Encoding Types in ML-like Languages [*]

Zhe Yang

Department of Computer Science
New York University
251 Mercer Street, New York, NY 10012, USA
E-mail: `zheyang@cs.nyu.edu`

Abstract

A Hindley-Milner type system such as ML's seems to prohibit type-indexed values, *i.e.*, functions that map a family of types to a family of values. Such functions generally perform case analysis on the input types and return values of possibly different types. The goal of our work is to demonstrate how to program with type-indexed values within a Hindley-Milner type system.

Our first approach is to interpret an input type as its corresponding value, recursively. This solution is type-safe, in the sense that the ML type system statically prevents any mismatch between the input type and function arguments that depend on this type.

Such specific type interpretations, however, prevent us from combining different type-indexed values that share the same type. To meet this objection, we focus on finding a value-independent type encoding that can be shared by different functions. We propose and compare two solutions. One requires first-class and higher-order polymorphism, and, thus, is not implementable in the core language of ML, but it can be programmed using higher-order functors in Standard ML of New Jersey. Its usage, however, is clumsy. The other approach uses embedding/projection functions. It appears to be more practical.

We demonstrate the usefulness of type-indexed values through examples including type-directed partial evaluation, C printf-like formatting, and subtype coercions. Finally, we discuss the tradeoffs between our approach and some other solutions based on more expressive typing disciplines.

1 Introduction

Over the last two decades, the Hindley-Milner type system [14, 20] has been widely used. For example, it underlies several major statically typed functional programming languages, such as ML [21] and Haskell [24]. Among other reasons, this popularity can be attributed to static typing (which serves as a static debugging facility,) and implicit polymorphism allowed by the *principal typing* scheme (which removes the burden of pervasive explicit type annotations). The simplicity of the type system, however, also restricts the class of typeable programs. For example, one cannot examine the type of a value at run-time, as in a dynamically typed language such as Scheme [4].

Functions that take type arguments and accordingly return values of possibly different types are used frequently in abstract formulations of certain algorithms. Such functions form an interesting class of programs that seem to be forbidden by the Hindley-Milner type system. In this article, we formulate such a function as a *type-indexed value*, viewing it as a value indexed by one or more type(s). Figure 1 illustrates a type-indexed value v indexed by one type argument: given a type τ, the corresponding value is v_τ of type T_τ. Usually, the family of types τ is inductively specified using a set of type constructors. Consequently, the type-indexed value v is naturally defined by case analysis on the type constructions. Since all types are implicit in a language with Hindley-Milner type system, one can only hope to use type encodings instead of types as the arguments of an ML function f_v that represents a type-indexed value v. We can reduce case analysis on type constructions to case analysis on value constructions, by encoding type arguments using a datatype. This, however, does not solve the problem, because different branches of the case analysis might have different types, and hence may not be typeable. A common strategy in such cases is to have tagged inputs and outputs of some user-defined datatype. However, this requires users to tag input values themselves, which is not only inconvenient and even unreasonable for cases when verbatim values are required, but also 'type-unsafe' in the sense that a run-time exception might be raised due to unmatched tags.

This problem has exposed the limitations of the Hindley-Milner type system and has motivated a lot of research exploring more expressive type systems. This article investigates what can be done within the framework of the Hindley-Milner type system; in particular, we demonstrate our methods with ML, though the techniques are equally applicable to any other language based on the Hindley-Milner type system. We show how interpreting types τ using corresponding values v_τ gives a type-safe solution to the problem. Based on our approach to type encodings, examples ranging from a printf-like formatting function[1] to type-directed par-

[*]This research was partially supported by National Science Foundation grant CCR-9616993 and the Danish National Research Foundation. Part of this work was carried out during a visit to the BRICS PhD School at the University of Aarhus in the fall of 1997 and during a visit to DIKU, the Department of Computer Science at the University of Copenhagen in January and February 1998.

To appear in the Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming, Baltimore, Maryland, September, 1998.
Copyright ©1998 by ACM, Inc.

[1]Initially devised by Olivier Danvy [6].

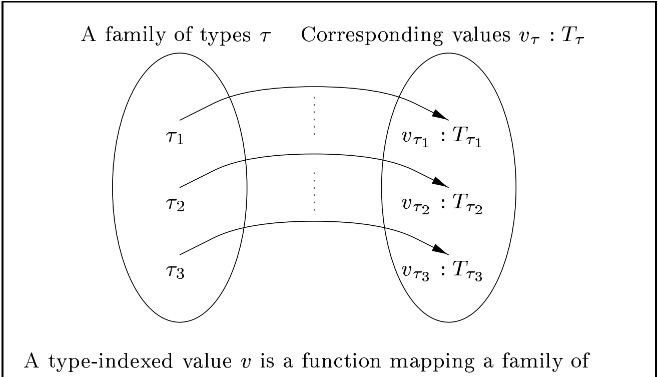
A family of types τ
Corresponding values $v_\tau : T_\tau$
τ_1
τ_2
τ_3
$v_{\tau_1} : T_{\tau_1}$
$v_{\tau_2} : T_{\tau_2}$
$v_{\tau_3} : T_{\tau_3}$
A type-indexed value v is a function mapping a family of types τ to a family of values v_τ of types T_τ.

Figure 1: A type-indexed value

tial evaluation can be programmed in ML successfully. As for their type safety, it is automatically ensured by the ML type system, statically.

The above type encoding is value-dependent. It is not suitable in modular programming practice when different type-indexed values sharing the same family of type indices need to be programmed separately and combined later. It is thus interesting to find a method of type encoding that is independent of any particular type-indexed value. A value-independent encoding of a specific type can be combined with the specification of a type-indexed value (which itself has a fixed type) to deliver the value at this type index. We present two methods of creating such a value-independent type encoding:

1. A type-indexed value is specified as a tuple of value constructions for all possible type constructors, and the encoding of a specific type recursively selects and applies components from the tuple. This gives rise to a Martin-Löf-style encoding of inductive types. The encoding uses first-class polymorphism and higher-order polymorphism, and can be implemented using the higher-order module language of Standard ML of New Jersey [3].

2. A type is encoded as the embedding and projection functions between verbatim values of that type and tagged values of a universal datatype. To encode a specific value v_τ of a type-indexed value v, we can first define its equivalent value, replacing types τ by the corresponding datatypes, and then coerce it to the specific value of the indexed type. We show that this type encoding is universal, *i.e.*, the coercion function can always be constructed from the embedding and projection functions of the indexed types.

In Section 2, we formalize the notion of type-indexed values, give examples, and discuss why it is difficult to program with them. In Section 3, with an understanding of type encodings as type interpretations, we characterize requirements for correct implementations of type-indexed values, and give an *ad hoc* approach to programming type-indexed values in ML. In Section 4, we present two approaches to value-independent type encodings, namely 1 and 2 above. We discuss related work in Section 5 and conclude in Section 6.

2 Type-Indexed Values

Type-indexed values are used in the formulation of algorithms in a type-indexed (or type-directed) fashion. Depending on input type arguments, specific values could have different types. For brevity, we mainly consider programs indexed by only one type argument. Multiple type arguments can be dealt with by bundling all type indices into one type index. This technique, however, could lead to code explosion. We will come back to a practical treatment for dealing with multiple type arguments in section 4.4.

