Extensible Records Without Subsumption

Robert W. Harper Benjamin C. Pierce

February 13, 1990
CMU-CS-90-102

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present a calculus of operations on *extensible records*, based on a polymorphic lambda calculus with constrained quantification. This system is related to Cardelli and Mitchell's calculus of operations on records, but expresses constraints on record operations using explicit predicates on types rather than a subtype relation.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD) and the Office of Naval Research under Contract N00014-84-K-0415.

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 the Defense Advanced Research Projects Agency or the U.S. government

Subject classification keywords:
F.4.1 Lambda calculus and related systems
D.3.1 Language Theory, Programming
F.3.3 Type structure
D.3.3 Data types and structures
Records

1 Introduction

In 1984, Cardelli [Car84, Car88] observed that some aspects of "multiple 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 t is a subtype of t', written $t \leq t'$, if any member of t may safely be used in a context where a member of 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*:

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

Cardelli and Wegner [CW85] extended this idea to a powerful second-order type system combining Cardelli's ordering on types with the usual notion of type quantification [Gir72, Rey74], using techniques developed by Mitchell [Mit84]. Wand [Wan87, Wan88] 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 a natural *record extension* operator. This work was refined by Jategaonkar and Mitchell [JM88, Jat89] and Stansifer [Sta88].

Rémy [Rém89] introduced the notion of positive and negative information and the intuition that adding 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 [DM82] to do type inference for programs involving extensible records.[1] Wand [Wan89] later extended Rémy's system to infinite label sets and studied type inference for a more general *record merge* operator.

Recently, Cardelli and Mitchell [CM89a, CM89b] have discovered a very 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.

At the beginning of the research described in this report, we set out to model positive and negative information, using a variant of bounded quantification where quantified type variables are constrained to lie *between* two given types rather than just beneath one given type. But in the course of investigating these issues, we found it was helpful to think in terms of more primitive constraints. For example, Cardelli and Mitchell's record extension operator is only defined on records where the field being added is not already present; to prevent run time type errors, the typing rule for the extension operator must check that this is the case. We found it simpler to express this constraint directly as "r lacks l" instead of encoding it as "r is less than some type lacking l." Another unsatisfying feature of systems that combine an order structure on types with ordinary polymorphism is that they allow the "same" polymorphic function to be written in two ways:

$\lambda x{:}R.\, e$
$\Lambda a{\leq}R.\, \lambda x{:}a.\, e$

We decided to look for a simpler system with no preorder on types at all, where genericity over record types would arise solely from quantification of type variables. (Essentially, this amounted to reverse-engineering Cardelli and Mitchell's system back in the direction of Rémy's, although we didn't realize it at the time. In fact, during the early stages of their work Cardelli and Mitchell seem to have independently developed a system similar to ours by extending the kind system of the polymorphic λ-calculus along the lines suggested by Rémy.)

[1] Strictly speaking, Rémy introduced two calculi: one based on ML type inference and one based on subtyping. Viewed from a suitable distance, the two systems can be considered the same for present purposes.

Working with record calculi, one soon realizes that there are many conceivable operations on records, including at least the following:[2]

Construction of new records from explicit lists of fields and their values.

Extraction of the value of a particular field of a record.

(Fresh) extension of a record by adding a value for a field that it does not already have.

(Unchecked) update where the existing value of a field, if any, is overwritten with a new one.

Consistent update where the new field must have the same type as the existing field of the record being updated.

Checked restriction of a record to omit one of its fields.

Unchecked restriction where the field being dropped need not be present.

Renaming of fields.

Retraction of all information about a field.

Symmetric (fresh) merge or concatenation of two records, where all the fields present in the first record must be absent in the second, and vice versa.

Asymmetric (unchecked) merge where fields occurring in both records being merged are given values from the rightmost record.

Consistent (hereditary) merge where fields occurring in both records must be records, which are recursively merged in the result.

Following Cardelli and Mitchell, we decided to take the empty record

$$\{\} \in \{\},$$

fresh extension

$$e|l{=}5 \in r|l{:}Int \qquad \text{(where } e \in r \text{ and } r \text{ lacks } l\text{)},$$

and checked restriction

$$e\backslash l \in r\backslash l \qquad \text{(where } e \in r \text{ and } r \text{ has } l : t \text{ for some } t\text{)}$$

as primitive.

One major difference between the two systems is that, in Cardelli and Mitchell's system, a field must *explicitly* be removed from a record before it can be extended at that field. In Cardelli and Mitchell's system, the empty record type gives no information at all beyond the fact that all values of that type are records: it neither has nor lacks any field. In the symmetric system, on the other hand, the empty record type lacks every field.

Our extension operator, like Cardelli and Mitchell's, is only legal on absent fields: $\{\}|l{=}5$ is allowed and $\{\}|l{=}3|l{=}5$ is illegal. The restriction operators differ in that we provide checked restriction—$\{\}|l{=}5\backslash l$ is allowed but $\{\}\backslash l$ is not—while theirs is unchecked (a record may always be restricted at any field).

Our system allows just one form of record polymorphism, arising from constrained type variables. For example,

$$\Lambda a \text{ lacks } j \text{ has } i{:}Int.\ \lambda x{:}a.\ (x|j{=}(x.i + x.i)) \quad \in \quad \forall a \text{ lacks } j \text{ has } i{:}Int.\ a{\rightarrow}(a|j{:}Int)$$

---

[2]Most of these are simpler to axiomatize in systems without type quantification. Some, such as asymmetric merge, have several interesting variants, depending on where type variables are allowed to appear in the types of their arguments.

denotes a function that may be applied to any record with an integer i field and no j field, and that returns a new record with a j field that is twice the value of the i field of the original, with all the other fields left unchanged. The general form of the constrained quantifier is

$$\Lambda a \text{ lacks } L^- \text{ has } L^+{:}T^+\ldots. \quad \in \quad \forall a \text{ lacks } L^- \text{ has } L^+{:}T^+\ldots.$$

where L^- is a set of absent labels and L^+ is a tuple of present labels of types T^+. The special case where both L^- and L^+ are empty corresponds to ordinary type quantification; the bound variable a then ranges over all types, not just record types. This agrees with the intuition that record types are precisely those types that provably lack or provably have at least one field.

Section 2 of this report defines the symmetric system in detail. Section 3 proves the decidability of typechecking. Section 4 compares the system to its closest relatives, the systems of Rémy and Cardelli and Mitchell. Section 5 suggests some directions for future work.

Our presentation of the symmetric system in Sections 2 and 3 stands on its own and should be comprehensible to anyone who can read inference rules. To understand the comparison to Cardelli and Mitchell and Rémy, readers should already be familiar with the details of those systems.

2 A Symmetric Calculus of Record Operations

2.1 Syntax

The metavariables t and r both range over types; a ranges over type variables; G ranges over environments; l ranges over a countable set of labels; e ranges over expressions; x ranges over variables; p ranges over primitive types; c ranges over constants. L, L^+, and L^- range over finite tuples of distinct labels; T and T^+ range over finite tuples of types. If L and T are tuples of the same length, then $L : T$ denotes the tuple $(L_1 : T_1, \ldots, L_n : T_n)$. We often use tuples as if they were sets, ignoring the order of their elements.

