Type Reconstruction for Variable-Arity Procedures

Hsianlin Dzeng* Christopher T. Haynes*

Computer Science Department
Lindley Hall, Indiana University
Bloomington, IN 47405
{dzeng,chaynes}@cs.indiana.edu

Abstract

An ML-style type system with variable-arity procedures is defined that supports both optional arguments and arbitrarily-long argument sequences. A language with variable-arity procedures is encoded in a core-ML variant with infinitary tuples. We present an algebra of infinitary tuples and solve its unification problem. The resulting type discipline preserves principal typings and has a terminating type reconstruction algorithm. The expressive power of infinitary tuples is illustrated.

1 Introduction

Most languages employing ML-style polymorphic type reconstruction do not support procedures with multiple arguments. Instead, multiple arguments are passed in an aggregate structure or via repeated application to a curried procedure. Extension of an ML-style type system to support higher, but fixed, arity procedures is straightforward. A *variable-arity procedure* accepts an indefinite number of arguments. Many languages provide variable-arity primitive procedures, and some allow creation of variable-arity procedures. For example, Ada [1] allows the definition of procedures with optional arguments, for which defaults are provided, Scheme [2] lambda expressions may have a "rest" parameter to which a list of all remaining arguments is bound, and Common Lisp [15] supports both optional and rest arguments. This paper presents a flexible method of supporting variable-arity procedures, with both optional and rest arguments, in the context of ML-style polymorphic type reconstruction.

When both optional and rest arguments are supported, the general form of a formal parameter declaration for a procedure, p, is some number, say i, of required parameters, followed by j optional parameters, each with an associated default value, followed optionally by a rest parameter, where i and j may be any natural number. Let $v_1, \ldots, v_n$ be the values of arguments in an application to p. Then $i \leq n$ is required, and so is $n \leq i+j$ in the absence of a rest parameter. The required parameters are associated with $v_1, \ldots, v_i$, as usual. The next $k = \min(j, n-i)$ argument values are associated with the first k optional parameters, while any remaining optional parameters assume default values. Finally, if a rest argument is provided, it is associated with the (possibly empty) sequence of values $v_{i+j+1}, \ldots, v_n$.

The default value of an optional parameter is given by an associated expression. This expression may be evaluated either when the associated procedure is evaluated or when the procedure is invoked. This does not affect typing unless, as is sometimes the case, the expression is evaluated in an environment that contains preceding required and optional arguments. Our mini-language does not reflect this latter option since it introduces inessential complications in the typing problem.

We adopt Dybvig and Hieb's rest parameter mechanism, in which rest variables may only be referenced following an **&rest** keyword at the end of an application, indicating that the values bound to the rest variable are to be passed as additional arguments in the call. [3] For example, if the sequence of values $v_1, \ldots, v_n$ is bound to the rest variable a, then (M_0 M_1 ... M_m **&rest** a) is equivalent to (M_0 M_1 ... M_m x_1 ... x_n), where $v_1, \ldots, v_n$ are bound to $x_1, \ldots, x_n$, respectively. Efficiency was the original motivation for not collecting the rest arguments of a call in a list (as in most Lisp dialects). In a statically-typed setting this approach can also be more general, allowing a sequence of rest arguments to be *heterogeneous* (of differing type).

The domain of an arity-n procedure is typically typed as a tuple of dimension n. For example, a procedure of the form (**lambda** (x y) M) would have a type of the form $\tau_x \times \tau_y \rightarrow \tau_M$ and in a procedure call of the form (M_0 M_1 M_2 M_3), M_0 must have a procedural type whose domain is of the form $\tau_{M_1} \times \tau_{M_2} \times \tau_{M_3}$. A procedure with optional or rest arguments is polymorphic in its arity. For example, the domain type of a procedure (**lambda** (x **&optional** (y M)) N) could be either τ_x or $\tau_x \times \tau_M$, and the domain of

$$(\textbf{lambda}\ (x\ \textbf{\&optional}\ (y\ M)\ \textbf{\&rest}\ a)\ N) \qquad (1)$$

may have an infinite number of types of the form τ_x, $\tau_x \times \tau_M$, $\tau_x \times \tau_M \times \tau_1$, $\tau_x \times \tau_M \times \tau_1 \times \tau_2$, ...

Variable-arity procedures may be typed by adapting techniques for typing extensible records. Wand [16] introduced *row variables* to express inclusion polymorphism on records. Rémy's projective polymorphism [13] generalizes classical record operations using infinitary records with defaults. Since tuple types may be encoded as record types with numerically labeled fields, the domain type of (1) may

*Work supported in part by the National Science Foundation under grant numbers CCR-8702117 and CDA-9312614.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

LISP 94 - 6/94 Orlando, Florida as USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

be encoded using Rémy's technique as $[\overline{1} : pre \cdot \tau_x \ ; \ \overline{2} : \varepsilon \cdot \tau_M \ ; \ \rho]$. Each field is marked present (*pre*) or absent (*abs*), and ε is a universally quantified type variable expressing either presence or absence, while ρ is a "row" expressing the remaining fields of the record type that belong to the rest parameter. For a call $(M_0\ M_1\ M_2)$ the domain type of M_0 is unified with $[\overline{1} : pre \cdot \tau_{M_1} \ ; \ \overline{2} : pre \cdot \tau_{M_2} \ ; \ \underline{abs} \circ \gamma]$, where the "base" row $\underline{abs} \circ \gamma$ can be viewed as equivalent to $\overline{3} : abs \cdot \tau_3 \ ; \ \overline{4} : abs \cdot \tau_4 \ ; \ \cdots$. The present marks are accepted by required and optional parameters, while absent marks are rejected by required parameters and accepted by optional parameters. The row variable γ expresses an infinite sequence of unknown types, which is matched against the types of default expressions associated with optional parameters. Dually, fixed-arity procedures have a base row element in their domain; for example, (**lambda** (x) M) has type $[\overline{1} : pre \cdot \tau_x \ ; \ \underline{abs} \circ \gamma] \rightarrow \tau_M$.

Next consider

(**lambda** (**&rest** *a*) ((*M* **&rest** *a*) (*N* 5 **&rest** *a*))) (2)

A type of the form $[\overline{1} : \tau_1 \ ; \ \overline{2} : \tau_2 \ ; \ \cdots]$ is assigned to a, which matches the domain type of M. The type of a must also be used to construct a tuple type of the form $[\overline{1} : int \ ; \ \overline{2} : \tau_1 \ ; \ \overline{3} : \tau_2 \ ; \ \cdots]$ to match the domain type of N. Since incrementing of the labels associated with τ_1, τ_2, ... is not possible with existing record-based type systems, we are prompted to introduce an infinitary tuple algebra.

"Infinitary tuples" admit the usual operations on lists (*car*, *cdr*, and *cons*), but may be heterogeneous. For example in (2), if a has a type of the form $[\tau_1 \ ; \ \tau_2 \ ; \ \cdots]$ then the domain type of N must be matched against a type of the form $[int \ ; \ \tau_1 \ ; \ \tau_2 \ ; \ \cdots]$. Though infinitary tuples do not have labels, the basic framework is still that of Rémy's record typing. Our infinitary tuple types may be defined through terms of an equational algebra related to Rémy's, and the unification problem remains tractable. The complexity of infinitary tuples appears justified by our desire to simultaneously support optional arguments, homogeneous rest sequences, and heterogeneous rest sequences. (Individually, each of the above could be encoded in an existing record mechanism.)

To illustrate the power of the type system we propose, consider the following programs (expressed in a Scheme-like syntax).

```
(define trace
  (lambda (fun &optional (mesg "hello "))
    (lambda (&rest args)
      (display mesg)
      (fun &rest args))))

(define traced-add1 (trace add1 "entering add1 "))
(define traced-cons (trace cons))
```

The procedure *trace* takes a procedure and an optional tracing message as arguments and returns a "traced" version of the original procedure argument. The type assigned subsumes types of the form $[([\rho] \rightarrow \tau)] \rightarrow ([\rho] \rightarrow \tau)$ and $[([\rho] \rightarrow \tau) \ ; \ string] \rightarrow ([\rho] \rightarrow \tau)$, where the rest parameter *args* consumes a possibly heterogeneous sequence of values denoted by the row ρ. Thus *trace* can be applied to any procedure. (The syntax of types used in this introduction is only intended to be suggestive. The full type system we propose requires a more powerful syntax.)

```
(define +
  (lambda (&rest nums)
    (let ((ls (list &rest nums)))
      (letrec
        ((loop (lambda (ls)
                 (if (null? ls)
                     0
                     (binary+ (car ls) (loop (cdr ls)))))))
        (loop ls)))))
```

The type of *list* in our system subsumes any type of the form $[\tau; \cdots] \rightarrow (\tau\ list)$. That is, *list* can take any number of arguments as long as they are of the same type, and returns a homogeneous list. The type of + can then be inferred to express $[int \ ; \ \cdots] \rightarrow int$.

```
(define ormap
  (lambda (pred ls)
    (if (null? ls)
        #f
        (or (pred (car ls)) (ormap pred (cdr ls))))))

(define transpose
  (lambda (a &rest ls)
    (let ((ls (list a &rest ls)))
      (letrec
        ((loop (lambda (ls)
                 (if (ormap null? ls)
                     nil
                     (cons (map car ls)
                           (loop (map cdr ls)))))))
        (loop ls)))))
```

The procedure *transpose* takes lists representing rows of a matrix as arguments and returns a list of lists. For example, (*transpose* '(1) '(2) '(3)) returns ((1 2 3)), and (*transpose* '(1 2) '(3 4)) returns ((1 3) (2 4)). The above definition requires the matrix's size to be at least 1 by 1 (one required parameter is specified). The type of *transpose* inferred by our system subsumes any type of the form $[(\tau\ list) \ ; \ (\tau\ list) \ ; \ \cdots] \rightarrow ((\tau\ list)\ list)$. The domain is a homogeneous sequence of lists.