A type-indexed value is defined by

$$v_\tau = e$$

where expression e is a case expression whose value depends on the form of type τ, and is defined using the values indexed at the component types of type τ. The family of types τ is inductively constructed in the following form:

$$\begin{array}{rcl} \tau & = & c_1(\tau_{11}, ..., \tau_{1m_1}) \\ & | & \cdots \\ & | & c_n(\tau_{n1}, ..., \tau_{nm_n}) \end{array} \qquad (1)$$

where c_i's are type constructors, representing a type construction in the underlying language (ML in our case), which builds type τ using *component types* τ_{i1} through τ_{im_i}. Without loss of generality, we assume that the case-analysis in expression e occurs at the outer-most level, which enables us to rewrite the specification of the type-indexed value v in the following pattern-matching form:

$$\begin{array}{rcl} v_{c_1(\tau_{11},\ldots,\tau_{1m_1})} & = & e_1(v_{\tau_{11}}, ..., v_{\tau_{1m_1}}) \\ & \vdots & \\ v_{c_n(\tau_{n1},\ldots,\tau_{nm_n})} & = & e_n(v_{\tau_{n1}}, ..., v_{\tau_{nm_n}}) \end{array} \qquad (2)$$

2.1 Running examples

We use the following two running examples to demonstrate the challenges posed by type-indexed values, and later to illustrate our methods for programming with them.

2.1.1 List flattening

The $\mathsf{flatten}$ program, which flattens arbitrary nested lists with integer elements, is a toy example often used to illustrate the intricacy of typing "typecase" (case study on types) in languages with Hindley-Milner type systems, and to motivate the use of datatypes. It can be written in an untyped language like Scheme (where type testing is allowed) as:

$$\begin{array}{rcll} \mathsf{flatten}\ x & = & [x] & \text{(where } x \text{ is atomic)} \\ \mathsf{flatten}\ [x_1, \ldots, x_n] & = & (\mathsf{flatten}\ x_1)@\cdots@(\mathsf{flatten}\ x_n) & \end{array}$$

where @ is the list concatenation operator. To write this function in ML, a natural solution is to use the ML datatype mechanism to define a "list" datatype, and use pattern matching facilities for case analysis. However, this requires a user to tag all the values, making it somewhat inconvenient to use. Is it possible to use verbatim values directly as the arguments? The term "verbatim values" refers to values whose types are formed using only native ML type constructors, and are hence free of user-defined value constructors.

Due to restrictions of the ML type system, a verbatim value of nested list type must be homogeneous, *i.e.*, all members of the list must have the same type (in the case that members are lists themselves, they must have the same nesting depth). Possible types τ of the argument of function $\mathsf{flatten}$ form the family $F^{int,list}$ of types generated by the following grammar.[2]

$$\tau \quad = \quad \mathsf{int} \mid \tau\ \mathsf{list}$$

The type-indexed function $\mathsf{flatten}$ is specified as:

$$\begin{array}{rcl} \mathsf{flatten} & : & \Lambda\tau \in F^{int,list}.\tau \to \mathsf{int\ list} \\ \mathsf{flatten}_{\mathsf{int}}\ x & = & [x] \\ \mathsf{flatten}_{\alpha\ \mathsf{list}}[x_1, ..., x_n] & = & (\mathsf{flatten}_\alpha\ x_1) + \cdots + (\mathsf{flatten}_\alpha\ x_n) \end{array}$$

Before trying to write the function $\mathsf{flatten}$, let us analyze how it might be used. A first attempt is to make the input value (of some arbitrary homogeneous nested list type) be the only argument. This requires that both expression `flatten 5` and expression `flatten [6]` type-check, so the function argument should be polymorphic and should generalize both type `int` and type `int list`, which must be a type variable α. But ML's parametric polymorphism disallows 'looking into' the type structure of a polymorphic value. Consequently it is impossible to write function $\mathsf{flatten}$ with the value to be flattened as the only argument.

The next attempt is to have an extra argument describing the input type, *i.e.*, a value that encodes the type. We expect to rewrite the aforementioned function invocations as `flatten Int 5` and `flatten (List Int) [6]`, respectively. One might try to encode the type using a datatype as:

```
datatype TypeExp = Int | List of TypeExp
```

The fixed type `TypeExp` of the type encoding, however, also makes the result of applying function $\mathsf{flatten}$ to the type encoding have a fixed ML type. As before, a simple argument shows that it is impossible to give a typeable solution in ML.

2.1.2 Type-directed partial evaluation

Type-directed partial evaluation, a surprisingly concise alternative to the traditional syntax-directed partial evaluation, offers a much more interesting and practical example of type-indexed values. In its simplest form, Danvy's type-directed partial evaluation (TDPE) is formulated in Figure 2. Here, we consider the family $F^{base,func}$ of types τ generated inductively by the following grammar.

$$\tau \quad = \quad \mathsf{base} \mid \tau_1 \to \tau_2$$

The two functions $\downarrow$ (reify) and $\uparrow$ (reflect) are type-indexed, recursively calling each other for the contravariant function argument. At first glance, their definitions do not fit into our canonical form of type-indexed values; however, pairing the two functions at each type index puts the definition into the standard form of a type-indexed value (Figure 3).

In his article [5], Danvy presents the Scheme code for this algorithm, where the type index is encoded as a value, thus reducing type analysis to case analysis. However, a direct transcription of that program into an ML program that requires its input arguments being tagged is not satisfactory for the following reasons:

[2]It is only for brevity that we use int as the base type, instead of a universally quantified type variable.

$$\begin{array}{llcl} \text{(reify)} & \downarrow^{\mathsf{base}} v & = & v \\ & \downarrow^{\tau_1\to\tau_2} v & = & \underline{\lambda} x_1.\downarrow^{\tau_2} (v@(\uparrow_{\tau_1} x_1)) \\ & & & \text{(where } x_1 \text{ is a fresh variable)} \\ \text{(reflect)} & \uparrow_{\mathsf{base}} e & = & e \\ & \uparrow_{\tau_1\to\tau_2} e & = & \lambda v_1.\uparrow_{\tau_2} (e\underline{@}(\downarrow^{\tau_1} v_1)) \end{array}$$

Figure 2: Type-directed partial evaluation

$$\begin{array}{rcl} (\downarrow,\uparrow) & : & \Lambda\tau \in F^{base,func}.(\tau \to \mathsf{Exp}) \times (\mathsf{Exp} \to \tau) \\ (\downarrow,\uparrow)_{\mathsf{base}} & = & (\lambda v.v, \lambda e.e) \\ (\downarrow,\uparrow)_{\tau_1\to\tau_2} & = & \mathbf{let}\ \ (\downarrow^{\tau_1},\uparrow_{\tau_1}) = (\downarrow,\uparrow)_{\tau_1} \\ & & \phantom{\mathbf{let}}\ \ (\downarrow^{\tau_2},\uparrow_{\tau_2}) = (\downarrow,\uparrow)_{\tau_2} \\ & & \mathbf{in}\ \ \ (\lambda v.\underline{\lambda} x_1.\downarrow^{\tau_2} (v@(\uparrow_{\tau_1} x_1)), \\ & & \phantom{\mathbf{in}}\ \ \ \ \lambda e.\lambda v_1.\uparrow_{\tau_2} (e\underline{@}(\downarrow^{\tau_1} v_1))) \\ & & \text{(where } x_1 \text{ is a fresh variable)} \end{array}$$

Figure 3: TDPE in the general form of type-indexed values

- Using type-directed partial evaluation, we expect to normalize a program in the source language and get the corresponding text. It is cumbersome for the user to tag/untag all the program constructs, so a verbatim program is much preferable in this case.
- Unlike the case of function $\mathsf{flatten}$, here the type argument must be explicit. The type index τ only appears as the codomain of the function $\uparrow$ (reflect), whereas its domain is always of type Exp. For the same input expression, varying the type argument results in different return values.