The syntax of types is given by the following abstract grammar:

t	::=	p	(primitive types)
	|	$t \to t'$	(function types)
	|	$\{\}$	(empty record type)
	|	$r\|l{:}t$	(record extension)
	|	$r\backslash l$	(record restriction)
	|	a	(type variable)
	|	$\forall a$ lacks L^- has $L^+{:}T^+.\, t$	(constrained quantification)

(We omit "lacks L^-" if L^- is empty and "has $L^+ : T : +$" if L^+ is empty. Ordinary type quantification is just the special case of record type quantification where both sets of constraints are empty.)

The syntax of terms is:

e	::=	c	(constants)
	|	x	(variables)
	|	$\lambda x{:}t.\, e$	(abstraction)
	|	$e_1\, e_2$	(application)
	|	$\{\}$	(empty record)
	|	$e.l$	(field selection)
	|	$e\|l{=}e'$	(record extension)
	|	$e\backslash l$	(record restriction)
	|	Λa lacks L^- has $L^+{:}T^+.\, e$	(type abstraction)
	|	$e[t]$	(type application)

The syntax of environments is:

$$
\begin{array}{lcl}
G & ::= & \emptyset \\
 & | & G, x : t \\
 & | & G, a \text{ lacks } L^- \text{ has } L^+ : T^+
\end{array}
$$

2.2 Inference Rules

The following judgements are defined by a set of mutually-recursive inference rules:

$\vdash G$ env	(G is a well-formed environment)
$G \vdash t$ type	(t is a well-formed type)
$G \vdash t \sim t'$	(t and t' are equivalent)
$G \vdash r$ rec	(r is a record type)
$G \vdash r$ has $l : t'$	(r has field l of type t')
$G \vdash r$ lacks l	(r does not have field l)
$G \vdash e \in t$	(e has type t)

The "has," "lacks," and "$\sim$" judgements could also be defined by a separate set of mutually-recursive rules.

2.2.1 Well-formed environments

$$\vdash \emptyset \text{ env} \qquad \text{(ENV-EMPTY)}$$

$$\frac{G \vdash t \text{ type} \quad x \notin \text{dom}(G)}{\vdash G, x : t \text{ env}} \qquad \text{(ENV-VAR)}$$

$$\frac{\vdash G \text{ env} \quad a \notin \text{dom}(G) \quad L^- \cap L^+ = \emptyset \quad \text{for all } t \in T^+, G \vdash t \text{ type}}{\vdash G, a \text{ lacks } L^- \text{ has } L^+ : T^+ \text{ env}} \qquad \text{(ENV-TVAR)}$$

2.2.2 Well-formed record types

$$\frac{\vdash G \text{ env}}{G \vdash \{\} \text{ rec}} \qquad \text{(REC-EMPTY)}$$

$$\frac{G \vdash r \text{ rec} \quad G \vdash r \text{ lacks } l \quad G \vdash t \text{ type}}{G \vdash r|l:t \text{ rec}} \qquad \text{(REC-EXT)}$$

$$\frac{G \vdash r \text{ rec} \quad G \vdash r \text{ has } l : t'}{G \vdash r \backslash l \text{ rec}} \qquad \text{(REC-RESTR)}$$

$$\frac{\vdash G, a \text{ lacks } L^- \text{ has } L^+ : T^+, G' \text{ env} \quad L^- \cup L^+ \neq \emptyset}{G, a \text{ lacks } L^- \text{ has } L^+ : T^+, G' \vdash a \text{ rec}} \qquad \text{(REC-TVAR)}$$

2.2.3 Well-formed types

$$\frac{\vdash G \text{ env}}{G \vdash p \text{ type}} \quad \text{(TYPE-PRIM)}$$

$$\frac{G \vdash t \text{ type} \quad G \vdash t' \text{ type}}{G \vdash t \rightarrow t' \text{ type}} \quad \text{(TYPE-ARROW)}$$

$$\frac{\vdash G \text{ env} \quad a \in \text{dom}(G)}{G \vdash a \text{ type}} \quad \text{(TYPE-TVAR)}$$

$$\frac{G, a \text{ lacks } L^- \text{ has } L^+{:}T^+ \vdash t \text{ type}}{G \vdash \forall a \text{ lacks } L^- \text{ has } L^+{:}T^+.t \text{ type}} \quad \text{(TYPE-ALL)}$$

$$\frac{G \vdash r \text{ rec}}{G \vdash r \text{ type}} \quad \text{(TYPE-REC)}$$

2.2.4 Type congruence

$$\frac{G \vdash t \text{ type}}{G \vdash t \sim t} \quad \text{(CONGR-REFL)}$$

$$\frac{G \vdash t \sim t'}{G \vdash t' \sim t} \quad \text{(CONGR-SYMM)}$$

$$\frac{G \vdash t \sim t' \quad G \vdash t' \sim t''}{G \vdash t \sim t''} \quad \text{(CONGR-TRANS)}$$

$$\frac{\vdash G \text{ env}}{G \vdash p \sim p} \quad \text{(CONGR-PRIM)}$$

$$\frac{G \vdash t_1 \sim t_1' \quad G \vdash t_2 \sim t_2'}{G \vdash t_1 \rightarrow t_2 \sim t_1' \rightarrow t_2'} \quad \text{(CONGR-ARROW)}$$

$$\frac{\vdash G \text{ env}}{G \vdash \{\} \sim \{\}} \quad \text{(CONGR-EMPTY)}$$

$$\frac{G \vdash r \backslash l \text{ rec} \quad G \vdash r' \backslash l \text{ rec} \quad G \vdash r \sim r'}{G \vdash r \backslash l \sim r' \backslash l} \quad \text{(CONGR-RESTR)}$$

$$\frac{G \vdash r \sim r' \quad G \vdash t \sim t' \quad G \vdash r|l{:}t \text{ rec} \quad G \vdash r'|l{:}t' \text{ rec}}{G \vdash r|l{:}t \sim r'|l{:}t'} \quad \text{(CONGR-EXT)}$$

$$\frac{G \vdash a \text{ type}}{G \vdash a \sim a} \qquad \text{(CONGR-VAR)}$$

$$\frac{\begin{array}{c}\text{for all } l_i : t_i \in L^+ : T^+ \text{ and } l_i : t'_i \in L^+ : T'^+, \quad G \vdash t_i \sim t'_i \\ G, a \text{ lacks } L^- \text{ has } L^+{:}T^+ \vdash t \sim t'\end{array}}{G \vdash (\forall a \text{ lacks } L^- \text{ has } L^+{:}T^+.t) \sim (\forall a \text{ lacks } L^- \text{ has } L^+{:}T'^+.t')} \qquad \text{(CONGR-ALL)}$$

2.2.5 Record type equivalence

$$\frac{G \vdash r\backslash l\backslash l' \text{ rec}}{G \vdash r\backslash l\backslash l' \sim r\backslash l'\backslash l} \qquad \text{(EQV-\-\')}$$

$$\frac{G \vdash r|l{:}t \text{ rec}}{G \vdash (r|l{:}t)\backslash l \sim r} \qquad \text{(EQV-|-\)}$$

$$\frac{\begin{array}{c}G \vdash (r|l{:}t)\backslash l' \text{ rec} \\ l \neq l'\end{array}}{G \vdash (r|l{:}t)\backslash l' \sim (r\backslash l')|l{:}t} \qquad \text{(EQV-|-\')}$$

$$\frac{G \vdash (r|l'{:}t')|l''{:}t'' \text{ rec}}{G \vdash (r|l'{:}t')|l''{:}t'' \sim (r|l''{:}t'')|l'{:}t'} \qquad \text{(EQV-|-|')}$$