In our type system the procedure *map* can be assigned a type of the form $[[\rho] \rightarrow \sigma \ ; \ (\rho\ list)] \rightarrow (\sigma\ list)$. This type is sufficiently generic to encompass all three applications in the following silly procedure.

```
(define lotsofmaps
  (lambda (&rest nums)
    (let ((ls (list &rest nums)))
      (map make-list
        (map add1 ls)
        (map + ls ls ls)))))
```

The two occurrences of the row ρ impose a type constraint between the domain of the first argument and the rest of the arguments passed to *map*. Without infinitary tuple types, it appears that three mapping procedures with distinct types would be required.

In Section 2 we present the static semantics of a simple language, ML^{va}, with variable-arity procedures. To solve the type reconstruction problem for ML^{va}, we introduce in Section 3 an embedding of ML^{va} into a subset of Standard ML [5] enriched with infinitary tuples and an appropriate equational algebra on its types, which we call ML^{Π}. The unification problem for the type terms of ML^{Π} is solved via

$$
\begin{aligned}
\textit{expression} &::= \textit{variable} \mid \textit{procedure_call} \mid \textit{let_expression} \mid \lambda\textit{_expression} \\
\textit{procedure_call} &::= (\textit{expression}\ \{\textit{expression}\}^*\ [\textbf{\&rest}\ \textit{rest_variable}]) \\
\textit{let_expression} &::= (\textbf{let}\ (\textit{variable expression})\ \textit{expression}) \\
\lambda\textit{_expression} &::= (\textbf{lambda}\ \textit{formals expression}) \\
\textit{formals} &::= (\{\textit{variable}\}^*\ [\textbf{\&optional}\ \{\textit{optional_part}\}^+]\ [\textbf{\&rest}\ \textit{rest_variable}]) \\
\textit{optional_part} &::= (\textit{variable expression})
\end{aligned}
$$

Figure 1: The syntax of ML^{va}

an algebra of infinitary tuples. Through type reconstruction in ML^{Π}, we obtain the principal typing theory and existence of a terminating type reconstruction algorithm for ML^{va}. Section 4 presents several extensions of ML^{va} that illustrate the expressive power of our system. Section 5 outlines some directions for future work, including an extension that appears to make the power of our type system accessible through a conservative extension of ML's pattern matching for tuples. Section 6 concludes.

2 The Language ML^{va}

In this section we introduce the ML^{va} language, which supports ML-style polymorphic type reconstruction along with optional and rest arguments.

The grammar in Figure 1 defines the syntax of ML^{va} expressions. The meta-variables M and N range over expressions, x and y over variables, and a and b over rest variables. Optional parameters have associated initial expressions that provide default values when the corresponding argument is not provided. Rest variables were discussed in the introduction. For simplicity, a let expression has only one binding, which is polymorphic in the manner of ML.

The following grammar defines the syntax of types in ML^{va}:

$$
\begin{array}{rll}
\text{type variable} & \alpha & \\
\text{mark variable} & \varepsilon & \\
\text{row variable} & \gamma_\mu,\ \gamma_\tau & \\
\\
\text{type} & \tau & ::= \alpha \mid \rho \to \tau \\
\text{row} & \rho & ::= \varphi :: \rho \mid \rho_\mu \circ \rho_\tau \\
\text{field} & \varphi & ::= \mu \cdot \tau \\
\text{mark} & \mu & ::= \varepsilon \mid abs \mid pre \\
\text{mark row} & \rho_\mu & ::= \gamma_\mu \mid \mu :: \rho_\mu \mid \underline{abs} \\
\text{type row} & \rho_\tau & ::= \gamma_\tau \mid \tau :: \rho_\tau
\end{array}
$$

The domain of a procedural type is a *row*, which is a sequence of *fields*. Each field contains a *mark* and a *type*. A row can also be generated by composing a *mark row* and a *type row*. Following Wand [16], we employ *row variables* to express indefinite sequences of terms, including both type rows and mark rows. The idea of infinite base rows (e.g. $\underline{abs}$) comes from Rémy's record terms [11, 14], but our rows do not have labels and are not sorted. (This last requirement is crucial in typing rest variable references.) Equality of types is defined by the following set, E_0, of axioms:

$$\underline{abs} \overset{E_0}{=} abs :: \underline{abs}$$

$$(\varepsilon :: \gamma_\mu) \circ (\alpha :: \gamma_\tau) \overset{E_0}{=} (\varepsilon \cdot \alpha) :: (\gamma_\mu \circ \gamma_\tau)$$

The E_0-equality relation on types generated by E_0 is written $=_{E_0}$.

Let $\mathcal{V}(\tau)$ be the set of type variables (regardless of sorts) occurring in τ. A *type scheme* is of the form $\forall W \cdot \tau$, where W is a finite set ranging over all sorts of type variables in $\mathcal{V}(\tau)$. We identify $\forall\emptyset \cdot \tau$ and τ. A *typing judgement* is of the form $A \vdash M : \tau$, where A is a *type environment* (an overloaded mapping from variables to type schemes and from rest variables to rows). $\mathcal{V}$ naturally extends to rows and type environments. We write $A,\ x : \tau$ or $A,\ a : \rho$ for extension of A. T denotes the collection of type terms of all sorts. σ and τ denote types, while ϱ and ρ denote rows. A *sort-respecting* substitution s from the set of variables W to T, written $s :: W \Rightarrow T$, substitutes terms only for variables of corresponding sort.

The typing relation $\vdash_{\mathrm{ML}^{va}}$, abbreviated $\vdash$ when there is no confusion, is the smallest relation satisfying the typing rules and rule prototypes in Figure 2. The (FUN) and (APP) prototypes are interpreted as rules by choosing either to include or omit the contents of option "$[\cdots]$" brackets, and choosing the left or right elements within choice "$\{\cdots \mid \cdots\}$" braces, with each choice made uniformly throughout a prototype.

3 Type Reconstruction

We introduce an extension of core ML, named ML^{Π}, incorporating a sorted regular equational theory on types patterned after the work of Rémy [12]. By translating expressions of ML^{va} into expressions of ML^{Π}, we reduce the type reconstruction problem for ML^{va} to that of typing ML^{Π}. The types of ML^{Π} can be derived from the raw terms of a parameterized algebra of infinitary tuples, whose unification problem we prove decidable and unitary.

In a procedure call of n arguments, we create an infinitary tuple whose first n fields are marked "present" and filled with the actual arguments. The rest of the tuple will be marked "absent" and filled with a default value. A procedure's domain type must likewise be an infinitary tuple. Elements of the tuple are projected based on the formal parameter specification.