  Since explicit type arguments must be present, the consistency of the type argument and the real tags of the input values cannot be guaranteed by static type checking of ML, and run-time 'type error' can arise in the form of pattern-mismatching exception. This problem is also present in the Scheme program.

3 Type-Indexed Values as Type Interpretations

Our first approach to programming type-indexed values v is based on interpreting specific types τ in the program as the values v_τ indexed by these types.

As we argued in the list flattening example (section 2.1.1), if verbatim arguments are required for an ML function representing a type-indexed value, a type encoding must be explicitly provided as an argument to the function, but this type encoding cannot have a fixed type. Now that the type encoding E_τ itself must have different types, a reasonable choice of these types should make them reflect the types τ being encoded. For each type construction c that constructs a type τ from types $\tau_1, \ldots, \tau_m$, its program encoding E_c is a function that transforms the type encodings $E_{\tau_1}, \ldots, E_{\tau_m}$ to the type encoding E_τ. In other words, the encodings of inductively constructed types form a particular interpretation of the types in value domains; if we use $[\![u]\!]$ instead of E_u to denote the interpretation, we can write down the requirements for the encodings:

$$\begin{array}{lrcl} \text{If} & \tau & = & c(\tau_1, \ldots, \tau_m) \\ \text{then} & [\![\tau]\!] & = & [\![c]\!]([\![\tau_1]\!], \ldots, [\![\tau_m]\!]) \end{array}$$

This can be understood as requiring the interpretations of type and type constructors to form a homomorphism, *i.e.*,

$$[\![c(\tau_1, \ldots, \tau_m)]\!] = [\![c]\!]([\![\tau_1]\!], \ldots, [\![\tau_m]\!]) \qquad (3)$$

A function f_v that represents a type-indexed value v using the above encoding should satisfy

$$v_\tau = f_v[\![\tau]\!] \qquad (4)$$

for all types τ in family F. Equations (3) and (4) precisely characterize program encodings of type-indexed values.

Definition 1 *The encodings $[\![c_i]\!]$ of type constructors c_i, along with function f_v, are said to* implement *type-indexed value v, if and only if they satisfy equations (3) and (4).*

The task of finding the type encodings now boils down to finding interpretations for the type constructors c_i. Observing the similarities of the general form of type-indexed values in the set of equations given by (2) and the interpretation of type constructors in Equation (3), it is not difficult to imagine the following approach to programming a type-indexed value: we interpret a type τ as the corresponding value v_τ, and interpret the type construction c_i using the value construction e_i in the set of equations given by (2), *i.e.*:

$$\begin{array}{rl} [\![\tau]\!] & = v_\tau \\ [\![c_i]\!] & = e_i \end{array}$$

Using the set of equations given by (2), it follows immediately that this interpretation satisfies equation (3). With this type encoding, the function that maps type encodings to the values is simply the identity function:

$$f_v[\![\tau]\!] = [\![\tau]\!]$$

Theorem 1 *A given type-indexed value v is implemented by interpretations $[\![c_i]\!] = e_i$ of type constructors and function $f_v = \lambda x.x$.*

3.1 Examples

The definition of function flatten gives rise to the following interpretations of type constructions:

$$\begin{array}{rcl} [\![\cdot]\!] & : & \Lambda\tau \in F^{int,list}.\tau \to \mathsf{int\ list} \\ [\![\mathsf{int}]\!] & = & \lambda x.[x] \\ [\![\alpha\ \mathsf{list}]\!] & = & \lambda[x_1, \ldots, x_n].[\![\alpha]\!]x_1 @ \cdots @ [\![\alpha]\!]x_n \end{array}$$

A direct coding of these interpretations of type construction into ML functions gives the following program:

```
val Int = fn x => [x]
fun List T = fn l => foldr (op @) [] (map T l)
fun flatten T l = T l
```

Since we choose the ML function names to be the type constructions they interpret, a type argument `List (List Int)` already has the value of

$$[\![(\mathsf{int\ list})\ \mathsf{list}]\!] = \mathsf{flatten}_{(\mathsf{int\ list})\ \mathsf{list}},$$

and function `flatten` is defined as the identity function. The function deals with verbatim values, *e.g.*, expression

```
datatype Exp = VAR of string
             | LAM of string * Exp
             | APP of Exp * Exp

infixr 5 -->
val Base = (fn v => v,
            fn e => e)
fun (T1 as (reify_1, reflect_1)) -->
    (T2 as (reify_2, reflect_2)) =
    let fun reify v =
        let val x1 = Gensym.fresh "x" in
            LAM(x1, reify_2 (v (reflect_1 (VAR x1))))
        end
        fun reflect e =
            fn v1 => reflect_2 (APP(e, reify_1(v1)))
    in
        (reify, reflect)
    end
fun reify (T as (reify_T, reflect_T)) v = reify_T v
```

Figure 4: Type-directed partial evaluation in ML

```
flatten (List (List Int)) [[1, 2], [], [3], [4, 5]]
```

evaluates to `[1,2,3,4,5]`.

We apply the same method to program type-directed partial evaluation (Figure 4) using the type interpretation $[\![\tau]\!] = (\downarrow, \uparrow)_\tau$ defined in Figure 3.

As an example, the expression

```
reify (Base --> Base)
      ((fn x => fn y => x y) (fn x => x) (fn x => x))
```

evaluates to a first-order representation of $\lambda x.x$ such as `LAM ("x7",VAR "x7")`.

3.2 Assessment of the approach

A type encoding in the above approach is essentially the type-indexed value specialized to the particular type index. There are several advantages to this approach:

- Type safety is automatically ensured by the ML type system: case-analysis on types, though it appears in the formulation, does not really occur; the encoding and also the value $[\![\tau]\!] = v_\tau$ of a particular type index τ already has the required type T_τ. If the value $[\![\tau]\!]$ is a function, taking some argument whose type depends on type τ, then the specific type of this argument will be manifested in the type T_τ. Hence, input arguments of illegal types would be rejected.

  For example, the expression

  ```
  reify (Base --> Base) (fn x => fn y => x)
  ```

  will cause a type error in ML, because expression

  ```
  reify (Base --> Base)
  ```

  has the domain type `(Exp -> Exp)`, which does not match type scheme $\Lambda\alpha.\Lambda\beta.(\alpha \to (\beta \to \alpha))$. If we use the expression

  ```
  reify (Base --> Base --> Base)
  ```

  instead, whose domain is of the type `(Exp -> Exp -> Exp)`, then the whole expression type-checks and it evaluates to a textual representation of $\lambda x.\lambda y.x$ like `LAM ("x7",LAM ("x8",VAR "x7"))`.

$$\begin{tikzcd}
\bullet \arrow[r, "\mathsf{super\_reverse}"] \arrow[d, "\mathsf{flatten}"'] & \bullet \arrow[d, "\mathsf{flatten}"] \\
\bullet \arrow[r, "\mathsf{reverse}"'] & \bullet
\end{tikzcd}$$

Figure 5: Composing functions super_reverse and flatten

- In some other approaches that do not make the type argument explicit (*e.g.*, using classes of an object-oriented language), one would need to perform case-analysis on tagged values (including dynamic dispatching), which would require the type index to appear at the input position. In our approach, however, the type index τ could appear at any arbitrary position in type T_τ.

But this simple solution has a major drawback: the loss of composability. One should be able to decompose the task of writing a large type-indexed function into writing several smaller type-indexed functions and then combining them. This would require that the encoding of a type be sharable by these different functions, each of which uses the encoding to get a specific value. However, the above simple solution of interpreting every type directly as the specific value would result in each type-indexed function having a different set of interpretations of type constructors, thereby disallowing sharing of the type encodings.