2.2.6 Present fields

$$\frac{\begin{array}{c}G \vdash r|l{:}t \text{ rec} \\ G \vdash t \sim t'\end{array}}{G \vdash (r|l{:}t) \text{ has } l : t'} \qquad \text{(HAS-|)}$$

$$\frac{\begin{array}{c}l \neq l' \\ G \vdash r \text{ has } l : t \\ G \vdash r|l'{:}t' \text{ rec}\end{array}}{G \vdash (r|l'{:}t') \text{ has } l : t} \qquad \text{(HAS-|')}$$

$$\frac{\begin{array}{c}l \neq l' \\ G \vdash r \text{ has } l : t \\ G \vdash r\backslash l' \text{ rec}\end{array}}{G \vdash (r\backslash l') \text{ has } l : t} \qquad \text{(HAS-\')}$$

$$\frac{\begin{array}{c}G, (a \text{ lacks } L^- \text{ has } L^+{:}T^+), G' \vdash \text{env} \\ l : t \in L^+ : T^+ \\ G \vdash t \sim t'\end{array}}{G, (a \text{ lacks } L^- \text{ has } L^+{:}T^+), G' \vdash a \text{ has } l : t'} \qquad \text{(HAS-TVAR)}$$

2.2.7 Absent fields

$$\frac{\vdash G \text{ env}}{G \vdash \{\} \text{ lacks } l} \quad \text{(LACKS-EMPTY)}$$

$$\frac{l \neq l' \quad G \vdash r|l':t' \text{ rec}}{G \vdash (r|l':t') \text{ lacks } l} \quad \text{(LACKS-|')}$$

$$\frac{G \vdash r\backslash l \text{ rec}}{G \vdash (r\backslash l) \text{ lacks } l} \quad \text{(LACKS-\)}$$

$$\frac{G \vdash r \text{ lacks } l \quad l \neq l'}{G \vdash (r\backslash l') \text{ lacks } l} \quad \text{(LACKS-\')}$$

$$\frac{\vdash G, (a \text{ lacks } L^- \text{ has } L^+:T^+), G' \text{ env} \quad l \in L^-}{G, (a \text{ lacks } L^- \text{ has } L^+:T^+), G' \vdash a \text{ lacks } l} \quad \text{(LACKS-TVAR)}$$

2.2.8 Well-typed terms

$$\frac{\vdash G \text{ env}}{G \vdash c \in \text{const-type}(c)} \quad \text{(CONST)}$$

$$\frac{\vdash G, x : t, G' \text{ env}}{G, x : t, G' \vdash x : t} \quad \text{(VAR)}$$

$$\frac{G, x : t \vdash e \in t'}{G \vdash (\lambda x{:}t.\, e) \in t \to t'} \quad \text{(ABS)}$$

$$\frac{G \vdash e_1 \in t \to t' \quad G \vdash e_2 \in t'' \quad G \vdash t \sim t''}{G \vdash (e_1\, e_2) \in t'} \quad \text{(APP)}$$

$$\frac{\vdash G \text{ env}}{G \vdash \{\} \in \{\}} \quad \text{(EMPTYREC)}$$

$$\frac{G \vdash e \in r \quad G \vdash e' \in t' \quad G \vdash r \text{ lacks } l}{G \vdash (e|l{=}e') \in (r|l:t')} \quad \text{(EXTEND)}$$

$$\frac{G \vdash e \in r \quad G \vdash r \text{ has } l : t}{G \vdash (e\backslash l) \in (r\backslash l)} \quad \text{(RESTRICT)}$$

$$\frac{G \vdash e \in r \quad G \vdash r \text{ has } l : t}{G \vdash e.l \in t} \quad \text{(SELECT)}$$

$$\frac{G,\ a \text{ lacks } L^- \text{ has } L^+{:}T^+ \vdash e \in t}{G \vdash (\Lambda a \text{ lacks } L^- \text{ has } L^+{:}T^+ . e) \in (\forall a \text{ lacks } L^- \text{ has } L^+{:}T^+ . t)} \quad \text{(TYPE-ABS)}$$

$$\frac{\begin{array}{c} G \vdash e \in (\forall a \text{ lacks } L^- \text{ has } L^+{:}T^+ . t) \\ \text{for all } l_i : t_i \in L^+ : T^+, \quad G \vdash t' \text{ has } l_i : t'_i \\ \text{for all } l_i : t_i \in L^+ : T^+, \quad G \vdash t_i \sim t'_i \\ \text{for all } l_j \in L^-, \quad G \vdash t' \text{ lacks } l_j \end{array}}{G \vdash (e[t']) \in t[t'/a]} \quad \text{(TYPE-APP)}$$

2.3 Alternative Rules

This version of the system is presented so that decidability is easy to show. As usual in systems of this kind, there is a more natural formulation where the equivalence assumptions in rules APP, TYPE-APP, HAS-TVAR, and HAS-| are omitted and two more general rules are added to the system:

$$\frac{\begin{array}{c} G \vdash e \in t \\ G \vdash t \sim t' \end{array}}{G \vdash e \in t'} \quad \text{(EQUIV)}$$

$$\frac{\begin{array}{c} G \vdash r \text{ has } l : t \\ G \vdash t \sim t' \end{array}}{G \vdash r \text{ has } l : t'} \quad \text{(HAS-EQUIV)}$$

3 Properties of the Symmetric System

3.1 Basic lemmas

1. Notation: We write "... ⊢ ..." for "... ⊢ ... is derivable." When $G \vdash r$ rec, we say that "r is a record type (under G)".

2. Lemma: $G \vdash r$ rec iff there is some l such that $G \vdash r$ lacks l or there are l and t such that $G \vdash r$ has $l : t$.

Proof: By straightforward inspection of the rules.

3. Lemma: If any of

$\vdash G, x : t$ env
$\vdash G, a$ env
$\vdash G, a$ lacks L^- has $L^+{:}T^+$ env

then also

$\vdash G$ env .

Proof: By induction on the structure of the hypothesized proof.

4. Lemma: If either $G \vdash t$ type or $G \vdash t$ rec then $\vdash G$ env.

Proof: By simultaneous induction on the sizes of proofs of $G \vdash t$ type and $G \vdash t$ rec.

5. Lemma: $G \vdash t \sim t'$ implies both $G \vdash t$ type and $G \vdash t'$ type.

Proof: By induction on the structure of the proof that $G \vdash t \sim t'$.

6. Lemma: Let G be an environment and r a record type under G. Then r has the form

$$r_0\ op_a\ lt_a\ op_b\ lt_b\ op_c\ lt_c \ldots op_n\ lt_n$$

where

- r_0 is either $\{\}$ or a variable a whose constraint under G is nontrivial (either L^- or L^+ is nonempty),
- op_n is either $\backslash$ or $|$, and
- lt_n is either l_n (if op_n is $\backslash$) or $l_n : t_n$ (if op_n is $|$).

Proof: By induction on the proof of the well-formedness of r in G. Since $G \vdash r$ rec, r must have one of the forms $\{\}$, $r'|l{:}t$, $r'\backslash l$, or a.

- if $r \equiv \{\}$, then it has the correct form.
- if $r \equiv r'|l{:}t$, then since $G \vdash r'$ lacks l, Lemma 2 and the induction hypothesis imply that r' has the correct form; $r'|l{:}t$ clearly does too.
- if $r \equiv r'\backslash l$, then since $G \vdash r'$ has $l : t''$, Lemma 2 and the induction hypothesis imply that r' has the correct form; $r'\backslash l$ clearly does too.
- if $r \equiv a$, then the well-formedness of G implies that $a \in \mathrm{dom}(G)$. Rule REC-TVAR guarantees that a's constraint under G is nontrivial.

