Haskellへの多相バリアントの導入について

Introduction of Polymorphic Variants into Haskell

香川 考司 †

Koji KAGAWA

† 香川大学工学部信頼性情報システム工学科

RISE, Kagawa University

kagawa@eng.kagawa-u.ac.jp

Haskell の型システムは、パラメトリック型クラス（あるいは関数従属性つきの型クラス）に拡張されることにより、多相バリアントを模倣するための表現力を十分持つ。しかし、多相バリアントが現実に使われることはほとんどない。これは、多相バリアントを模倣するためのコードが巨大になりがちであることと、曖昧さの問題に悩まされることが原因と考えられる。

本稿ではまず、多相バリアントを Haskell の型システム（Haskell 98 + ポピュラーな拡張）で模倣する方法を説明し、次に多相バリアントを直接扱うことのできる Haskell の型クラスの拡張について説明する。この型システムは型推論の結果として、通常の Haskell のコードを生成することができ、拡張言語のトランスレータとして振舞うことができる。プログラマは曖昧さのような厄介な問題に悩まされることなく多相バリアントを利用することができる。

1 はじめに

Hindley-Milner の型システムの多相レコード/バリアントへの拡張は活発に研究され、その成果もよく知られている（例 [8, 9]）。

バリアントとレコードはプログラミング言語の理論のなかで双対な概念である。多相バリアントは拡張可能な代数的データ型とみなすことができる。

多相バリアントは潜在的にさまざまなアプリケーション領域 — 例えば領域特化言語のインタプリタ、グラフィカルユーザインタフェース (GUI) ライブラリ、データベースインタフェースなど、— 前もって必要な構成子の数が決定できない場合に有用である。例えば、Haskell でインタプリタを作成することは、特にモナドの普及以来、ひじょうに容易になった。ひじょうに小さな言語のインタプリタを書くとき、次のようなデータ型が必要になる。

```
data Expr = Var String | App Expr Expr
          | Lambda String Expr
```

ここで `Lambda "x" (Var "x")` は “$\lambda x.x$” という式の内部表現である。

あとで、例えば引数の揃った関数適用を特別扱いするために、新しい構成子を追加したくなるかもしれない。

```
data ExprF extends Expr
  = FullApp ExprF [ExprF]
```

(**注:** この記法はあくまでも暫定的なものである。)

別のバリエーションでは、言語に手続き的な特徴を導入したいかもしれない。

```
data ExprS extends Expr =
  Setq String ExprS | Read | Write ExprS
```

このように、核となる構成子を共有するが、少しずつ異なる構成子をもつ拡張を考えることができる。

このように多相バリアントは潜在的に様々な用途があるが、現実には使われることは少ない。著者の知る限り関数型言語ではわずかに Objective Caml[2] がバージョン 3 以降多相バリアントをサポートしているのみである。多相バリアントを導入する際にはなぜこれまで多相バリアントがあまり使われなかったかを考慮する必要がある。

本稿では、まず多相バリアントが Haskell でどのように表現されるかを説明する（2 節）。この表現は Haskell98 の拡張を必要とする。しかし、この拡張（関数従属性多引数型クラス [6]）は、少なくともメジャーな 2 つの Haskell の実装である GHC と Hugs で利用可能である。このエンコーディングは難解ではないが、少なくともプログラマが手で書くには面倒すぎる。そこで、多相バリアントを直接扱える型クラスの拡張を提案する（3 節）。型クラス宣言の特

別な場合として、多相バリアントの宣言と、さらにレコードとバリアントを対称的に扱う新しい形式のインスタンス宣言を導入する。最後にまとめを述べる（4 節）。

2 Haskell での多相バリアントの表現

2.1 関数従属性をもつ型クラス

Haskell の型クラスは一般的で強力なシステムであるが、Haskell98 で定められたシステムそのものでは多相レコード/バリアントを表現することはできない。しかし、パラメトリック型クラス [1] を使用すれば多相レコード/バリアントを表現できることは知られている。さらに関数従属性を持つ型クラスはパラメトリック型クラスをさらに一般化したもので、主な Haskell の処理系に導入されている。型パラメータの間の従属性はクラス定義のなかの | の後の部分で表される。

```
class Foo a b c | a b → c where ...
```

