt{r}_i}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \# (\mathtt{s_1} || \mathtt{s_2})}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{s}_i}$$

$$\frac{\mathtt{T} \vdash \mathtt{s_1} \# \mathtt{s_2} \qquad \mathtt{T} \vdash \mathtt{r} \# \mathtt{s_1} \qquad \mathtt{T} \vdash \mathtt{r} \# \mathtt{s_2}}{\mathtt{T} \vdash \mathtt{r} \# (\mathtt{s_1} || \mathtt{s_2})}$$

$$\frac{\mathtt{l} \neq \mathtt{l'} \qquad \mathtt{T} \vdash \mathtt{l{:}t} \text{ record} \qquad \mathtt{T} \vdash \mathtt{l'{:}t'} \text{ record}}{\mathtt{T} \vdash \mathtt{l{:}t} \# \mathtt{l'{:}t'}}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \text{ record}}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{Empty}}$$

Figure 3: Selected compatibility rules

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 $\mathtt{r}_i$ of R because the records in R need not themselves be mutually compatible.

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 element of 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 [r/a]t, as usual. 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.

The compatibility relation (see Figure 3) plays a central role in $\lambda^{||}$. Informally, T ⊢ r # s holds iff r and s are mergeable, that is, iff r lacks every field possessed by s and s lacks every field possessed by r. The definition is made relative to a context T since the compatibility for a type variable a is determined by the constraint set associated with a in T. It follows from this informal description that compatibility is symmetric, respects type equivalence, and is insensitive to the types ascribed to fields of a record. In particular, l:t is compatible with l′:t′ iff l is different from l′.

$$\mathtt{r}||\mathtt{Empty} \sim \mathtt{r}$$

$$\mathtt{r_1} \,||\, (\mathtt{r_2}||\mathtt{r_3}) \sim (\mathtt{r_1}||\mathtt{r_2}) \,||\, \mathtt{r_3}$$

$$\mathtt{r_1} \,||\, \mathtt{r_2} \sim \mathtt{r_2} \,||\, \mathtt{r_1}$$

$$\mathtt{r \backslash l \backslash l'} \sim \mathtt{r \backslash l' \backslash l}$$

$$(\mathtt{l{:}t})\backslash\mathtt{l} \sim \mathtt{Empty}$$

$$\frac{\mathtt{r_1 \text{-} l} \downarrow}{(\mathtt{r_1}||\mathtt{r_2})\backslash\mathtt{l} \sim (\mathtt{r_1}\backslash\mathtt{l}||\mathtt{r_2})}$$

$$\frac{\mathtt{R} \sim \mathtt{R'} \qquad \mathtt{t} \sim \mathtt{t'}}{\forall \mathtt{a\#R.\ t} \sim \forall \mathtt{a\#R'.\ t'}}$$

Figure 4: Selected type equivalence rules

$$\mathtt{r, (r', R)} \sim \mathtt{r', (r, R)}$$

$$\mathtt{Empty, R} \sim \mathtt{R}$$

$$(\mathtt{r_1}||\mathtt{r_2}), \mathtt{R} \sim \mathtt{r_1, (r_2, R)}$$

$$\mathtt{r, (r, R)} \sim \mathtt{r, R}$$

$$\mathtt{l{:}t, R} \sim \mathtt{l{:}t', R}$$

Figure 5: Selected constraint list equivalence rules

Equivalence of types (Figure 4) is defined as an equivalence relation compatible with all type-forming constructors, such that the merge operation is commutative and associative and has Empty as unit. Field restriction operations eliminate fields in the expected way. Constraint list equivalence (Figure 5) is important due to the presence of constraint lists on quantified types. Besides the equivalences induced by equivalence of types, we identify constraint lists that differ only in the order and multiplicity of constraints and allow for the breakdown of merge types and the elimination of **Empty**.

2.3 Properties

In this section we sketch a proof of the decidability of type checking for $\lambda^{||}$. Due to space limitations, we give only a brief overview of the development. A more detailed account appears in the full paper.

