GADTs for the OCaml Masses

(Functional Pearl)

Yitzhak Mandelbaum
AT&T Labs - Research
yitzhak@research.att.com

Aaron Stump
University of Iowa
astump@acm.org

Abstract

GADTs are a modest extension to datatypes which support static typing of a larger class of programs than those possible with standard datatypes. However, programming languages like OCAML lack native support for GADTs, which has impeded their adoption. In this pearl, we investigate a flexible encoding of GADTs in the second-order polymorphic lambda calculus, and demonstrate how to implement this encoding to capture the power of GADTs in OCAML.

1. Introduction

GADTs, or Generalized Abstract Datatypes, have recently gained popularity within the functional programming community as a modest extension to the concept of datatypes that permits static typing of a number of useful paradigms which were previously thought to require individual, and often complex, language extensions. Common examples include generic programming, indexed lists and staged computation, although many more have been illustrated in the literature. In general, GADTs allow the user to statically track *more* information about their datatypes. This extended static checking allows both for more careful checking of existing paradigms (e.g. indexed lists) and static checking of previously uncheckable paradigms (e.g. typed `printf`).

Unfortunately, because of the need to extend a language in order to support GADTs, they have yet to be implemented in many languages. A traditional way of dealing with such a problem is to find an encoding of the desired feature in terms of more primitive and, hopefully, available language features. Indeed, a number of other language concepts which can be captured with GADTs have been independently shown to be encodeable; most notably, intensional polymorphism (Weirich 2001, 2006), generic programming (Yang 1998; Hinze 2004; Fernández et al. 2008), and tagless, staged interpreters (Carette et al. 2007).

In this paper, we generalize those results by describing a straightforward recipe for encoding GADTs in System F extended with recursion and second-order, impredicative polymorphism (alternatively known as higher-rank, first-class polymorphism)[1]. Moreover, we choose an encoding that is subtly different than those chosen in most previous works. Generally, the essence of each encoding is a Church-style encoding of some GADT. In contrast, we use Dana Scott's lesser-known 1963 encoding of datatypes (Curry et al. 1972, page 504). We choose this encoding because it supports a more natural form of programming in some of the more complex examples. In Section 2, we give a more detailed comparison of Church and Scott encodings.

Finding an appropriate encoding of GADTs is, in essence, a two-part problem. The first part is to find an appropriate *computational* encoding of GADTs. The second challenge is to find an appropriate encoding of GADT *types*. Of course, the two challenges are related as the type must suit the computational representation and vice-versa. Indeed, we will show how we can start from either direction (the types or the terms) and still arrive in the same place. Specifically, we will start with Scott-encoded data and give it appropriate types, and then we will sketch how to translate the GADTs in Xi et al.'s $\lambda_{2,G\mu}$ into our encoding and arrive at the same types and terms.

We aim to accomplish two things with this pearl: first, to provide the reader with a clear list of sufficient language features for capturing the power of GADTs. We hope these "sufficient conditions" will be useful to end-user and language implementer alike. Second, we describe a concrete realization of the encoding in OCAML, which weaves together the core language, the module system, type abstraction and a safe use of unsafe cast.

We will begin in Section 2 by discussing the Church and Scott encodings of datatypes in the lambda calculus. We choose to work with the Scott encoding, and, in Section 3, we show how to effectively translate a polymorphic, recursive datatype into a corresponding (second-order) polymorphic lambda calculus type. Next, we discuss the restricted class of datatypes represented by GADTs and derive an appropriate type for them as well. Then, in Section 5, we turn around and approach the problem from the other direction. We start with Xi et al. (2003)'s calculus of guarded recursive datatype constructors (GADTs by another name), specifically the specification of the GADT types. We then show how, using the familiar trick of encoding of existential types based on their elimination rule, we can arrive at the same type encoding as earlier. The Scott encoding then falls out as obvious inhabitant of that type.