ここで →の左辺に現れる a と b が独立なパラメータで、右辺に現れる c が a と b に従属するパラメータである。つまり、もし第 1、第 2 引数を共通にもつ 2 つの述語 Foo x y z と Foo x y w がひとつの述語集合にあれば、z と w も一致しなければならない。

多相バリアントは単純に唯一の独立型パラメータがメンバ関数の戻り値の位置に出現する型クラスとして表現できる。

```
class List s x | s → x where
  cons :: x → s → s
  nil  :: s
```

さらにサブクラスで構成子を追加することができる。

```
class List s x ⇒
   AppendList s x | s → x  where
  unit   :: x → s
  append :: s → s → s
```

さらに構成子の型がこれらのクラスの型に一致する“標準インスタンス型” とでも呼ぶべきこれらの型クラスのインスタンスを定義することができる。

```
data T_List x =
  Cons_List x (T_List x) | Nil_List

data T_AppendList x =
    Cons_AppendList x (T_AppendList x)
  | Nil_AppendList
  | Unit_AppendList x
  | Append_AppendList (T_AppendList x)
                      (T_AppendList x)
```

当然、これらの間の自明なインスタンス宣言も必要である。

```
instance List (T_List x) x where
   cons = Cons_List
   nil  = Nil_List
instance List (T_AppendList x) x where
   cons = Cons_AppendList
   nil  = Nil_AppendList
instance AppendList (T_AppendList x) x
where unit   = Unit_AppendList
      append = Append_AppendList
```

多相バリアントを引数として受け取る関数を、この “標準インスタンス型” をパターン部分に利用して定義することができる。

```
lengthL Nil_List         = 0
lengthL (Cons_List _ xs) = 1 + lengthL xs

sumA Nil_AppendList = 0
sumA (Cons_AppendList x xs) = x + sumA xs
sumA (Unit_AppendList x)    = x
sumA (Append_AppendList xs ys) =
   sumA xs + sumA ys
```

例えば sumA は append に対するケースを明示的に持ち、lengthL は append に対するケースを持たない。そして、クラス List に定義されたメンバ関数は lengthL (cons 1 nil) と sumA (cons 2 nil) のようにどちらの関数にも渡すことができる。これが多相バリアントの意味するところである。

2.2 関数の再利用

List に対して定義された関数を AppendList に対して再利用するために次のように新しい関数を定義したくなるかもしれない。

```
lengthA (Append_AppendList xs ys) =
  lengthA xs + lengthA ys
lengthA (Unit_AppendList x) = 1
-- もともとある構成子
```

```
  lengthA (Cons_AppendList z zs) =
    lengthL (Cons_List z zs)
  lengthA Nil_AppendList = lengthL Nil_List
```

しかし、これはもちろんうまくいかない。zs が T_List 型ではないからである。

次のような型変換関数

```
coerce_AppendList_List
  :: T_AppendList x → T_List x
coerce_AppendList_List
  (Append_AppendList Nil_AppendList ys) =
  coerce_AppendList_List ys
coerce_AppendList_List
   (Append_AppendList xs ys) =
  Cons (hdA xs) (coerce_AppendList_List
        (Append_AppendList (tlA xs) ys))

lengthA xs =
      lengthL (coerce_AppendList_List xs)

-- 以下の関数の定義は省略
hdA :: T_AppendList x → x
tlA :: T_AppendList x → x
```

を定義すると length の場合はうまくいくが、一般的には良い方法ではない。なぜなら型変換がディープ、つまり T_AppendList の下位の構成要素までが T_List に型変換されてしまうからである。一般的には T_AppendList の下位の構成要素はその型を維持することが望ましい。例えば、

```
tlA xs = tlL (coerce_ApeendList_List xs)
```

は明らかに良い定義でない。なぜならばその型は、

```
  T_AppendList x → T_List x
-- T_AppendList x → T_AppendList x ではない
```

となるからである。

このように一般に List に対して定義された関数をそれを拡張したデータ型で利用することは容易ではない。このことがこのような拡張可能な代数的データ型がそれほど利用されない理由であると考えられる。つまり、既存の型を拡張してもあまり意味がなく、結局既存の型を定義し直し、それに応じて既存の関数を書き直すことになる。

```
data List x = Nil | Cons x (List x)
   | Append (List x) (List x) | Unit x
```

