Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 315 (2004) 151 – 190

Theoretical Computer Science

www.elsevier.com/locate/tcs

Encoding types in ML-like languages[☆]

Zhe Yang

Department of Computer Science, New York University, USA

Abstract

This article presents several general approaches to programming with type-indexed families of values within a Hindley–Milner type system. A type-indexed family of values is a function that maps a family of types to a family of values. The function performs a case analysis on the input types and returns values of possibly different types. Such a case analysis on types seems to be prohibited by the Hindley–Milner type system. Our approaches solve the problem by using *type encodings*. The compile-time types of the type encodings reflect the types themselves, thereby making the approaches type-safe, in the sense that the underlying type system statically prevents any mismatch between the input type and the function arguments that depend on this type.

A type encoding could be either value-dependent, meaning that the type encoding is tied to a specific type-indexed family, or value-independent, meaning that the type encoding can be shared by various type-indexed families. Our first approach is value-dependent: we simply interpret a type as its corresponding value. Our second approach provides value-independent type encodings through embedding and projection functions; they are universal type interpretations, in that they can be used to compute other type interpretations. We also present an alternative approach to value-independent type encodings, using higher-order functors.

We demonstrate our techniques through applications such as C printf-like formatting, type-directed partial evaluation, and subtype coercions.

© 2003 Elsevier B.V. All rights reserved.

1. Introduction

Over the last two decades, the Hindley–Milner type system [17,26] evolved into the most popular type basis of functional languages. It underlies several major higher-order, statically typed functional programming languages, such as ML [27] and Haskell [31].

[☆] A preliminary version of this paper appeared in *Proceedings* of the 1998 International Conference on Functional Programming, ACM Press, 1998.

E-mail address: zhey@microsoft.com (Z. Yang).

0304-3975/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2003.11.017

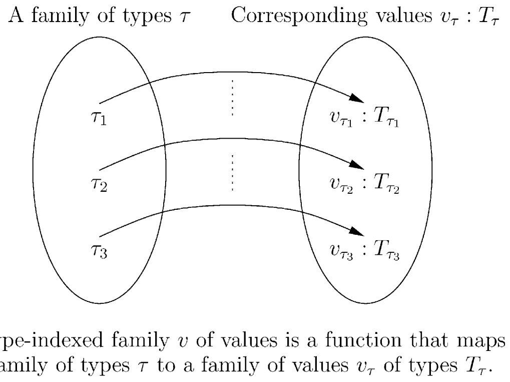
A family of types τ
Corresponding values $v_\tau : T_\tau$
τ_1
τ_2
τ_3
$v_{\tau_1} : T_{\tau_1}$
$v_{\tau_2} : T_{\tau_2}$
$v_{\tau_3} : T_{\tau_3}$
A type-indexed family v of values is a function that maps a family of types τ to a family of values v_τ of types T_τ.

Fig. 1. A type-indexed family of values.

This popularity can be attributed to, among other reasons, (1) static typing, which serves as a static debugging facility, and (2) implicit polymorphism (made possible 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, it is impossible to examine the type of a value at run-time, as in a dynamically typed language such as Scheme [24].

Functions that take type arguments and return values of possibly different types accordingly appear frequently in the abstract formulations of certain algorithms. These functions form an interesting class of programs, which seem to be beyond the capability of the Hindley–Milner type system. In this article, we formulate such a function as a *type-indexed family of values*, i.e., a family of values indexed by one or more type argument(s). Fig. 1 illustrates a type-indexed family v indexed by one type argument τ: $v = \{v_\tau\}_{\tau \in F}$, where τ ranges over a family F of types. For a given type τ, the corresponding value is v_τ of type T_τ.

Usually, the family F of types is inductively specified using a set of type constructors. Correspondingly, the F-indexed family v of values is defined by case analysis on the type constructions. Since all types are implicit in a language with Hindley–Milner type system, only value encodings of types, instead of types themselves, can serve as the arguments of a function that represents a type-indexed family. One might try to reduce case analysis on type constructions to case analysis on value constructions, by encoding the type arguments using inductive data types. This would not work, however, because different branches of the case-expression might have different types, and hence the case-expression may not be typeable. A common strategy in such a situation is to use tagged inputs and outputs, which are of some inductive data types as well. However, this solution puts the burden of tagging on the user, which is not only inconvenient, probably unusable when verbatim values are required, but also "type-unsafe", in that a run-time exception might be raised due to unmatched tags.

The problem of programming with type-indexed families of values has exposed the limitations of the Hindley–Milner type system, and it has motivated a line of research that explores more expressive type systems, notably intensional type analysis [15] and polytypic typing [19]. This article, in contrast, investigates what can be done *within* the framework of the Hindley–Milner type system. We demonstrate our methods with ML, though the techniques are equally applicable to any other functional language based on the Hindley–Milner type system.

We first show that interpreting types τ using corresponding values v_τ gives a type-safe solution to the problem. With this approach to type encodings, we show how examples ranging from a printf-like formatting function[1] to type-directed partial evaluation can be programmed in ML. Their type safety is statically ensured by the ML type system.

But with this first approach, a type encoding is application-specific, or *value-dependent*, i.e., the type encoding is tied to a specific family of values. Such a type encoding is not suitable in the practice of modular programming: one should be able to program different type-indexed families that share the same family of type indices separately, and combine them later. It is therefore interesting to find a method of type encoding that is independent of any particular type-indexed family. A value-independent encoding of a specific type τ can then be combined with the representation of a type-indexed family, say v, to deliver the value v_τ. We present two methods of creating such a value-independent type encoding:

1. A type-indexed family of values is specified as a tuple of value constructing functions, one for each possible type constructor, 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 τ is encoded as the embedding and projection functions between verbatim values, of type τ, and tagged values, of a universal data type U. To encode a specific value v_τ of a type-indexed family v, we can first define its equivalent value, whose type substitutes the universal data type for type τ, 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, we view type encodings as type interpretations, characterize requirements for correct implementations of type-indexed values, and give the value-dependent approach to programming type-indexed values in ML. In Section 4, we present the two approaches to value-independent type encodings, and argue that the second approach is universal and more practical. We discuss related work in Section 5 and conclude Section 6.

[1] Originally devised by Danvy [6].

2. Type-indexed families of values

2.1. The notion

Type-indexed families of values (or simply type-indexed families) are used in the formulation of algorithms given in a type-indexed fashion. Depending on input type arguments, specific values within the same family could have different types. For brevity, we mainly consider those families indexed by only one type argument. Multiple type arguments can be reduced to a single type argument by bundling all type indices, but this solution could lead to code explosion. We will return with a practical treatment for dealing with multiple type arguments in Section 4.4.

A type-indexed family is usually defined by induction on the type τ, i.e., it is specified in the form

$$v_\tau = e,$$

where expression e performs a case analysis on type τ, and uses the values indexed at the component types of type τ. This notion is spelled out in the following series of definitions.

Definition 1 (Family of types). An (inductive) family of types is inductively constructed in the following form:

$$\begin{aligned} \tau \; = \; & c_1(\tau_{11}, \ldots, \tau_{1m_1}) \\ \mid \; & \cdots \\ \mid \; & c_n(\tau_{n1}, \ldots, \tau_{nm_n}) \end{aligned} \tag{1}$$

where each c_i is a type constructor, representing a type construction in the underlying language (ML in our case), which builds a type τ using *component types* τ_{i1} through τ_{im_i}. We write $F^{c_1,\ldots,c_n}$ for the family of types.

In most of our examples, the constructors c_i are named as the type constructions they represent, such as $\times$ (product) and $\rightarrow$ (function). It is not uncommon, however, that the constructors c_i are named according to the application: In the string-formatting example (Section 3.4), intuitive formatting directives are used for the constructor names. For these cases, one can take the altervative view that the constructors define a syntactic family, and there is a compositional mapping from this syntactic family to the family of types; we make this mapping notationally explicit whenever necessary.

Without loss of generality, we assume that the case analysis used in the definition of the type-indexed family occurs at the outer-most level of the right-hand-side expression. This assumption accommodates the following definition.

Definition 2 (Type-indexed family of values). A family of values indexed by the family $F^{c_1,\ldots,c_n}$ of types is specified inductively in the following pattern-matching form:

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

In the other words, specific values in the family are constructed inductively using the expressions e_1 to e_n. For a given type τ, we write v_τ to represent the unique term obtained by unfolding the specification according to τ. We require that each expression e_i be expressible in the underlying language, which is ML in our case; this condition guarantees that each v_τ must be an ML-expression. The types of each expression e_i, consequently, might only contain type variables that are bound at the top level, i.e., the type must be rank-1 polymorphic type of the form $\forall\vec{\alpha}.\tau$.

It is the possible variation in the types of the values that makes programming with a type-indexed family challenging and interesting. We therefore should make precise the types of the values in a type-indexed family of values. We observe that, in the applications that we encounter, the types of the values in the same family stay essentially the *same*, modulo the occurrences of the type index. That is, there is a type Q with a free type variable α, such that the type of value v_τ is $T_\tau = Q\{\tau/\alpha\}$.[2] Returning to Definition 2, we can infer that the type scheme of the expression e_i should be instantiatable to $T_{\tau_1} \times \cdots \times T_{\tau_{m_i}} \to T_\tau = Q\{\tau_1/\alpha\} \times \cdots \times Q\{\tau_{m_i}/\alpha\} \to Q\{c_i(\tau_1,\ldots,\tau_{m_i})/\alpha\}$. By abstracting over the types τ_1 through τ_{m_i}, we can write the type scheme for a type-indexed family of values in the following form.

Definition 3 (Type scheme). A type-indexed family of values v, as given in Definition 2, is assigned a *type scheme* as v: $\forall\alpha \in F^{c_1,\ldots,c_n}.Q$, if expression e_i has the (Hindley–Milner) type scheme $\forall\alpha_1,\ldots,\alpha_{m_i}.(Q\{\alpha_1/\alpha\} \times \cdots \times Q\{\alpha_{m_i}/\alpha\}) \to Q\{c_i(\alpha_1,\ldots,\alpha_{m_i})/\alpha\}$ for all $1 \leqslant i \leqslant n$.

2.2. Running examples

This section introduces some examples to demonstrate the challenges posed by type-indexed families. They are revisited in later sections to illustrate our methods for programming with type-indexed families.