In Section 6, encoding in hand, we tackle the problem of realizing the encoding in OCAML, whose core language lacks support for one of the crucial ingredients of the encoding. We then demonstrate use of the encoded GADTs with a number of examples in Section 7. We discuss related work and suggestions for further reading in Section 8 and conclude in secrefsec:conclusion.

2. Datatype Encodings

Inductive datatypes can be represented in untyped lambda calculus using either the Church or the Scott encoding (this section is

[1] This system is sometimes referred to as F_3.

ICFP '09 August-September 2009, Edinburgh, Scottland

based on a longer discussion in (Stump 2008)). The Church encoding represents the n constructors of an inductive datatype as follows (Church 1941, Chapter 3). For a simple example, consider the following datatype of unary natural numbers:

```
type nat = Z | S of nat
```

We Church-encode numerals built with these constructors as follows:

$$\begin{array}{lcl} 0 & \equiv & \lambda s\, z.\, z \\ 1 & \equiv & \lambda s\, z.\, s\, z \\ 2 & \equiv & \lambda s\, z.\, s\, (s\, z) \\ 3 & \equiv & \lambda s\, z.\, s\, (s\, (s\, z)) \end{array}$$

The numeral N is encoded as a λ-abstraction that takes two terms, s and z, and applies iteratively applies s N times to z. Each numeral can be thought of as its own interpretation function: given interpretations of the constructors, the numeral will compute its interpretation. For example, given a suitable definition for S (just below), we can add two numerals n and m with the term $(n\ S\ m)$. This will interpret iteratively apply S n times to m, which indeed adds n and m. Suitable definitions of S and Z are:

$$\begin{array}{lcl} S & \equiv & \lambda x_1.\, \lambda s\, z.\, s\, (x_1\, s\, z) \\ Z & \equiv & \lambda s\, z.\, z \end{array}$$

More generally speaking, suppose we have a datatype with n constructors, where the i'th constructor C_i has arity $a(i)$. We encode C_i by the following lambda term, where we write $\bar{c}$ for $c_1\ \ldots\ c_n$:

$$\lambda x_1\ \ldots\ x_{a(i)}.\, \lambda c_1\ \ldots\ c_n.\, c_i\ (x_1\ \bar{c})\ \ldots (x_{a(i)}\ \bar{c})$$

This term takes in the $a(i)$ arguments to constructor C_i as $x_1, \ldots,$ $x_{a(i)}$. It then returns a lambda abstraction (call it M) which accepts n arguments $c_1, \ldots, c_n$, one for each constructor of the datatype. In contrast, the Scott encoding encodes constructor C_i by the following lambda term (Curry et al. 1972, page 504):

$$\lambda x_1\ \ldots\ x_{a(i)}.\, \lambda c_1\ \ldots\ c_n.\, c_i\, x_1\ \ldots\ x_{a(i)}$$

The crucial difference between this term and the corresponding term from the Church encoding is that here, the inputs $x_1, \ldots, x_{a(i)}$ are not interpreted using the constructor interpretations. For the natural number datatype, the definition specializes as follows:

$$\begin{array}{lcl} S & \equiv & \lambda x_1.\, \lambda s\, z.\, s\, x_1 \\ Z & \equiv & \lambda s\, z.\, z \end{array}$$

With this encoding, we may obtain the first few numerals by call-by-value reduction using S and Z. We here write $\downarrow_{cbv}$ for joinability using a call-by-value operational semantics:

$$\begin{array}{lclcl} 0 & \equiv & \lambda s\, z.\, z & \downarrow_{cbv} & Z \\ 1 & \equiv & \lambda s\, z.\, s\, 0 & \downarrow_{cbv} & S\, Z \\ 2 & \equiv & \lambda s\, z.\, s\, 1 & \downarrow_{cbv} & S\, (S\, Z) \\ 3 & \equiv & \lambda s\, z.\, s\, 2 & \downarrow_{cbv} & S\, (S\, (S\, Z)) \end{array}$$