The proof proceeds largely along standard lines: type checking is reduced to checking equivalence and compatibility of types by way of a syntax-directed set of type synthesis rules. Equivalence of well-formed types is established using Huet's method of confluence modulo an equivalence relation [12]. The main idea is to handle the associative and commutative rules for merge types and the permutation and idempotency equations for constraints lists by segregating the "proper reductions" from the "pure equations," so that the equivalence problem is reduced to checking a simple form of equivalence of normal forms. Compatiblity checking is based on a simple characterization of normal forms of record types and constraint lists, with the main complications stemming from the need to take account of all of the consequences of assuming that a variable is compatible with a complex record type.

2.3.1 Type and Constraint List Equivalence

As a technical convenience in the presentation of the type checking algorithm, we extend constraint lists to admit "bare labels." The metavariable ϕ is used to range over type variables and bare labels; the metavariables ρ and σ are used to range over types and bare labels. 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 $\mathtt{T} \vdash \mathtt{r} \# \mathtt{l{:}t}$ for any well-formed type $\mathtt{t}$.

Let $\twoheadrightarrow$ denote the least reflexive, transitive relation containing the relation given in Figure 6 (compatibly extended to all type constructors); let $\approx$ denote the least congruence containing the relation given in Figure 7.

Theorem 2.3.1.1: The restriction of $\twoheadrightarrow$ to well-typed terms is confluent modulo $\approx$.

Corollary 2.3.1.2:

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 $\mathtt{t}' \approx \mathtt{u}'$.

$$\mathtt{Empty} \,||\, \mathtt{r} \twoheadrightarrow \mathtt{r}$$

$$\mathtt{r} \,||\, \mathtt{Empty} \twoheadrightarrow \mathtt{r}$$

$$\mathtt{l{:}t \backslash l} \twoheadrightarrow \mathtt{Empty}$$

$$(\mathtt{r} \,||\, \mathtt{s})\mathtt{\backslash l} \twoheadrightarrow (\mathtt{r\backslash l}) \,||\, \mathtt{s} \quad \text{if } \mathtt{r}_{-}\mathtt{l}\downarrow$$

$$(\mathtt{r} \,||\, \mathtt{s})\mathtt{\backslash l} \twoheadrightarrow \mathtt{r} \,||\, (\mathtt{s\backslash l}) \quad \text{if } \mathtt{s}_{-}\mathtt{l}\downarrow$$

$$\rho||\sigma,\mathtt{R} \twoheadrightarrow \rho,\sigma,\mathtt{R}$$

$$\mathtt{Empty},\mathtt{R} \twoheadrightarrow \mathtt{R}$$

$$\mathtt{l{:}t},\mathtt{R} \twoheadrightarrow \mathtt{l},\mathtt{R}$$

Figure 6: Proper reduction rules

$$\mathtt{r\backslash l\backslash l'} \approx \mathtt{r\backslash l'\backslash l}$$

$$\mathtt{r} \,||\, \mathtt{s} \approx \mathtt{s} \,||\, \mathtt{r}$$

$$\mathtt{r} \,||\, (\mathtt{s} \,||\, \mathtt{t}) \approx (\mathtt{r} \,||\, \mathtt{s}) \,||\, \mathtt{t}$$

$$\phi,\phi,\mathtt{R} \approx \phi,\mathtt{R}$$

$$\rho,\sigma,\mathtt{R} \approx \sigma,\rho,\mathtt{R}$$

Figure 7: Pure equivalences

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

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

Theorem 2.3.1.4:

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

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

up to associativity and commutativity of $||$, 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).

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

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

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. (That is, $\mathtt{R}^*$ 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.