2.2.1. List flattening and polytypic printing

The flatten program, which flattens arbitrary nested lists with integer elements, is a toy example often used to illustrate the intricacy of typing "typecases" (case studies on types) in languages with Hindley–Milner type systems. It can be written in a dynamically typed language like Scheme (where type testing is allowed) as:

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

[2] The type Q could contain other free type variables, which, however, can only be used monomorphically.

where $\otimes$ is the list concatenation operator. To write this function in ML, a natural solution is to use the ML data type mechanism to define a data type for nested lists, 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" here refers to values whose types are formed using only native ML type constructors, and are 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 particular case that members are lists themselves, they must have the same nesting depth. The flatten function for verbatim values can therefore be specified as a family of flatten functions indexed by the possible type argument τ.

Example 4 (flatten). The family $F^{\mathsf{int},\mathsf{list}}$ of types is generated by the following grammar.[3]

$$\tau = \mathsf{int} \mid \tau_1\ \mathsf{list}$$

The flatten function can then be specified as an $F^{\mathsf{int},\mathsf{list}}$-indexed family of function values.

$$\begin{aligned}
&\mathsf{flatten} &&: \forall\alpha \in F^{\mathsf{int},\mathsf{list}}.\,\alpha \to \mathsf{int\ list}\\
&\mathsf{flatten}_{\mathsf{int}}\, x &&= [x]\\
&\mathsf{flatten}_{\tau_1\ \mathsf{list}}[x_1,\ldots,x_n] &&= (\mathsf{flatten}_{\tau_1}\, x_1) \otimes \cdots \otimes (\mathsf{flatten}_{\tau_1}\ x_n)
\end{aligned}$$

This definition conforms to Definition 2, since we can also write:

$$\begin{aligned}
&\mathsf{flatten}_{\mathsf{int}} &&= e_1\\
&\mathsf{flatten}_{\tau_1\ \mathsf{list}} &&= e_2(\mathsf{flatten}_{\tau_1})
\end{aligned}$$

where

$$\begin{aligned}
&e_1 : \mathsf{int} \to \mathsf{int\ list} &&= \lambda x.[x]\\
&e_2 : \forall\alpha.(\alpha \to \mathsf{int\ list}) \to (\alpha\ \mathsf{list} \to \mathsf{int\ list}) &&= \lambda f.\lambda[x_1,\ldots,x_n].f x_1 \otimes \cdots \otimes f x_n
\end{aligned}$$

It should be easy to check that flatten has the declared type scheme (Definition 3).

Before trying to write the function flatten, let us analyze how it might be used. A first attempt is to make the input value (of some arbitrary homogeneously nested list type) the only argument. This would require that both the expression `flatten 5` and the expression `flatten [6]` type-check, so the function argument should be of a polymorphic type that generalizes both type `int` and type `int list`—this polymorphic type can only be a type variable α. But ML's parametric polymorphism disallows inspecting the type structure of a polymorphic value. Consequently, it is impossible to write function flatten with the value to be flattened as the only argument.

[3] It is for the ease of presentation that we use int for the base case here, instead of a universally quantified type variable. We discuss type variable and polymorphism in Section 3.5.

The next attempt is to use an extra argument for 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, constrains the result of applying function $\mathsf{flatten}$ to the type encoding to a fixed ML type again. A simple reasoning like the one for the first attempt shows that it is still impossible to give a typeable solution in ML.

A similar, but probably more useful example of type-indexed family, is polytypic printing. The functionality of polytypic printing is to convert data of arbitrary verbatim types to a string representation suitable for output, according to the type. For the type constructors, we consider integer, string, product, and list.

Example 5 (polytypic printing). The family $F^{\mathsf{toStr}} = F^{\mathsf{int},\mathsf{str},\times,\mathsf{list}}$ of types is generated by the following grammar:

$$\tau = \mathsf{int} \mid \mathsf{str} \mid \tau_1 \times \tau_2 \mid \tau_1 \ \mathsf{list}$$

Polytypic printing is specified as a F^{toStr}-indexed family of functions.

$$\begin{array}{lll}
\mathsf{toStr} & : & \forall\tau \in F^{\mathsf{toStr}}.\tau \to \mathsf{str} \\
\mathsf{toStr}_{\mathsf{int}}\, x & = & \mathsf{intToStr}\, x \\
\mathsf{toStr}_{\mathsf{Str}}\, s & = & s \\
\mathsf{toStr}_{\tau_1\times\tau_2}(x_1, x_2) & = & \text{“(”} \wedge (\mathsf{toStr}_{\tau_1}\, x_1) \wedge \text{“,”} \wedge (\mathsf{toStr}_{\tau_2}\ x_2) \wedge \text{“)”} \\
\mathsf{toStr}_{\tau_1\ \mathsf{list}}[x_1, \ldots, x_2] & = & \text{“[”} \wedge (\mathsf{toStr}_{\tau_1}\, x_1) \wedge \cdots \wedge \text{“,”} \wedge (\mathsf{toStr}_{\tau_1}\ x_n) \wedge \text{“]”}
\end{array}$$

where $\wedge$ is the string concatenation function.

Having specified polytypic printing, we can use it to define a C printf-style formatting function, also as a type-indexed family. The formatting function, however, is slightly more involved. In order not to distract the reader with its details at the current stage, we postpone its development to Section 3.4, where the basic programming technique has already been introduced.

2.2.2. Type-directed partial evaluation

Partial evaluation is an automatic program transformation technique that specializes programs with respect to partial input, thereby improving the performance of the programs when running on the remaining input. Traditional partial evaluators are syntax-directed [23]: they work by simplifying the partially applied subject program through symbolic computation. An efficient alternative partial evaluation technique is type-directed partial evaluation (TDPE) [4,11]. In the case of TDPE, the simplification process for the partially applied subject program is carried out using a reduction-free normalizer: instead of performing reduction, it extracts the normal form of a program from a suitably chosen interpretation of the program. In this interpretation, the base types are interpreted as code (or expression) types.

$$\begin{array}{lll}
\text{(reify)} & \downarrow^{\mathsf{exp}} v & = v \\
 & \downarrow^{\tau_1 \to \tau_2} f & = \underline{\lambda} x.\, \downarrow^{\tau_2} (f(\uparrow_{\tau_1} x)) \\
 & & \quad \text{(where } x \text{ is a fresh variable)} \\
\text{(reflect)} & \uparrow_{\mathsf{exp}} e & = e \\
 & \uparrow_{\tau_1 \to \tau_2} e & = \lambda v_1.\, \uparrow_{\tau_2} \; (e \underline{@} (\downarrow^{\tau_1} v_1))
\end{array}$$

Fig. 2. Type-directed partial evaluation.

We shall not go into detail of the TDPE technique itself, but devote our attention to the type-indexed family of the extraction functions, called *reification*. Given the interpretation of base types as code types, the extraction functions can be viewed as coercions from higher-order types constructed from the code type using various type constructors to the code type. In this paper, we consider only the function type constructor, and TDPE only for pure simply-typed λ-terms.

Example 6 (Type-directed partial evaluation). The family $F^{\mathsf{exp},\to}$ of types is generated by the following grammar:

$$\tau = \mathsf{exp} \mid \tau_1 \to \tau_2$$

where the type `exp` is an inductive type that provides representation for generated code.

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

To use ML with `exp` as an informal two-level language for describing code-generation algorithms, we write $\underline{\lambda} x.E$ as shorthand for `LAM(`x`, `E`)`, $E_1 \underline{@} E_2$ as shorthand for `APP(`E_1`, `E_2`)`, and an occurrence of $\underline{\lambda}$-bound variable x as shorthand for `VAR(`x`)`.

The extraction function for type-directed partial evaluation is defined as two families of type-indexed functions $\downarrow$ (reify) and $\uparrow$ (reflect) (Fig. 2), which recursively call each other for the contravariant function argument. At first glance, their definitions do not fit into the canonical form of a type-indexed family (Definition 2); however, pairing the two functions at each type index puts the definition into the standard form of a type-indexed family (Fig. 3).

In the following, we write "$\bullet$" as a shorthand for the type `exp`.

It might be helpful, for a rough intuition of the TDPE algorithm, to work out some simple examples, such as $\downarrow^{\bullet \to \bullet \to \bullet \to \bullet} ((\lambda x.\lambda y.x)(\lambda x.\lambda y.x))$ and $\downarrow^{(\bullet \to \bullet) \to (\bullet \to \bullet)}$ $(\lambda f.\lambda x.f(f(x)))$. Detailed accounts of type-directed partial evaluation can be found in the literature [8,11,13].

In his article [4], Danvy presents the Scheme code for this algorithm. He represents the type index as a value—which is an S-expression and is similar to a value of inductive data type in ML—thereby reducing type analysis to case analysis. A direct transcription of that program into an ML program, however, would require the input

$$\begin{array}{lll}
(\downarrow, \uparrow) & : & \forall \tau \in F^{\mathsf{exp}, \rightarrow}.(\tau \rightarrow \mathsf{exp}) \times (\mathsf{exp}) \rightarrow \tau) \\
(\downarrow, \uparrow)_{\mathsf{exp}} & = & (\lambda v.v, \lambda e.e) \\
(\downarrow, \uparrow)_{\tau_1 \rightarrow \tau_2} & = & \textbf{let} \quad (\downarrow^{\tau_1}, \uparrow_{\tau_1}) = (\downarrow, \uparrow)_{\tau_1} \\
 & & \qquad\quad (\downarrow^{\tau_2}, \uparrow_{\tau_2}) = (\downarrow, \uparrow)_{\tau_2} \\
 & & \textbf{in} \quad\;\; (\lambda f.\underline{\lambda} x. \downarrow^{\tau_2} (f(\uparrow_{\tau_1} x)), \\
 & & \qquad\quad \lambda e.\lambda v. \uparrow_{\tau_2} \; (e \underline{@} (\downarrow^{\tau_1} v)) \\
 & & (\text{where } x \text{ is a fresh variable})
\end{array}$$

Fig. 3. TDPE in the general form of type-indexed family.

arguments to be tagged. Such a solution is not satisfactory for the following reasons:

- Using type-directed partial evaluation, one expects to normalize a program in the source language with minimum modification. It is cumbersome for the user to tag/untag all the program constructs. A verbatim program is much preferable in this case.
- Unlike the function flatten, the function $\uparrow$ (reflect) requires the type argument explicitly. The type index τ only appears as the codomain of the function $\uparrow$, whereas its domain is always of type exp. For the same input expression, varying the type argument results in different return values of different types.

  Because explicit type arguments must be present, static type checking of ML cannot guarantee the consistency of the type argument and the tags attached to the input values cannot be guaranteed by static type checking of ML: run-time "type error" can arise in the form of a pattern mismatch exception. This problem is also present in the Scheme program.