We finish with two more examples:

```
type data = Num of int | String of string
type ty = Int | Arrow of ty * ty
```

become

$$\begin{array}{lcl} \texttt{Num} & \equiv & \lambda i.\, \lambda n\, s.\, n\, i \\ \texttt{String} & \equiv & \lambda x.\, \lambda n\, s.\, s\, x \\ \\ \texttt{Int} & \equiv & \lambda i\, a.\, i \\ \texttt{Arrow} & \equiv & \lambda t_1\, t_2.\, \lambda i\, a.\, a\, t_1\, t_2 \end{array}$$

2.1 Comparison

The Church encoding seems to be more widely known in Computer Science than the Scott encoding. For example, the Church encoding is presented in detail in standard programming languages textbooks like Pierce's, while the Scott encoding is not mentioned (Pierce 2002). The main advantage of the Church encoding is that Church-encoded data and many common operations on them are typable in System F, and hence strongly normalizing (Girard et al. 1990). Scott-encoded data are typable in System F (M.Abadi et al. 1993). It is not clear how to represent operations on them in System F, however, since those seem to require recursion instead of iteration for Scott encodings.

An advantage of Scott encodings is that constructor terms evaluate to their intended encodings in call-by-value lambda calculus. This is not true for the Church encoding, where reduction inside the bodies of λ-abstractions is needed to reduce them to the desired normal forms. For example, call-by-value reduction of Church-encoded 1 yields $\lambda s\ z.\ (s\ (Z\ s\ z))$, not $\lambda s\ z.\ (s\ z)$. Similarly, operations like addition on Church-encoded numerals evaluate with call-by-value reduction to terms which will carry out the addition when applied to interpretations of successor and zero, but which are not themselves identical to the desired resulting numeral.

Constant-time selector functions are easily definable for the Scott encoding, while with the Church encoding, known implementations of operations like predecessor are rather complicated, and run in linear time. A final advantage of Scott encodings over Church encodings is that with Church encodings, to invoke an interpretation f from within the definition of another interpretation g, one must write g so that each piece of data d is interpreted as a pair (d, v), where v is the desired resulting value. Otherwise, the data d itself is not available within the definition of g to give to f. Scott encodings do not suffer from this aesthetic limitation.

3. Typed Scott Encodings

We now turn to the question of the appropriate type for Scott encoded data. More specifically, we will be looking for an encoding of a datatype T with the following typed constructors:

$$\begin{array}{c} C_1 : \overline{\tau_1} \to T \\ \cdots \\ C_n : \overline{\tau_n} \to T \end{array}$$

where $\overline{x} \equiv x_1 \ldots x_k$ (for some k).

We will begin by considering the datatype `data` of the previous section. Suppose we want to write a function that will convert values of type `data` to strings. Using native datatypes with pattern matching, we could write:

```
let print_data data =
case data of
  Num i -> string_of_int i
| String s -> s
```

assuming a built-in function `string_of_int`. Using the Scott encoding, we could write:

```
let print_data data =
data
  (λi. string_of_int i)
  (λs. s)
```

Notice the "active" role of the value `data`, which is a function that is applied to the match cases.

What is the type of values `data`? Let's start by finding types to its arguments:

```
λi. string_of_int i : int -> string
λs. s               : string -> string
```

So,

```
data : (int -> string) -> (string -> string) -> string
```

Generalizing to any datatype, we would specify

$$T = (\overline{\tau_1} \rightarrow \texttt{string}) \rightarrow \cdots \rightarrow (\overline{\tau_n} \rightarrow \texttt{string}) \rightarrow \texttt{string}$$

Yet, while this type for `data` is fine in the context of a function for converting instances to strings, it is insufficient for other return types. To generalize to any result type, we can generalize T by replacing `string` with a type variable ρ (for "result").

$$T = (\overline{\tau_1} \rightarrow \rho) \rightarrow \cdots \rightarrow (\overline{\tau_n} \rightarrow \rho) \rightarrow \rho$$

But how is ρ bound? Is T a type constructor with argument α ($T = \lambda\, \rho.\ ...$), in which case every application of T is monomorphic, or is T a type, universally quantified over α ($T = \forall\, \rho.\ ...$), in which case our language would require first-class polymorphism to support members of T as first-class values?