Consider the following toy example: on the family $F^{int,list}$ of types, we define yet another type-indexed function super_reverse, which recursively reverses a list at each level. The function is defined through the following type interpretation:

$$\begin{array}{rcl}
[\![.]\!] & : & \Lambda\tau \in F^{int,list}.\tau \to \tau \\
[\![\mathsf{int}]\!] & = & \lambda x.x \\
[\![\alpha\ \mathsf{list}]\!] & = & \lambda[x_1, \ldots, x_n].[[\![\alpha]\!]x_n, \ldots, [\![\alpha]\!]x_1]
\end{array}$$

which is implemented in ML as,

```
fun Int x = x
fun List T = rev o (map T)
fun super_reverse T l = T l
```

Each of function flatten and function super_reverse can be used separately, but we cannot use an expression such as

```
fn T => (flatten T) o (super_reverse T)
```

to combine them. We cannot reverse a list recursively and then flatten the result, because the functions `Int` and `List` are defined differently in the two programs. (Notice that the effect of composing function super_reverse and function flatten amounts to reversing the flattened form of the original list (Figure 5).)

This problem can be evaded in a non-modular fashion, if we know in advance *all* possible type-indexed values $v, v' \ldots$ indexed by the same family of types, by tupling all the values together as the type interpretation. Every function f_{v_i} simply projects the appropriate component from the type interpretation. Our previous program of type-directed partial evaluation (Figure 4) illustrates such a tupling.

3.3 Other applications of the approach

Sometimes, the types of certain function arguments are determined by other arguments which embody related type information. In these cases, extra type arguments are redundant, and it is sufficient to interpret the arguments determining types.

As an example, a C printf-style formatting function specifies the type of its arguments through its formatting specification, which is a sequence of field specifiers, represented here as a list. The (simplified) grammar of a formatting specification is given below:

$$\begin{array}{rcl}
\mathit{Spec} & ::= & \mathtt{NIL} \mid \mathit{Field} :: \mathit{Spec} \\
\mathit{Field} & ::= & \mathtt{LIT}\ s \mid \mathtt{\%}\ \tau
\end{array}$$

where s is a string literal and `%` τ specifies an input field argument of type τ. We want to write a function format such that, for instance, the expression

```
format (% Str ++ LIT " is " ++ % Int ++
        LIT "-year old.")
       "Mickey" 80
```

evaluates to the string `"Mickey is 80-year old."`.

Our function is indexed by a formatting specification fs. A specialized format_{fs} has type $\tau_1 \to \tau_2 \ldots \to \tau_n \to \mathsf{string}$, where τ_i's are from all the field specifiers "`%` τ_i" in the specification fs in the order of their appearance. We make use of an auxiliary function format', which introduces one extra argument b as a string buffer; the function will append its output to the end of this input string buffer to get the output string. The functions format and format' can be formulated as follows.

$$\begin{array}{rcl}
\mathsf{format}'_{fs} & : & \mathsf{string} \to T(fs) \\
\textbf{where} & & \\
T(\mathtt{NIL}) & = & \mathsf{string} \\
T(\mathtt{LIT}\ s :: fs) & = & T(fs) \\
T(\mathtt{\%}\ \tau :: fs) & = & \tau \to T(fs) \\
 & & \\
\mathsf{format}'_{\mathtt{NIL}} b & = & b \\
\mathsf{format}'_{\mathtt{LIT}\ s::fs} b & = & \mathsf{format}'_{fs}(b\,\hat{}\,s) \\
\mathsf{format}'_{\mathtt{\%}\ \tau::fs} b & = & \lambda(x:\tau).\mathsf{format}'_{fs}(b\,\hat{}\,\mathsf{toStr}_\tau x) \\
 & & \\
\mathsf{format}_{fs} & : & T(fs) \\
\mathsf{format}_{fs} & = & \mathsf{format}'_{fs}(\text{`` ''})
\end{array}$$

In these declarations, each function $\mathsf{toStr}_\tau : \tau \to \mathsf{string}$ converts a value of type τ to its string representation. Since format' is inductively defined over the formatting specification, we can make it the interpretation of the formatting specification. Each individual field specification f can be viewed as a constructor for formatting specifications, similar to the type constructors in the previous section. Therefore $[\![f]\!]$ should be a transformer from $[\![fs]\!] = \mathsf{format}'_{fs}$ to $[\![f :: fs]\!] = \mathsf{format}'_{f::fs}$, *i.e.*,

$$\mathsf{format}'_{f::fs} = [\![f]\!]\ \mathsf{format}'_{fs}$$

It is now easy to give the interpretation of different individual field specifiers:

$$\begin{array}{rl}
[\![\mathsf{LIT}\ s]\!] & = \lambda\mathsf{format}'_{fs}.\lambda b.\mathsf{format}'_{fs}(b\,\hat{}\,s) \\
[\![\%\ \tau]\!] & = [\![\%]\!]\ \mathsf{toStr}_\tau \\
 & = \lambda\mathsf{format}'_{fs}.\lambda b.\lambda(x:\tau).\mathsf{format}'_{fs}(b\,\hat{}\,\mathsf{toStr}_\tau x)
\end{array}$$

To complete the construction, we define a function `++` to compose such transformers (similar to the function append for lists), and we can define a function format, which supplies the interpretation of the empty field specification $[\![\texttt{NIL}]\!] = \mathsf{format}'_{\texttt{NIL}}$ to a transformer, along with an empty string as the initial buffer. Let us move directly to the ML code:

```
infix 5 ++

fun LIT s p = fn b => p (b ^ s)
fun % toStr_t p = fn b => fn x => p (b ^ toStr_t x)
fun f1 ++ f2 = f1 o f2
fun format fs = fs (fn b => b) ""

fun Int n = Int.toString n
fun Str s = s
```

Unlike the C `printf` function, the above ML implementation is type-safe; for example, the type of the expression

```
format (% Int ++ LIT ":  " ++ % Str)
```

is int $\rightarrow$ string $\rightarrow$ string, thus ensuring that exactly two arguments, one of type int, the other of type string, can be supplied.

The power of a higher-order functional language with static typing like ML also enables the construction of field specifiers for different types: for the type-indexed function toStr, we can use the standard type interpretation method to allow type constructions such as product types and list types.

```
fun Pair toStr1 toStr2 =
    fn (x1, x2)
    => "(" ^ (toStr1 x1) ^ ", " ^ (toStr2 x2)  ^ ")"
fun List toStr l =
    let fun mkTail []
          = "]"
        | mkTail [e]
          = (toStr e) ^ "]"
        | mkTail (e :: el)
          = (toStr e) ^ ", " ^ (mkTail el)
    in "[" ^ (mkTail l)
    end
```

This enables us to construct field specifiers for compound types. The following example illustrates its usage:

```
format (%(List (Pair Str (List Str))))
       [("N", ["Prince", "8", "14"]),
        ("P", ["Newport", "Christopher", "9"])]
```

It should be clear that for any given type τ, we can have different functions to translate a value of type τ to its string representation. It is easy to define a more complicated field specifier which determines formatting issues such as choosing various paddings or parameterizing the constructors of compound types over delimiters—*i.e.*, a pretty-printer.

Danvy observed that such an implementation of `format` out-performs the library version of formatting functions provided with SML/NJ and Objective Caml, without even applying partial evaluation to remove interpretive overhead [6]. Intuitively, the efficiency comes from the elimination of case-analysis by using function "dispatching" instead, which is similar to the practice of eliminating conditionals by hard-wiring data into code, or using jump-tables in machine language.