2.3.2 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 8. 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 and completeness of the algorithm are stated by the following theorem:

$$\mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} \mathtt{Empty}$$

$$\mathtt{T} \vdash \mathtt{Empty} \#^{\Rightarrow} \mathtt{r}$$

$$\frac{\mathtt{T} \vdash \mathtt{s}_1 \#^{\Rightarrow} \mathtt{r} \qquad \mathtt{T} \vdash \mathtt{s}_2 \#^{\Rightarrow} \mathtt{r}}{\mathtt{T} \vdash (\mathtt{s}_1 \,||\, \mathtt{s}_2) \#^{\Rightarrow} \mathtt{r}}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} \mathtt{s}_1 \qquad \mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} \mathtt{s}_2}{\mathtt{T} \vdash \mathtt{r} \#^{\Rightarrow} (\mathtt{s}_1 \,||\, \mathtt{s}_2)}$$

$$\frac{\mathtt{l} \neq \mathtt{l}'}{\mathtt{T} \vdash \mathtt{l}{:}\mathtt{t} \#^{\Rightarrow} \mathtt{l}'{:}\mathtt{t}'}$$

$$\frac{\mathtt{a} \in \mathtt{S}^*}{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2, \mathtt{b}\#\mathtt{S}, \mathtt{T}_3 \vdash \mathtt{a} \#^{\Rightarrow} \mathtt{b}}$$

$$\frac{\mathtt{a} \in \mathtt{S}^*}{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2, \mathtt{b}\#\mathtt{S}, \mathtt{T}_3 \vdash \mathtt{b} \#^{\Rightarrow} \mathtt{a}}$$

$$\frac{\mathtt{l} \in \mathtt{R}^*}{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2 \vdash \mathtt{a} \#^{\Rightarrow} \mathtt{l}{:}\mathtt{t}}$$

$$\frac{\mathtt{l} \in \mathtt{R}^*}{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2 \vdash \mathtt{l}{:}\mathtt{t} \#^{\Rightarrow} \mathtt{a}}$$

Figure 8: Algorithmic compatibility rules

Theorem 2.3.2.1:

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 $\mathtt{T} \vdash \mathtt{r} \# \mathtt{r}'$. Conversely, if $\mathtt{T} \vdash \mathtt{r} \# \mathtt{s}$, then $\mathtt{T} \vdash \mathtt{r}^* \#^{\Rightarrow} \mathtt{s}^*$.

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 $\mathtt{T} \vdash \mathtt{r} \# \mathtt{R}$. Conversely, if $\mathtt{T} \vdash \mathtt{r} \# \mathtt{R}$, then $\mathtt{T} \vdash \mathtt{r}^* \#^{\Rightarrow} \mathtt{R}^*$.

The decidability of the algorithmic formulation follows from the fact that it is "almost syntax-directed": although some rules overlap, either choice leads to the same conclusion.

2.3.3 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 $\mathtt{T}\,;\,\mathtt{G} \vdash \mathtt{e} \Rightarrow \mathtt{t}$, 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

$$\mathtt{T_1, a\#R, T_2} \vdash \mathtt{a} \Rightarrow \text{record}$$

$$\frac{\mathtt{T} \vdash \mathtt{r_1} \Rightarrow \text{record} \quad \mathtt{T} \vdash \mathtt{r_2} \Rightarrow \text{record} \quad \mathtt{T} \vdash \mathtt{r_1} \#^{\Rightarrow} \mathtt{r_2}}{\mathtt{T} \vdash \mathtt{r_1} \,||\, \mathtt{r_2} \Rightarrow \text{record}}$$

$$\frac{\mathtt{T} ; \mathtt{G} \vdash \mathtt{e_1} \Rightarrow \mathtt{r_1} \quad \mathtt{T} ; \mathtt{G} \vdash \mathtt{e_2} \Rightarrow \mathtt{r_2} \quad \mathtt{T} \vdash \mathtt{r_1^*} \#^{\Rightarrow} \mathtt{r_2^*}}{\mathtt{T} ; \mathtt{G} \vdash \mathtt{e_1} \,||\, \mathtt{e_2} \Rightarrow \mathtt{r_1} \,||\, \mathtt{r_2}}$$

$$\frac{\mathtt{T} ; \mathtt{G} \vdash \mathtt{e} \Rightarrow \forall \mathtt{a\#R.\, t} \quad \mathtt{T} \vdash \mathtt{r} \Rightarrow \text{record} \quad \mathtt{T} \vdash \mathtt{r^*} \#^{\Rightarrow} \mathtt{R^*}}{\mathtt{T} ; \mathtt{G} \vdash \mathtt{e[r]} \Rightarrow [\mathtt{r/a}]\mathtt{t}}$$

Figure 9: Selected type synthesis rules

or not a derivation exists. A representative set of rules from the definition of the type synthesis algorithm is given in Figure 9.