7. Lemma: If $G \vdash r$ has $l : t$ is derivable then $G \vdash r$ lacks l is not derivable, and vice versa.

Proof: Assume, for a contradiction, that both are derivable. Lemma 2 implies that r is a record type, so r has the form given by Lemma 6. Now reason by induction on the structure of r, keeping in mind that the rules for "has" and "lacks" are syntax-directed:

- if $r \equiv a$, then because r is well formed (by Lemma 2 and rule REC-TVAR) we may assume that a's constraint in G is "lacks L^- has $L^+{:}T^+$," with L^- and L^+ disjoint. But then one of LACKS-TVAR and HAS-TVAR must fail to apply, contradicting the assumption.
- if $r \equiv \{\}$, then $G \vdash r$ has $l : t$ cannot be derived and the contradiction is immediate.
- if $r \equiv r''\backslash l$, none of the HAS rules apply.
- if $r \equiv r''\backslash l'$, where $l' \neq l$, then by HAS-\' and LACKS-\', both $G \vdash r''$ has $l : t$ and $G \vdash r''$ lacks l are derivable. This contradicts the induction hypothesis.
- if $r \equiv r''|l{:}t'$, none of the LACKS rules apply.
- if $r \equiv r''|l'{:}t'$, where $l' \neq l$, then then by HAS-|' and LACKS-|', both $G \vdash r''$ has $l : t$ and $G \vdash r''$ lacks l are derivable. This contradicts the induction hypothesis.

3.2 Normalization

8. Definition: The rewrite rules NORM-\-\' through NORM-|-|' are formed by orienting rules EQV-\-\' through EQV-|-|' from left to right as follows:

$$\frac{G \vdash r\backslash l\backslash l' \text{ type} \quad l' \text{ alphabetically less than } l}{G \vdash r\backslash l\backslash l' \rightsquigarrow r\backslash l'\backslash l} \qquad \text{(NORM-\-\')}$$

$$\frac{G \vdash r|l{:}t \text{ type}}{G \vdash (r|l{:}t)\backslash l \rightsquigarrow r} \qquad \text{(NORM-|-\)}$$

$$\frac{G \vdash (r|l{:}t)\backslash l' \text{ type} \quad l \neq l'}{G \vdash (r|l{:}t)\backslash l' \rightsquigarrow (r\backslash l')|l{:}t} \qquad \text{(NORM-|-\')}$$

$$\frac{G \vdash (r|l'{:}t')|l''{:}t'' \text{ type} \quad l'' \text{ alphabetically less than } l'}{G \vdash (r|l'{:}t')|l''{:}t'' \rightsquigarrow (r|l''{:}t'')|l'{:}t'} \qquad \text{(NORM-|-|')}$$

9. Definition: If G is an environment, $C[\,]$ is a type context (a type with a hole), and r and r' are record types such that $G \vdash C[r]$ type and $G \vdash r \rightsquigarrow r'$ by one of the above rewrite rules, then $C[r]$ *reduces in one step* to $C[r']$ under G, written $G \vdash C[r] \rightsquigarrow C[r']$.

10. Definition: The *reduction* relation for a fixed G is the transitive, reflexive closure of one-step reduction. If a well-formed type t reduces to t' under G, we write $G \vdash t \rightsquigarrow^* t'$.

11. Fact: A well-formed record type r in an environment G is in normal form with respect to the reduction relation iff it has the form

$$r_0\backslash l_1 \ldots \backslash l_m | l_{m+1}{:}t_{m+1} \ldots | l_n{:}t_n$$

where

1. r_0 is either $\{\}$ or a variable,
2. the sequences $l_1 \ldots l_m$ and $l_{m+1} \ldots l_n$ are both in strictly increasing lexicographic order.

Proof: By Lemma 6, r can be written as

$$r_0\ op_a\ lt_a\ op_b\ lt_b\ op_c\ lt_c \ldots op_n\ lt_n$$

If any op_i is $|$, then every op_j with $j > i$ must also be $|$, since otherwise there would be an adjacent pair "$\ldots|l_{j-1}{:}t_{j-1}\backslash l_j \ldots$" to which either NORM-|-\ or NORM-|-\' would apply, contradicting the assumption that none of the normalization rules apply to r. So r has the form

$$r_0\backslash l_1 \ldots \backslash l_m | l_{m+1}{:}t_{m+1} \ldots | l_n{:}t_n$$

Furthermore,

1. r_0 is $\{\}$ or a variable, by Lemma 6.
2. If either $l_1 \ldots l_m$ or $l_{m+1} \ldots l_n$ is out of strict lexicographic order, then there must be an adjacent pair of elements that are out of order. One of the rules NORM-\-\', or NORM-|-|' will apply to this pair, contradicting the assumption.

12. Lemma: If $G \vdash t \rightsquigarrow^* t'$, then $G \vdash t \sim t'$.

Proof: Each proof that $G \vdash t \rightsquigarrow^* t'$ corresponds to a proof that $G \vdash t \sim t'$, replacing uses of the transitive and reflexive closure conditions in the definition of $\rightsquigarrow^*$ with instances of CONGR-REFL and CONGR-TRANS and replacing subproofs of the form $G \vdash t'' \rightsquigarrow t'''$ by the corresponding proof that $G \vdash t'' \sim t'''$, using the un-oriented versions of the one-step reduction and congruence rules.

13. Definition: The *outer rank* of a well-formed record type r in an environment G, written outer-rank(r), is defined as follows. Write r as

$$r_0\ op_a\ lt_a\ op_b\ lt_b\ op_c\ lt_c \ldots op_n\ lt_n.$$

Take the sum, over all pairs of elements lt_i and lt_j, with $i < j$, of $c(lt_i, lt_j)$, where

$$\begin{array}{lcll} c(l, l') & = & 0 & \text{if } l \text{ and } l' \text{ are strictly in alphabetical order,} \\ & & 1 & \text{otherwise,} \\ c(l:t, l':t') & = & 0 & \text{if } l \text{ and } l' \text{ are in alphabetical order,} \\ & & 1 & \text{otherwise,} \\ c(l:t, l') & = & 0 & \text{if } l \neq l', \\ & & 1 & \text{if } l = l', \\ c(l, l':t') & = & 0. & \end{array}$$

The idea is that if $r \rightsquigarrow r'$ by one of the normalization rules being applied on the "outer layer" of r, then outer-rank(r) > outer-rank(r').

14. Definition: Let t be a type and r a record-typed subphrase of t. If r's occurrence in t is not of the form $r|l{:}t'$ or $r\backslash l$, then r is said to be *maximal*.

15. Definition: The *rank* of a well-formed type t in an environment G is the sum of the outer ranks of all of its maximal record-typed subphrases. Formally,

$$\begin{array}{lcl} \text{rank}(p) & = & 0 \\ \text{rank}(t{\rightarrow}t') & = & \text{rank}(t) + \text{rank}(t') \\ \text{rank}(a) & = & 0 \\ \text{rank}(\forall a \text{ lacks } L^- \text{ has } L^+{:}T^+.t) & = & \text{rank}(t) \\ \text{rank}(r) & = & \text{outer-rank}(r) + \text{inner-rank}(r) \\ & & \\ \text{inner-rank}(\{\}) & = & 0 \\ \text{inner-rank}(r|l{:}t') & = & \text{inner-rank}(r) + \text{rank}(t') \\ \text{inner-rank}(r\backslash l) & = & \text{inner-rank}(r) \\ \text{inner-rank}(a) & = & 0. \end{array}$$

16. Proposition: If $G \vdash t \rightsquigarrow t'$, then t' has smaller rank than t.

Proof: The proof that $G \vdash t \rightsquigarrow t'$ consists of a context C and types r and r' such that $t \equiv C[r]$, $t' \equiv C[r']$, and $G \vdash r \rightsquigarrow r'$ by one of the NORM rules. The hole in $C[\,]$ appears in exactly one maximal record-typed subcontext, say $R[\,]$.