Danvy also makes an interesting comparison of the type-indexed formatting function and the two formatting library functions of SML/NJ and of OCaml. In SML/NJ, the user is required to embed all arguments into a universal datatype and to collect the result in a list. Any mistake in the embedding or in the size of the list results in a run-time error. In OCaml, the formatting function is itself type-unsafe. Applying it to a formatting specification, however, yields a type-safe curried function that can be used on untagged values. Programming a formatting function as a type-indexed value yields the same effect as in OCaml (convenience and verbatim values), but with the added benefit that the formatting function itself can be statically type-checked in ML.

4 Value-Independent Type Encoding

In this section, we further develop two approaches to encode types independent of the type-indexed values defined on them, *i.e.*, we should be able to define the encodings $[\![\tau]\!]$ of a family F of types τ, so that given any value v indexed by this family of types, a function f_v that satisfies equation (4) can be constructed. In contrast to the solution in the previous section, which interprets types τ using values v_τ directly and is value-dependent, a value-independent type encoding enables different type-indexed values $v, v', \ldots$ to share a family of type encodings, resulting in more modular programs using type-indexed values. We present the following two approaches to value-independent type encoding:

- as an abstraction of the formulation of a type-indexed value, and
- as a universal interpretation of types as tuples of embedding and projection functions between verbatim values and tagged values.

4.1 Abstracting type encodings

If the type encoding is value-independent, the function f_v representing type-indexed value v should carry the information of the value constructions e_i in a specification in the form of the set of equations given in (2). This naturally leads to the following approach to type encoding: a type-indexed value v is specified as an n-ary tuple $\vec{e} = (e_1, \ldots, e_n)$ of the value constructions, and the value-independent type interpretation $[\![\tau]\!]$ maps this specification to the specific value v_τ.

$$[\![\tau]\!]\vec{e} = v_\tau \tag{5}$$

With Equation (3), we require the encoding of type constructors c_i to satisfy

$$\begin{array}{cll} & [\![c_i]\!]([\![\tau_1]\!], \ldots, [\![\tau_m]\!])\vec{e} & \\ = & [\![c_i(\tau_1, \ldots, \tau_m)]\!]\vec{e} & \text{by (3)} \\ = & v_{c_i(\tau_1, \ldots, \tau_m)} & \text{by (5)} \\ = & e_i(v_{\tau_1}, \ldots, v_{\tau_m}) & \text{by (2)} \\ = & e_i([\![\tau_1]\!]\vec{e}, \ldots, [\![\tau_m]\!]\vec{e}) & \text{by (5)} \end{array}$$

By this derivation, we have

Theorem 2 *The value-independent encodings of type constructors*

$$[\![c_i]\!] = \lambda(x_1, \ldots, x_m).\lambda\vec{e}.e_i(x_1\vec{e}, \ldots, x_m\vec{e})$$

and the function $f_v(x) = x(e_1, \ldots, e_n)$ implement the corresponding type-indexed value v.

This approach seems to be readily usable as the basis of programming type-indexed values in ML. However, the restriction of ML type system that universal quantifiers on type variables must appear at the top level again makes this approach infeasible. For example, let us try to encode types in the family $F^{base,func}$, and use them to program type-directed partial evaluation in ML (Figure 6).

```
val Base = fn (base_v, func_v) => base_v
fun T1 --> T2 = fn (spec_v as (base_v, func_v))
               => func_v (T1 spec_v) (T2 spec_v)

fun reify T =
    let val (reify_T, _) =
        T ((fn v => v, fn e => e),       (* base_v *)
                                          (* func_v *)
           fn (reify_T1, reflect_T1) =>
           fn (reify_T2, reflect_T2) =>
              ...          (* (reify_T, reflect_T) *)
          )
    in reify_T end
```

Figure 6: An unsuccessful encoding of $F^{base,func}$ and TDPE

The definition of `reify` and `reflect` at higher types is as before and omitted here for brevity. This program will not work, because the λ-bound variable `spec_v` can only be used monomorphically in the function body. This forces all uses of `func_v` to have the same monotype; as an example, the type encoding `Base --> (Base --> Base)` causes a type error, because the two uses of variable `func_v` (one being applied, the other being passed to lower type interpretations) have different monotypes.

Indeed, the type of the argument of `reify`, a type encoding $[\![\tau]\!]$ constructed using `Base` and `-->`, is somewhat involved:

$$
\begin{array}{rll}
[\![\tau]\!] & : & \Lambda \mathsf{obj} : * \to *. \\
 & & \Lambda \mathsf{base\_type} : *. \\
 & & (\mathsf{base\_type}\ \mathsf{obj}\ \times \qquad\qquad\qquad (*\ \mathsf{base\_v}\ *) \\
 & & \ (\Lambda\alpha : *, \beta : *.(\alpha\ \mathsf{obj}) \to (\beta\ \mathsf{obj}) \to \quad (*\ \mathsf{func\_v}\ *) \\
 & & \qquad\qquad\qquad ((\alpha \to \beta)\ \mathsf{obj}))) \to \\
 & & \tau\ \mathsf{obj}
\end{array}
$$

Here, the type constructor obj constructs the type T_τ of the specific value v_τ from a type index τ, and the type base_type gives the base type index. What we need here is *first-class polymorphism*, which allows nested quantified types, as used in the type of argument `func_v`. Substantial work has been done in this direction, such as allowing selective annotations of λ-bound variables with polymorphic types [23] or packaging of these variables using polymorphic datatype components [16]. Moreover, *higher-order polymorphism* [15] is needed to allow parameterizing over a type constructor, *e.g.*, the type constructor obj.

In fact, such type encodings are similar to a Martin-Löf-style encoding of inductive types using the corresponding elimination rules in System F_ω, which does support both first-class polymorphism and higher-order polymorphism in an explicit form [10, 25].

4.2 Explicit first-class and higher-order polymorphism in SML/NJ

The module system of Standard ML provides an explicit form of first-class polymorphism and higher-order polymorphism. Quantifying over a type or a type constructor is done by specifying the type or type constructor in a signature, and parameterizing functors with this signature. To recast the higher-order functions in Figure 6 into functors, we also need to use higher-order functors which allows functors to have functor arguments or results. Such higher-order functors [31] are supported by Standard ML of New Jersey [3]. Below we give a program for type-directed partial evaluation using higher-order functors.

```
signature SpecValue =
    sig
        type 'a obj
        type my_type
        val v: my_type obj
    end

signature IndValue =
    sig
        type 'a obj
        type base_type
        val Base : base_type obj
        val Arrow: 'a obj -> 'b obj ->
                   ('a -> 'b) obj
    end

signature Type =
    sig
        functor F(Obj: IndValue): SpecValue
           where type 'a obj = 'a Obj.obj
    end

structure Base: Type =
    struct
        functor F(Obj: IndValue): SpecValue =
            struct
                type 'a obj = 'a Obj.obj
                type my_type = Obj.base_type
                val v = Obj.Base
            end
    end

functor Arrow(T1: Type) (T2: Type): Type =
    struct
        functor F(Obj: IndValue): SpecValue =
            struct
                type 'a obj = 'a Obj.obj
                structure v_T1 = T1.F(Obj)
                structure v_T2 = T2.F(Obj)
                type my_type = v_T1.my_type ->
                               v_T2.my_type
                val v = Obj.Arrow v_T1.v v_T2.v
            end
    end

structure reify_reflect: IndValue =
    struct
        type 'a obj = ('a -> Exp) * (Exp -> 'a)
        type base_type = Exp
        val Base = (fn v => v, fn v => v)
        fun Arrow (reify_1, reflect_1)
                  (reify_2, reflect_2) =
            ...
    end
```

Here, a `Type` encoding is a functor from a structure with signature `IndValue`, which is a specification of type-indexed values, to a structure with signature `SpecValue`, which denotes a value of the specific type. The type `my_type` gives the particular type index τ, and the type `base_type` and the type constructor `obj` are as described in the last section.