この場合、書き直すべき関数はソースプログラムの中に散らばっている。さらに悪い場合にはソースがなくコンパイルされた形でしか利用できないこともある。

2.3 開再帰

Objective Caml はバージョン 3.0 以降多相バリアントが導入されており、上の問題に対してはガリグにより次のような開再帰を用いる方法が提案されている [3]。

```
leL_aux le_rec Nil = 0
leL_aux le_rec (Cons _ xs) = le_rec xs

lengthL = leL_aux lengthL
```

ここで、le_rec が再帰呼出しを抽象している。

```
leA_aux le_rec (Append xs ys) =
  le_rec xs + le_rec ys
leA_aux le_rec Nil = leL_aux le_rec Nil
leA_aux le_rec (Cons x xs) =
  leL_aux le_rec (Cons x xs)
```

そして、“結び目をゆわえる” ことにより関数を再利用することができる。

```
lengthA = leA_aux lengthA
```

2.4 再び型クラス

しかし、この方法では、拡張可能なバリアント型に対して再帰関数を定義するたびに、常に再帰呼出しを抽象した高階関数を用意しなければならない。Haskell の型クラスは、もともとこのような高階関数とそれに伴う面倒な操作からプログラマを解放するためものである。この精神に沿った方法として、Haskell で length を新しい型クラスのメンバ関数として定義する。

```
class Length a where
  length :: a → Int

instance Length (T_List x) where
  length Nil_List         = length_Nil
  length (Cons_List x xs) = length_Cons x xs

length_Nil = 0
length_Cons x xs = 1 + length xs
```

```
instance Length (T_AppendList x) where
  length Nil_AppendList = length_Nil
  length (Cons_AppendList x xs) =
    length_Cons x xs
  length (Unit_AppendList x) =
    length_Unit x
  length (Append_AppendList xs ys) =
    length_Append xs ys

length_Unit x = 1
length_Append xs ys =
  length xs + length ys
```

これは開再帰を用いるのと本質的にはおなじ方法である。しかし、もし次のような式を書くと、

```
length (cons 1 (cons 2 nil))
```

Haskell の型チェッカはこの式が次のような曖昧な型を持つというメッセージを出す。

```
(Length a, List a Int) => Int
```

ここで “曖昧” とは $\pi \Rightarrow \tau$ という形の型の中で π の中の変数で τ の中に現れないものがあるということである。するとプログラマはプログラムの意味を確定するために、型を明示的に示さなければならない。この場合は a に対する具体的な型 T_List Int を示すことができる。

```
length (cons 1 (cons 2 nil) :: T_List Int)
```

もし、次のように型パラメータが完全に具体化されないときは、少しトリッキーなコードを書く必要がある。

```
asList :: T_List x → T_List x
asList x = x

length (asList (cons x (cons y nil)))
```

ここでは 2 つの大きな問題があると考えられる。

- プログラマがこのような模倣コードを手で書くのは面倒である。
- 一般的には、明示的な型を全ての必要な場所に追加するのは簡単ではない。

Haskell プログラマは本能的に曖昧な型を避けるようである。これが、この模倣方法がこれまで利用されなかった原因のひとつであろう。しかし、この場合は、曖昧な型は病的な状況を示しているわけではない。実際、次のように曖昧な型変数を別の候補に具体化したとしてもプログラムの意味が変わるわけではない。

```
length (cons 1 (cons 2 nil)
          :: T_AppendList Int)
```

3 レコード/バリアント宣言

そこで、多相バリアントを直接提供し、しかも前節で述べたエンコーディングと同じ効果を持つ型システムを設計する。具体的には、この節で多相バリアントを導入するための宣言形式、多相バリアントに対する関数（メソッド）を宣言するための形式、そして両者の間のインスタンス宣言を導入する。