By the definition of rank, we can write

$$\begin{array}{lcl} \text{rank}(C[r]) & = & c + \text{outer-rank}(R[r]) + \text{inner-rank}(R[r]) \\ \text{rank}(C[r']) & = & c + \text{outer-rank}(R[r']) + \text{inner-rank}(R[r']), \end{array}$$

where c is a constant depending on $C[\,]$, and

$$\begin{array}{lcl} \text{inner-rank}(R[r]) & = & d + \text{inner-rank}(R[r]) \\ \text{inner-rank}(R[r']) & = & d + \text{inner-rank}(R[r']), \end{array}$$

where d is a constant depending on $R[\,]$.

From the definitions of outer-rank and the NORM rules, it is clear that outer-rank($R[r]$) > outer-rank($R[r']$) and inner-rank(r) $\geq$ inner-rank(r').

17. Corollary: All reduction sequences terminate.

18. Proposition: The reduction relation is locally confluent. For any environment G and types t, t', and t'' such that $G \vdash t \rightsquigarrow t'$ and $G \vdash t \rightsquigarrow t''$, there is a type t''' such that $G \vdash t' \rightsquigarrow^* t'''$ and $G \vdash t'' \rightsquigarrow^* t'''$.

Proof: It suffices to examine critical pairs of redexes in t—situations where the redexes in t overlap so that after the reduction from t to t' the redex used to reduce t to t'' no longer exists in t' and vice versa. We consider all possible overlapping applications of the rules NORM-\-\' through NORM-|-|':

1. $((r_0\backslash l_1)\backslash l_2)\backslash l_3$ with $l_3 < l_2 < l_1$:

$$\begin{array}{lll} \text{left redex:} & ((r_0\backslash l_1)\backslash l_2)\backslash l_3 & \rightsquigarrow\ ((r_0\backslash l_2)\backslash l_1)\backslash l_3 \\ & & \rightsquigarrow\ ((r_0\backslash l_2)\backslash l_3)\backslash l_1. \\ \text{right redex:} & ((r_0\backslash l_1)\backslash l_2)\backslash l_3 & \rightsquigarrow\ ((r_0\backslash l_1)\backslash l_3)\backslash l_2 \\ & & \rightsquigarrow\ ((r_0\backslash l_2)\backslash l_3)\backslash l_1. \end{array}$$

2. $((r_0|l_1{:}t_1)\backslash l_1)\backslash l_2$ with $l_2 < l_1$:

$$\begin{array}{lll} \text{left redex:} & ((r_0|l_1{:}t_1)\backslash l_1)\backslash l_2 & \rightsquigarrow\ r_0\backslash l_2. \\ \text{right redex:} & ((r_0|l_1{:}t_1)\backslash l_1)\backslash l_2 & \rightsquigarrow\ ((r_0|l_1{:}t_1)\backslash l_2)\backslash l_1 \\ & & \rightsquigarrow\ ((r_0\backslash l_2)|l_1{:}t_1)\backslash l_1 \\ & & \rightsquigarrow\ r_0\backslash l_2. \end{array}$$

3. $((r_0|l_1{:}t_1)\backslash l_2)\backslash l_3$ with $l_3 < l_2$ and $l_1 \neq l_2$:

   If $l_1 \neq l_3$:

$$\begin{array}{lll} \text{left redex:} & ((r_0|l_1{:}t_1)\backslash l_2)\backslash l_3 & \rightsquigarrow\ ((r_0\backslash l_2)|l_1{:}t_1)\backslash l_3 \\ & & \rightsquigarrow\ ((r_0\backslash l_2)\backslash l_3)|l_1{:}t_1 \\ & & \rightsquigarrow\ ((r_0\backslash l_3)\backslash l_2)|l_1{:}t_1. \\ \text{right redex:} & ((r_0|l_1{:}t_1)\backslash l_2)\backslash l_3 & \rightsquigarrow\ ((r_0|l_1{:}t_1)\backslash l_3)\backslash l_2 \\ & & \rightsquigarrow\ ((r_0\backslash l_3)|l_1{:}t_1)\backslash l_2 \\ & & \rightsquigarrow\ ((r_0\backslash l_3)\backslash l_2)|l_1{:}t_1. \end{array}$$

   If $l_1 = l_3$:

$$\begin{array}{lll} \text{left redex:} & ((r_0|l_1{:}t_1)\backslash l_2)\backslash l_3 & \rightsquigarrow\ ((r_0\backslash l_2)|l_1{:}t_1)\backslash l_3 \\ & & \rightsquigarrow\ r_0\backslash l_1. \\ \text{right redex:} & ((r_0|l_1{:}t_1)\backslash l_2)\backslash l_3 & \rightsquigarrow\ ((r_0|l_1{:}t_1)\backslash l_3)\backslash l_2 \\ & & \rightsquigarrow\ r_0\backslash l_1. \end{array}$$

4. $((r_0|l_1{:}t_1)|l_2{:}t_2)\backslash l_2$ with $l_1 \neq l_2$:

$$\begin{array}{lll} \text{left redex:} & ((r_0|l_1{:}t_1)|l_2{:}t_2)\backslash l_2 & \rightsquigarrow\ ((r_0|l_2{:}t_2)|l_1{:}t_1)\backslash l_2 \\ & & \rightsquigarrow\ ((r_0|l_2{:}t_2)\backslash l_2)|l_1{:}t_1 \\ & & \rightsquigarrow\ r_0|l_1{:}t_1. \\ \text{right redex:} & ((r_0|l_1{:}t_1)|l_2{:}t_2)\backslash l_2 & \rightsquigarrow\ r_0|l_1{:}t_1. \end{array}$$

5. $((r_0|l_1{:}t_1)|l_2{:}t_2)\backslash l_3$ with $l_2 < l_1$:

If $l_3 \neq l_1$:

$$\begin{array}{lll}
\text{left redex:} & ((r_0|l_1{:}t_1)|l_2{:}t_2)\backslash l_3 & \rightsquigarrow ((r_0|l_2{:}t_2)|l_1{:}t_1)\backslash l_3 \\
 & & \rightsquigarrow ((r_0|l_2{:}t_2)\backslash l_3)|l_1{:}t_1 \\
 & & \rightsquigarrow ((r_0\backslash l_3)|l_2{:}t_2)|l_1{:}t_1. \\
\text{right redex:} & ((r_0|l_1{:}t_1)|l_2{:}t_2)\backslash l_3 & \rightsquigarrow ((r_0|l_1{:}t_1)\backslash l_3)|l_2{:}t_2 \\
 & & \rightsquigarrow ((r_0\backslash l_3)|l_1{:}t_1)|l_2{:}t_2 \\
 & & \rightsquigarrow ((r_0\backslash l_3)|l_2{:}t_2)|l_1{:}t_1.
\end{array}$$