3.1 The Intermediate Language ML^{Π}

The ML^{Π} language extends a subset of Standard ML with the following additions: a new sort of type term, named *row*, a new type constructor, $\Pi\langle_\rangle$, which takes a row as argument, and a set of constants operating on *infinitary tuples*, which behave as heterogeneous lists of unbounded length. To encode variable-arity procedures we require three

$$\frac{A \vdash M : \sigma \qquad \sigma =_{E_0} \tau}{A \vdash M : \tau} \quad \text{(EQUAL)} \qquad \frac{A \vdash a : \varrho \qquad \varrho =_{E_0} \rho}{A \vdash a : \rho} \quad \text{(Row-EQUAL)}$$

$$\frac{s :: W \Rightarrow T}{A,\ x : \forall W \cdot \tau \vdash x : s(\tau)} \quad \text{(VAR)} \qquad A,\ a : \rho \vdash a : \rho \quad \text{(Row-VAR)}$$

$$\frac{A \vdash M : \tau \qquad A,\ x : \forall W \cdot \tau \vdash N : \sigma \qquad W \cap \mathcal{V}(A) = \emptyset}{A \vdash (\textbf{let}\ (x\ M)\ N) : \sigma} \quad \text{(LET)}$$

$$\frac{\begin{array}{c}\{\ A \vdash a : \rho \mid \ \} \\ A \vdash N_1 : \tau_1 \quad \ldots \quad A \vdash N_n : \tau_n \\ A \vdash M : (pre \cdot \tau_1 :: \cdots :: pre \cdot \tau_n :: \{\ \rho \mid \underline{abs} \circ \rho_\tau\ \}) \rightarrow \tau\end{array}}{A \vdash (M\ N_1\ \ldots\ N_n\ \{\textbf{\&rest}\ a \mid \}) : \tau} \quad \text{(APP)}$$

$$\frac{\begin{array}{c}[\ A \vdash M_1 : \sigma_1 \quad \ldots \quad A \vdash M_m : \sigma_m\ ] \\ A,\ x_1 : \tau_1,\ \ldots,\ x_n : \tau_n[,\ y_1 : \sigma_1,\ \ldots,\ y_m : \sigma_m]\{,\ a : \rho \mid \ \} \vdash N : \tau\end{array}}{\begin{array}{c}A \vdash (\textbf{lambda}\ (x_1\ \ldots\ x_n\ [\textbf{\&optional}\ (y_1\ M_1)\ \ldots\ (y_m\ M_m)]\ \{\textbf{\&rest}\ a \mid \})\ N) \\ : (pre \cdot \tau_1 :: \cdots :: pre \cdot \tau_n[:: \mu_1 \cdot \sigma_1 :: \cdots :: \mu_m \cdot \sigma_m\ ] :: \{\rho \mid \underline{abs} \circ \rho_\tau\}) \rightarrow \tau\end{array}} \quad \text{(FUN)}$$

Figure 2: The static semantics of ML^{va}

concrete data types that may be expressed in Standard ML as:

```
datatype abs = Abs
     and pre = Pre
     and 'a opt = None | One of 'a
```

The values Abs and Pre serve as marks specifying the absence or presence of an argument in a tuple field. The values stored in the fields of a tuple are of type 'a opt, where the constructor None is used as the value of an absent field, and One tags a value in a present field.

The types we consider are generated by the following grammar:

$$\begin{array}{lrl} \text{type} & \tau ::= & \alpha \mid \tau \rightarrow \tau \mid \tau * \tau \\ & & \mid \tau\ \mathtt{opt} \mid \mathtt{abs} \mid \mathtt{pre} \mid \Pi\langle\rho\rangle \\ \text{row} & \rho ::= & \gamma \mid \tau ; \rho \mid \rho \underline{*} \rho \mid \rho\ \underline{opt} \mid \underline{abs} \end{array}$$

α (and sometimes β) denotes a type variable. The $\rightarrow$ and $*$ operators construct function and pair types as in ML. The ML types are augmented by *infinitary tuple types*, $\Pi\langle\rho\rangle$. The type constructor $\Pi\langle_\rangle$ takes a row argument denoting an infinite sequence of types. γ (and sometimes δ) denotes a row variable, $(_ ; _)$ constructs a new row by adding a type to an existing row, $(_ \underline{*} _)$ constructs a row of pairs from two rows, $\underline{opt}$ constructs a row of opt types from another row, and $\underline{abs}$ denotes a row composed entirely of abs types. Equality of types is defined by the following set, E_1, of axioms:

$$\underline{abs} \stackrel{E_1}{=} \mathtt{abs} ; \underline{abs}$$

$$(\alpha ; \gamma)\ \underline{opt} \stackrel{E_1}{=} \alpha\ \mathtt{opt} ; \gamma\ \underline{opt}$$

$$(\alpha ; \gamma) \underline{*} (\beta ; \delta) \stackrel{E_1}{=} (\alpha * \beta) ; (\gamma \underline{*} \delta)$$

The E_1-equality relation on the type terms generated by E_1 is written $=_{E_1}$. The typing rules for ML^{Π} are those in ML extended with the rule:

$$\frac{A \vdash_{\mathrm{ML}^{\Pi}} M : \sigma \qquad \sigma =_{E_1} \tau}{A \vdash_{\mathrm{ML}^{\Pi}} M : \tau} \quad \text{(EQUAL)}$$

We assume the following constants on tuples:

$$\begin{array}{rl} \mathtt{nil} : & \Pi\langle \underline{abs} \underline{*} \gamma\ \underline{opt}\rangle \\ \mathtt{car} : & \Pi\langle \alpha ; \gamma\rangle \rightarrow \alpha \\ \mathtt{cdr} : & \Pi\langle \alpha ; \gamma\rangle \rightarrow \Pi\langle\gamma\rangle \\ \mathtt{cons} : & \alpha \rightarrow \Pi\langle\gamma\rangle \rightarrow \Pi\langle \alpha ; \gamma\rangle \end{array}$$

The value nil is an infinitary tuple whose elements are all the value (Abs,None). The procedure car retrieves the first element of an infinitary tuple, cdr returns an infinitary tuple obtained by removing the first element of a given infinitary tuple, and cons constructs a new infinitary tuple from a given value and infinitary tuple. Aside from nil, there are no infinitary tuples whose elements are all of the same value (such as those returned by Rémy's elevation operator [13]).

3.2 Transformation Rules

We now give rules for translation of expressions of ML^{va} into expressions of ML^{Π}. The following syntactic sugar is used:

$$M/\overline{n} \stackrel{\text{def}}{=} \mathtt{car}(\underbrace{\mathtt{cdr}(\ldots}_{n-1 \text{ times}}\ M)\ldots) \qquad n \geq 1$$

$$M\backslash\overline{n} \stackrel{\text{def}}{=} \underbrace{\mathtt{cdr}(\ldots M)\ldots}_{n \text{ times}} \qquad n \geq 0$$

$$<\underbrace{M, \ldots}_{n \text{ times}}\ ; N> \stackrel{\text{def}}{=} \underbrace{\mathtt{cons}\ M\ (\ldots\ N)\ldots}_{n \text{ times}} \qquad n \geq 0$$

$M/\overline{n}$ returns the n-th element of the tuple M. $M\backslash\overline{n}$ returns a tuple containing all but the first n elements of M. $< M, \ldots ; N >$ constructs a tuple by adjoining the objects $M, \ldots$ to the tuple N.

The transformation function $\mathcal{T} :: \mathrm{ML}^{va} \Rightarrow \mathrm{ML}^{\Pi}$ is defined by the rules and rule prototypes of Figure 3. The option "[$\cdots$]" and choice "{$\cdots \mid \cdots$}" prototype operators are interpreted as in Figure 2. Variable references

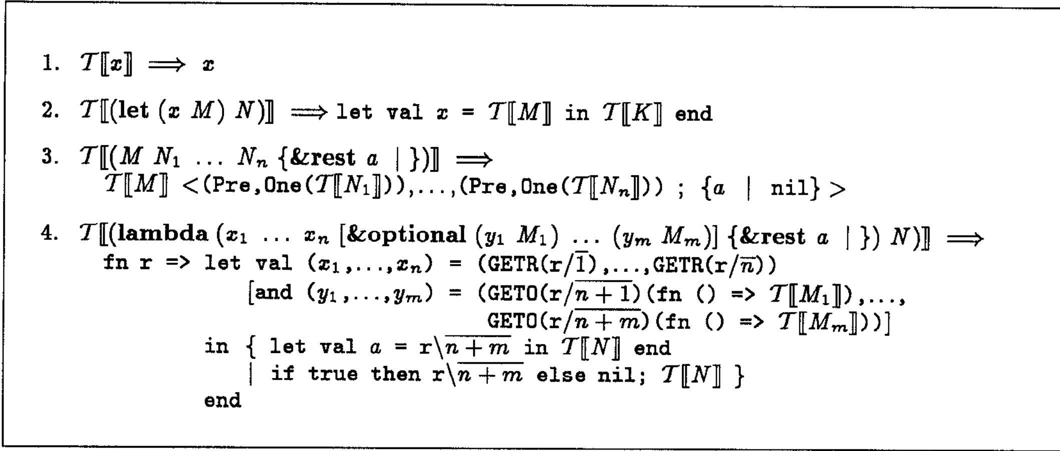

Figure 3: Transformation of ML^{va} expressions to ML^{Π} expressions

are unchanged by the transformation. The translation of let expressions is straightforward. A procedure call in ML^{va} is translated into an ML^{Π} application with one argument, which is a tuple with encoded fields. Each field is represented by a pair consisting of a present or absent mark and a tagged value. The number of Pre's in the encoded tuple indicates the minimal number of arguments passed in the procedure call. If a rest variable is involved in the procedure call, the tuple will be generated by appending the actual arguments (now encoded fields) to the tuple value bound to the rest variable. Otherwise, nil will be used instead as the base tuple. As described earlier, nil is equivalent to <(Abs,None), (Abs,None), $\cdots$ >, and its type may be viewed as $\Pi\langle \mathtt{abs} * \alpha_1\ \mathtt{opt}\ ;\ \mathtt{abs} * \alpha_2\ \mathtt{opt}\ ;\ \cdots\ \rangle$. The auxiliary functions GETR and GETO used in the code generated by translation of lambda expressions may be defined in Standard ML as follows:

```
val GETR : pre * 'a opt -> 'a
  = fn (left,right) =>
      case left of x
        as Pre => case right of y
                    as One(v) => v

val GETO : 'a * 'b opt -> (unit -> 'b) -> 'b
  = fn (_,right) =>
      fn e => case right of x
                as One(v) => v
                 | None => e();
```

Both GETR and GETO take a field of an encoded infinitary tuple. GETR checks for the present mark and then projects the stored value. It is used for extracting the required arguments. GETO does not refer to the present/absent mark of the field, so fields passed to GETO may represent optional arguments. If the field is present, the stored value will be projected and returned. Otherwise, GETO uses its second argument, which is a delayed expression that evaluates to a default value. Those fields that do not correspond to required or optional parameters are bound to the rest variable, if one is provided. Otherwise, the if construct forces unification of the type of $\mathtt{r}\backslash\overline{n+m}$ with the type of nil.