3. Type-indexed families as type interpretations

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

3.1. Implementing indexed families with type encodings: the idea

As we argued in the list-flattening example (Section 2.2.1), if verbatim arguments are required for an ML function that represents a type-indexed family, then (1) a type encoding must be explicitly provided as an argument to the function, and (2) this type encoding cannot have a fixed type. Now that the type encodings, say E_τ for all $\tau \in F$, must have different types themselves, a reasonable choice of these types should make them reflect the types τ being encoded.

A type family usually consists of infinitely many types, therefore we should not encode types directly, but encode type constructors, such that the encoding of a type is constructed inductively by using the encoding of the type constructors. To be more precise, for each type constructor c of arity m (i.e., it constructs a type τ from types $\tau_1, \ldots, \tau_m$), its encoding E_c as a term (in ML) 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 should form a particular syntactic interpretation, in the underlying language. If we use $\langle\!|u|\!\rangle$ instead of E_u to denote the interpretation, we can write down the requirements for the encodings:

$$\begin{array}{lcl} \text{If} \quad \tau & = & c(\tau_1, \ldots, \tau_m) \\ \text{then} \ \langle\!|\tau|\!\rangle & \equiv & \langle\!|c|\!\rangle(\langle\!|\tau_1|\!\rangle, \ldots, \langle\!|\tau_m|\!\rangle). \end{array}$$

where $\equiv$ represents strict syntactic equality. This can be understood as requiring the interpretations of type and type constructors to form a homomorphism, i.e.,

$$\langle\!|c(\tau_1, \ldots, \tau_m)|\!\rangle \equiv \langle\!|c|\!\rangle(\langle\!|\tau_1|\!\rangle, \ldots, \langle\!|\tau_m|\!\rangle) \tag{3}$$

This way, for the encoding of a whole family of types, it suffices to give the encoding of the type constructors, since the encoding of every type is uniquely determined.

Definition 7 (Encoding a family of types). The encoding of a family of types $F^{c_1,\ldots,c_n}$ (as given in Definition 1) is specified as the encoding of the constructors c_1 through c_n as ML-terms $\langle\!|c_1|\!\rangle$ through $\langle\!|c_n|\!\rangle$, such that the encoding $\langle\!|\tau|\!\rangle$ of every type, induced according to Eq. (3), is typeable.

With such an encoding $\langle\!| \cdot |\!\rangle$ of a type family F, a F-indexed family v of values can then be represented as a function f_v that takes the type encoding as an argument.

Definition 8 (Implementation of type-indexed families). Let F be a family of types, v be an F-indexed family of values, and $\langle\!| \cdot |\!\rangle$ be an encoding of F (given on the constructors). An ML function f_v *implements* a type-indexed value v through the encoding $\langle\!| \cdot |\!\rangle$, if the following equation holds for all $\tau \in, F$.

$$v_\tau = f_v \langle\!|\tau|\!\rangle. \tag{4}$$

The equality used in Eq. (4) should be the appropriate semantic equality in the implementation language: $\beta\eta$-equivalence for the pure λ-calculus, λ_c-equivalence for a language with computational effects [29], such as ML.

3.2. The ad hoc approach

The task of finding the type encodings now boils down to finding suitable interpretations for the type constructors c_i. The close similarities between the general form of type-indexed values in the set of equations given by (2) on p. 8 and the interpretation of type constructors in Eq. (3) hints at an immediate solution to programming with type-indexed families: We can interpret a type τ as the corresponding value v_τ, which is, in turn, achieved by interpreting the type construction c_i using the value construction e_i in the set of equations given by (2).

```
val Int: int -> int list                                   (* ⦇int⦈ *)
    = fn x => [x]
fun List (T:'a -> int list): ('a list -> int list)         (* ⦇list⦈ *)
    = fn l => foldr (op @) [] (map T l)
fun flatten T = T                                 (* f_flatten ≜ λx.x *)
```

Fig. 4. flatten in ML: ad hoc encoding.

Proposition 9 (Ad hoc encoding). *Let $F^{c_1,\dots,c_n}$ be a family of types and v an $F^{c_1,\dots,c_n}$-indexed family of values, with associated data as presented in Definition 2, i.e., type constructors c_i and corresponding data constructions e_i. The following equation defines a type encoding for F.*

$$\langle\!|c_i|\!\rangle \triangleq e_i$$

It induces the following encoding of types:

$$\langle\!|\tau|\!\rangle \equiv v_\tau$$

Proof. That $\langle\!|\tau|\!\rangle \equiv v_\tau$ follows immediately from Eqs. (2) and (3). The typeability of v_i implies the typeability of $\langle\!|\tau|\!\rangle$. □

This type encoding is ad hoc, in the sense that it is specific to the F-indexed family v. The implementation of the indexed family is immediate.

Theorem 10. *Let $F^{c_1,\dots,c_n}$ be a family of types and v a $F^{c_1,\dots,c_n}$-indexed family of values, with associated data as presented in Definition 2. The identity function $f_v \triangleq \lambda x.x$ implements v through the ad hoc encoding $\langle\!|c_i|\!\rangle \triangleq e_i$.*

3.3. Examples

Let us demonstrate the ad hoc type-encoding method with the running examples.

Example 11 (flatten). The definition of the function flatten (Example 4) gives rise to the following interpretations of type constructions:

$$\begin{aligned}\langle\!|.|\!\rangle &: \forall\tau \in F^{\mathsf{int},\mathsf{list}}.\tau \to \mathsf{int\ list}\\ \langle\!|\mathsf{int}|\!\rangle &= \lambda x.[x]\\ \langle\!|\alpha\ \mathsf{list}|\!\rangle &= \lambda[x_1,\dots,x_n].\langle|\alpha|\rangle x_1 @ \cdots @ \langle\!|\alpha|\!\rangle x_n\end{aligned}$$

A direct coding of these interpretations of type construction as ML functions leads to the program in Fig. 4.

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

$$\langle\!|(\mathsf{int\ list})\mathsf{list}|\!\rangle = \mathsf{flatten}_{(\mathsf{int\ list})\ \mathsf{list}},$$

```
type 'a ts = 'a -> string                        (* Type scheme : α → str *)
val Int: int ts                                                 (* ⦇int⦈ *)
    = fn n => Int.toString n
fun Str: string ts                                              (* ⦇str⦈ *)
    = fn s => s
fun Pair (toStr1: 'a ts) (toStr2: 'b ts)                        (* ⦇×⦈ *)
    : ('a * 'b) ts
    = fn (x1: 'a, x2: 'b) =>
        "(" ^ (toStr1 x1) ^ ", " ^ (toStr2 x2)  ^ ")"
fun List (toStr: 'a ts): ('a list ts)                          (* ⦇list⦈ *)
    = fn (l: 'a list) =>
      let fun mkTail []  = "]"
            | mkTail [e] = (toStr e) ^ "]"
            | mkTail (e :: el)
              = (toStr e) ^ ", " ^ (mkTail el)
      in "[" ^ (mkTail l)
      end
fun toStr T = T                                          (* f_toStr ≜ λx.x *)
```

Fig. 5. Polytypic printing in ML: ad hoc encoding.

and function `flatten` can be defined just as the identity function. As desired, the function takes verbatim values as input. For example, the expression

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

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

This example exhibits the basic pattern of the ad hoc approach to programming with a type-indexed family: for each type constructor, we bind to its corresponding ML name the expression e_i through a value or function definition. Their associated types, as given in Definition 3, are rank-1 and therefore accepted by the ML type system.[4]

The same method works for the examples of polytypic printing and type-directed partial evaluation.

Example 12 (polytypic printing). Fig. 5 shows the ML implementation of polytypic printing, as formulated in Example 5, using the type encoding $\langle\!|\tau|\!\rangle \triangleq \mathsf{toStr}_\tau$.

As an example, evaluating the expression

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

yields `"[(N, [Prince, 8, 14]), (P, [Newport, Christopher, 9])]"`.

[4] The type annotations are not necessary, since the ML type system infers the most general type scheme anyway; we include them for the sake of clarity.

```
datatype exp = VAR of string                                    (* exp *)
             | LAM of string * exp
             | APP of exp * exp

type 'a rr = ('a -> exp) * (exp -> 'a)
                                (* Type scheme : (α → exp) × (exp → α) *)

infixr 5 -->
val a': exp rr                                                  (* ⟅exp⟆ *)
    = (fn v => v, fn e => e)
fun (T1 as (reif1, refl1): 'a rr) -->                           (* ⟅ → ⟆ *)
    (T2 as (reif2, refl2): 'b rr): ('a -> 'b) rr
    =  (fn (f: 'a -> 'b) =>
           let val x = Gensym.new()
           in  LAM(x, reif2 (f (refl1 (VAR x))))
           end,
        fn (e: exp) =>
           fn (v: 'a) => refl2 (APP(e, reif1 v)))
fun reify (T as (reif_T, refl_T)) = reif_T                      (* f↓ *)
fun reify_init T v = (Gensym.init(); reify T v)
                                (* reify, with name counter initialized *)
```

Fig. 6. Type-directed partial evaluation in ML.

Example 13 (type-directed partial evaluation). Fig. 6 shows the ML implementation of type-directed partial evaluation, as described in Example 6 and formulated as a type-indexed family of values in Fig. 3, using the type encoding $\langle\!|\tau|\!\rangle = (\downarrow, \uparrow)_\tau$. The structure `Gensym` provides a function `new` for generating fresh names using a counter, and a function `init` to initialize this counter.

As an example, the expression

```
reify_init (a' -> a' -> a' -> a')
       ((fn x => fn y => x) (fn x => fn y => x))
```

evaluates to `LAM("x1", LAM("x2", LAM("x3", VAR "x2")))`, which represents the λ-expres sion $\lambda x_1.\lambda x_2.\lambda x_3.x_2$.

3.4. Printf-style String formatting

We can apply the ad hoc type encoding method to program a type-safe formatting function in the style of the C printf function. In fact, this is an example where it is more natural not to view the indices as types directly, but to view them as syntactic phrases that are translated to the types that they represent under a compositional mapping, say $\mathscr{R}$.