If $l_3 = l_1$:

$$\begin{array}{lll}
\text{left redex:} & ((r_0|l_1{:}t_1)|l_2{:}t_2)\backslash l_3 & \rightsquigarrow ((r_0|l_2{:}t_2)|l_1{:}t_1)\backslash l_3 \\
 & & \rightsquigarrow r_0|l_2{:}t_2. \\
\text{right redex:} & ((r_0|l_1{:}t_1)|l_2{:}t_2)\backslash l_3 & \rightsquigarrow ((r_0|l_1{:}t_1)\backslash l_3)|l_2{:}t_2 \\
 & & \rightsquigarrow r_0|l_2{:}t_2.
\end{array}$$

6. $((r_0|l_1{:}t_1)|l_2{:}t_2)|l_3{:}t_3$ with $l_3 < l_2 < l_1$:

$$\begin{array}{lll}
\text{left redex:} & ((r_0|l_1{:}t_1)|l_2{:}t_2)|l_3{:}t_3 & \rightsquigarrow ((r_0|l_2{:}t_2)|l_1{:}t_1)|l_3{:}t_3 \\
 & & \rightsquigarrow ((r_0|l_2{:}t_2)|l_3{:}t_3)|l_1{:}t_1 \\
 & & \rightsquigarrow ((r_0|l_3{:}t_3)|l_2{:}t_2)|l_1{:}t_1. \\
\text{right redex:} & ((r_0|l_1{:}t_1)|l_2{:}t_2)|l_3{:}t_3 & \rightsquigarrow ((r_0|l_1{:}t_1)|l_3{:}t_3)|l_2{:}t_2 \\
 & & \rightsquigarrow ((r_0|l_3{:}t_3)|l_1{:}t_1)|l_2{:}t_2 \\
 & & \rightsquigarrow ((r_0|l_3{:}t_3)|l_2{:}t_2)|l_1{:}t_1.
\end{array}$$

19. Proposition: The reduction relation is confluent: for every environment G and types t, t', and t'' such that $G \vdash t \rightsquigarrow^* t'$ and $G \vdash t \rightsquigarrow^* t''$, there is some type t''' such that $G \vdash t' \rightsquigarrow^* t'''$ and $G \vdash t'' \rightsquigarrow^* t'''$.

Proof: By Corollary 17 and Proposition 18.

20. Corollary: Every well-formed type has a unique normal form and all maximal reduction sequences terminate in this normal form.

21. Proposition: Let t and t' be well formed types in an environment G. Then $G \vdash t \sim t'$ iff there is a type t_{nf} such that t_{nf} is in normal form, $G \vdash t \rightsquigarrow^* t_{nf}$, and $G \vdash t' \rightsquigarrow^* t_{nf}$.

Proof:

($\Leftarrow$) By Lemma 12 and CONGR-TRANS.

($\Rightarrow$) By induction on the length of the proof that $G \vdash t \sim t'$.

If the last step is an application of one of the congruence rules CONGR-PRIM through CONGR-ALL, use the induction hypothesis to find common normal forms of all corresponding subphrases of t and t'. By the definition of reduction, these normal subphrases can be substituted into the outer structure of t (or t') to produce a type t'' such that $G \vdash t \rightsquigarrow^* t''$ and $G \vdash t' \rightsquigarrow^* t''$. By Corollary 20, t'' can be further reduced to a unique normal form t_{nf}.

If the last step is an application of CONGR-SYMM, use the induction hypothesis, reversing the roles of t and t'.

If the last step is an application of CONGR-REFL, the result follows directly from Corollary 20.

If the last step is an application of one of the record equivalence rules EQV-\-\' through EQV-|-|', then by the corresponding normalization rule either $G \vdash t \rightsquigarrow t'$ or $G \vdash t' \rightsquigarrow t$ and the result again follows from Corollary 20.

Otherwise, the last step is an application of CONGR-TRANS. Call the middle type t_{mid}. Apply the induction hypothesis to t and t_{mid} to obtain a normal-form type t''_{nf} such that $G \vdash t \rightsquigarrow^* t''_{nf}$ and $G \vdash t_{mid} \rightsquigarrow^* t''_{nf}$, and again to t_{mid} and t' to obtain a normal-form type t'''_{nf} such that $G \vdash t_{mid} \rightsquigarrow^* t'''_{nf}$ and $G \vdash t' \rightsquigarrow^* t'''_{nf}$. By Corollary 20, t''_{nf} and t'''_{nf} are identical, being common reducts of t_{mid}.

22. Corollary: Equivalence of well-formed types is decidable.

3.3 Decidability of typechecking

23. Theorem: All the judgements of this system are decidable.

Proof: The rules for the judgements

$$\vdash G \text{ env}$$
$$G \vdash t \text{ type}$$
$$G \vdash r \text{ rec}$$
$$G \vdash r \text{ has } l : t'$$
$$G \vdash r \text{ lacks } l$$
$$G \vdash e \in t$$

are all syntax-directed, and hence are decidable given that

$$G \vdash t \sim t'$$

is decidable. (In checking this, it is important to observe that the equivalence of two types need only be decided when both are already known to be well formed.) By Lemma 22, all the judgements are decidable.

4 Related Work

The closest relatives of the symmetric system are Rémy's calculi [Rém89] (taking into account Wand's observation [Wan89] that Rémy's system can be extended to infinite label sets) and Cardelli and Mitchell's work on operations on records in a calculus with bounded second-order polymorphism [CM89a, CM89b].

4.1 Rémy

The symmetric system is can be thought of as essentially just the instantiation of Rémy's ideas in a pure second-order λ-calculus, since genericity in both systems arises solely from polymorphism. In Cardelli and Mitchell's calculus the function that extracts the x field of any record that has an x field of type *Int* can be written either

$$\lambda r{:}(x : Int).\ r.x$$

or

$$\Lambda a <: (x : Int).\ \lambda r{:}a.\ r.x$$

The symmetric system allows only the second version:

$$\Lambda R \text{ has } x{:}Int.\ \lambda r{:}R.\ r.x$$

Rémy's system, which permits only ML-style polymorphism, leaves the Λ implicit:

$$x : Int.\lambda r{:}\{l_1 : \varepsilon_1.\alpha_1,\ l_i : \triangle.Int,\ l_n : \varepsilon_n.\alpha_n,\}.\ r.l_i$$

One difference between Rémy's system and the symmetric system is apparent from this example: Rémy quantifies over flags corresponding to the presence or absence of individual labels. A field whose flag is an unconstrained type variable may be instantiated to either "present of type t" or "absent." In the symmetric system, a single type variable stands for a whole "row" of unknown fields. This seems to be an artifact of the treatment of infinite label sets in the two systems (ours is like Cardelli and Mitchell's rather than Wand's), rather than an essential point of divergence.

Another difference is that Rémy considers a slightly different set of primitives (in particular, his update operation is unchecked), though his system can be extended to include consistent update as well.