To decide, let's see what we need in order to type `print_data` if `data` is a type constructor:

$$\begin{array}{l}\texttt{type}\ \ \rho\, \texttt{data} = (\texttt{int} \rightarrow \rho) \rightarrow (\texttt{string} \rightarrow \rho) \rightarrow \rho \\ \texttt{print\_data} : \texttt{string data} \rightarrow \texttt{string}\end{array}$$

So, we can successfully type `print_data` if `data` is a type constructor, but is this enough? Notice that the type of `print_data` requires a value of type `string data`. Is this too specific?

The answer depends on the intended use of elements of `data`. For limited uses of `data`, specifically, where we only intend to use datatype elements with an apriori fixed result type (i.e. monomorphically), as in `print_data`, then it is sufficient. However, if the user intends to eliminate elements of the datatype with different result types at different parts of the program, than the element must be universally quantified over the result type α. For example, consider two functions, `f` and `g`, which both take `data` as an argument, but match against the value with different result types:

```
val f: string data → string
val g: int data → int
```

What happens if we want to use `f` and `g` together, in a new function `h`?

```
let h data = (f data, g data)
```

What is the type of `h`? More specifically, what is the type of its argument `data`? The function `f` requires that it be `string data`, while the function `g` requires that it be `int data`, and we're stuck. In order to satisfy both functions, the `data` type must be *universally* quantified over its return type. The intuition is that a given `data` value doesn't care how it will be ultimately be used, so its type should work for *any* result, and not be tied to a specific one.

Therefore, the general form of an encoded datatype is:

$$T = \forall\, \rho.(\overline{\tau_1} \rightarrow \rho) \rightarrow \cdots \rightarrow (\overline{\tau_n} \rightarrow \rho) \rightarrow \rho$$

Returning to our example,

$$\texttt{type data} = \forall\, \rho.(\texttt{int} \rightarrow \rho) \rightarrow (\texttt{string} \rightarrow \rho) \rightarrow \rho$$

3.1 Polymorphic and Recursive Datatypes

At this point, we are nearly done the exercise of encoding ordinary datatypes. We have specified an encoding of datatypes that works for all result types and allows for arbitrary mixing and matching of functions over datatype values. However, "true" ML-style datatypes can be both polymorphic and recursive, so we have not finished our job until we show an encoding that can support both of these paradigms. Fortunately, supporting them is a straightforward extension of the current encoding.

We change the encoded type just as we would an ordinary datatype – we simply parameterize over the argument type. For example, the classic option type

```
type α option = None | Some of α
```

can be encoded

$$\alpha\ \texttt{option} \equiv \lambda\alpha.\forall\rho.\rho \rightarrow (\alpha \rightarrow \rho) \rightarrow \rho$$

Recursion is similarly straighforward. For example, the datatype of unary natural numbers

```
type nat = Zero | Succ of nat
```

becomes

$$\texttt{nat} \equiv \mu\tau.\forall\rho.\rho \rightarrow (\tau \rightarrow \rho) \rightarrow \rho$$

We can encode both polymorphism and recursion simultaneously as well. For example, the standard list datatype

```
type α list = Empty | Cons of α * α list
```

becomes

$$\alpha\ \texttt{list} \equiv \lambda\alpha.\mu\tau.\forall\rho.\rho \rightarrow (\alpha \rightarrow \tau \rightarrow \rho) \rightarrow \rho$$

or, potentially supporting polymorphic recursion:

$$\alpha\ \texttt{list} = \mu\tau.\lambda\alpha.\forall\rho.\rho \rightarrow (\alpha \rightarrow \tau\alpha \rightarrow \rho) \rightarrow \rho$$

Generalizing to any polymorphic, recursive datatype, we have:

$$(\overline{\alpha})\ T \equiv \mu\tau.\lambda\overline{\alpha}.\forall\rho.(\overline{\tau_1} \rightarrow \rho) \rightarrow \cdots \rightarrow (\overline{\tau_n} \rightarrow \rho) \rightarrow \rho$$