It is however cumbersome and time-consuming to use such functor-based encodings. The following example illustrates how to partially evaluate (residualize) the function $\lambda x.x$ with type $(\mathsf{base} \to \mathsf{base}) \to (\mathsf{base} \to \mathsf{base})$.

```
local structure T   = Arrow(Arrow(Base)(Base))
                           (Arrow(Base)(Base))
      structure v_T = T.F(reify_reflect)
in
      val result = #1(v_T.v) (fn x => x)
end
```

Furthermore, since ML functions cannot take functors as arguments, we must define functors to use such functor-encoded type arguments. Therefore, even though this approach is conceptually simple and gives clean, type-safe and value-independent type encodings, it is not very practical for programming in ML.

4.3 Embedding/projection functions as type interpretation

The alternative approach to value-independent type encodings is (maybe somewhat surprisingly) based on programming with tagged values of user-defined universal datatypes. Before describing this approach, let us look at how tagged values are often used to program functions with type arguments.

First of all, for a type-indexed value v whose type index τ appears at the position of input arguments, the tags attached to the input arguments are enough to guide the computation. For examples, the tagged-value version of functions `flatten` and `super_reverse` is as follows:

```
datatype tagIntList =
    INT of int
  | LST of tagIntList list

fun flattenTg (INT x)
    = [x]
  | flattenTg (LST l)
    = foldr (op @) [] (map (fn x => flattenTg x) l)
fun super_reverseTg (INT v)
    = INT v
  | super_reverseTg (LST l)
    = LST (rev (map super_reverseTg l))
```

In more general cases, if the type index τ can appear at any position of the type T_τ of specific values v_τ, then a description of type τ using a datatype must be provided as a function argument. However, this approach suffers from several drawbacks:

1. Verbatim values cannot be directly used.
2. If an explicit encoding of a type τ is provided, one cannot ensure at compile time its consistency with other input arguments whose types depend on type τ; in other words, run-time 'type-errors' can happen due to unmatched tags.

Can we avoid these problems while still using universal datatypes? To solve the first problem, we want the program to automatically tag a verbatim value according to the type argument. To solve the second problem, if all tagged values are generated from verbatim values under the guidance of type arguments, then they are guaranteed to conform to the type encoding, and run-time 'type-errors' can be avoided.

The automatic tagging process that embeds values of various types into values of a universal datatype is called an *embedding* function. Its inverse process, which removes tags and returns values of various types, is called a *projection* function. Interestingly, these functions are type-indexed themselves, thus they can be programmed using the *ad hoc* method described in Section 3. Using the embedding function and projection function of a type τ as its encoding gives another value-independent type encoding method for type-indexed values.

For each family T of types τ inductively defined in the form of equation (1), we first define a datatype U of tagged values, as well as a datatype *typeExp* (type expression) to represent the type structure. Next, we use the following interpretation as the type encoding:

$$\begin{array}{rcll} [\![\tau]\!] & = & \langle \mathit{emb}_\tau, \mathit{proj}_\tau, \mathit{tE}_\tau \rangle & \\ \mathit{emb}_\tau & : & \tau \to U & \text{(embedding function)} \\ \mathit{proj}_\tau & : & U \to \tau & \text{(projection function)} \\ \mathit{tE}_\tau & : & \mathit{typeExp} & \text{(type expression)} \end{array} \qquad (6)$$

Finally, we use the embedding and projection functions as basic coercions to convert a value based on a universal datatype to type T_τ corresponding to the type index τ.

The important question that remains is how we can define the embedding/projection function pair of a type τ in terms of those of its component types τ_i. In general, for a covariant component type τ_i, emb_τ and proj_τ should be defined in terms of emb_{τ_i} and proj_{τ_i}, respectively; for a contravariant component type τ_i, emb_τ and proj_τ should be defined in terms of proj_{τ_i} and emb_{τ_i}, respectively. More involved cases of embedding and projection functions between special types and universal tagged datatypes are studied in detail in [13].

4.3.1 Examples

Taking the family $F^{int,list}$ of types, we can encode the type constructors as:

```
datatype typeExpL = tInt | tLst of typeExpL
val Int = (fn x => INT x, fn (INT x) => x, tInt)
fun List (T as (emb_T, proj_T, tE_T)) =
    (fn l => LST (map emb_T l),
     fn LST l => map proj_T l,
     tLst tE_T)
```

and then the functions `flatten` and `super_reverse` are defined as

```
fun flatten (T as (emb, _, _)) v = flattenTg (emb v)
fun super_reverse (T as (emb, proj, _)) v =
        proj (super_reverseTg (emb v))
```

Now that the type encoding is neutral to different type-indexed values, they can be combined, sharing the same type argument. For example, the function

```
fn T => (flatten T) o (super_reverse T)
```

defines a type-indexed function that composes `flatten` and `super_reverse`.

The other component of the interpretation, the type expression `tE` is used for those functions where the type indices do not appear at the input argument positions, such as the `reflect` function. In these cases, a tagged-value version of the type-indexed value must perform case analysis on the type expression `tE`. As an example, the code of type-directed partial evaluation using this new type interpretation is presented below.

```
datatype 'base tagBaseFunc =
    BASE of 'base
  | FUNC of ('base tagBaseFunc) -> ('base tagBaseFunc)
datatype typeExpF =
    tBASE
  | tFUNC of typeExpF * typeExpF

val Base = (fn x => (BASE x), fn (BASE x) => x, tBASE)
fun ((T1 as (I_T1, P_T1, tE1)) -->
     (T2 as (I_T2, P_T2, tE2))) =
    (fn f => FUNC (fn tag_x => I_T2 (f (P_T1 tag_x))),
     fn FUNC f => (fn x => P_T2 (f (I_T1 x))),
     tFUNC(tE1,tE2))

val rec reifyTg =
  fn (tBASE, BASE v) => v
   | (tFUNC(tE1,tE2), FUNC v) =>
       let val x1 = Gensym.fresh "x" in
         LAM(x1, reifyTg
                 (tE2, v (reflectTg (tE1, (VAR x1)))))
       end
and reflectTg =
  fn (tBASE, e) => BASE(e)
   | (tFUNC(tE1,tE2), e) =>
       FUNC(fn v1 => reflectTg
                     (tE2, APP (e, reifyTg (tE1, v1))))

fun reify (T as (emb, _, tE)) v = reifyTg(tE, emb v)
```

Recall that the definition of functions `reifyTg` and `reflectTg` will cause matching-inexhaustive compilation warnings, and invoking them might cause run-time exceptions. Function `reify` is safe, however, in the sense that if the argument `v` type-checks with the domain type of the embedding function `emb`, then, the resulting tagged expression must comply with the type expression `tE`. This value-independent type encoding can be used for the 'type specialization' described in [7], where the partial evaluator and the projection function are type-indexed by the same family of types.

4.3.2 Comments

Finally, we briefly argue that the above approach based on embedding and projection functions is universal, in the sense that the type index τ can appear at any position of the type T_τ of the value v_τ.

We assume the following conditions about the types:

1. All the type constructions c_i build a type only from component types covariantly and/or contravariantly. As shown in the TDPE example, the same component type can be used both covariantly and contravariantly.