We consider the formatting specification as a sequence of field specifiers. The grammar of formatting specification is given below:

$$\begin{aligned} \mathit{Spec} &::= \texttt{NIL} \mid \mathit{Field} :: \mathit{Spec} \\ \mathit{Field} &::= \texttt{LIT}\ s \mid \texttt{\%}\ \tau \end{aligned}$$

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 "-years old.")
       "Mickey" 80
```

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

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

Example 14. The syntactic family F^{fs} of formatting specifications is given by the following grammar (in concrete syntax):

$$fs ::= \texttt{NIL} \mid \texttt{LIT}\ s :: fs' \mid \texttt{\%}\ \tau :: fs' \quad (\tau \in F^{\mathsf{toStr}})$$

Here, the 0-ary constructor NIL and the binary constructor :: are used for building sequences, while the unary constructors LIT s and % τ are used for building individual field specifiers.

A formatting specification fs determines the type of format_{fs}, through the compositional translation $\mathscr{R}$, defined as follows:

$$\begin{aligned} \mathscr{R}(\texttt{NIL}) &= \mathsf{str} \\ \mathscr{R}(\texttt{LIT}\ s :: fs') &= \mathscr{R}(fs') \\ \mathscr{R}(\texttt{\%}\ \tau :: fs') &= \tau \to \mathscr{R}(fs') \end{aligned}$$

We specify format' as an F^{fs}-indexed family via $\mathscr{R}$, and use it to define format.

$$\begin{aligned} \mathsf{format}' &: \forall fs \in F^{fs}.\mathsf{str} \to \mathscr{R}(fs) \\ \mathsf{format}'_{\texttt{NIL}} b &= b \\ \mathsf{format}'_{\texttt{LIT}\ s::fs'} b &= \mathsf{format}'_{fs'}(b\,\hat{}\,s) \\ \mathsf{format}'_{\texttt{\%}\ \tau::fs'} b &= \lambda(x:\tau).\mathsf{format}'_{fs'}(b\,\hat{}\,\mathsf{toStr}_\tau x) \end{aligned}$$

$$\begin{aligned} \mathsf{format} &: \forall fs \in F^{fs}.\mathscr{R}(fs) \\ \mathsf{format}_{fs} &= \mathsf{format}'_{fs}(\text{“ ”}) \end{aligned}$$

where the F^{toStr}-indexed family toStr is from Example 5.

```
infix 5 ++
type 'a fmt = string -> 'a
                                   (* Type scheme (of format'): str → ℛ(fs) *)

fun LIT s: ('a fmt -> 'a fmt)                                (* ⦉ LIT s⦊ *)
    = fn (p: 'a fmt) => fn b => p (b ^ s)
fun % (toStr_t: 'c ts): ('a fmt -> ('c -> 'a) fmt)
                                                               (* ⦉ % τ⦊ *)
    = fn (p: 'a fmt) => fn b =>
         fn (x: 'c) => p (b ^ toStr_t x)
fun f1 ++ f2 = f1 o f2
val NIL: (string fmt)                                          (* ⦉NIL⦊ *)
    = fn b => b
fun format (ftrans: string fmt -> 'a fmt)
    = ftrans NIL ""
```

Fig. 7. printf-style formatting in ML: ad hoc encoding.

Following the ad hoc method, we need to interpret each individual field specification f (LIT s or % τ), which is a constructor for formatting specifications. We can define the function $\llparenthesis f \rrparenthesis$ as a transformer from format'_{fs} to $\mathsf{format}'_{f::fs}$, i.e.,

$$\mathsf{format}'_{f::fs} = \llparenthesis f \rrparenthesis\ \mathsf{format}'_{fs}$$

We obtain the interpretation of individual field specifiers by abstracting over format'_{fs}:

$$\begin{aligned}\llparenthesis \texttt{LIT}\ s \rrparenthesis &= \lambda \mathsf{format}'_{fs}.\lambda b.\mathsf{format}'_{fs}(b\,\hat{}\,s)\\ \llparenthesis \texttt{\%}\ \tau \rrparenthesis &= \lambda \mathsf{format}'_{fs}.\lambda b.\lambda(x:\tau).\mathsf{format}'_{fs}(b\,\hat{}\,\mathsf{toStr}_\tau x)\end{aligned}$$

On the practical side, it is undesirable to use the field specifications as nested prefix constructors that build formatting specifications from NIL. For this purpose, we define a function ++ to compose such transformers (similar to the function append for lists), and we can define the function format to take such specification transformer, instead of a specification, and to supply the interpretation of the empty field specification $\llparenthesis \texttt{NIL} \rrparenthesis = \mathsf{format}'_{\texttt{NIL}}$, along with an empty string as the initial buffer. Putting all the pieces together, we have the ML implementation of the string formatter in Fig. 7; it uses the implementation of toStr in Fig. 5.

Unlike the C printf function, the above ML implementation is type-safe: compilation would reject mismatched field specification and input argument. For example, the type of the expression

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

is $\mathsf{int} \to \mathsf{str} \to \mathsf{str}$, which ensures that exactly two arguments, one of type int, the other of type str, can be supplied.

The printf example also showcases the power of a higher-order functional language with the possibility of building constructing field specifiers for compound types, through toStr. The following expression, following Example 12, illustrates this flexibility.

```
format (LIT "MTA/NJT: " ++ %(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. By defining more complicated field specifiers, which, e.g., allow variations in paddings, delimiters for the layout of compound types, and by refining the output type from string to more sophisticated formatting objects, we can construct sophisticated pretty-printers like John Hughes's [18], through the convenient printf-style interface.

Let us conclude this section with a comparison made by Danvy [5] 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. When applied to a formatting specification, however, the function will be specialized by the compiler, through some dedicated mechanism, into 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, and remains highly extensible.

3.5. Variations

The ad hoc method easily adapts to several variations of type-indexed families.

Multiple translations of indices. In the examples up until now, the type schemes either uses the indices directly, or a single translation of them (such as $\mathscr{R}$ in the string-formatting example). We shall see that having multiple translations of the indices occurring in the type scheme does not add to the complexity.

Other type constructions. The type constructions used in the earlier examples include product, function, and list. One might wonder to what other type constructions the ad hoc method is applicable. In fact, since we left unspecified the type constructions in our formulation of type-indexed families and their implementations, type-indexed families with any type constructions can be implemented using the ad hoc method, as long as they can be cast into the form of Definition 2 (where e_i's could be rank-1 polymorphic); we shall see an example that uses both the reference type and the continuation type (Example 15). Type constructions that bind type variables, however, do not conform to our formulation of type-indexed families.

- Polymorphic types: suppose that the universal quantifier $\forall\beta$ could be used as a constructor. It should construct from a type τ parameterized over a free type variable β (or, equivalently, a type constructor T such that $\tau = T(\beta)$), the type $\forall\beta.\tau$. A possible type for the corresponding value construction $e_{\forall\alpha.}$ could be $(\forall T : * \rightarrow *.(\forall\beta.Q\{T(\beta)/\alpha\} \rightarrow Q\{\forall\beta.T(\beta)/\alpha\}))$. Unfortunately, the polymorphism of this type is not rank-1, and it is higher-order; therefore universal qualification cannot be used as a type construction.

This analysis, however, does not rule out using a type variable as a base type (0-ary type constructor), universally quantified over at the top level of the type expression. In fact, in the flatten example, replacing the base type int by a type variable would cause no problem.

Instead of performing analysis over the universal quantifier, one might want to parameterize an index over another index or a constructor. There is no problem with abstracting over an index, since the corresponding encoding is used only monomorphically in building other type encodings. Taking the polytypic printing example (Example 12), we can code $\lambda\tau.\tau \times \tau\,\mathsf{list}$ as `fn t => Pair t (List Int)`. In contrast, to abstract over a constructor is generally not permissible, since we want to use the encoding of type constructors polymorphically in building type encodings. Again taking the polytypic printing example, coding $\lambda c : * \rightarrow *.c((c(\mathsf{int}))\ \mathsf{list})$ as `fn x => x (List (x Int))` results in a type error.

- Recursive types: as we have seen in the flatten example, we can use a recursive type, such as list, in the type construction; we can as well replace list with a user-defined recursive type, such as Tree. On the other hand, by the same reason we cannot perform type analysis over a universal quantifier, we cannot perform type analysis over the recursive type constructor itself.

Let us demonstrate some of these possible variations through another example: ironing (or, indirection elimination), which eliminates indirections of either reference type or double-continuation type. It showcases not only some extra type constructions, but also the use of two separate translations from the indices, one for the domain type, and one for the codomain type.

Example 15 (iron). The family F^{iron} of indexes is generated by the following grammar:

$$\tau = \mathsf{int} | \tau_1\ \mathsf{list} | \tau_1 \times \tau_2 |\ \tau_1\ \mathsf{ref} | \neg\neg(\tau_1)$$

The indices are almost the domain types, except for the constructor $\neg\neg$, which is mapped into a double-continuation construction. The indices are mapped into the domain types and codomain types through the compositional translations $\mathscr{R}_I$ and $\mathscr{R}_O$, respectively.

$$\begin{aligned}
\mathscr{R}_I(\mathsf{int}) &= \mathsf{int}\\
\mathscr{R}_I(\tau_1\ \mathsf{list}) &= \mathscr{R}_I(\tau_1)\ \mathsf{list}\\
\mathscr{R}_I(\tau_1 \times \tau_2) &= \mathscr{R}_I(\tau_1) \times \mathscr{R}_I(\tau_2)\\
\mathscr{R}_I(\tau_1\ \mathsf{ref}) &= \mathscr{R}_I(\tau_1)\ \mathsf{ref}\\
\mathscr{R}_I(\neg\neg(\tau_1)) &= \mathscr{R}_I(\tau_1)\ \mathsf{cont}\ \mathsf{cont}
\end{aligned}$$

$$\begin{array}{ll}
\mathsf{iron} & : \forall\tau \in F^{\mathsf{iron}}.\mathscr{R}_I(\tau) \to \mathscr{R}_O(\tau) \\
\mathsf{iron}_{\mathsf{int}} x & = x \\
\mathsf{iron}_{\tau_1\ \mathsf{list}}[x_1,\ldots,x_n] & = [\mathsf{iron}_{\tau_1} x_1,\ldots,\mathsf{iron}_{\tau_1} x_n] \\
\mathsf{iron}_{\tau_1\times\tau_2}(x_1,x_2) & = (\mathsf{iron}_{\tau_1} x_1, \mathsf{iron}_{\tau_2} x_2) \\
\mathsf{iron}_{\tau_1\ \mathsf{ref}} x & = \mathsf{iron}_{\tau_1}(!x) \\
\mathsf{iron}_{\neg\neg(\tau_1)} c & = \mathsf{iron}_{\tau_1}(\mathsf{callcc}(\lambda k.\mathsf{throw}\, c k)
\end{array}$$

Fig. 8. The indexed family of ironing functions.

$$\begin{array}{ll}
\mathscr{R}_O(\mathsf{int}) & = \mathsf{int} \\
\mathscr{R}_O(\tau_1\ \mathsf{list}) & = \mathscr{R}_O(\tau_1)\ \mathsf{list} \\
\mathscr{R}_O(\tau_1 \times \tau_2) & = \mathscr{R}_O(\tau_1) \times \mathscr{R}_O(\tau_2) \\
\mathscr{R}_O(\tau_1\ \mathsf{ref}) & = \mathscr{R}_O(\tau_1) \\
\mathscr{R}_O(\neg\neg(\tau_1)) & = \mathscr{R}_O(\tau_1)
\end{array}$$

The ironing function is defined as an indexed family of functions iron in Fig. 8. The double-continuation construction probably is not very useful by itself; but in conjunction with the reference type, it can be used as the type for implementing coroutine-style iterators. That is, the type τ iter $\triangleq \tau$ cont cont ref can be used to implement an iterator over inductive data structures such as trees, which enumerates elements of type τ upon "ironing". We leave the detail of implementing such iterators to Example 16.