To show the translation from ML^{va} to ML^{Π} preserves the typings of ML^{va} in terms of typings of ML^{Π}, we first define an injection function, h, from types in ML^{va} to types in ML^{Π} by the following rules:

$$h(\rho \to \tau) = \Pi\langle h(\rho)\rangle \to h(\tau)$$

$$\begin{aligned} h(\varphi :: \rho) &= h(\varphi)\ ;\ h(\rho) \\ h(\mu :: \rho_\mu) &= h(\mu)\ ;\ h(\rho_\mu) \\ h(\tau :: \rho_\tau) &= h(\tau)\ \mathtt{opt}\ ;\ h(\rho_\tau) \end{aligned}$$

$$\begin{aligned} h(\rho_\mu \circ \rho_\tau) &= h(\rho_\mu) \mathbin{\underline{*}} h(\rho_\tau) \\ h(\mu \cdot \tau) &= h(\mu) * h(\tau)\ \mathtt{opt} \\ h(\gamma_\tau) &= \gamma_\tau\ \underline{\mathtt{opt}} \end{aligned}$$

$$h(t) = t \qquad \text{otherwise}$$

The function h is overloaded on fields, marks, and various rows, and we assume its natural extensions to type schemes and type environments.

Let B be the ML^{Π} type environment supplying the type schemes of Abs, Pre, None, One, nil, car, cdr, and cons. We write A, A' for the composition of type environments A and A'. The type-consistency of the translation from ML^{va} to ML^{Π} may now be expressed:

Lemma 3.1 (Soundness) *For every M, A in ML^{va} and τ in ML^{Π}, if $B, h(A) \vdash_{\mathrm{ML}^{\Pi}} \mathcal{T}[\![M]\!] : \tau$, then there exists a type σ in ML^{va} such that $h(\sigma) = \tau$ and $A \vdash_{\mathrm{ML}^{va}} M : \sigma$.*

Lemma 3.2 (Completeness) *For every M, A, σ in ML^{va}, if $A \vdash_{\mathrm{ML}^{va}} M : \sigma$, then $B, h(A) \vdash_{\mathrm{ML}^{\Pi}} \mathcal{T}[\![M]\!] : h(\sigma)$.*

Both proofs are by structural induction on the derivation tree of a typing judgement and follow directly from the transformation rules, the $\vdash_{\mathrm{ML}^{va}}$ and $\vdash_{\mathrm{ML}^{\Pi}}$ relations, and the definition of h. Additional details of these proofs and others in this paper will be reported elsewhere [4].

3.3 Infinitary Tuple Algebra

The terms of types in ML^{Π} can be derived from the raw terms of an equational algebra, which we call the "algebra of infinitary tuples." We prove the unification problem for the terms of this algebra is decidable and unitary unifying. This result provides the basis for the type reconstruction algorithm of ML^{Π} and, by extension, ML^{va}.

$$
\begin{array}{rcll}
f^{Row}((\alpha_1 \,;\, \gamma_1), \,\ldots,\, (\alpha_n \,;\, \gamma_n)) & \stackrel{E}{=} & f^{Type}(\alpha_1,\ldots,\alpha_n) \,;\, f^{Row}(\gamma_1,\ldots,\gamma_n) & (f \;\triangleleft\triangleright\; ;) \\
f^{Row}(\partial(\alpha_1),\ldots,\partial(\alpha_n)) & \stackrel{E}{=} & \partial(f^{Type}(\alpha_1,\ldots,\alpha_n)) & (f \;\triangleleft\triangleright\; \partial) \\
\alpha \,;\, \partial(\alpha) & \stackrel{E}{=} & \partial(\alpha) & (; \;\triangleleft\triangleright\; \partial)
\end{array}
$$

where $f \in \mathcal{C}$ and $\varpi(f) = n$

Figure 4: The set, E, of axioms for the infinitary tuple algebra

The algebra is parameterized by a set, $\mathcal{C} = \bigcup_{n\geq 0} \mathcal{C}_n$, of symbols, where an element $f \in \mathcal{C}_n$ has arity $\varpi(f) = n$. Let $\mathcal{K}$ be the set composed of the sorts *Type* and *Row*, and Σ be the signature composed of the following symbols of the indicated sorts:

$$
\begin{array}{rcll}
\Pi\langle_\rangle & :: & Row \Rightarrow Type & \\
(_ \,;\, _) & :: & Type \otimes Row \Rightarrow Row & \\
\partial & :: & Type \Rightarrow Row & \\
f^{\iota} & :: & \iota^{\varpi(f)} \Rightarrow \iota & f \in \mathcal{C},\ \iota \in \mathcal{K}
\end{array}
$$

The ∂ symbol is an addition to the row constructors given in ML^{Π} earlier that may be viewed as a row of shared types analogous to those used by Rémy [13, 14]. It is required in our algebra because unification would not be unitary without its presence. An example on this is given later. Let $\mathcal{V}$ be a denumerable set of variables with infinitely many variables of every sort in $\mathcal{K}$. The set of terms of the infinitary tuple algebra generated by Σ and $\mathcal{V}$ is denoted $T(\Sigma, \mathcal{V})$, or simply T.

Let E be the set of axioms in Figure 4. All axioms are *collapse-free*; that is, none are of the form $x \stackrel{E}{=} t$ where x is a variable that occurs in t (but $x \neq t$). They are also *regular*, since the terms on either side of each axiomatic equation contain the same set of variables. The E-equality relation on T generated by E is written $=_E$.

Definition The *algebra of infinitary tuples* is the equational theory $T(\Sigma, \mathcal{V})/E$ (a quotient of terms by E-equality).

A set of axioms is *syntactic* if any two equal terms can be proved equal by applying axioms at most once at the top occurrence. [9, 10] See Appendix A for a more formal definition.

Theorem 3.1 *E is syntactic.*

The proof is tedious, involving twelve cases on the outermost structures of both sides of an equivalence. See Appendix A for more detail.

The resulting equational theory, $=_E$, is thus syntactic. From Kirchner's theory of unification for syntactic equational theories [8], we deduce that unification can be expressed for our equational theory via rewrite rules in which unification of terms with conflicting top symbols is resolved via the following rules:

$$
\frac{f^{Row}(\rho_1,\ldots,\rho_n) \stackrel{?}{=} (\tau \,;\, \varrho) \stackrel{?}{=} e}{\exists(\alpha_i,\gamma_i)_{i\in[1,n]} \cdot \bigwedge \begin{cases} (\tau \,;\, \varrho) \stackrel{?}{=} e \\ \tau \stackrel{?}{=} f^{Type}(\alpha_1,\ldots,\alpha_n) \\ \varrho \stackrel{?}{=} f^{Row}(\gamma_1,\ldots,\gamma_n) \\ \rho_i \stackrel{?}{=} \alpha_i \,;\, \gamma_i & \forall\, i \in [1,n] \end{cases}}
$$

$$
\frac{f^{Row}(\rho_1,\ldots,\rho_n) \stackrel{?}{=} \partial(\tau) \stackrel{?}{=} e}{\exists(\alpha_i)_{i\in[1,n]} \cdot \bigwedge \begin{cases} \partial(\tau) \stackrel{?}{=} e \\ \tau \stackrel{?}{=} f^{Type}(\alpha_1,\ldots,\alpha_n) \\ \rho_i \stackrel{?}{=} \partial(\alpha_i) & \forall\, i \in [1,n] \end{cases}}
$$

$$
\frac{(\tau \,;\, \rho) \stackrel{?}{=} \partial(\sigma) \stackrel{?}{=} e}{\partial(\sigma) \stackrel{?}{=} e \;\wedge\; \tau \stackrel{?}{=} \sigma \;\wedge\; \rho \stackrel{?}{=} \partial(\sigma)}
$$

Since the rows in our algebra are all of one sort (unlike Rémy's algebra of record terms), some of the axioms are *subterm-collapsing* (there are axioms of the form $t \stackrel{E}{=} t'$ in E such that t is a subterm of t'). Hence $=_E$ is not *strict* (there are terms in T that are E-equal to a subterm of themselves). Though this behavior complicates the process of obtaining a terminating unification algorithm, we have the following property of non-strictness in our algebra:

Lemma 3.3 *If $\rho =_E \varrho$ and ρ is a (proper) subterm of ϱ, then there exists a term τ such that $\rho =_E \partial(\tau)$.*

The proof is based on the forms of the axioms in E and Corollary A.1 in Appendix A.