このシステムは次のような役割を舞台裏で果たす必要がある。

- バリアントに対し “標準インスタンス型” を定義する、
- “標準インスタンス型” を関連するクラスのインスタンスとして宣言する。

プログラマは標準インスタンス型やその構成子の具体的な名前を知らないので、これらはすべて舞台裏で行われなければならない。

またこのシステムは

- 曖昧な型が現れる場所に明示的な型を挿入する

必要がある。これは自明なことではなく、型推論アルゴリズムを拡張することにより、この型情報の挿入を行う。

4 バリアント宣言

多相バリアントを定義する新しい宣言形式を導入する。この形式はパラメトリック型クラスの宣言の形式とほとんど同じである。

variant $\overline{\pi} \Rightarrow \alpha \in \mathit{VariantName}\ \overline{\beta}$ **where**
  $\mathit{Constr}_1 :: \tau_1^1 \to \cdots \to \tau_1^{n_1} \to \alpha$
  ...
  $\mathit{Constr}_m :: \tau_m^1 \to \cdots \to \tau_m^{n_m} \to \alpha$

この宣言は新しいシンボル $Constr_1, \ldots, Constr_m$ を導入する。バリアント宣言に関する制限は、次のようになる。

- 独立な型パラメータ α はすべての関数の戻り値の型として出現しなければならない。つまり関数は $\cdots \to \alpha$ という型を持たなければならない。

例えば拡張可能なリスト型は次のように定義できる。

```
variant xs ∈ List x where
  Nil  :: xs
  Cons :: x → xs → xs
```

通常の **data** 宣言との違いは新しい構成子を追加できることである。

```
variant xs ∈ List x ⇒ xs ∈ List2 x where
  Cons2 :: x → x → xs → xs

variant xs ∈ List x ⇒ xs ∈ AppendList x
where Unit   :: x → xs
      Append :: xs → xs → xs
```

ここでは多相バリアントを型クラスの特別な場合として導入しているが、（標準インスタンス型以外の）データ型をバリアントクラスのインスタンスとして宣言することはできない。

4.1 レコード宣言

ここで、多相バリアントを引数とする関数を定義するために、レコード宣言と呼ぶ新しい宣言形式を導入する。これはレコード宣言で定義されるクラスを通常の型クラスとわけなければ曖昧性の問題が残ってしまうからである。レコードをわけておけば、曖昧な型変数にバリアントクラスとレコードクラスの述語のみが関連しているとき、式の意味はその型変数が具体化される型に依存しない。

“A Second Look at Overloading” [7] で、Odersky らは型クラスの曖昧性の問題を解決するために多重定義されるシンボルの型に単純な制限を課した。つまり、独立な型変数が関数の第 1 引数の型として出現しなければならない、とした。（つまり、α を独立な型変数とするとき、関数は $\alpha \to \cdots$ という型を持たなければならない。）

本稿で提案するシステムは、レコード宣言に対して Oderskey らのシステムと同じ制限を課す。バリアント宣言に関する制限が多重定義された記号が α を独立な型変数とするとき、$\cdots \to \alpha$ という型を持たなければならない、というものだったことを考えると、本稿で提案するシステムは Odersky らのシステムの対称的な拡張と考えることができる。

レコード宣言は **class** の代りに **record** というキーワードを使うことを除けば型クラス宣言と同じである。

```
record π̄ ⇒ α ∈ RecordName β̄ where
  method_1 :: α → τ_1
  ...
  method_m :: α → τ_m
```

これで、新しいシンボル $method_1, \ldots, method_m$ が導入される。α が独立な型変数で $\bar{\beta}$ が α に依存する型変数である。例えば、次のように宣言する。

```
method a ∈ Length where
   length :: a → Int
```

4.2 インスタンス宣言

これまで **record** と **variant** 宣言を **class** 宣言の特別な場合として、導入してきた。しかし、レコード/バリアント間のインスタンス宣言はオリジナルの型クラスのものとは異なるものである。

つまり、本システムではバリアントクラスは通常の意味のインスタンスを持たず、そのかわり、バリアントクラス (*v*) をレコードクラス (*r*) のインスタンスと宣言することができる。その宣言は次のような形式を持つ。

```
vrinstance π̄ ⇒ v τ̄ ∋ υ ∈ r σ̄ where
  method_m (Constr_n  p_1 ... p_{k_n}) = e_m
  ...
```

ここで “∋” と “∈” の間の υ は直感的には “self 型”（$method_m$ の第 1 引数の型）を表し、$\bar{\pi}$ の中に出現しても良い。