3.2 Simplifying with Record Types

There is one remaining step we'd like to take before declaring us done, with respect to ordinary datatypes. This step is optional, but can lead to somewhat more readable types. We bundle all of the match-case functions together in a record, and give that record type a name. We choose records instead of ordinary tuples because they allow us to give a name to each match case, providing further (helpful) annotations.

$$(\overline{\alpha})\ T \equiv \mu t.\lambda\overline{\alpha}.\forall\rho.\{\ c_1 : \overline{\tau_1} \rightarrow \rho; \ldots ; c_n : \overline{\tau_n} \rightarrow \rho\} \rightarrow \rho$$

Notice that the general definition of each constructor, for constructors $\overline{C}$ and projection function π_c, is

$$\frac{\mathsf{arity}(C_i) = n}{C_i = \lambda x_1 \ldots x_n\, m.\,(\pi_{c_i}\, m)\ \overline{x}}$$

4. Encoding GADTs

We now turn to the challenge of encoding *generalized* datatypes. We first note that GADTs are only meaningful when the datatype has at least one type parameter. Otherwise, they are regular, old datatypes. The essential generalization of GADTs is that different constructors for the datatype are free to instantiate the type argument(s) as they please. Furthermore, they can each specify their *own* set of type arguments, for use in instantiating the parameter(s).

For example, we can revisit the `ty` type from Section 2.

```
type ty = Int | Arrow of ty * ty
```

We can restrict the set of values that can inhabit this type by adding a type parameter to `ty` and then instantiating that parameter selectively, based on the particular constructor. Notice that we now explicitly annotate each alternative with the instance of `ty` constructed.

```
type α ty =
    Int: int ty
  | Arrow of α ty * β ty : (α → β) ty
```

How can our encoding support this added precision? Each constructor must be allowed to parameterize over its own set of type parameters. Since constructors in our encoding are expressed in terms of their corresponding match case – a function – we must assign those match-case functions first-class polymorphic types. The example above therefore becomes

$$\begin{aligned}\texttt{type ty} = \quad & \lambda\alpha.\forall\rho.\{\\ & \quad \texttt{int} : \rho;\\ & \quad \texttt{arrow} : \forall\alpha,\beta.\ \alpha\ \texttt{ty} * \beta\ \texttt{ty} \rightarrow \rho;\\ & \} \rightarrow \rho\end{aligned}$$

We're half-way there – each constructor can choose its own parameters, but they are unable to involve those parameters in their result type. This deficiency is clearest in the definition of `ty`, where the type parameter α goes unused.

The second step is to generalize the result type ρ, changing it from a type to a *type constructor*[2]. Then, for each case, we instantiate ρ with types appropriate to that case (and, usually, constructed from that case's type arguments). Revising our example, then, we have:

$$\begin{aligned}\texttt{type ty} = \quad & \mu t.\lambda\alpha.\forall\rho :: * \rightarrow *.\{\\ & \quad \texttt{int} : \texttt{int}\ \rho;\\ & \quad \texttt{arrow} : \forall\alpha,\beta.\ \alpha\ t * \beta\ t \rightarrow (\alpha \rightarrow \beta)\ \rho;\\ & \} \rightarrow \alpha\ \rho\end{aligned}$$

Notice that we've annotated ρ with its kind, $* \rightarrow *$, read "a function from type to type". Two examples of such functions are the type signatures for type-indexed marshalling and unmarshalling functions:

$$\rho = \lambda\,\alpha.\alpha \rightarrow \texttt{bits}$$
$$\rho = \lambda\,\alpha.\texttt{bits} \rightarrow \alpha$$

Furthermore, the types of the `ty` constructor functions are now:

```
int : int ty
arrow : ∀α,β. α ty → β ty → (α → β) ty
```