For strict equational theories, cycles are handled by the so-called "occurs-check" rule. Since our theory is not strict, we instead use the following rules, which are justified by Theorem 3.1 and Lemma 3.3 (a *cycle variable* is one that participates in a cycle [8]):

$$
\frac{\begin{array}{c} f^{Row}(\rho_1,\ldots,\rho_n) \stackrel{?}{=} \gamma \stackrel{?}{=} e \\ \gamma \text{ is a cycle variable in } f^{Row}(\rho_1,\ldots,\rho_n) \end{array}}{\exists(\alpha_i)_{i\in[1,n]} \cdot \bigwedge \begin{cases} \partial(f^{Type}(\alpha_1,\ldots,\alpha_n)) \stackrel{?}{=} \gamma \stackrel{?}{=} e \\ \rho_i \stackrel{?}{=} \partial(\alpha_i) & \forall\, i \in [1,n] \end{cases}}
$$

$$
\frac{(\tau \,;\, \rho) \stackrel{?}{=} \gamma \stackrel{?}{=} e \qquad \gamma \text{ is a cycle variable in } (\tau \,;\, \rho)}{\partial(\tau) \stackrel{?}{=} \rho \stackrel{?}{=} \gamma \stackrel{?}{=} e}
$$

$$
\frac{\begin{array}{c} t \stackrel{?}{=} x \stackrel{?}{=} e \\ x \text{ is a cycle variable in } t \\ t \text{ is not of the form } (\tau \,;\, \rho) \text{ or } f^{Row}(\rho_1,\ldots,\rho_n) \end{array}}{\textbf{Failure}}
$$

As mentioned earlier, the addition of the ∂ symbol (and its associated axioms) is crucial for unification in our algebra. For example, with only the $(f \;\triangleleft\triangleright\; ;)$ axiom in Figure 4, the solution to a unification problem such as $(\alpha \,;\, \gamma) \stackrel{?}{=} \gamma$ would not be unitary.

The basis for obtaining a terminating, syntactically sound and complete type reconstruction algorithm for the type systems of ML^{Π} and ML^{va} may now be stated.

Theorem 3.2 *Unification in the algebra of infinitary tuples is decidable and unitary (every solvable unification problem has a principal unifier).*

The unification algorithm is based on the above rewrite rules. The algorithm is unitary since no rule introduces a disjunction. Termination is proved by showing a decrease of the number of symbols f^{Row} and $(_;_)$, lexicographically ordered by increasing rank (corresponding to the depth of their nesting).

3.4 Principal Typing Theory and Type Reconstruction Algorithm

We show that the type system of ML^{Π} has the principal typing property and a terminating type reconstruction algorithm, and demonstrate that ML^{va} inherits these results.

Instantiate the algebra of infinitary tuples given in the last section by letting $\mathcal{C}_0 = \{pre,\ abs\}$, $\mathcal{C}_1 = \{opt\}$, $\mathcal{C}_2 = \{(_ \to _),\ (_ * _)\}$, and $\mathcal{C}_n = \emptyset$ for $n \geq 3$. Let the resulting signature, Σ', be restricted to contain only the following symbols, given with their sorts ($\Pi\langle_\rangle$, $(_;_)$ and ∂ are not listed):

$$\begin{aligned} pre^{Type} &:: Type \\ abs^{Type} &:: Type \\ opt^{Type} &:: Type \Rightarrow Type \\ (_ * _)^{Type} &:: Type \otimes Type \Rightarrow Type \\ (_ \to _)^{Type} &:: Type \otimes Type \Rightarrow Type \end{aligned}$$

$$\begin{aligned} \underline{abs}^{Row} &:: Row \\ \underline{opt}^{Row} &:: Row \Rightarrow Row \\ (_ \underline{*} _)^{Row} &:: Row \otimes Row \Rightarrow Row \end{aligned}$$

Let pre^{Type}, abs^{Type}, opt^{Type}, $(_ * _)^{Type}$, and $(_ \to _)^{Type}$ be synonymous with **pre**, **abs**, **opt**, $*$, and $\to$; and let abs^{Row}, opt^{Row}, and $(_ * _)^{Row}$ be synonymous with $\underline{abs}$, $\underline{opt}$, and $\underline{*}$. The resulting equational theory $T(\Sigma', \mathcal{V})/\overline{E}$ is named the *algebra of* ML^{Π} *types*.

Corollary 3.1 *Unification in the algebra of* ML^{Π} *types is decidable and unitary.*

Theorem 3.3 (Rémy [12]) *Extension of the ML type systm by allowing a sorted regular equational theory on types preserves principal typings if unification in the theory is decidable and unitary.*

Since the equational theory $=_{E}$ is regular and its unification is decidable and unitary, by applying Theorem 3.3, we obtain:

Corollary 3.2 *There exists a terminating, syntactically sound and complete type reconstruction algorithm for* ML^{Π} *that deduces principal types.*

Hence the ML^{va} type system has the desired property:

Theorem 3.4 *There exists a terminating, syntactically sound and complete type reconstruction algorithm for* ML^{va} *that deduces principal types.*

The proof follows directly from Corollary 3.2 and Lemmas 3.1 and 3.2.

4 Extensions

In this section we consider several extensions of ML^{va} that illustrate the power of its variable-arity typing framework. First we extend the type terms of ML^{va} to accommodate ML base types and type constructors, such as *int* and *list*:

$$\begin{array}{llll} \text{type} & \tau & ::= & \cdots \mid b \mid \tau\ c \\ \text{type row} & \rho_\tau & ::= & \cdots \mid \underline{b} \mid \rho\ \underline{c} \mid \partial(\tau) \end{array}$$

where b and $\underline{b}$ denote base types and base rows, and c and $\underline{c}$ denote unary type constructors and row constructors. The set, E_0, of type equality axioms must then be extended to include the following:

$$\underline{b} \overset{E_0}{=} b :: \underline{b}$$
$$(\alpha :: \gamma)\ \underline{c} \overset{E_0}{=} (\alpha\ c) :: (\gamma\ \underline{c})$$

$$\underline{b} \overset{E_0}{=} \partial(b)$$
$$\partial(\alpha)\ \underline{c} \overset{E_0}{=} \partial(\alpha\ c)$$

$$\partial(\alpha) \overset{E_0}{=} \alpha :: \partial(\alpha)$$

It is easy to check that Theorem 3.4 still holds with the addition of these rules to the equational theory $=_{E_0}$.

Some examples of variable-arity procedures that may be typed in ML^{va} follow:

$$\begin{aligned} + &: (\gamma \circ \underline{int}) \to int \\ list &: (\gamma \circ \partial(\alpha)) \to (\alpha\ list) \\ map &: (pre \cdot ((\gamma \circ \delta) \to \alpha) :: (\gamma \circ (\delta\ \underline{list}))) \to (\alpha\ list) \end{aligned}$$

The primitive + takes any number of arguments as long as they are all of type *int*. The row variable, γ, is universally quantified and may be instantiated to any mark row of the form $pre :: \cdots :: pre :: \underline{abs}$, expressing the number of arguments in a given call. The primitive *list* creates a homogeneous list by putting a shared type restriction on its arguments. The call (*list* **&rest** *a*) may be used to package an unspecified number of rest arguments bound to *a* into a list and forces *a* to be homogeneous.