The soundness and completness of the algorithm are stated by the following theorem:

Theorem 2.3.3.1: (Soundness) If $\vdash$ T ok and T $\vdash$ r $\Rightarrow$ record, then T $\vdash$ r record. If $\vdash$ T ok and T $\vdash$ t $\Rightarrow$ type, then T $\vdash$ t type. If $\vdash$ T ok and T $\vdash$ R $\Rightarrow$ ok, then T $\vdash$ R ok. If T $\vdash$ G ok and T ; G $\vdash$ e $\Rightarrow$ t, then T ; G $\vdash$ e $\in$ t.

(Completeness) If T $\vdash$ r record, then T $\vdash$ r $\Rightarrow$ record. If T $\vdash$ t type, then T $\vdash$ t $\Rightarrow$ type. If T $\vdash$ R ok, then T $\vdash$ R $\Rightarrow$ ok. If T $\vdash$ e $\in$ t, then T $\vdash$ e $\Rightarrow$ t′ for some t′ such that t′ $\sim$ t.

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

One motivation for studying record calculi is the potential application to typed object-oriented programming, as suggested by Cardelli [2] and Wand [27] and further developed by a number of authors [1, 7, 17]. In particular, Wand has demonstrated that a simple form of object-oriented programming may be expressed in a language with record concatenation and recursive types, and the members of the ABEL group at HP Labs have introduced an extension of the notion of bounded quantification, known as *F-bounded quantification,* in order to capture certain object-oriented idioms. We consider here two examples, one taken from Wand, the other from Canning, et al. (Further examples are presented in the full version [10].)

We need to work in an extension of $\lambda^{||}$, called $\lambda^{||\mu}$, that includes full polymorphism (quantification over all types, not just record types), a fixed point operator at all functional types, and recursive types. Of these, only recursive types present any difficulties. Recursive types are written μ a. t(a), where t(a) is an arbitrary type expression possibly involving the variable a. Type equivalence for $\lambda^{||\mu}$ is defined by considering a recursive type μ a. t(a) as denoting the (possibly infinite) regular tree obtained by "unrolling" the recursion and applying the equivalences on type expressions given in Appendix A. Although this description of type equivalence is sufficient for the examples to follow, it should be emphasized that we have not studied the decidability of type equivalence; the decidability of type checking for $\lambda^{||\mu}$ remains open.

Here is a simple example from Wand [26], illustrating the use of record extension to model inheritance. Define the "class" A to be

```
λx:int.
  Λa#sum,n.
    λself:(a||sum:int||n:int).
      sum=(x + self.n)
```

with type

```
int → ∀a#sum,n.
          (a||sum:int||n:int) → (sum:int).
```

Here x is a parameter of the "class instantiation" operation: when A is instantiated, the value supplied for x becomes a hidden component of the new object. The row variable a is a placeholder for any fields that may be added by subclasses of A. The parameter self provides a name within the new instance of A for the instance itself, which is supplied at instantiation time using the fixed point operator.

As it stands, the class A cannot actually be instantiated since it lacks a method for n. Define the class B to be a subclass of A with a method for n:

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

with type

```
int → ∀a#sum,n.
          (a||sum:int||n:int) → (sum:int||n:int).
```

We may now instantiate B by writing, for example, B(3)[Empty] $\in$ sum:int||n:int $\rightarrow$ sum:int||n:int, so that (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:

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

The instantiation expression fix(C(3)[Empty]) results in an object with sum and n fields agreeing with B and with an additional m field whose value is 10.