The ML implementation is shown in Fig. 9. As an example, the program

```
let val v1 = ref [3,4]
    and v2 = ref [5,6]
in
    iron (List (Ref (List Int))) [v1, v2, v1, v2]
end
```

evaluates to `[[3, 4], [5, 6], [3, 4], [5, 6]]`.

Example 16 (Coroutine-style iterators). A double continuation type, say τ cont cont, can be understood as the type of one-shoot τ-typed value producers. To invoke such a producer, i.e., to retrieve the τ-typed value, the caller should 'throw' to the producer, which is a continuation, its own current continuation, which is of type τ cont. This is realized through the definition of $\mathsf{iron}_{\neg\neg(\tau)}$ in Fig. 8.

If we further make the double continuation type mutable through a reference cell, i.e., employ the type τ cont cont ref, we can implement a producer thread by changing the stored continuation when suspending the thread. The structure `Iterator` Fig. 10 realizes this idea. In particular, the function `makeIterator` converts a "producer" function to an iterator of the mutable double continuation type. This producer function takes as arguments a function to yield the control of the thread and "produce" an answer. The function `makeIterator` takes also a second argument to provide a default value, which the iterator should produce after the return from the "producer" function.

```
val Int: int -> int                                        (* ⦇int⦈ *)
    = fn x => x
fun List (T: 'a -> 'b): ('a list -> 'b list)               (* ⦇list⦈ *)
    = fn l => map T l
fun Pair (T1: 'a1 -> 'b1) (T2: 'a2 -> 'b2)                 (* ⦇ × ⦈ *)
    : ('a1 * 'a2) -> ('b1 * 'b2)
    = fn (x1, x2) => (T1 x1, T2 x2)
fun Ref (T: 'a -> 'b): ('a ref -> 'b)                      (* ⦇ref⦈ *)
    = fn cell => T (! cell)
fun DblNeg (T: 'a -> 'b): ('a cont cont -> 'b)             (* ⦇¬¬⦈ *)
    = fn (c: 'a cont cont) =>
         T (callcc (fn (k: 'a cont) => throw c k))
fun Iter (T: 'a -> 'b): ('a cont cont ref -> 'b)           (* ⦇iter⦈ *)
    = Ref (DblNeg T)

fun iron T = T                                             (* f_iron *)
```

Fig. 9. iron in ML: ad hoc encoding.

Fig. 11 presents an example of using structure `Iterator`; it builds an iterator for a binary tree. Finally, as an example for both ironing and iterator, the program

```
val iter1
  = tree2Iter (ND(ND(ND(LF 1, LF 2), LF 3),
                  ND(LF 4, ND(LF 5, LF 6)))) ~1
val iter2
  = tree2Iter (ND(ND(LF 5, LF 4),
                  ND(LF 3, ND(LF 2, LF 1)))) ~2
val pI    = [iter1, iter2]
val zipped
  = iron (List (List (Iter Int)))
         [pI, pI, pI, pI, pI, pI, pI, pI]
```

zips together the traversals of the two trees to produce the following result.

```
[[1,5],[2,4],[3,3],[4,2],[5,1],[6,~2],[~1,~2],[~1,~2]]
```

3.6. Assessment of the approach

The ad hoc encoding of a type used in the previous sections is exactly the specialized value in the indexed family at 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 during program execution. For a type-indexed family of values $v : \forall\beta \in F.Q$, the encoding $\llparenthesis\tau\rrparenthesis = v_\tau$ of a particular type index τ already has the required type $T_\tau = Q\{\tau/\beta\}$. Often, the value v_τ is a function that takes some argument whose type depends on type τ. Since the specific type of this argument is manifested in the type T_τ, input arguments of illegal types are rejected.

```
structure Iterator =
struct
  local open SMLofNJ.Cont in
  type 'a iterator = 'a cont cont ref
  fun dblNeg (v: 'a): 'a cont cont
    = callcc (fn k => throw (callcc (fn c => throw k c)) v)

  fun makeIter (producer: ('a -> unit) -> unit) (dflt: 'a)
    : 'a cont cont ref
    = let val stream = ref (dblNeg dflt)
          val _ =
            callcc(fn back: unit cont =>
              let val consumer_k =
                    ref (callcc(fn init_c =>
                          (stream := init_c;
                           throw back ())))
              in
                (producer (fn v =>
                  (consumer_k :=
                    callcc(fn (c: 'a cont cont) =>
                      ((stream := c);
                       (throw (!consumer_k) v))))));
                (stream := dblNeg dflt);
                (throw (!consumer_k) dflt)
              end)
      in  stream
      end
  end
end
```

Fig. 10. Iterators from mutable double continuation.

- 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 (which include 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; this has been used, for example, in the implementation of the $\uparrow$ functions for type-directed partial evaluation.

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 obtain the specific value indexed at this type, but from different indexed families. 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, hence disallowing sharing of the type encodings.

```
structure BTIterator =
struct
  structure I = Iterator
  datatype 'a BTree = ND of 'a BTree * 'a BTree | LF of 'a
  fun tree2Iter (T: 'a BTree) (dflt: 'a): 'a I.iterator
    = I.makeIter
        (fn (produce: 'a -> unit) =>
          let fun traverse (ND(ltree, rtree))
                = ((traverse ltree); (traverse rtree))
                | traverse (LF(n))
                = produce n
          in  traverse T
          end)
        dflt
end
```

Fig. 11. Iterators for binary trees.

```
fun Int x = x                                        (* ⟨|int|⟩ *)
fun List T = rev o (map T)                           (* ⟨|list|⟩ *)
fun super_reverse T = T                              (* f_super_reverse *)
```

Fig. 12. super_reverse in ML: ad hoc encoding.

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

Example 17 (Super reverse). Let us consider the type-indexed family of functions super_reverse, defined as follows.

$$\begin{aligned}
\mathsf{super\_reverse} &: \forall\alpha \in F^{\mathsf{int},\mathsf{list}}.\alpha \to \alpha \\
\mathsf{super\_reverse}_{\mathsf{int}} &= \lambda x.x \\
\mathsf{super\_reverse}_{\tau_1\ \mathsf{List}} &= \lambda[x_1,\ldots,x_n].[\mathsf{super\_reverse}_{\tau_1} x_n,\ldots,\mathsf{super\_reverse}_{\tau_1} x_1]
\end{aligned}$$

By interpreting the types, we obtain the ML implementation in Fig. 12.

Function flatten and function super_reverse can be used separately, but we cannot write an expression such as

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

to use them in combination, i.e., to reverse a list recursively and then flatten the result, because the encoding 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, i.e., $\mathsf{flatten} \circ \mathsf{super\_reverse} = \mathsf{super\_reverse} \circ \mathsf{flatten}$.)

This problem can be evaded in a *non-modular* fashion, if we know in advance *all* possible families $v, v' \dots$ of values that are indexed by the same family of (type) indices: we can simply tuple all the values together for the type interpretation. Every function f_{v_i}, then, is defined to project out the appropriate component from the type interpretation. Indeed, our previous program of type-directed partial evaluation (Fig. 6) illustrates such a tupling.

4. Value-independent type encoding

In this section, we develop two approaches to encoding types independently of the type-indexed families of values indexed by them. That is, we should be able to define the encodings $\langle\!|\tau|\!\rangle$ of a family F of types τ, so that given any F-indexed family v of values, a function f_v that satisfies Eq. (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 $v, v', \dots$ to share a family of type encodings, resulting in more modular programs with 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 characterized as an n-ary tuple $\vec{e} = (e_1, \dots, e_n)$ of the value constructions, and the value-independent type interpretation $\langle\!|\tau|\!\rangle$ maps this specification to the specific value v_τ.

$$\langle\!|\tau|\!\rangle \vec{e} = v_\tau \tag{5}$$

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

$$\begin{aligned} &\langle\!|c_i|\!\rangle(\langle\!|\tau_1|\!\rangle, \dots, \langle\!|\tau_m|\!\rangle)\vec{e} \\ &= \langle\!|c_i(\tau_1, \dots, \tau_m)|\!\rangle \vec{e} && (by\ (3)) \\ &= v_{c_i(\tau_1,\dots,\tau_m)} && (by\ (5)) \\ &= e_i(v_{\tau_1}, \dots, v_{\tau_m}) && (by\ (2)) \\ &= e_i(\langle\!|\tau_1|\!\rangle \vec{e}, \dots, \langle\!|\tau_m|\!\rangle \vec{e}) && (by\ (5)) \end{aligned}$$

By this derivation, we have

Theorem 18. *The value-independent encodings of type constructors*

$$\langle\!|c_i|\!\rangle = \lambda(x_1, \dots, x_m).\lambda\vec{e}.e_i(x_1\vec{e}, \dots, x_m\vec{e})$$

```
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
```

Fig. 13. An unsuccessful encoding of $F^{\mathsf{exp},\to}$ and TDPE.

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 impossible: we cannot abstract over a value of a polymorphic type, such as the e_i's.

For an example, let us try to encode types in the family $F^{\mathsf{exp},\to}$, and use them to program type-directed partial evaluation in ML (Fig. 13).