インスタンス宣言に関する型ルールを単純に保つために、次のような制限を課す。

- 多相バリアントの構成子は **vrinstance** 宣言のメソッド定義の第 1 引数のトップレベルにのみ出現することができる。

これはレコードクラスのメソッドがバリアントクラスの（あとで追加されるかもしれない）構成子を受取

ることを保証するためのもっとも簡単な制限だと考えられる。もし多相バリアントの構成子がパターンの他の場所に出現することを許せば、メソッドがバリアントクラスの全ての構成子を受取ることを保証するために、より精密な規則が必要になる。さらに

- 多相バリアントの構成子は **vrinstance** 宣言以外の場所でパターン中には使用できない。

v が他のクラスのサブクラスのとき、v で追加された構成子に対する場合のみ **vrinstance** 宣言で定義する。よって、*method*/*Constr* のペアには唯一の定義が割り当てられる。

```
vrinstance List x ∋ xs ∈ Length where
   length Nil         = 0
   length (Cons x xs) = 1 + length xs

vrinstance List2 x ∋ xs ∈ Length where
   length (Cons2 x y xs) =
     length (Cons x (Cons y xs))

vrinstance AppendList x ∋ xs ∈ Length
where length (Unit x) = 1
      length (Append xs ys) =
        length xs + length ys
```

$method_m$ の意味は型に依らないので曖昧性の問題は生じない。

また **vrinstance** 宣言の型付け規則では次のことが保証されなければならない。

- $method_m$ と $Constr_n$ の型を下記のバリアント/レコード宣言のとおりとすると、

  variant $\alpha \in v\ \overline{\beta}$ **where**
  $Constr_n :: \kappa_1 \to \cdots \to \kappa_{k_n} \to \alpha$

  record $\alpha \in r\ \overline{\gamma}$ **where**
  $method_m :: \alpha \to \mu_m$

  上の **vrinstance** 宣言の $method_m$の型は

  $$\overline{\rho} \Rightarrow \upsilon \to \mu_m[\overline{\sigma}/\overline{\gamma}]$$

  でなければならない。ここで $Constr_n$ の型を

  $$\kappa_1[\overline{\tau}/\overline{\beta},\ \upsilon/\alpha] \to \cdots \to \kappa_{k_n}[\overline{\tau}/\overline{\beta},\ \upsilon/\alpha] \to \upsilon$$

  と仮定する。そして、$\overline{\pi} \cup \{\, \upsilon \in v\ \overline{\tau},\ \upsilon \in r\ \overline{\sigma}\,\}$ が $\overline{\rho}$ を導出しなければならない。ここで υ は “self 型” を意味する。

4.3 型推論

基本的には型推論アルゴリズムの基本的な部分はパラメトリック型クラスのものと変わらない。しかし、インスタンス宣言の形式が変更されているので、“文脈整理” の振舞いを変更する必要がある。

直感的には型推論アルゴリズムは曖昧な型を発見し、曖昧な型変数が具体的な “標準インスタンス型” に具体化できるかどうかチェックする。そして可能なときは実際に具体化を行う。

Haskell の文脈整理は Jones[4] の用語で**単純化**の特別な場合とみなすことができる。本稿のシステムでは、対応するプロセスは**単純化**と**改良**が融合したものと考えることができる。ここではこのプロセスを *impr* という関数として形式化する。この関数は型代入と述語集合の対を返す。2 つの補助関数 *check* と *find* を *impr* の定義の中で利用する。

impr(P) =
  let V = all variant class constraints in P
    R = all record class constraints in P
    $VR = \{\, (v\,\overline{\sigma},\ \alpha,\ r\,\overline{\tau}) \mid (\alpha \in v\,\overline{\sigma}) \in V, (\alpha \in r\,\overline{\tau}) \in R \,\}$
  in
  if $\forall\, (v\,\overline{\sigma},\ \alpha,\ r\,\overline{\tau}) \in VR.\ check(v\,\overline{\sigma},\ \alpha,\ r\,\overline{\tau}, P)$
  then ($idSubst, P$)
  else let ($v\,\overline{\sigma},\ \alpha,\ r\,\overline{\tau}$) **be** an arbitrary pair
      s.t.$\neg$*check*($v\,\overline{\sigma},\ \alpha,\ r\,\overline{\tau},\ P$)
      $(Q,\ v\,\overline{\gamma},\ \zeta,\ r\,\overline{\delta})$ = *find*($v,\ r$)
      S = $mgu((\overline{\gamma},\ \zeta,\ \overline{\delta}), (\overline{\sigma},\ \alpha,\ \overline{\tau}))$
      (S', P') = $impr(S\ (P \cup Q))$ **in**
    $(S' \circ S, P')$