2. The type T_τ is constructed by covariant and/or contravariant type constructions from type variable τ exclusively.

We use the following systematic method of implementing type-indexed values v. First, define an ML datatype U following the recursive construction of the type domain. Then, program the type interpretation in the form of equation (6). This can be achieved because by Condition 1, all the type constructions are covariant/contravariant in all their arguments. The embedding and projection functions serve as two basic coercions between type τ and the type U:

$$\begin{cases} emb_\tau & : \quad \tau \rightsquigarrow U \\ proj_\tau & : \quad U \rightsquigarrow \tau \end{cases}$$

Given this type encoding (independent of any particular type-indexed value), we can write a universal datatype version of the type-indexed v, and use the above pair of coercion functions to construct a coercion from the universal-typed value to the value of particular type. The existence of such a coercion is ensured by Condition 2 by a straightforward structural induction on the type T_τ.

By the construction, we have

Theorem 3 *The approach described above, based on interpreting types as embedding/projection functions, gives a type-safe and value-independent solution to type encodings and implementing type-indexed values.*

This new approach to value-independent type encodings is general and practical. Though this approach is based on universal datatype solutions using tagged values, it overcomes the two original problems of directly using universal datatypes:

- Though the universal datatype version of the indexed value is not type-safe, the coerced value is type-safe in general. This is because verbatim input arguments of various types are mapped into the universal datatype by the embedding function, whose type acts as a filter of input types. Unmatched tags are prevented this way.

- Users do not need to tag the input and/or untag the output; this is done automatically by the program f_v using the embedding and projection functions. From another perspective, this provides a method of *external tagging* using the type structure. Such external tags are much smaller than the internal tags and are much easier to acquire (in our case, one can simply use the result of type inference from the compiler).

This approach is not as efficient as the *ad hoc*, value-dependent approach, due to the lengthy tagging and untagging operations and the introduction of extra intermediate data structures. This problem can be overcome using program transformation techniques such as partial evaluation [18], by specializing the general functions with respect to certain type encodings at compile time, and removing all the tagging/untagging operations. In particular, Danvy showed how it can be naturally combined with type-directed partial evaluation to get a 2-level embedding/projection function [7].

4.4 Multiple type indices

Though our previous examples only demonstrate type-indexed values which have only one type index, the embedding/projection-based approach can be readily applied to implementing values indexed by more than one type indices. Here let us take the example of writing an ML function that performs subtype coercion [22]. Given a from-type, a to-type, a list of subtype coercions at base types, and a value of the from-type, this function coerces the value to the to-type and return it.

Following the general pattern, we first write a function `univ_coerce`, which performs the coercions on tagged values. The function `coerce` then wraps up function `univ_coerce`, by embedding the input argument and projecting the output. For brevity, we have omitted the obvious definition of the related datatypes, and the type interpretations as embedding/projection functions and type expressions of `Int`, `Str`, `List`, `-->`, `**`, some of which have already appeared in previous examples.

```
exception nonSubtype of typeExp * typeExp

fun lookup_coerce [] tE1 tE2
    = raise nonSubtype(tE1, tE2)
  | lookup_coerce ((t, t', t2t')::Others) tE1 tE2
    = if t = tE1 andalso t' = tE2 then
        t2t'
      else
        lookup_coerce Others tE1 tE2

fun univ_coerce cl (tFUN(tE1_T1, tE2_T1))
                   (tFUN(tE1_T2, tE2_T2)) (FUN v) =
    FUN (fn x => univ_coerce cl tE2_T1 tE2_T2
        (v (univ_coerce cl tE1_T2 tE1_T1 x)))
  | univ_coerce cl (tLST tE_T1) (tLST tE_T2) (LST v) =
    LST (map (univ_coerce cl tE_T1 tE_T2) v)
  | univ_coerce cl (tPR(tE1_T1, tE2_T1))
                   (tPR(tE1_T2, tE2_T2)) (PR (x, y)) =
    PR (univ_coerce cl tE1_T1 tE1_T2 x,
        univ_coerce cl tE2_T1 tE2_T2 y)
  | univ_coerce cl x y v =
    if x = y then
        v
    else
        (lookup_coerce cl x y) v

fun coerce cl (T1 as (emb_T1, proj_T1, tE_T1))
              (T2 as (emb_T2, proj_T2, tE_T2)) v =
    proj_T2 (univ_coerce cl tE_T1 tE_T2 (emb_T1 v))
```

The example below builds a subtype coercion C : string $\to$ string $\leadsto$ int $\to$ string, given a base coercion int $\leadsto$ string, so that, for example, the expression `C (fn x => x ^ x) 123` evaluates to `"123123"`.

```
val C = coerce [(tINT, tSTR,
                 fn (INT x) => STR (Int.toString x))]
               (Str --> Str) (Int --> Str)
```

Again, this approach can be combined with type-directed partial evaluation to obtain 2-level functions, as done by Danvy for coercion functions and by Vestergaard for "à la Kennedy" conversion functions [19, 32].

5 Related work

5.1 Using more expressive type systems

The problem of programming type-indexed values in a statically typed language like ML motivated several earlier works that introduce new features to the type systems. In the following sections, we briefly go through some of these frameworks that provide solutions to type-indexed values.

5.1.1 Dynamic typing

Realizing that static typing is too restrictive in some cases, there is a line of work on adding dynamic typing [1, 2] to languages with static type systems. Such an approach introduces a universal type `Dynamic` along with two operations for constructing values of type `Dynamic` and inspecting the type tag attached to these values. A dynamic typing approach extends user-defined datatypes in several ways: the set of type constructions does not need to be known in advance—the type `Dynamic` is extensible; it also allows polymorphism in the represented data. Processing dynamic values is however similar to processing tagged values of user-defined type—both require operations that wrap values and case analysis that removes the wrapping.

A recent approach along the line of dynamic typing, *staged type inference* [28] proposes to defer the type inference of some expressions until run-time when all related information is available. In particular, this approach is naturally combined with the framework of staged computation [9, 30] to support type-safe code generation at run-time. Staged programming helped to solve some of the original problems of dynamic typing, especially those concerning usages.

However, the way type errors are prevented at run-time is to require users to provide 'default values' that have expected types of expressions whose actual types are inferred at run-time; when type-inference fails, or the inferred type does not match the context, the default values are used. This is effectively equivalent to providing default exception handlers for run-time exceptions resulting from type inference. The approach is still a dynamic-typing approach, so that the benefit of static debugging offered by a static typing system is lost. For example, the formatting function in [28] will simply return an error when field specifiers do not match the function arguments. On the other hand, it is also because of this possibility of run-time 'type error' that dynamic typing disciplines give extra power, as shown in applications such as meta-programming and higher-level data/code transferring in distributed programming.

5.1.2 Intensional type analysis

Intensional type analysis [12] directly supports type-indexed values in the language λ_i^{ML} in order to compile polymorphism into efficient unboxed representations. The language λ_i^{ML} extends a predicative variant of Girard's System F_ω with primitives for intensional type analysis, by providing facilities to define constructors and terms by structural induction on monotypes. However, the language λ_i^{ML} is explicitly polymorphic, requiring pervasive type annotations throughout the program and thus making it inconvenient to directly program in this language. Not surprisingly, the language λ_i^{ML} is mainly used as a typed-intermediate language.

5.1.3 Haskell type classes

The *type-class* mechanism in Haskell [11] also makes it easy to program type-indexed values: the declaration of a type class should include all the type-indexed value needed, and every value construction e_i should be implemented as an instance declaration for the constructed type, assuming the component types are already instances of the type class. One way of implementing type classes is to translate the use of type classes to arguments of polymorphic functions (or in logic terms, to translate existential quantifiers to universal quantifiers at dual position), leading to programs in the same style as handwritten ones following the *ad hoc* approach of Section 3. The type-class-based solution, like the *ad hoc* approach, is not value-independent, because all indexed values need to be declared together in the type class. Also, because each type can only have one instance of a particular type class, it does not seem likely to support, *e.g.*, defining various formatting functions for the same types of arguments.