Our second example illustrates the flexibility of constrained quantification by showing that the motivating examples of F-bounded quantification may readily be expressed in the pure $\lambda^{||\mu}$ calculus.

Members of the ABEL group have argued persuasively 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 consider two classes of situations in which problems arise, one when the recursion variable occurs negatively, the other when it occurs positively. To deal with these problems, they propose an extended notion, called F-bounded quantification, where the pure bounded quantifier of the form $\forall$a $\leq$ r. t is generalized to the form $\forall$a $\leq$ F(a). t, where F is a function from types to types.

For example, in the class of situations 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. Consider the type

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

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

min $\in$ $\forall$a $\leq$ PartialOrder. a$\rightarrow$a$\rightarrow$a.

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

Number = μnum. {leq : num$\rightarrow$Bool, other : t}.

But by the usual rule for subtyping on recursive types, 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

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

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

```
min  =  Λa ≤ FPartialOrder(a).
          λx:a. λy:a.
            if x.leq(y) then x else y
     ∈  ∀a ≤ FPartialOrder(a). a→a→a.
```

The same intention — to express min so that it can be applied generically to members of any class possessing at least an leq operation mapping another member of the same class to a truth value — can be realized directly in $\lambda^{||\mu}$. We write:

```
min    ∈ ∀a#leq. μb. (a || leq:b→Bool)
                → μb. (a || leq:b→Bool)
                → μb. (a || leq:b→Bool)
Number = μnum. (leq:num→Bool || other:t)
five   ∈ Number.
```

To type the application

min five five

we need to restrict away the leq field from Number:

min [Number\leq] five five.

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

Number $\sim$ μb. (Number\leq) || leq:b$\rightarrow$Bool.

Unrolling the definition of Number and applying the rule for elimination of restriction operations, this reduces to showing

Number $\sim$ μb. (other:[Number/num]t)
  || leq:b$\rightarrow$Bool.

But these type expressions have the same infinite unrolling and hence are equal under our interpretation of recursive types. This argument may be made precise by considering a definition of type equivalence similar to the definition of bisimulation equivalence in CCS [15]. To establish the above equivalence, it suffices to show that it is consistent to assume that it holds, where the consistency constraints ensure, for example, that two records are equal only if they have the same fields and corresponding fields are equal.

4 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" [11] 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, something like F-bounded quantification seems to be required. In the negative setting we do not need an analogue of F-bounded quantification, since we 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 can be carried out almost verbatim in Cardelli and Mitchell's calculus.

5 Acknowledgements

We are grateful for productive discussions with Val Breazu-Tannen, Peter Buneman, Luca Cardelli, Carl Gunter, Frank Pfenning, Didier Rémy, and John Reynolds.

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

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

$$\diamond \text{ ok}$$

$$\frac{\mathtt{T} \vdash \mathtt{R} \text{ ok}}{\mathtt{T}, \mathtt{a}\#\mathtt{R} \text{ ok}}$$

A.3 Well-formed constraint lists

$$\frac{\mathtt{T} \text{ ok}}{\mathtt{T} \vdash \diamond \text{ ok}}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \text{ record} \qquad \mathtt{T} \vdash \mathtt{R} \text{ ok}}{\mathtt{T} \vdash \mathtt{r}, \mathtt{R} \text{ ok}}$$

A.4 Well-formed term contexts

$$\frac{\mathtt{T} \text{ ok}}{\mathtt{T} \vdash \diamond \text{ ok}}$$

$$\frac{\mathtt{T} \vdash \mathtt{G} \text{ ok} \qquad \mathtt{T} \vdash \mathtt{t} \text{ type}}{\mathtt{T} \vdash \mathtt{G}, \mathtt{x}{:}\mathtt{t} \text{ ok}}$$