For example,

```
arrow int int : (int → int) ty
```

where

$$(\texttt{int} \rightarrow \texttt{int})\ \texttt{ty} = \forall\rho :: * \rightarrow *.\ \{\ldots\} \rightarrow (\texttt{int} \rightarrow \texttt{int})\ \rho$$

Once again, we generalize our results to any datatype:

$$T \equiv \mu t.\lambda\overline{\alpha}.\forall\rho :: \overline{*} \rightarrow *.\{\ \ldots c_i : \forall\overline{\alpha_i}.\overline{\tau_i} \rightarrow (\overline{\sigma_i})\ \rho; \ldots\} \rightarrow (\overline{\alpha})\rho$$

where, forall i, the $\overline{\sigma_i}$ types are functions of the $\overline{\alpha_i}$ parameters.

Notice that the one-line definition above provides a succint specification of "sufficient conditions" for GADT types:

recursive types
type constructors
second-order polymorphism
impredicative polymorphism
record types (optional)

If your language has all of these features, then you are all set. Unfortunately, it would seem that the one well-known language which has all these features, already has native support for GADTs (yes, GHC Haskell). But, what can you do if your language has some, but not all of these features? In Section 6, we discuss how to encode GADTs in OCAML, which has most, but not quite all, of these features in its core language. However, before jumping on to practical matters, we will consider an alternative way of arriving at the definition presented above.

5. From Types to Terms

In the previous section, we started with a set of terms and devised types for those terms capable of capturing the features of GADTs. However, we can actually start from the other direction – GADT types – and arrive at the same place. We'll also find an interesting alternative encoding along the way.

What is the type of a GADT $(\overline{\alpha})T$? Xi et al. (2003) provide an account which we will use here:

$$(\overline{\alpha})T \equiv \mu t.\lambda\overline{\alpha}.(\exists[\overline{\alpha_1}, \overline{\sigma_1} = \overline{\alpha}].\tau_1 + \cdots + \exists[\overline{\alpha_n}, \overline{\sigma_n} = \overline{\alpha}].\tau_n)$$

where every occurence of T in the body of the fixpoint is replaced with t. We can see from this definition that the key novelty of GADTs over standard datatypes is that their branches are individually existentially quantified and constrained in relation to the type parameters of the GADT.

In order to encode this type in a language without existentials, we use the familiar trick of converting existentials to universals by reifying their elimination rule. The elimination rule specified by Xi et al. for these existentials is:

$$\frac{\Delta;\Gamma \vdash e_1 : \exists\Delta_1.\tau_1 \quad \Delta,\Delta_1;\Gamma,x:\tau_1 \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash \texttt{unpack}\ e_1\ \texttt{as}\ (|\Delta_1|, e_2)\ \texttt{in}\ e_2 : \tau_2}\ (\exists\text{−elim})$$

assuming a typing judgment of the form $\Delta;\Gamma \vdash e : \tau$, where Δ contains bound type variables and type constraints, and $|\Delta|$ is the set of bound variables from Δ (that is, the erasure of the constraints).

In essence, we can encode an existential as a function from the remaining hypothesis in its elimination rule to its result. To do so, we must first find a term which captures the conditions of the hypothesis. We do so in two steps, using standard System F typing rules. Starting with the hypothesis, we apply a λ introduction rule:

$$\frac{\Delta,\Delta_1;\Gamma,x:\tau_1 \vdash e_2 : \tau_2}{\Delta,\Delta_1;\Gamma \vdash \lambda x:\tau_1.e_2 : \tau_1 \rightarrow \tau_2}$$

Next, we apply a polymorphic-function introduction rule:

$$\frac{\Delta,\Delta_1;\Gamma \vdash \lambda x:\tau_1.e_2 : \tau_1 \rightarrow \tau_2}{\Delta;\Gamma \vdash \Lambda\Delta_1.\lambda x:\tau_1.e_2 : \forall\Delta_1.\tau_1 \rightarrow \tau_2}$$

Note that none of the type variables in Δ_1 are free in τ_2. Now, since the result type of ($\exists$−elim) is τ_2, we have

$$\exists[\Delta].\tau \equiv (\forall[\Delta].\tau \rightarrow \tau_2) \rightarrow \tau_2$$