It is interesting to note that type classes and value-independent types (or type encodings) form two dimensions of extensibility.

- A type class fixes the set of indexed values, but the types in the type classes can be easily extended by introducing new instances.

- A value-independent type fixes the family of types, but new values indexed by the family can be defined without changing the type declarations.

It would be nice to allow both kinds of extensibility at the same time. But this seems to be impossible—consider the problem of defining a function when possible new types of arguments the function need to handle are not known yet. A linear number of function and type definitions cannot result in a quadratic number of independent variations.

5.1.4 Conclusion

The approaches above (described in section 5.1.1 through section 5.1.3) give satisfactory solutions to the problem of type-indexed values. However, since ML-like languages dominate large-scale program development in the functional programming community, our approach is immediately usable and pragmatic in common programming practice.

5.2 Type-directed partial evaluation

Partial evaluation is an automatic program transformation technique that removes the run-time interpretive overhead of a general-purpose program and generates an efficient special-purpose program. A traditional partial evaluator is syntax-directed, intensionally working on the program text by propagating constant values through the program text and carrying out static computations to yield a simplified program. On the contrary, type-directed partial evaluation is an extensional approach which amounts to normalizing the expression through evaluating the given expression in a suitable context, given the type of residual program. Guided by the type information, the functions defined in Figure 2 eta-expand a value into a two-level lambda expression. The underlined constructs are dynamic constructs, which represent code-generating computations, while other constructs are static constructs, which represent computations during partial evaluation (hence the alternative name *normalization by evaluation* [8]).

Andrzej Filinski first implemented type-directed partial evaluation in ML in 1995. In his presentations of type-directed partial evaluation, Danvy always challenged the attendees to program it in a typed language such as ML or Haskell. The author answered the challenge in 1996, which, according to Danvy, is the first solution after Filinski's. The third person to have solved it is Morten Rhiger [26]. Since then, Kristoffer Rose has programmed it in Haskell, using type classes [27].

An interesting common pattern shared by type-directed partial evaluation and the embedding/projection-based approach is the use of types as external tags (see section 4.3.2): loosely speaking, one external type tag in type-directed partial evaluation replaces pervasive binding-time annotations in the preprocessed program texts. The two-level eta-expansion process then follows the external type tag to place appropriate binding-time annotations to the program.

6 Conclusions

We have presented a notion of type-indexed values, which formalize functions having type arguments. We have formulated type-encoding-based implementations of type-indexed values in terms of type interpretations. According to this formulation, we presented three approaches that enable type-safe programming of type-indexed values in ML or similar languages.

- The first approach directly uses the specific values of a given type-indexed value as the type interpretation. It gives value-dependent type encodings, not sharable by different values indexed by the same family of types. However, its efficiency makes it a suitable choice both for applications where all type-indexed values using the same family of types are known in advance, and for the target form of a translation from a source language with explicit support for type-indexed values.
- The second approach is value-independent, abstracting the specification of a type-indexed value from the first approach. Apart from its elegant form, it is not very practical because it requires first-class and higher-order polymorphism.
- The third approach applies the first approach to tune a usual tagged-value-based, type-unsafe approach to give a type-safe and syntactically convenient approach, by interpreting types as the embedding/projection functions. Though it is less efficient than the first approach due to all the tagging/untagging operations, it allows different type-indexed values to be combined. Therefore, we prefer this approach to the other approaches for practical programming in a modular fashion.

On one hand, we showed in this article that with appropriate type encoding, type-indexed values can be programmed in ML-like languages; on the other hand, our investigation also feedbacks to the design of new features of type systems. For example, implicit first-class and higher-order polymorphism seem to be useful in applications such as type encodings. The question of what is an expressive enough and yet convenient type system will only be answered by various practical applications.

Concerning programming methodologies, we note the similarity between type-directed partial evaluation and our third approach in externalizing internal tags. Requiring only a single external tag not only alleviates the burden of manually annotating the program or data with internal tags, but also increases the consistency of these tags. We would like to generalize this idea to other applications.

Acknowledgments

I especially thank Olivier Danvy for his challenge, and for his encouragement that led to this article, and for productive discussions. I am also grateful to Hseu-Ming Chen, Deepak Goyal, Fritz Henglein, Bob Paige, and the anonymous referees for their helpful comments. Thanks also go to other researchers from BRICS, from the DIKU TOPPS group, and from New York University for fruitful discussions.

Figure 5 was drawn with Kristoffer Rose's XY-pic package.

References

[1] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268., April 1991.

[2] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[3] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Maluszyński and Martin

Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991. Springer-Verlag.

[4] William Clinger and Jonathan Rees, editors. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[5] Olivier Danvy. Type-directed partial evaluation. In Steele [29], pages 242–257.

[6] Olivier Danvy. Formatting strings in ML. Research Series RS-98-5, BRICS, Department of Computer Science, University of Aarhus, March 1998. To appear in the Journal of Functional Programming.

[7] Olivier Danvy. A simple solution to type specialization. Research Series RS-98-1, BRICS, Department of Computer Science, University of Aarhus, January 1998. To appear in the *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming.*

[8] Olivier Danvy and Peter Dybjer, editors. *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98,* (Göteborg, Sweden, May 8–9, 1998), number NS-98-1 in BRICS Notes Series, BRICS, Department of Computer Science, University of Aarhus, May 1998.

[9] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Steele [29], pages 258–283.

[10] Jean-Yves Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192, 1986.

[11] Cordelia Hall, Kevin Hammond, Simon Peyton-Jones, and Philip Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[12] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In Peter Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, January 1995. ACM Press.

[13] Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

[14] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[15] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995. An earlier version appeared in FPCA '93.

[16] Mark P. Jones. First-class polymorphism with type inference. In Jones [17], pages 483–496.

[17] Neil D. Jones, editor. *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997. ACM Press.

[18] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, International Series in Computer Science, June 1993.

[19] Andrew Kennedy. Relational parametricity and units of measure. In Jones [17], pages 442–455.

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

[21] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[22] John C. Mitchell. Coercion and type inference. In Ken Kennedy, editor, *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, Salt Lake City, Utah, January 1984.

[23] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In Steele [29], pages 54–67.

[24] John Peterson, Kevin Hammond, et al. Report on the programming language Haskell, a non-strict purely-functional programming language, version 1.4. Available at the Haskell homepage: http://www.haskell.org, April 1997.

[25] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, number 19 in Lecture Notes in Computer Science, pages 408–425, Paris, France, April 1974. Springer-Verlag.

[26] Morten Rhiger. A study in higher-order programming languages. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1997.

[27] Kristoffer Rose. Type-directed partial evaluation in a pure functional language. In Danvy and Dybjer [8].

[28] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing as staged type inference. In Luca Cardelli, editor, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 289–302, San Diego, California, January 1998. ACM Press.

[29] Guy L. Steele, editor. *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996. ACM Press.

[30] Walid Taha and Tim Sheard. Multi-stage programming. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 321–321, Amsterdam, The Netherlands, June 1997. ACM Press.

[31] Mads Tofte. Principal signatures for higher-order program modules. *Journal of Functional Programming*, 4(3):285–335, July 1994.

[32] René Vestergaard. From proof normalization to compiler generation and type-directed change-of-representation. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 1997.