Though + and *list* may be typed with much simpler variable-arity type systems, *map* illustrates the power of ML^{va}. It takes at least one argument, which must be a procedure. The remaining arguments must be homogeneous lists of certain types that are compatible with the domain type of the procedure. The two occurrences of row variable δ guarantee this congruence. The number of list arguments must also be compatible with the arity of the procedure, which is assured by the two occurrences of row variable γ. For instance, consider

$$(map\ add1\ (list\ 1\ 2\ 3)) \tag{3}$$
$$(map\ +\ (list\ 1\ 2)\ (list\ 3\ 4)) \tag{4}$$
$$(map\ make\text{-}list\ (list\ 1\ 2)\ (list\ \text{"a"}\ \text{"b"})) \tag{5}$$

The primitive *add1* takes one required integer argument only. In typing (3) the type of *map* is instantiated by substituting $(pre :: \underline{abs})$ for γ, $(int :: \delta')$ for δ, and *int* for α. In (4)

the type of $+$ is instantiated as $(pre :: pre :: \underline{abs}) \circ \underline{int} \to int$ and the type of *map* is instantiated as

$$(pre \cdot ((pre :: pre :: \underline{abs}) \circ \underline{int} \to int) :: (pre :: pre :: \underline{abs}) \circ (\underline{int}\ \underline{list})) \to (int\ list)$$

by substituting $(pre :: pre :: \underline{abs})$ for γ, $\underline{int}$ for δ, and int for α. The primitive *make-list*, for which the type scheme $(pre \cdot int :: pre \cdot \alpha :: \underline{abs} \circ \gamma) \to (\alpha\ list)$ is assigned, takes an integer argument n and a second argument v and returns a list of v's with length n. In typing (5) the type of *make-list* is instantiated by substituting *string* for α and δ' for γ, while the type of *map* is instantiated by substituting $(pre :: pre :: \underline{abs})$ for γ, $(int :: string :: \delta')$ for δ, and $(string\ list)$ for α.

The ML^{va} *optional_part* declaration may be extended to include a group of optional variables along with an optional boolean-valued variable indicating the presence or absence of actual arguments for the group:

optional_part ::= ([*variable*] {(*variable expression*)}$^+$)

The ML^{va} type system has the power to assure that all arguments corresponding to the same group of optional parameters are either present or absent in a given call. The (FUN) rule is easily modified to give the same field mark μ to all optional parameters in the same group, instead of distinct marks. For example, if

(lambda (&optional $((x\ 0)\ (y\ 0)))\ (+\ x\ y))$

is bound to f, the calls (f) and $(f\ 1\ 2)$ are both well-typed, but $(f\ 1)$ is not.

Let expressions may be extended to allow polymorphic binding of rest variables. We extend the ML^{va} *let_expression* syntax to allow binding of a sequence of values to a rest variable:

let_expression ::= (**let** *binding expression*)
binding ::= (*variable expression*)
| (*rest_variable* {*expression*}* [**&rest** *rest_variable*])

Semantically, (**let** $(a\ M_1 \ldots M_n$ **&rest** $b)\ M)$ is equivalent to ((**lambda** (**&rest** a) M) $M_1 \ldots M_n$ **&rest** b). The second *rest_variable* in a *binding* allows extension of a sequence of values provided by another rest variable. The type environment is extended by mapping rest variables to *row schemes* and it is straightforward to extend the (RowVAR) and (LET) rules of the ML^{va} type system. Let-bound rest variables then may be referenced polymorphically. For example,

(**let** $(a\ 1\ 2)$
  $(+\ (M\ M_1 \ldots M_m$ **&rest** $a)$
    $(N\ N_1 \ldots N_n$ **&rest** $a)))$

could then be well-typed, while its semantically equivalent counterpart is ill-typed.

Another possible extension is to explicitly specify absent information while passing a sequence of arguments in a procedure call:

procedure_call ::= (*expression* {*argument*}* [**&rest** *rest_variable*])
argument ::= *expression* | ?

The transformation function, $\mathcal{T}$, encodes any argument of the form ? into `(Abs,None)`. For example, let

(lambda (&optional $(x\ 1)\ (y\ 2)\ (z\ 3))\ (+\ x\ y\ z))$

be bound to g. The call $(g\ 1\ ?\ -1)$ would then return 2.

Extension of ML^{va} to support keyword parameters with default values presents no difficulty. Garrigue and Aït-Kaci [7] use a very different record calculus to support keyword parameters. Their procedures are implicitly curried, which dramatically alters the type inference problem.

5 Future Work

The complexity of ML's syntax makes the details messy, but it appears that based on this work a conservative extension of ML to support "enhanced tuples" is possible. This would make all the power of the type system proposed in this paper accessible in ML. The tuple construction syntax would be extended with a form such as $(e_1, \ldots, e_n$[, `&rest` a]) and tuple pattern syntax such as

$(p_1, \ldots, p_n$[, `&optional` $p'_1 = e_1, \ldots, p'_m = e_m$][, `&rest` a])

would be supported. The static (and dynamic) semantics of this extension could, it seems, be expressed with a straightforward translation into ML^{Π} similar to that of Figure 3.

Since lambda-bound variables are not generic in ML^{va} (as in ML), arity-polymorphism may be limited. For example,

((**lambda** $(f)\ (+\ (f\ 1)\ (f\ 2\ 3)))\ +)$

is ill-typed, since f is not allowed to take both one and two arguments, though the semantically equivalent let expression is well-typed. Subtyping may solve this problem: if we introduce an *unknown* mark that is a super-type of both *pre* and *abs*, written $pre \preceq unknown$ and $abs \preceq unknown$, then the lambda-bound variable f above could be typed $(\varepsilon \cdot int :: unknown \cdot int :: \gamma \circ \delta) \to int$ with the set of type constraints $pre \preceq \varepsilon$ and $\underline{abs} \preceq \gamma$.

In such a subtyping system, we could also define *list* (and + similarly) using standard primitives as follows:

(**define** *list*
  (**lambda** (**&optional** (*pre?* (*v nil*)) **&rest** *vals*)
    (**if** *pre?*
      (*cons v* (*list* **&rest** *vals*))
      v)))

The procedure *list* would then be typed $(\underline{unknown} \circ \partial(\alpha)) \to (\alpha\ list)$, which is as general as the type given in Section 4.

With mark subtyping we would also be able to type the primitive *apply*, which takes a procedure and a list as arguments and returns the result of passing the values in the list argument to the procedure argument:

$$apply : (pre \cdot (\underline{unknown} \circ \partial(\alpha) \to \beta) :: pre \cdot (\alpha\ list) :: \underline{abs} \circ \gamma) \to \beta$$

The procedure passed as the first argument to *apply* would be required to accept any number of arguments, all of the same type. We might then define + as follows:

(**define** +
  (**lambda** (**&rest** *nums*)
    (**let** ((*ls* (*list* **&rest** *nums*)))
      (**if** (*null? ls*)
        0
        (*binary+* (*car ls*) (*apply* + (*cdr ls*)))))))

Another approach might be to treat *abs* as a field rather than a field mark and employ the subtype relation $abs \preceq unknown \cdot \tau$ (i.e. types in absent fields are forgotten). This would solve the above problems and also enhance the polymorphism of lambda-bound rest variables. Whether such type systems have desirable properties, such as principal types and decidable type reconstruction, remains to be investigated.

The procedure *map* introduced in Section 4 may be provided as a primitive with the indicated type. Though our type system allows *map* to be defined using standard primitive procedures, it appears that it is not possible to do so in a manner that yields a type as general as that given in Section 4. It remains to be investigated whether an extension of our technique, or any other, can support the inference of the desired type for the *map* procedure from its definition in terms of more primitive procedures.

Rémy conjectures that his projection and elevation operators may be extended to nested records [13]. An analogous extension of our system to support polymorphism of nested infinitary tuples may be possible. Infinitary tuples probably have applications in contexts other than variable-arity procedures in which such an extension may prove especially desirable. Possible areas include type systems for multiple returned values and scaling operators applied to trees, arrays, and other compound structures.

6 Conclusion

We have defined an extension of core ML that supports variable-arity procedures along with a flexible polymorphic type reconstruction system. The extension preserves principal typings and has a terminating type reconstruction algorithm that is sound and complete. A number of examples have illustrated the expressive power of this type system. Our work employs elements of the record typing systems of Wand and Rémy, as well as the theory of equational unification. The typing mechanism presented in this paper has been implemented in Infer [6], a statically-typed dialect of Scheme. Though limited support for variable-arity procedures may be obtained with substantially-simpler type systems, we believe our algebra of infinitary tuples is justified by its expressive power.

References

[1] Military Standard Ada Programming Language. No. ANSI/MIL-STD-1815 A, American National Standards Institute, New York, N.Y., 1983.

[2] W. Clinger and J. Rees (Ed.). Revised[4] Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, 1991.

[3] R. K. Dybvig and R. Hieb. A New Approach to Procedures with Variable Arity. *Lisp and Symbolic Computation: An International Journal*, 3(3):229–244, September 1990.

[4] H. Dzeng and C. T. Haynes. Typing Variable-Arity Procedures with Infinitary Tuples. Technical Report, Computer Science Department, Indiana University, 1994. To appear.

[5] R. Harper, R. Milner, and M. Tofte. *The Definition of Standard ML.* MIT Press, 1990.

[6] C. T. Haynes. Infer: A Statically-typed Dialect of Scheme (preliminary). Technical Report 367, Computer Science Department, Indiana University, revised 1994.