The definition of `reify` and `reflect` at higher types is as before and omitted here for brevity. This program is not typeable in ML, 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 $\langle\!|\tau|\!\rangle$ constructed using `Base` and `-->`, is somewhat involved:

$$\begin{array}{lll} \langle\!|\tau|\!\rangle : & \forall \mathsf{obj} : * \to *. & \\ & \forall \mathsf{base\_type} : *. & \\ & (\mathsf{base\_typeobj} \quad \times & (* \ \mathsf{base\_v} \ *) \\ & \ (\forall \alpha : *, \beta : *.(\alpha \ \mathsf{obj}) \to (\beta \ \mathsf{obj}) \to ((\alpha \to \beta) \ \mathsf{obj}))) \to & (* \ \mathsf{func\_v} \ *) \\ & \tau \ \mathsf{obj} & \end{array}$$

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 [30] or packaging of these variables using polymorphic datatype components [21]. Moreover, *higher-order* *polymorphism* [20] 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 [12,32].

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 Fig. 13 into functors, we also need to use higher-order functors which allows functors to have functor arguments or results. Such higher-order modules are supported by Standard ML of New Jersey [3], which extends Standard ML with higher-order functors [38]. Figs. 14 and 15 gives a program for type-directed partial evaluation using higher-order functors.

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 Section 4.1.

It is, unfortunately, somewhat cumbersome to use such functor-based encodings. The following example illustrates just how much code it takes partially evaluate (residualize) the function $\lambda x.x$ with type (base $\to$ base) $\to$ (base $\to$ 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, the syntactic overhead in using the type system makes the approach somewhat tedious to be used for programming in ML. On the other hand, a systematic application of this approach has been used in realizing self-application of TDPE in ML [13].

4.3. Embedding/projection functions as type interpretation

The second approach to value-independent type encodings is, perhaps 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.

```
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
```

Fig. 14. Encoding $F^{\mathsf{exp},\rightarrow}$ using higher-order functors.

```
structure TDPE: IndValue =
  struct
    type 'a obj = ('a -> exp) * (exp -> 'a)
    type base_type = exp
    val Base = (fn v => v, fn e => e)
    fun Arrow (reif1, refl1) (reif2, refl2)
      = (fn (f: 'a -> 'b) =>
            let val x = Gensym.new()
            in  LAM(x, reif2 (f (refl1 (VAR x))))
            end,
         fn (e: exp) =>
            fn (v: 'a) => refl2 (APP(e, reif1 v)))
    end
```

Fig. 15. Type-directed partial evaluation using the functor-based encoding.

```
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))
```

Fig. 16. Tagged version of functions `flatten` and `super_reverse`.

First of all, for a type-indexed value v whose type index τ appears at the input positions, 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 shown in Fig. 16.

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 also 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.

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

$$\begin{aligned} \langle\!|\tau|\!\rangle &= \langle emb_\tau, proj_\tau, tE_\tau\rangle \\ emb_\tau &: \tau \to U \quad \text{(embedding function)} \\ proj_\tau &: U \to \tau \quad \text{(projection function)} \\ tE_\tau &: typeExp \ \text{(type expression)} \end{aligned} \tag{6}$$

Finally, we use the embedding and projection functions as basic coercions to convert a value based on a universal datatype to the type of the specific value v_τ.

We continue to illustrate the approach in Section 4.3.1, and then formally present the general approach in Section 4.3.2.

4.3.1. Examples

Taking the family $F^{\mathsf{int,list}}$ 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 define the functions `flatten` and `super_reverse` 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 to share 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`.