check($v\,\overline{\sigma},\ \alpha,\ r\,\overline{\tau},\ P$) =
  let $(Q,\ v\,\overline{\gamma},\ \zeta,\ r\,\overline{\delta})$ = *find*($v,\ r$) **in**
  if there is a substitution S
    s.t. $S\ (\overline{\gamma}, \zeta, \overline{\delta}) = (\overline{\sigma}, \alpha, \overline{\tau})$ **and** $S\ \pi \in P$
  then *True* **else** *False*

find($v,\ r$) =
  if there is an vrinstance declaration:

vrinstance $Q \Rightarrow v\,\overline{\gamma} \ni \zeta \in r\,\overline{\delta}$ **where** ...

then $(Q,\ v\,\overline{\gamma},\ \zeta,\ r\,\overline{\delta})$

else *failure*

パラメトリック型クラスに対する通常の “改良” はこの関数の前に実行されていると仮定する。

impr の戻り値の第 1 要素は型代入で型と型環境に適用される。第 2 要素はもとの述語集合を置き換える。Jones の記法では、このことは次のように書かれる。

$$\frac{Q \mid TA \vdash^W E : v \qquad (T', P) = impr(Q)}{P \mid T'TA \vdash^W E : T'v}$$

α が $P \Rightarrow \tau$ という型の中の曖昧な型変数であるとき、α を P の中の α に関連するバリアントクラスから計算される “標準インスタンス型” で置き換えることができる。すると、P の中の “$\alpha \in \ldots$” という形の述語は安全に取り除くことができる。同時に型推論アルゴリズムがソース中に明示的な型を挿入することができる。

本稿の型推論アルゴリズムのプロトタイプ実装を “Typing Haskell in Haskell” [5] の推論エンジンを拡張して実装した。ほとんどの変更はパラメトリック型クラスを導入するためのもので、本質的な拡張はほとんど上に示した *impr* 関数の部分のみである。

5 まとめ

本稿では、多相バリアントを Haskell の型クラスを使ってエンコードする方法を説明し、さらに多相レコードとバリアントを直接扱える型システムを提案した。

1. 多相バリアントを導入する宣言形式をパラメトリック型クラスの特別な場合として導入した。
2. レコードクラスとバリアントクラスの間の新しいインスタンス宣言形式とそれに対応する新しい “文脈整理” 規則を提案した。

プログラムの意味は型と独立に与えられるため、曖昧な型に関して頭を悩ませる必要はない。本稿のシステムは曖昧な型を避けるのではなく、むしろ曖昧さを積極的に利用しているといえる。

さらに、このシステムは通常の Haskell（正確には Haskell 98 + 関数従属性を持つ型クラス）のコードを型推論の結果として出力するため、プリプロセッサとして動作させることができる。

謝辞

本稿は APLAS’02 (The Third Asian Workshop on Programming Languages and Systems) で発表したシステムをより単純化したものである。このワークショップの聴衆からのコメント、および本稿の旧版に対する査読者のコメントはアイデアを単純化し改良するのに役立った。ここに感謝する。

参考文献

[1] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *ACM Conf. on LISP and Functional Programming*, June 1992.

[2] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, September 1998.

[3] Jacques Garrigue. Code reuse through polymorphic variants. In *FOSE 2000*, November 2000.

[4] Mark P. Jones. Simplifying and improving qualified types. Reserch Report YALEU/DCS/RR-1040, Yale University, June 1994.

[5] Mark P. Jones. Typing haskell in haskell. In *Proceedings of the 1999 Haskell Workshop*, pages 9–22, October 1999.

[6] Mark P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming*, March 2000. LNCS 1782.

[7] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, pages 135–146, June 1995.

[8] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

[9] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Annual ACM Symp. on Principles of Prog. Languages*, pages 77–88, January 1989.