[7] J. Garrigue and H. Aït-Kaci. The Typed Polymorphic Label-Selective λ-Calculus. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 35–47, January 1994.

[8] J.-P. Jouannaud and C. Kirchner. Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic*, pages 257–321. MIT Press, 1991.

[9] C. Kirchner. Computing Unification Algorithms. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 206–216, 1986.

[10] C. Kirchner and F. Klay. Syntactic Theories and Unification. In *Proceedings of the Fifth Symposium on Logic in Computer Science*, pages 270–277, 1990.

[11] D. Rémy. Type Inference for Records in a Natural Extension of ML. Technical Report 1431, INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France, May 1991.

[12] D. Rémy. Extension of ML Type System with a Sorted Equational Theory on Types. Technical Report 1766, INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France, October 1992.

[13] D. Rémy. Projective ML. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, January 1992.

[14] D. Rémy. Syntactic Theories and the Algebra of Record Terms. Technical Report 1869, INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France, March 1993.

[15] G. L. Steele Jr. *Common Lisp: The Language.* Digital Press, 1984.

[16] M. Wand. Complete Type Inference for Simple Objects. In *Proceedings of the Second Symposium on Logic in Computer Science*, pages 37–44, 1987.

A Appendix: Syntactic Theory

Consider the infinitary tuple algebra $T(\Sigma, \mathcal{V})/E$ of Section 3.3. First let E be extended with the trivial axiom $x \stackrel{E}{=} x$. We write $t \leftrightarrow s$ for the relation such that s can be obtained by replacing a subterm of t with the result of applying an axiom in E. Note that $\leftrightarrow$ is symmetric. The equational theory of E is the transitive closure of the above relation, written $\leftrightarrow^*$ (thus $=_E$ is equivalent to $\leftrightarrow^*$). We write $\stackrel{0}{\leftrightarrow}$ for the sub-relation of $\leftrightarrow$ such that the substitution of an axiom is only applied to the outermost term, while $\stackrel{n,\infty}{\longleftrightarrow}$ is the sub-relation of $\leftrightarrow$ where the substitution is only applied to a subterm at depth n or greater. Juxtaposition of relations denotes their composition, and $X \subseteq Y$ indicates that X is a sub-relation of Y.

Definition E is *syntactic* if and only if

$$\leftrightarrow^* \subseteq (\stackrel{1,\infty}{\longleftrightarrow})^* \stackrel{0}{\leftrightarrow} (\stackrel{1,\infty}{\longleftrightarrow})^*$$

Lemma A.1 (Rémy [14]) *E is syntactic if and only if*

$$\overset{0}{\leftrightarrow} (\overset{1,\infty}{\longleftrightarrow})^* \overset{0}{\leftrightarrow} \;\subseteq\; (\overset{1,\infty}{\longleftrightarrow})^* \overset{0}{\leftrightarrow} (\overset{1,\infty}{\longleftrightarrow})^*$$

We obtain a rewrite system, R, over T from the equations of Figure 4 by rewriting from right to left in the first two equations and left to right in the third equation, naming these rewrite rules $(;\triangleright f)$, $(\partial \triangleright f)$, and $(;\triangleright\partial)$, respectively. R defines a relation $\rightarrow$ on terms of T, which is a sub-relation of $\leftrightarrow$ and is anti-symmetric. We say t *contracts to* s if $t \rightarrow s$ and t *reduces to* s if $t \rightarrow^* s$. A term t is in *normal form* if it does not contract to any term.

Definition The *size* of a term t in T, written $\Theta(t)$, is defined by the following rules:

$$\begin{aligned}
&\Theta(x) = 1 \qquad \forall\, x \in \mathcal{V}\\
&\Theta(\Pi\langle\rho\rangle) = 1 + \Theta(\rho)\\
&\Theta(f^{Type}(\tau_1, \ldots, \tau_n)) = 1 + n + \Theta(\tau_1) + \ldots + \Theta(\tau_n)\\
&\Theta(f^{Row}(\rho_1, \ldots, \rho_n)) = 1 + \Theta(\rho_1) + \ldots + \Theta(\rho_n)\\
&\Theta(\tau\ ;\ \rho) = 1 + \Theta(\tau) + \Theta(\rho)\\
&\Theta(\partial(\tau)) = 1 + \Theta(\tau)
\end{aligned}$$

"Induction on T" will mean induction on the size of T. The basis case considers terms of size 1, which includes all variables and terms constructed by nullary-arity function symbols.

Theorem A.1 (Strong Normalization) *There are no infinite reduction sequences in R.*

<u>Proof</u>: It is easy to prove that for every t and s in T if $t \rightarrow s$ then $\Theta(t) > \Theta(s)$. The proof then follows by induction on T.

□

Theorem A.2 (Confluence) *If $t \rightarrow^* s$ and $t \rightarrow^* u$, then there exists a term v such that $s \rightarrow^* v$ and $u \rightarrow^* v$.*

<u>Proof</u>: By induction on T.

□

From these theorems we have:

Corollary A.1 *For every t, s in T, if $t =_E s$, then there exists a unique term u in normal form such that $t \rightarrow^* u$ and $s \rightarrow^* u$.*

In other words, terms that are E-equal reduce to the same normal form.

An equational theory is always a *congruence*; that is, if $t_i =_E s_i$ for $i = 1, \ldots, n$, then $f(t_1, \ldots, t_n) =_E f(s_1, \ldots, s_n)$. Our next lemma states that the converse also holds for our theory. That is, for every t and s in T, if t and s have the same top function symbol and $t =_E s$, then their corresponding components are E-equal to each other.

Lemma A.2 *If $f(t_1, \ldots, t_n) =_E f(s_1, \ldots, s_n)$, then $t_i =_E s_i$ for $i = 1, \ldots, n$.*

<u>Proof</u>: By induction on T.
Basis: Trivial.
Induction Step: Assume $t =_E s$ where t and s have the same top function symbol, then by Corollary A.1 there exists a term u such that $t \rightarrow^* u$ and $s \rightarrow^* u$.

Case 1: $t = f^{Row}(\rho_1, \ldots, \rho_n) \ \wedge\ s = f^{Row}(\varrho_1, \ldots, \varrho_n)$

$$\begin{aligned}
&\Longrightarrow\ u = f^{Row}(u_1, \ldots, u_n) \qquad \text{(by } R\text{)}\\
&\Longrightarrow\ \rho_i \rightarrow^* u_i \ \wedge\ \varrho_i \rightarrow^* u_i \quad \forall\, i \in [1, n]\\
&\Longrightarrow\ \rho_i =_E \varrho_i \quad \forall\, i \in [1, n]
\end{aligned}$$

Case 2: $t = \partial(\tau) \ \wedge\ s = \partial(\sigma)$

(Subcase 1) $u = \partial(v)$

$$\begin{aligned}
&\Longrightarrow\ \tau \rightarrow^* v \ \wedge\ \sigma \rightarrow^* v \qquad \text{(by } R\text{)}\\
&\Longrightarrow\ \tau =_E \sigma
\end{aligned}$$

(Subcase 2) $u = f^{Row}(u_1, \ldots, u_n)$

$$\begin{aligned}
&\Longrightarrow\ \tau = f^{Type}(\tau_1, \ldots, \tau_n) \ \wedge\ \sigma = f^{Type}(\sigma_1, \ldots, \sigma_n)\\
&\qquad \wedge\ \tau_i \rightarrow^* \tau'_i \ \wedge\ \sigma_i \rightarrow^* \sigma'_i \quad \forall\, i \in [1, n]\\
&\qquad \wedge\ t \rightarrow^* f^{Row}(\partial(\tau'_1), \ldots, \partial(\tau'_n))\\
&\qquad \wedge\ s \rightarrow^* f^{Row}(\partial(\sigma'_1), \ldots, \partial(\sigma'_n)) \qquad \text{(by } R\text{)}\\
&\Longrightarrow\ \partial(\tau'_i) =_E \partial(\sigma'_i) \quad \forall\, i \in [1, n] \qquad \text{(by Case 1)}\\
&\Longrightarrow\ \tau'_i =_E \sigma'_i \quad \forall\, i \in [1, n] \qquad \text{(by Hypothesis)}\\
&\Longrightarrow\ \tau_i =_E \tau'_i =_E \sigma'_i =_E \sigma_i \quad \forall\, i \in [1, n]\\
&\Longrightarrow\ \tau =_E \sigma \qquad \text{(by Congruence)}
\end{aligned}$$

Case 3: $t = (\tau\ ; \rho) \ \wedge\ s = (\sigma\ ;\ \varrho)$
Involves three subcases similar to the previous cases.

□

We are now prepared to prove our main result (presented earlier as Theorem 3.1).