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

infixr 5 -->

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)
```

Fig. 17. Embedding/projection-based encoding for TDPE.

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 need to 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 in Fig. 17.

Recall that the definition of the functions `reifyTg` and `reflectTg` will cause matching-inexhaustive compilation warnings, and invoking them might cause run-time exceptions. In contrast, function `reify` is safe 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 example, to implement a form of type specialization [7], where the partial evaluator and the projection function are type-indexed by the same family of types.

4.3.2. Universality

In this section, we argue that the above approach based on embedding and projection functions indeed provides a value-independent encoding for a majority of type constructors. The idea is that the embedding/projection encoding forms a *universal* type-indexed family of values, in that any other family of values indexed by the same family can be constructed from this particular family.

We assume the following conditions about the types:

1. All the type constructions c_i (in Eqs. (2)) build a type only from component types covariantly and/or contravariantly. The constructed type can use the same component type both covariantly and contravariantly at its different occurrences, as in the example of type-directed partial evaluation. This condition rules out, for example, the reference type constructor.
2. The type Q is covariant or contravariant in every occurrence of the type variable τ.

Universal Type and Embedding/projection Pairs. We first define an ML datatype U to distinctively represent values of different types in the type family F. This is done by tagging all the branches of type constructions c_i. Without loss of generality, we assume no other type variable freely occurring in the type constructions c_i. Type variables can be dealt with by parameterization over type U.

$$\begin{aligned} \textbf{datatype } U &= tag_{c_1} \textbf{ of } c_1(\overbrace{U,\ldots,U}^{m_1}) \\ &\ \ \vdots \\ &\ \ \mid tag_{c_n} \textbf{ of } c_n(\underbrace{U,\ldots,U}_{m_n}) \end{aligned}$$

We also define a datatype $typeExpU$ to describe the structure of a particular type in the type family F:

$$\begin{aligned} \textbf{datatype } typeExpU &= tEc_1 \textbf{ of } (typeExpU)^{m_1} \\ &\ \ \vdots \\ &\ \ \mid tEc_n \textbf{ of } (typeExpU)^{m_n} \end{aligned}$$

Condition 1 ensures the existence of the embedding/projection pairs between every inductively constructed type index $\tau \in F$ and the universal type U. Such pairs witness the apparent isomorphisms between the values of type τ, denoted as $Val(\tau)$, and the corresponding set of tagged values of type U, denoted as $UVal(\tau)$.

To see that such pairs always exist, consider the category **Type** whose objects are types and whose morphisms are coercions (see Section 2.1 of Henglein's article [16]). Every type construction c_i is interpreted as a multi-functor that is either covariant or contravariant in each of its argument:

$$C_i : \mathbf{Type}^{p_1} \times \mathbf{Type}^{p_2} \times \cdots \times \mathbf{Type}^{p_{m_i}} \to \mathbf{Type}$$

$$\begin{array}{rcll}
c(\tau_1,\dots,\tau_m) & = & \tau_j \qquad (j=1,\dots,m) & (\mathit{Type\ argument}) \\
& | & \beta & (\mathit{Type\ variable}) \\
& | & A & (\mathit{Atomic\ types}) \\
& | & c_1(\tau_1,\dots,\tau_m) \to c_2(\tau_1,\dots,\tau_m) & (\mathit{Function}) \\
& | & c_1(\tau_1,\dots,\tau_m)\ \mathsf{list} & (\mathit{List}) \\
& | & c_1(\tau_1,\dots,\tau_m) * \cdots * c_l(\tau_1,\dots,\tau_m) & (\mathit{Tuple\ of\ length\ } l)
\end{array}$$

Fig. 18. Formation of a type construction c.

Here p_j's are the polarities of the arguments: if the type construction c_i is covariant in its jth argument, then $p_j = +$ and $\mathbf{Type}^{p_i} = \mathbf{Type}$; if it is contravariant in its jth argument, then $p_j = -$ and $\mathbf{Type}^{p_j} = \mathbf{Type}^{op}$. Given pairs of embedding $emb_{\tau_j} : \tau_j \rightsquigarrow U$ and projection $proj_{\tau_j} : U \rightsquigarrow \tau_j$ at the component type, we can apply the functor C_i to induce another pair of coercions for the constructed type $\tau = c_i(\tau_1,\dots,\tau_{m_i})$.

$$\varepsilon_\tau = C_i(emb_{\tau_1}^{p_1},\dots,emb_{\tau_{m_i}}^{p_{m_i}}) : \tau \rightsquigarrow c_i(\overbrace{U,\dots,U}^{m_i})$$
$$\pi_\tau = C_i(proj_{\tau_1}^{p_1},\dots,proj_{\tau_{m_i}}^{p_{m_i}}) : c_i(\underbrace{U,\dots,U}_{m_i}) \rightsquigarrow \tau$$

Here, $emb_\tau^+ = emb_\tau$, $emb_\tau^- = proj_\tau$, $proj_\tau^+ = proj_\tau$ and $proj_\tau^- = emb_\tau$ for all types τ. Composing coercions ε_τ and π_τ with the tagging and untagging operations for the tag tag_{c_i} respectively gives the embedding/projection pair at the type τ. By structural induction, all types τ in the type family T are equipped with the embedding/projection pairs.

Let us make concrete the above construction for the type constructions in ML. For Condition 1, we assume that every type construction c_i is inductively generated using the ML type constructions in Fig. 18.

Lemma 19. *Let c be a m-ary type construction generated by the grammar in Fig.* 18. *Given types τ_j (for $j = 1,\dots,m$), and for each type a pair of embedding $emb_j : \tau_j \rightsquigarrow U$ and projection $proj_j : U \rightsquigarrow \tau_j$ (defined on $UVal(\tau_j)$), which are inverse to each other between $Val(\tau_j)$ and $UVal(\tau_j)$, one can induce a pair of functions $\varepsilon_\tau : \tau \rightsquigarrow c(U,\dots,U)$ and $\pi_\tau : c(U,\dots,U) \rightsquigarrow \tau$ where $\tau = c(\tau_1,\dots,\tau_m)$, which are inverse to each other between the set $Val(\tau)$ and $UVal(\tau)$.*

Proof. By structural induction on type $\tau = c(\tau_1,\dots,\tau_m)$.

$c(\tau_1,\dots,\tau_m) = \tau_j$. Define $\varepsilon_\tau = emb_j : \tau_j \rightsquigarrow U$ and $\pi_\tau = proj_j : U \rightsquigarrow \tau_j$. They are inverse to each other by the condition of the lemma.

$c(\tau_1,\dots,\tau_m) = \beta$. Here β is a freely occurring type variable, thus $c(U,\dots,U) = \beta$. Define $\varepsilon_\tau = \pi_\tau = \lambda x.x : \beta \rightsquigarrow \beta$, which are inverse to each other.

$c(\tau_1,\dots,\tau_m) = A$. Since $c(U,\dots,U) = A$, setting $\varepsilon_\tau = \pi_\tau = \lambda x.x : A \rightsquigarrow A$ gives the pair.

$c(\tau_1,\dots,\tau_m) = c_1(\tau_1,\dots,\tau_m) \to c_2(\tau_1,\dots,\tau_m)$. By the induction hypotheses, we have $\varepsilon_{c_1(\tau_1,\dots,\tau_m)} : c_1(\tau_1,\dots,\tau_m) \rightsquigarrow c_1(U,\dots,U)$ and its inverse $\pi_{c_1(\tau_1,\dots,\tau_m)} : c_1(U,\dots,U) \rightsquigarrow c_1(\tau_1,\dots,\tau_m)$, together with $\varepsilon_{c_2(\tau_1,\dots,\tau_m)} : c_2(\tau_1,\dots,\tau_m) \rightsquigarrow c_2(U,\dots,U)$ and its inverse $\pi_{c_2(\tau_1,\dots,\tau_m)} : c_2(U,\dots,U) \rightsquigarrow c_2(\tau_1,\dots,\tau_m)$. Now, define

$$\varepsilon_\tau f = \varepsilon_{c_2(\tau_1,\dots,\tau_m)} \circ f \circ \pi_{c_1(\tau_1,\dots,\tau_m)}$$
$$\pi_\tau f = \pi_{c_2(\tau_1,\dots,\tau_m)} \circ f \circ \varepsilon_{c_1(\tau_1,\dots,\tau_m)}$$

It is easy to verify that these two functions have the required types, and they are inverse to each other.

$c(\tau_1,\dots,\tau_m) = c_1(\tau_1,\dots,\tau_m)$ list. By the induction hypothesis, we have the embedding $\varepsilon_{c_1(\tau_1,\dots,\tau_m)} : c_1(\tau_1,\dots,\tau_m) \rightsquigarrow c_1(U,\dots,U)$ and as its inverse the projection $\pi_{c_1(\tau_1,\dots,\tau_m)} : c_1(U,\dots,U) \rightsquigarrow c_1(\tau_1,\dots,\tau_m)$. Now, let

$$\varepsilon_\tau L = \mathbf{map}\ L\ \varepsilon_{c_1(\tau_1,\dots,\tau_m)}$$
$$\pi_\tau L = \mathbf{map}\ L\ \pi_{c_1(\tau_1,\dots,\tau_m)}$$

It is easy to verify that these two functions have the required types, and they are inverse to each other.

$c(\tau_1,\dots,\tau_m) = c_1(\tau_1,\dots,\tau_m) * \cdots * c_l(\tau_1,\dots,\tau_m)$. By induction hypothesis, we have embeddings $\varepsilon_{c_i(\tau_1,\dots,\tau_m)}$ and projections $\pi_{c_i(\tau_1,\dots,\tau_m)}$, which are pairwise inverse. Now, let

$$\varepsilon_\tau(x_1,\dots,x_l) = (\varepsilon_{c_1(\tau_1,\dots,\tau_m)} x_1,\dots,\varepsilon_{c_l(\tau_1,\dots,\tau_m)} x_l)$$
$$\pi_\tau(x_1,\dots,x_l) = (\pi_{c_1(\tau_1,\dots,\tau_m)} x_1,\dots,\pi_{c_l(\tau_1,\dots,\tau_m)} x_l)$$

It is easy to verify that these two functions have the required types, and they are inverse to each other. □

Theorem 20. *For all types $\tau \in F$, there is a pair of embedding $emb_\tau : \tau \rightsquigarrow U$ and projection $proj_\tau : U \rightsquigarrow \tau$ which are inverse to each other between $Val(\tau)$ and $UVal(\tau)$.*

Proof. By induction on type τ.

$\tau = c_i(\tau_1,\dots,\tau_{m_i})$. The induction hypotheses for every type τ_j where $1 \leqslant j \leqslant m_i$ says that $emb_{\tau_j} : \tau_j \rightsquigarrow U$ and $proj_{\tau_j} : U \rightsquigarrow \tau_j$ exist and are inverse to each other. By Lemma 19, we can induce a pair of inverse functions $\varepsilon_\tau : \tau \rightsquigarrow c_i(U,\dots,U)$ and $\pi_\tau : c_i(U,\dots,U) \rightsquigarrow \tau$. Define

$$\begin{aligned} emb_\tau &: \tau \rightsquigarrow U \\ emb_\tau(x) &= tag_{c_i}(\varepsilon_\tau(x)) \\ proj_\tau &: U \rightsquigarrow \tau \\ proj_\tau(tag_{c_i}(x)) &= \pi_\tau(x) \end{aligned}$$

It is easy to verify that the two functions are inverse to each other. □

The proof above essentially gives an algorithm for computing the embedding/projection pair for every type τ. Notice this algorithm itself is specified in a type-

$Q = \alpha$	(*indexed type* τ)
$\mid A$	(*Built-in atomic types*)
$\mid Q_1 \to Q_2$	(*Function*)
$\mid Q_1$ list	(*List*)
$\mid Q_1 \times \cdots \times Q_l$	(*Tuple of length* l)

Fig. 19. Formation of the type Q of a type-indexed value.

indexed form, that the pair for a constructed type is computed from the pairs for its component types; therefore, we can always use the ad hoc approach to program the type interpretation in the form of Eq. (6).

Embedding/projection for the result type. The type encoding $\langle\!|\tau|\!\rangle$ of a type $\tau \in F$ gives the pairs of embedding and projection between this type and the universal type U. Now, for a type-indexed family v of values with the type scheme $\forall\alpha \in F.Q$, we need to compute the embedding and the projection between type $Q\{\tau/\alpha\}$ and type $Q\{U/\alpha\}$ for any given type $\tau \in F$ from its type encoding $\langle\!|\tau|\!\rangle$. This makes it possible to first compute the universal version of the value v_τ, which is of type $Q\{U/\alpha\}$, and then project it to the specific type $Q\{\tau'/\alpha\}$. Condition 2 ensures the existence of these embedding/projection pairs; in fact, the pair can be constructed using the multi-functor $Q\{-/\alpha\}$.

Like before, Condition 2 can be made concrete in terms of ML type constructions. We assume that the type Q is inductively generated using the ML type constructions in Fig. 19.

Theorem 21. *Let Q be a type with a free type variable α, generated by the grammar in Fig.* 19. *Given a type τ and the pair of inverse functions $emb_\tau : \tau \rightsquigarrow U$ and $proj_\tau : U \rightsquigarrow \tau$, one can induce a pair of functions $e_\tau^Q : Q\{\tau/\alpha\} \rightsquigarrow Q\{U/\alpha\}$ and $p_\tau^Q : Q\{U/\alpha\} \rightsquigarrow Q\{\tau/\alpha\}$ which are inverse to each other.*

Proof. By induction on type Q. The proof is similar to the previous ones; for brevity, here we simply gives the construction of e_τ^Q and p_τ^Q. It is straightforward to verify that each of them is a pair of inverse functions.
$Q = \tau$. Define $e_{\tau'}^Q = emb_{\tau'}$ and $p_{\tau'}^Q = proj_{\tau'}$.
$Q = A$. Define $e_{\tau'}^Q = p_{\tau'}^Q = \lambda x.x : A \to A$.
$Q = Q_1 \to Q_2$. Define $e_{\tau'}^Q\ f = e_{\tau'}^{Q_1} \circ f \circ p_{\tau'}^{Q_2}$ and $p_{\tau'}^Q\ f = p_{\tau'}^{Q_1} \circ f \circ e_{\tau'}^{Q_2}$.
$Q = Q_1$ list. Define $e_{\tau'}^Q\ L = \textbf{map}\ L\ e_{\tau'}^{Q_1}$ and $p_{\tau'}^Q\ L = \textbf{map}\ L\ p_{\tau'}^{Q_1}$.
$Q = Q_1 * \cdots * Q_l$. Define $e_{\tau'}^Q\ (x_1,\ldots,x_l) = (e_{\tau'}^{Q_1}\ x_1,\ldots,e_{\tau'}^{Q_l}\ x_l)$
and $p_{\tau'}^Q\ (x_1,\ldots,x_l) = (p_{\tau'}^{Q_1}\ x_1,\ldots,p_{\tau'}^{Q_l}\ x_l)$. $\square$

In fact, the proof itself provides an algorithm for computing e_τ^Q and p_τ^Q for a fixed type Q from the embedding/projection pair of type τ, which is included in the "universal" encoding encoding $\langle\!|\tau|\!\rangle$.

Type-indexed values from universal values. Now, we can write a function $f_v^U : type\ ExpU \rightarrow Q\{U/\alpha\}$, the universal-datatype version of the type-indexed value v, such that for each type $\tau \in T$, the value $f_v^U(tE_\tau)$ is equivalent to the verbatim value v_τ embedded into the universal type $Q\{U/\alpha\}$.

$$f_v^U(tE_\tau) = e_\tau^Q(v_\tau)$$

It is then sufficient to define the function f_v as

$$f_v\langle emb_\tau, proj_\tau, tE_\tau\rangle = p_\tau^Q(f_v^U(tE_\tau)) \qquad (7)$$

where p_τ^Q is constructed from emb_τ and $proj_\tau$ by Theorem 21. Function f_v and the universal encoding $\langle\!\langle \cdot \rangle\!\rangle$ do implement the type-indexed family v (Definition 8); this follows from the fact that $p_\tau^Q \circ e_\tau^Q$ is the identity function, also by Theorem 21.

Function f_v^U can be induced from the specification in the form of Eq. (2) as follows:

$$f_v^U(tEc_1(tE_{\tau_1},\ldots,tE_{\tau_{m_1}})) = e_1^U(f_v^U(tE_{\tau_1}),\ldots,f_v^U(tE_{\tau_{m_1}}))$$
$$\vdots$$

where $e_i^U : (Q\{U/\alpha\})^{m_i} \rightarrow Q\{U/\alpha\}$ is a properly instrumented version of $e_i : \forall\alpha_1,\ldots,\alpha_{m_i}.(Q\{\alpha_1/\alpha\} \times \cdots \times Q\{\alpha_{m_i}/\alpha\}) \rightarrow Q\{c_i(\alpha_1,\ldots,\alpha_{m_i})/\alpha\}$ by adding tagging and untagging operations. This process can be purely mechanical: instantiating all the type variables α_i to type U, and then applying a coercion function induced from the data constructor tag_{c_i} of type U.

Note that the case analysis on the types in Eq. (2) has been turned into case analysis on their value representations, which are of the uniform type *typeExpU*. This way, the program f_v^U fits into the Hindley–Milner type system.

The following theorem summarizes the approach based on the above construction.

Theorem 22. *Encoding types as embedding/projection functions gives a value-indepen dent type encoding for a type family F. Every F-indexed family v of values is implemented by the function f_v defined in Eq.* (7) *and the type encoding.*

4.3.3. Comments

The 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. While internal tagging incurs a syntactic overhead proportional to the size of the term, external tagging incurs a syntactic overhead proportional to only the size of the type, which is usually much smaller.