A.5 Well-formed types

$$\frac{\mathtt{T} \text{ ok}}{\mathtt{T} \vdash \mathtt{p} \text{ type}}$$

$$\frac{\mathtt{T} \vdash \mathtt{t}_1 \text{ type} \qquad \mathtt{T} \vdash \mathtt{t}_2 \text{ type}}{\mathtt{T} \vdash \mathtt{t}_1 \rightarrow \mathtt{t}_2 \text{ type}}$$

$$\frac{\mathtt{T}, \mathtt{a}\#\mathtt{R} \ \vdash \mathtt{t} \text{ type}}{\mathtt{T} \vdash \forall \mathtt{a}\#\mathtt{R}.\, \mathtt{t} \ \text{ type}}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \text{ record}}{\mathtt{T} \vdash \mathtt{r} \text{ type}}$$

A.6 Well-formed record types

$$\frac{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2 \text{ ok}}{\mathtt{T}_1, \mathtt{a}\#\mathtt{R}, \mathtt{T}_2 \vdash \mathtt{a} \text{ record}}$$

$$\frac{\mathtt{T} \vdash \mathtt{t} \text{ type}}{\mathtt{T} \vdash \mathtt{l}{:}\mathtt{t} \text{ record}}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \text{ record} \qquad \mathtt{r}_{\text{-}}\mathtt{l}\downarrow}{\mathtt{T} \vdash \mathtt{r} \backslash \mathtt{l} \text{ record}}$$

$$\frac{\mathtt{T} \text{ ok}}{\mathtt{T} \vdash \mathtt{Empty} \text{ record}}$$

$$\frac{\mathtt{T} \vdash \mathtt{r}_1 \# \mathtt{r}_2}{\mathtt{T} \vdash \mathtt{r}_1 \,||\, \mathtt{r}_2 \text{ record}}$$

A.7 Constraint list satisfaction

$$\frac{\mathtt{T} \vdash \mathtt{r} \text{ record}}{\mathtt{T} \vdash \mathtt{r} \# \diamond}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{r}_i \qquad \mathtt{T} \vdash \mathtt{r} \# \mathtt{R}}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{r}_i, \mathtt{R}}$$

A.8 Compatibility

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{s} \qquad \mathtt{r} \sim \mathtt{r}' \qquad \mathtt{s} \sim \mathtt{s}'}{\mathtt{T} \vdash \mathtt{r}' \# \mathtt{s}'}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{s}}{\mathtt{T} \vdash \mathtt{s} \# \mathtt{r}}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \# \mathtt{l}{:}\mathtt{t} \qquad \mathtt{T} \vdash \mathtt{t}' \text{ type}}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{l}{:}\mathtt{t}'}$$

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

$$\frac{\mathtt{T} \vdash \mathtt{r} \# (\mathtt{s}_1 || \mathtt{s}_2)}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{s}_i}$$

$$\frac{\mathtt{T} \vdash \mathtt{s}_1 \# \mathtt{s}_2 \qquad \mathtt{T} \vdash \mathtt{r} \# \mathtt{s}_1 \qquad \mathtt{T} \vdash \mathtt{r} \# \mathtt{s}_2}{\mathtt{T} \vdash \mathtt{r} \# (\mathtt{s}_1 || \mathtt{s}_2)}$$

$$\frac{\mathtt{l} \neq \mathtt{l}' \qquad \mathtt{T} \vdash \mathtt{l}{:}\mathtt{t} \text{ record} \qquad \mathtt{T} \vdash \mathtt{l}'{:}\mathtt{t}' \text{ record}}{\mathtt{T} \vdash \mathtt{l}{:}\mathtt{t} \# \mathtt{l}'{:}\mathtt{t}'}$$

$$\frac{\mathtt{T} \vdash \mathtt{r} \text{ record}}{\mathtt{T} \vdash \mathtt{r} \# \mathtt{Empty}}$$