Theorem A.3 *E is syntactic.*

<u>Proof</u>: The proof, based on Lemma A.1 and Lemma A.2, is divided into twelve cases for all possible pairs of terms of the relation $\overset{0}{\leftrightarrow} (\overset{1,\infty}{\longleftrightarrow})^* \overset{0}{\leftrightarrow}$. For each such pair of terms τ and σ, it is proved that if $\tau \overset{0}{\leftrightarrow} (\overset{1,\infty}{\longleftrightarrow})^* \overset{0}{\leftrightarrow} \sigma$ then $\tau\ (\overset{1,\infty}{\longleftrightarrow})^* \overset{0}{\leftrightarrow} (\overset{1,\infty}{\longleftrightarrow})^*\ \sigma$. Symbols written under $\leftrightarrow$ indicate the associated axiom of Figure 4.

Assume $\tau \overset{0}{\leftrightarrow} (\overset{1,\infty}{\longleftrightarrow})^* \overset{0}{\leftrightarrow}\ \sigma$.

Case 1: $(\underset{f\lhd\rhd;}{\overset{0}{\longleftrightarrow}} (\overset{1,\infty}{\longleftrightarrow})^* \underset{;\lhd\rhd f}{\overset{0}{\longleftrightarrow}})$

If $\tau = f^{Row}((\tau_1; \rho_1), \ldots, (\tau_n; \rho_n))$,
then $\sigma = f^{Row}((\sigma_1; \varrho_1), \ldots, (\sigma_n; \varrho_n))$ and

$$\begin{aligned}
&\tau \overset{0}{\leftrightarrow} (f^{Type}(\tau_1, \ldots, \tau_n)\ ;\ f^{Row}(\rho_1, \ldots, \rho_n))\\
&\quad (\overset{1,\infty}{\longleftrightarrow})^* (f^{Type}(\sigma_1, \ldots, \sigma_n)\ ;\ f^{Row}(\varrho_1, \ldots, \varrho_n))\\
&\quad \overset{0}{\leftrightarrow}\ \sigma\\
&\Longrightarrow\ f^{Row}(\rho_1, \ldots, \rho_n) \leftrightarrow^* f^{Row}(\varrho_1, \ldots, \varrho_n)\\
&\qquad \wedge\ \tau_i \leftrightarrow^* \sigma_i \quad \forall\, i \in [1, n]\\
&\Longrightarrow\ \text{(by Lemma A.2)}\\
&\qquad \rho_i \leftrightarrow^* \varrho_i \quad \forall\, i \in [1, n]\\
&\Longrightarrow\ \tau\ (\overset{2,\infty}{\longleftrightarrow})^*\ \sigma\\
&\Longrightarrow\ \tau\ (\overset{1,\infty}{\longleftrightarrow})^* \overset{0}{\leftrightarrow} (\overset{1,\infty}{\longleftrightarrow})^*\ \sigma
\end{aligned}$$

Case 2: $(\underset{;\lhd\rhd f}{\overset{0}{\longleftrightarrow}} (\overset{1,\infty}{\longleftrightarrow})^* \underset{f\lhd\rhd;}{\overset{0}{\longleftrightarrow}})$

If $\tau = (f^{Type}(\tau_1, \ldots, \tau_n)\ ;\ f^{Row}(\rho_1, \ldots, \rho_n))$,
then $\sigma = (f^{Type}(\sigma_1, \ldots, \sigma_n)\ ;\ f^{Row}(\varrho_1, \ldots, \varrho_n))$ and

$$\tau \overset{0}{\leftrightarrow} f^{Row}((\tau_1;\rho_1),\ldots,(\tau_n;\rho_n))$$
$$(\overset{1,\infty}{\longleftrightarrow})^* f^{Row}((\sigma_1;\varrho_1),\ldots,(\sigma_n;\varrho_n))$$
$$\overset{0}{\leftrightarrow} \sigma$$
$$\Longrightarrow (\tau_i \; ; \; \rho_i) \leftrightarrow^* (\sigma_i \; ; \; \varrho_i) \quad \forall\, i \in [1,n]$$
$$\Longrightarrow \text{(by Lemma A.2)}$$
$$\tau_i \leftrightarrow^* \sigma_i \;\wedge\; \rho_i \leftrightarrow^* \varrho_i \quad \forall\, i \in [1,n]$$
$$\Longrightarrow \tau \, (\overset{2,\infty}{\longleftrightarrow})^* \, \sigma$$
$$\Longrightarrow \tau \, (\overset{1,\infty}{\longleftrightarrow})^* \overset{0}{\leftrightarrow} (\overset{1,\infty}{\longleftrightarrow})^* \, \sigma$$

Case 3: $(\underset{\partial \triangleleft\triangleright f}{\overset{0}{\longleftrightarrow}} (\overset{1,\infty}{\longleftrightarrow})^* \underset{f \triangleleft\triangleright \partial}{\overset{0}{\longleftrightarrow}})$
If $\tau = \partial(f^{Type}(\tau_1,\ldots,\tau_n))$,
then $\sigma = \partial(f^{Type}(\sigma_1,\ldots,\sigma_n))$ and

$$\tau \overset{0}{\leftrightarrow} f^{Row}(\partial(\tau_1),\ldots,\partial(\tau_n))$$
$$(\overset{1,\infty}{\longleftrightarrow})^* f^{Row}(\partial(\sigma_1),\ldots,\partial(\sigma_n))$$
$$\overset{0}{\leftrightarrow} \sigma$$
$$\Longrightarrow \partial(\tau_i) \leftrightarrow^* \partial(\sigma_i) \quad \forall\, i \in [1,n]$$
$$\Longrightarrow \text{(by Lemma A.2)}$$
$$\tau_i \leftrightarrow^* \sigma_i \quad \forall\, i \in [1,n]$$
$$\Longrightarrow \tau \, (\overset{2,\infty}{\longleftrightarrow})^* \, \sigma$$
$$\Longrightarrow \tau \, (\overset{1,\infty}{\longleftrightarrow})^* \overset{0}{\leftrightarrow} (\overset{1,\infty}{\longleftrightarrow})^* \, \sigma$$

Case 4: $(\underset{f \triangleleft\triangleright \partial}{\overset{0}{\longleftrightarrow}} (\overset{1,\infty}{\longleftrightarrow})^* \underset{\partial \triangleleft\triangleright f}{\overset{0}{\longleftrightarrow}})$
If $\tau = f^{Row}(\partial(\tau_1),\ldots,\partial(\tau_n))$,
then $\sigma = f^{Row}(\partial(\sigma_1),\ldots,\partial(\sigma_n))$ and

$$\tau \overset{0}{\leftrightarrow} \partial(f^{Type}(\tau_1,\ldots,\tau_n))$$
$$(\overset{1,\infty}{\longleftrightarrow})^* \partial(f^{Type}(\sigma_1,\ldots,\sigma_n))$$
$$\overset{0}{\leftrightarrow} \sigma$$
$$\Longrightarrow \tau_i \leftrightarrow^* \sigma_i \quad \forall\, i \in [1,n]$$
$$\Longrightarrow \tau \, (\overset{2,\infty}{\longleftrightarrow})^* \, \sigma$$
$$\Longrightarrow \tau \, (\overset{1,\infty}{\longleftrightarrow})^* \overset{0}{\leftrightarrow} (\overset{1,\infty}{\longleftrightarrow})^* \, \sigma$$

Case 5: $(\underset{; \triangleleft\triangleright \partial}{\overset{0}{\leftrightarrow}} (\overset{1,\infty}{\longleftrightarrow})^* \underset{\partial \triangleleft\triangleright f}{\overset{0}{\leftrightarrow}})$
If $\tau = (f^{Type}(\tau_1,\ldots,\tau_n) \; ; \; \partial(f^{Type}(\tau_1,\ldots,\tau_n)))$,
then $\sigma = f^{Row}(\partial(\sigma_1),\ldots,\partial(\sigma_n))$ and

$$\tau \overset{0}{\leftrightarrow} \partial(f^{Type}(\tau_1,\ldots,\tau_n))$$
$$(\overset{1,\infty}{\longleftrightarrow})^* \partial(f^{Type}(\sigma_1,\ldots,\sigma_n))$$
$$\overset{0}{\leftrightarrow} \sigma$$
$$\Longrightarrow \tau_i \leftrightarrow^* \sigma_i \quad \forall\, i \in [1,n]$$
$$\Longrightarrow \tau \, (\overset{1,\infty}{\longleftrightarrow})^* f^{Type}(\sigma_1,\ldots,\sigma_n); f^{Row}(\partial(\sigma_1),\ldots,\partial(\sigma_n))$$
$$\overset{0}{\leftrightarrow} f^{Row}((\sigma_1;\partial(\sigma_1)),\ldots,(\sigma_n;\partial(\sigma_n)))$$
$$(\overset{1,\infty}{\longleftrightarrow})^* \sigma$$

Cases 6–12 are similar.

□