```
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))
```

Fig. 20. Type-safe coercion function.

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 [23], 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 index. Fig. 20 presents an example: an ML function that performs subtype coercion [28]. 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 returns 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.

The example below builds a subtype coercion `C`: $\mathsf{str} \to \mathsf{str} \rightsquigarrow \mathsf{int} \to \mathsf{str}$ from a base coercion $\mathsf{int} \rightsquigarrow \mathsf{str}$, so that, e.g., the expression `C (fn x => x ^ x) 123` evaluates to `"123 123"`.

```
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 [25,39].

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* [35] 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 [10,37] 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 the ease of use.

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 [35] 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 provide extra flexibility, as shown in applications such as meta-programming and high-level data/code transfer in distributed programming.

5.1.2. Intensional type analysis

Intensional type analysis [15] directly supports programming with type-indexed families of 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 [14] also makes it easy to program type-indexed family of 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 Sections 5.1.1–5.1.3) 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 often immediately usable 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 Fig. 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* [9]).

Apart from raising the interest of programming type-indexed families in ML,[5] type-directed partial evaluation shares an interesting common pattern with the embedding/projection-based approach: both use types as external tags (see Section 4.3.3). Loosely speaking, one external type tag in type-directed partial evaluation replaces pervasive binding-time annotations in the pre-processed 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 family of values that captures functions that take type arguments. We have formulated type-encoding-based implementations of type-indexed values in terms of type interpretations. According to this formulation, we

[5] 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 [33]. Since then, Kristoffer Rose has programmed it in Haskell, using type classes [34].

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 family of values as the type interpretation. It gives value-dependent type encodings, not sharable by different families 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. Though simple in theory, it might be not very practical because it requires first-class and higher-order polymorphism. But with some efforts, such advanced forms of polymorphism are embeddable in some dialects of ML, such as SML/NJ and Moscow ML.
- The third approach applies the first approach to tune a usual tagged-value-based, type-unsafe approach to give a type-safe and yet syntactically convenient approach, by interpreting types as 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 without going beyond the Hindley–Milner type system.

On one hand, we showed in this article that with appropriate type encodings, type-indexed values can be programmed in ML-like languages; on the other hand, our investigation also feeds back 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.

Acknowledgements

I especially thank Olivier Danvy for his challenge on programming TDPE in ML, 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 TCS reviewers 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.

Part of this work was carried out during a visit to the BRICS PhD School[6] at the University of Aarhus in the fall of 1997 and during a visit to the Department of Computer Science at the University of Copenhagen in January and February 1998. The work was partly supported by NSF CCR-9970909.

[6] Basic Research in Computer Science, Centre of the Danish National Research Foundation. Home page: http://www.brics.dk

References

[1] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin, Dynamic typing in a statically typed language, ACM Trans. Programming Languages and Systems 13 (2) (1991) 237–268.

[2] M. Abadi, L. Cardelli, B. Pierce, D. Rémy, Dynamic typing in polymorphic languages, J. Funct. Programming 5 (1) (1995) 111–130.

[3] A.W. Appel, D.B. MacQueen, Standard ML of New Jersey, in: J. Maluszyński, M. Wirsing (Eds.), Third Internat. Symp. on Programming Language Implementation and Logic Programming, Lecture Notes in Computer Science, Vol. 528 Passau, Germany, August 1991, Springer, Berlin, pp. 1–13.

[4] O. Danvy, Type-directed partial evaluation, in: G.L. Steele (Ed.), Proc. 23rd Annu. ACM Symp. on Principles of Programming Languages, St. Petersburg, Beach, FL, January 1996, pp. 242–257.

[5] O. Danvy, Private Communication, 1998.

[6] O. Danvy, Functional unparsing, J. Funct. Programming 8 (1998) 621–625.

[7] O. Danvy, A simple solution to type specialization, in: K.G. Larsen, S. Skyum, G. Winskel (Eds.), Proc. 25th Internat. Colloq. on Automata, Languages, and Programming, Lecture Notes in Computer Science, Vol. 1443, Springer, Berlin, 1998, pp. 908–917.

[8] O. Danvy, Type-directed partial evaluation, in: J. Hatcliff, T.E. Mogensen, P. Thiemann (Eds.), Partial Evaluation—Practice and Theory; Proc. 1998 DIKU Summer School, Lecture Notes in Computer Science, Vol. 1706, Copenhagen, Denmark, July 1998, Springer, Berlin, pp. 367–411.

[9] O. Danvy, P. Dybjer (Eds.), Preliminary Proc. 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98, Göteborg, Sweden, May 8–9, 1998, BRICS Notes Series, Vol. NS-98-1, BRICS, Department of Computer Science, University of Aarhus, May 1998.

[10] R. Davies, F. Pfenning, A modal analysis of staged computation, in: G.L. Steele (Ed.), Proc. 23rd Annu. ACM Symp. on Principles of Programming Languages, St. Petersburg, Beach, FL, January 1996, pp. 258–283; Extended version, J. ACM 48 (2001) 555–604.

[11] A. Filinski, A semantic account of type-directed partial evaluation, in: G. Nadathur (Ed.), Internat. Conf. Principles and Practice of Declarative Programming, Lecture Notes in Computer Science, Vol. 1702, Paris, France, September 1999, pp. 378–395.

[12] J.-Y. Girard, The system F of variable types, fifteen years later, Theoret. Comput. Sci. 45 (2) (1986) 159–192.

[13] B. Grobauer, Z. Yang, The second Futamura projection for type-directed partial evaluation, in: J.L. Lawall (Ed.), Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics—Based Program Manipulation, SIGPLAN Notices, Vol. 34 (11), Boston, MA, November 2000, ACM Press, pp. 22–32; Extended version, J. Higher-Order Symbolic Comput. 14 (2/3) (2001).

[14] C. Hall, K. Hammond, S. Peyton-Jones, P. Wadler, Type classes in Haskell, ACM Trans. Program. Languages Systems 18 (2) (1996) 109–138.

[15] R. Harper, G. Morrisett, Compiling polymorphism using intensional type analysis, in: P. Lee (Ed.), Proc. 22nd Annu. ACM Symp. on Principles of Programming Languages, San Francisco, California, January 1995, pp. 130–141.

[16] F. Henglein, Dynamic typing: syntax and proof theory, Sci. Comput. Program. 22 (3) (1994) 197–230.

[17] J.R. Hindley, The principal type-scheme of an object in combinatory logic, Trans. Amer. Math. Soc. 146 (1969) 29–60.

[18] J. Hughes, The design of a pretty-printing library, in: J. Jeuring, E. Meijer (Eds.), Advanced Functional Programming, Lecture Notes in Computer Science, Vol. 925, Springer, Berlin, 1995, pp. 53–96.

[19] P. Jansson, J. Jeuring, PolyP—a polytypic programming language extension, in: N.D. Jones (Ed.), Proc. 24th Annu. ACM Symp. on Principles of Programming Languages, Paris, France, January 1997, pp. 470–482.

[20] M.P. Jones, A system of constructor classes: overloading and implicit higher-order polymorphism, J. Funct. Program. 5 (1) (1995) 1–35, An earlier version appeared in FPCA '93.

[21] M.P. Jones, First-class polymorphism with type inference, in: N.D. Jones (Ed.), Proc. 24th Annu. ACM Symp. on Principles of Programming Languages, Paris, France, January 1997, pp. 483–496.

[22] N.D. Jones (Ed.), Proc. 24th Annu. ACM Symp. on Principles of Programming Languages, Paris, France, January 1997.

[23] N.D. Jones, C.K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice-Hall International, Englewood Cliffs, NJ, 1993, Available online at http://www.dina.kvl.dk/∼sestoft/pebook/pebook.html.

[24] R. Kelsey, W. Clinger, J. Rees (Eds.), Revised^5 Report on the algorithmic language Scheme, Higher-Order and Symbol. Comput. 11 (1) (1998) 7–105; Also appears in ACM SIGPLAN Notices 33 (9) (1998); Available online at http://www.brics.dk/∼hosc/11-1/.

[25] A. Kennedy, Relational parametricity and units of measure, in: N.D. Jones (Ed.), Proc. 23rd Annu. ACM Symp. on Principles of Programming Languages, Paris, France, January 1997, pp. 442–455.

[26] R. Milner, A theory of type polymorphism in programming, J. Comput. System Sci. 17 (1978) 348–375.

[27] R. Milner, M. Tofte, R. Harper, D. MacQueen, The Definition of Standard ML (Revised), The MIT Press, Cambridge, MA, 1997.

[28] J.C. Mitchell, Coercion and type inference, in: K. Kennedy (Ed.), Proc. 11th Annu. ACM Symp. on Principles of Programming Languages, Salt Lake City, Utah, January 1984, pp. 175–185.

[29] E. Moggi, Computational lambda-calculus and monads, in: R. Parikh (Ed.), Proc. 4th Annu. IEEE Symp. on Logic in Computer Science, Pacific Grove, California, June 1989, pp. 14–23.

[30] M. Odersky, K. Läufer, Putting type annotations to work, in: G.L. Steele (Ed.), Proc. 23rd Annu. ACM Symp. on Principles of Programming Languages, St. Petersburg, Beach, FL, January 1996, pp. 54–67.

[31] J. Peterson, K. 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.

[32] J.C. Reynolds, Towards a theory of type structure, in: Programming Symp., Lecture Notes in Computer Science, Vol. 19, Springer, Berlin, Paris, France, April 1974, pp. 408–425.

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

[34] K. Rose, Type-directed partial evaluation in a pure functional language, in: O. Danvy, P. Dybjer (Eds.), Preliminary Proc. 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98, Göteborg, Sweden, May 8–9, 1998, BRICS Notes Series, Vol. NS-98-1, BRICS, Department of Computer Science, University of Aarhus, May 1998.

[35] M. Shields, T. Sheard, S.P. Jones, Dynamic typing as staged type inference, in: L. Cardelli (Ed.), Proc. 25th Annu. ACM Symp. on Principles of Programming Languages, San Diego, California, January 1998, pp. 289–302.

[36] G.L. Steele (Ed.), Proc. 23rd Annu. ACM Symp. on Principles of Programming Languages, St. Petersburg, Beach, FL, January 1996.

[37] W. Taha, T. Sheard, Multi-stage programming, in: M. Tofte (Ed.), Proc. 1997 ACM SIGPLAN Internat. Conf. on Functional Programming, Amsterdam, The Netherlands, June 1997, pp. 321–321.

[38] M. Tofte, Principal signatures for higher-order program modules, J. Funct. Program. 4 (3) (1994) 285–335.

[39] R. 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.