A.9 Constraint list equivalence

$$R \sim R$$

$$\frac{R \sim R'}{R' \sim R}$$

$$\frac{R \sim R' \qquad R' \sim R''}{R \sim R''}$$

$$\frac{R \sim R' \qquad r \sim r'}{r, R \sim r', R'}$$

$$r, (r', R) \sim r', (r, R)$$

$$\mathtt{Empty}, R \sim R$$

$$(r_1 || r_2), R \sim r_1, (r_2, R)$$

$$r, (r, R) \sim r, R$$

$$l{:}t, R \sim l{:}t', R$$

A.10 Type equivalence

$$r || \mathtt{Empty} \sim r$$

$$r_1 \,||\, (r_2 || r_3) \sim (r_1 || r_2) \,||\, r_3$$

$$r_1 \,||\, r_2 \sim r_2 \,||\, r_1$$

$$r \backslash l \backslash l' \sim r \backslash l' \backslash l$$

$$(l{:}t) \backslash l \sim \mathtt{Empty}$$

$$\frac{r_1 \_ l \downarrow}{(r_1 || r_2) \backslash l \sim (r_1 \backslash l || r_2)}$$

A.11 Type equivalence (congruence)

$$t \sim t$$

$$\frac{t' \sim t}{t \sim t'}$$

$$\frac{t \sim t' \qquad t' \sim t''}{t \sim t''}$$

$$\frac{t_1 \sim t_1' \qquad t_2 \sim t_2'}{t_1 \rightarrow t_2 \sim t_1' \rightarrow t_2'}$$

$$\frac{R \sim R' \qquad t \sim t'}{\forall a \# R.\ t \sim \forall a \# R'.\ t'}$$

$$\frac{t \sim t'}{l{:}t \sim l{:}t'}$$

$$\frac{r \sim r}{r \backslash l \sim r' \backslash l}$$

$$\frac{r_1 \sim r_1' \qquad r_2 \sim r_2'}{r_1 \,||\, r_2 \sim r_1' \,||\, r_2'}$$

A.12 Well-typed terms

$$\frac{T; G \vdash e \in t \qquad T \vdash t' \text{ type} \qquad t \sim t'}{T; G \vdash e \in t'}$$

$$\frac{T \vdash G_1, x{:}t, G_2 \text{ ok}}{T; G_1, x{:}t, G_2 \vdash x \in t}$$

$$\frac{T; G, x{:}t \vdash e \in t'}{T; G \vdash \lambda x{:}t.\ e \in t \rightarrow t'}$$

$$\frac{T; G \vdash e_1 \in t \rightarrow t' \qquad T; G \vdash e_2 \in t}{T; G \vdash e_1\ e_2 \in t'}$$

$$\frac{T; G \vdash e \in t}{T; G \vdash l{=}e \in l{:}t}$$

$$\frac{T; G \vdash e \in r \qquad r \_ l \downarrow}{T; G \vdash e \backslash l \in r \backslash l}$$

$$\frac{T \vdash G \text{ ok}}{T; G \vdash \mathtt{empty} \in \mathtt{Empty}}$$

$$\frac{T; G \vdash e_1 \in r_1 \qquad T; G \vdash e_2 \in r_2 \qquad T \vdash r_1 \# r_2}{T; G \vdash e_1 || e_2 \in r_1 || r_2}$$

$$\frac{T; G \vdash e \in r \qquad r \_ l \downarrow}{T; G \vdash e.l \in r \_ l}$$

$$\frac{T, a \# R; G \vdash e \in t}{T; G \vdash \Lambda a \# R.e \in \forall a \# R.\ t}$$

$$\frac{T; G \vdash e \in \forall a \# R.\ t \qquad T \vdash r \# R}{T; G \vdash e[r] \in [r/a]t}$$

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 (summary). In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of Fifth International Conference on Mathematical Foundations of Programming Language Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 22–52, Tulane University, New Orleans, March 1989. Springer Verlag. To appear in Mathematical Structures in Computer Science.

[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, 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 Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. Technical Report CMU-CS-90-157, Carnegie Mellon University, August 1990.

[11] 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.

[12] 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.

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

[14] 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.

[15] Robin Milner. *A Calculus of Communicating Systems.* Springer-Verlag, LNCS 92, 1980.

[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 manuscript, 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, November 1988.

[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.