4.2 Cardelli and Mitchell

The relationship between the symmetric system and Cardelli and Mitchell's is less straightforward. In a sense they are siblings, since both can be seen as extensions of Rémy's ideas to a system with second-order quantification and both provide the same set of primitive record operations. But the fact that the core calculus of the symmetric system is based on pure second-order quantification, while Cardelli and Mitchell's is based on bounded quantification, leads to a number of important differences. The principal differences are:

- The symmetric system has no explicit order structure on types. This regains unicity of types, and makes it possible to express functions taking arguments of "exact" types.
- Whereas extraction types are an integral part of Cardelli and Mitchell's system without which it loses a great deal of power, adding them to the symmetric system would increase its power very little. Surprisingly, some of the "canonical" uses of extraction types in Cardelli and Mitchell can be expressed in the symmetric system without extraction. In particular, we show below that renaming and consistent update are expressible.
- The constraints on the extension and restriction operators in the symmetric system are exactly symmetric: a record can be extended only on a field that it lacks and restricted only on a field that it has. There is no known symmetric version of Cardelli and Mitchell's calculus (where, for example, $\langle\rangle\backslash l \sim \langle\rangle$) in which fresh extension is also definable [Car89].
- The symmetric system seems somewhat simpler overall than Cardelli and Mitchell's and is arguably more "primitive" because it provides exactly one way to type certain functions that have two different typings in Cardelli and Mitchell's system. However, the symmetric system is considerably less concise in many cases because it uses quantifiers to do things that can often be done with subsumption in Cardelli and Mitchell's system.
- The lack of subsumption may make the symmetric system less expressive in practice.

We conjecture that if a rule of subsumption were added to the symmetric system, it would end up looking very much like Cardelli and Mitchell's calculus. Indeed, Cardelli and Mitchell apparently developed a system similar to the symmetric system during the early stages of their work.

To understand the relation between the symmetric system and its closest relative it is useful to try encoding each in the other.

We sketch translations in both directions. (N.b., we have not checked the correctness of these translations in full detail. Also, the notation used for examples in Cardelli and Mitchell's calculus in the following has been altered slightly to highlight similarities with the symmetric system.)

The simpler of the two translations is from the symmetric system to Cardelli and Mitchell's. Define the translation function $(-)^*$ from types, environments, terms, and judgements in the symmetric system to the corresponding ones in Cardelli and Mitchell's system as follows:

Types:

$$
\begin{array}{lcl}
p^* & = & p \\
(t_1 \rightarrow t_2)^* & = & t_1^* \rightarrow t_2^* \\
\{\}^* & = & \langle\rangle \\
(r|l{:}t)^* & = & r^*|l{:}t^* \\
(r\backslash l)^* & = & r^*\backslash l \\
a^* & = & a \\
(\forall a \text{ lacks } L^- \text{ has } L^+{:}T^+.t)^* & = & \forall a < (\langle\rangle\backslash(L^- \cup L^+))|L^+{:}(T^+)^*.t^*
\end{array}
$$

Environments:

$$
\begin{array}{lcl}
\emptyset^* & = & \emptyset \\
(G,\, x : t)^* & = & G^*,\, x : t^* \\
(G,\, a \text{ lacks } L^- \text{ has } L^+{:}T^+)^* & = & G^*,\, a <: (\langle\rangle \backslash (L^- \cup L^+)) | L^+ : (T^+)^* . t^*
\end{array}
$$

Terms:

$$
\begin{array}{lcl}
c^* & = & c \\
x^* & = & x \\
(\lambda x{:}t.\, e)^* & = & \lambda x{:}t^* .\, e^* \\
(e_1\, e_2)^* & = & e_1^*\, e_2^* \\
\{\}^* & = & \langle\rangle \\
(e.l)^* & = & (e^*).l \\
(e | l{=}e')^* & = & e^* | l{=}e'^* \\
(e \backslash l)^* & = & e^* \backslash l \\
(\Lambda a \text{ lacks } L^- \text{ has } L^+{:}T^+ .\, e)^* & = & \Lambda a <: (\langle\rangle \backslash (L^- \cup L^+)) | L^+ : (T^+)^* .\, t^*
\end{array}
$$

Judgements:

$$
\begin{array}{lcl}
(\vdash G \text{ env})^* & = & \vdash G^* \text{ env} \\
(G \vdash t \text{ type})^* & = & G^* \vdash t^* \text{ type} \\
(G \vdash t \sim t')^* & = & G^* \vdash t^* \sim t'^* \\
(G \vdash r \text{ rec})^* & = & G^* \vdash r^* <: \langle\rangle \\
(G \vdash r \text{ has } l : t')^* & = & G^* \vdash r^* <: \langle\rangle \backslash l | l{:}t'^* \\
(G \vdash r \text{ lacks } l)^* & = & G^* \vdash r^* <: \langle\rangle \backslash l \\
(G \vdash e \in t)^* & = & G^* \vdash e^* \in t^*
\end{array}
$$

By translating proofs in the symmetric system into proofs in Cardelli and Mitchell's system, it is fairly easy to show that the translation preserves derivability—for example, that if $G \vdash e \in t$ is derivable in the symmetric system then $(G \vdash e \in t)^*$ is derivable under Cardelli and Mitchell's rules. (The disjointness condition in rule ENV-TVAR is needed to make the case for type application go through.) On the other hand, the translation does not reflect derivability because there are terms such as

$$(\lambda x{:}\{i : Int\}.\, 2 + x.i)\{i = 5,\, j = 9\}$$

that are ill typed in the symmetric system but that become well typed when the rule of subsumption is allowed.

The idea of the translation from Cardelli and Mitchell's system into the symmetric system is to regard $r <: r'$ as equivalent to a set of "has" and "lacks" assertions expressing the following constraints on r (our use of "has" and "lacks" here is informal):

1. If r' has $l{:}t$, then r has $l : t$ with $t <: t'$.
2. If r' lacks l, then r lacks l.

We use the derivation of $r <: r'$ to generate the requisite assertions. The condition on t and t' reflects the "hereditary" nature of systems like Cardelli and Mitchell's.

The essence of the translation is that every use of subsumption is translated into a type application involving a bounded quantifier. We begin by assuming that derivations have been translated into a normal form where the only use of the subsumption rule is immediately before application, to promote the type of the argument of a function so that it matches the domain type. Each λ-abstraction is then transformed so that its domain type is a variable

$$(\lambda x{:}r.\, e)^{\#} \;=\; \Lambda a \text{ lacks } \ldots \text{ has } \ldots .\, \lambda x{:}a.(e^{\#})$$

where the sets of present and absent labels are determined by the outer-level structure of r. For example,

$$(\lambda x{:}\{i{:}Int, j{:}Bool\}\backslash k.e)^{\#} \;=\; \Lambda a \text{ lacks } k \text{ has } i{:}Int, j{:}Bool.\ \lambda x{:}a.\ e^{\#}.$$

Applications are translated so that the "actual" (minimal) type of the argument, which can be read off from the derivation, is explicitly passed as an extra parameter:

$$(e_1\ e_2)^{\#} \;=\; e_1^{\#}\ [(\text{type-of}(e_2))^{\#}]\ e_2^{\#}$$

If the domain type of the function is a primitive, then the use of subsumption at the application must be trivial (by our assumption above) and the extra type parameter can be dropped from the translation of both the function and the application. On the other hand, subsumption can work not only at the outer level of record types, but also on record types embedded in the fields of larger record types. In general, the translation of a function involves one type quantifier for *each* record-typed subphrase of its domain type. For example,

$$(\lambda x{:}\{i : \{j : Bool\}\}.\ e)^{\#} \;=\; \Lambda a \text{ has } j{:}Bool.\ \Lambda b \text{ has } i{:}a.\ \lambda x{:}b.\ e^{\#}.$$

Obviously, the two translations are not inverses. Neither

$$(e^{*})^{\#} = e$$

nor

$$(e^{\#})^{*} = e$$

holds in general. However, we believe that the translation from Cardelli and Mitchell's system to the symmetric system both preserves and reflects derivability of judgements—e.g., that if $G \vdash e \in t$ is derivable under Cardelli and Mitchell's rules then $(G \vdash e \in t)^{\#}$ is derivable in the symmetric system, and conversely that if $(G \vdash e \in t)^{\#}$ is derivable in the symmetric system then $G \vdash e \in t$ is derivable under Cardelli and Mitchell's rules.

4.3 Some examples

Both Cardelli and Mitchell's system and the symmetric system can express the operation of adding an x field to any record that doesn't already have one:

C&M: $\Lambda R <: \langle\!|\rangle\backslash x.$
$\lambda r{:}R.\ r|x{=}5$

Symmetric: ΛR lacks x.
$\lambda r{:}R.\ r|x{=}5$

Cardelli and Mitchell's record operations allow the "deep update" function that negates the b field of the a field of a record to be written in two different ways. The first version has an analog in the symmetric system, but the second, which is terser and more elegant, relies crucially on extraction types to express the result type of the function. In the symmetric system, deep update

requires an extra type parameter for every level of nesting.

C&M:
$$\Lambda R <: \langle\!\langle b : Bool \rangle\!\rangle.\ \Lambda S <: \langle a : R \rangle.\ \lambda r{:}S.\ r\backslash a|a{=}(r.a\backslash b|b{=}not(r.a.b))$$
using quantification

$$\Lambda S <: \langle\!\langle a : \langle\!\langle b : Bool \rangle\!\rangle \rangle\!\rangle.\ \lambda s{:}S.\ (s \rightarrow a = (s.a \rightarrow b = not(s.a.b)))$$
$$\in \langle\!\langle S \leftarrow a : \langle\!\langle S.a \leftarrow b : Bool \rangle\!\rangle \rangle\!\rangle$$
using extraction types

Symmetric:
$$\Lambda R \text{ has } b{:}Bool.\ \Lambda S \text{ has } a{:}R.\ \lambda r{:}S.\ r\backslash a|a{=}(r.a\backslash b|b{=}not(r.a.b))$$

The identity function on all records with *only* a field x (of any type) is expressible only in the symmetric system, since Cardelli and Mitchell have no way of *preventing* a larger record from being given the more restricted type by the rule of subsumption:

$$\Lambda R.\ \lambda r{:}\{x : R\} \cdot r$$

The function accepting any record with at least integer x and y fields and returning the stripped record with only x and y fields can be expressed in both systems, but Cardelli and Mitchell's version is shorter:

C&M:
$$\lambda r{:}\langle\!\langle x : Int, y : Int \rangle\!\rangle.\ r$$

Symmetric:
$$\Lambda R \text{ has } x{:}Int, y : Int.\ \lambda r{:}R.\ \{x = r.x, y = r.y\}$$

Since one of the biggest differences between our system and Cardelli and Mitchell's is that we do not provide extraction types (they seem not to add significant expressive power in systems that do not also have subsumption), it is important to assess how much of what they do with extraction types is expressible in other ways in our system. Probably the most important use of extraction types in Cardelli and Mitchell is in defining the "consistent update" operator, which takes a record and gives a new value to one of its fields, where the new value must have the same type as the existing contents of that field. The result then has exactly the same type as the original record.

In both systems, the update operation requires a new syntactic form with a separate rule of inference giving its type:

$$\frac{E \vdash r \in R <: \langle\!\langle\rangle\!\rangle \qquad E \vdash a \in R.x}{E \vdash r.x :- a \in R} \qquad \text{(CM-UPDATE)}$$

$$\frac{G \vdash e \in r \qquad G \vdash r \text{ has } l : t' \qquad G \vdash e' \in t'' \qquad G \vdash t' \sim t''}{G \vdash (e \text{ upd } l = e') \in r} \qquad \text{(SYMM-UPDATE)}$$

A less critical but convenient record operation is "renaming": altering a record so that all of its values are the same but one of its fields has been renamed. In Cardelli and Mitchell's calculus, renaming is definable as syntactic sugar:

$$R[x \to y] \stackrel{\text{def}}{=} \langle\!\langle R \backslash x | y{:}R.x \rangle\!\rangle$$
$$r[x \to y] \stackrel{\text{def}}{=} \langle\!\langle r \backslash x | y{=}r.x \rangle\!\rangle$$

The symmetric system requires a new inference rule for renaming at the level of types because there is no way to use simple syntactic sugar to get a name for the type of the x field of the original record type:

$$\frac{\begin{array}{c} G \vdash r \text{ rec} \\ G \vdash r \text{ has } x : t \\ G \vdash r \backslash x \text{ lacks } y \end{array}}{G \vdash r[x \to y] \sim (r \backslash x) | y{:}t} \qquad \text{(SYMM-TYPE-RENAME)}$$

But at the level of values, syntactic sugar suffices:

$$r[x \to y] \stackrel{\text{def}}{=} r \backslash x | y{=}r.x$$

The obvious typing rule for the rename operator,

$$\frac{\begin{array}{c} G \vdash e \in r \\ G \vdash r \text{ has } x : t \\ G \vdash r \text{ lacks } y \end{array}}{G \vdash e[x \to y] \in r[x \to y]} \qquad \text{(SYMM-VAL-RENAME)}$$

is derivable.

5 Future Work

Our investigation of this calculus suggests a number of profitable avenues for further research:

- In many situations, it is possible to translate expressions involving quantifiers with positive constraints on their bound variable into equivalent expressions involving only negative constraints on variables. For example,

  $$(\Lambda a \text{ has } l{:}\mathit{Int}.\ \lambda x{:}a.\ x.l + 2)\ [\{\}|l{:}\mathit{Real}]\ (\{\}|l{=}3)$$

  can be translated as

  $$(\Lambda a \text{ lacks } l.\ \lambda x{:}(a|l{:}\mathit{Int}).\ x.l + 2)\ [\{\}]\ (\{\}|l{=}3).$$

  This suggests that there may be an equivalent formulation of the symmetric system where quantifiers have only negative constraints. If there is, we conjecture that this system can be encoded in an even simpler, more basic system whose only record operator is the symmetric merge.

- Our original investigation of how Cardelli and Mitchell's ideas can be combined with double-bounded quantification still presents a number of fascinating questions. Interesting issues may also arise from combining Cardelli and Mitchell's system with one involving intersections (also called "meets" or "conjunctions") of types [CDV80, Pie89].

- All the schemes for record operations in extensions of ML's type system involve some kind of type reconstruction. The ramifications of record operations for type reconstruction in a second-order type system have not yet been considered in depth (either in our work or, as far as we know, by Cardelli and Mitchell).

- In view of the bewildering variety of conceivable possible operations on records, it would be very helpful to have a comprehensive taxonomy of natural programming examples where each operation can be used.

References

[Car84] 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.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Car89] Luca Cardelli. Personal communication, 1989.

[CDV80] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and lambda calculus semantics. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 535–560, New York, 1980. Academic Press.

[CM89a] 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.

[CM89b] Luca Cardelli and John C. Mitchell. Operations on records. Research report 48, Digital Equipment Corporation, Systems Research Center, August 1989.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, December 1985.

[DM82] 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.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[Jat89] Lalita A. Jategaonkar. ML with extended pattern matching and subtypes. Master's thesis, MIT, August 1989.

[JM88] 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. ACM.

[Mit84] John C. Mitchell. Coercion and type inference (summary). In *Proc. 11-th ACM Symp. on Principles of Programming Languages*, pages 175–185, January 1984.

[Pie89] Benjamin C. Pierce. Bounded quantification and intersection types. Thesis proposal, September 1989.

[Rém89] 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.

[Rey74] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[Sta88] 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.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.

[Wan88] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

[Wan89] 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.