We now have a type which is familiar, except for the type constraints in Δ. We could attempt to eliminate those constraints, by defining a substitution θ and using it to substitute every occurence of α_i in $\tau \rightarrow \tau_2$ with the corresponding σ_i:

$$\exists[\overline{\beta}, \overline{\sigma} = \overline{\alpha}].\tau \equiv (\forall[\overline{\beta}].\tau[\theta] \rightarrow \tau_2[\theta]) \rightarrow \tau_2$$

However, this definition is incomplete, because τ_2 is left open. Yet, we cannot simply quantify the definition by all τ_2, because we would then need to include the substitution explicitly in the type itself, which would violate the syntax of System F types. Instead, rather than quantifying over a *type* τ_2, we can abstract over a type *constructor* ρ, whose arity is that of T. Then, we can instantiate ρ in the nested result type according to the constraints specified in the existential:

$$\exists[\overline{\beta}, \overline{\sigma} = \overline{\alpha}].\tau \equiv \forall\rho :: \overline{*} \rightarrow *.(\forall\overline{\beta}.\tau[\theta] \rightarrow (\overline{\sigma})\rho) \rightarrow (\overline{\alpha})\rho$$

We note two things. First, the application of θ to τ is not a concern, because τ is a concrete type specified in the existential, and θ can therefore be applied as part of the encoding process. Equivalently, we could restrict $\overline{\alpha}$ from appearing in τ, and thereby drop the need for θ, because any appearance of an α_i can simply be replaced by a fresh type variable β_i and a corresponding constraint $\beta_i = \alpha_i$. Second, notice that ρ is instantiated differently in the nested function type as in the top-level function type. This difference implicitly captures the constraints which appear explicitly in the existential.

With this encoding in hand, we now have an interesting choice which we did not encounter in Section 4. One potential next step

[2] This parameterization over type constructors is what is variously referred to as higher-order polymorphism or higher kinds.

is to eliminate sums from the definition of T using the same technique as we did for existentials. We would then arrive at the same definition of T as the one we constructed in Section 4. The Scott encoding falls out as an intuitive inhabitant of that type.

An alternative step would be to stop here and present an encoding which relies upon regular datatypes for the sum and the encoding for the existentials:

$$\begin{aligned}(\overline{\alpha})T \equiv \mu t.\lambda\overline{\alpha}.&\\ &(\forall\rho.(\forall\overline{\beta_1}.\tau_1 \rightarrow (\overline{\sigma_1})\rho) \rightarrow (\overline{\alpha})\rho)\\ &+ \ldots\\ &+ (\forall\rho.(\forall\overline{\beta_n}.\tau_n \rightarrow (\overline{\sigma_n})\rho) \rightarrow (\overline{\alpha})\rho)\end{aligned}$$

(omitting the kind of ρ to reduce clutter). The encoding of the data itself would then be datatypes where each branch is a polymorphic function. The `ty` example of the previous section would look something like this (where ρ has kind $* \rightarrow *$):

```
datatype α ty =
   Int of ∀ρ.int ρ → α ρ
|  Arrow of ∀ρ.(∀α,β.α ty * β ty → (α → β) ρ) → α ρ
```

While this latter choice seems appealling in that it would give GADTs a somewhat more natural feel, in practice it complicates matters because any patttern-match code will need to handle instantiating a second-order polymorphic value in each branch, rather than dealing with it once, as is the case with the other encoding.

6. OCAML Implementation

OCAML, along with other dialects of ML, does not support higher kinds in its core language. So, core-language types and functions cannot be parametrized by type constructors. However, ML's rich module system *does* support higher-kinds, after a fashion[3]. Specifically, an ML module can be parametrized by another module containing type constructors (among other things). Therefore, one can use the module system to manually parametrize the necessary types and functions and then manually instantiate them. Given that type inference for first-class polymorphism and higher kinds is, in general, undecidable, any system supporting both will require some amount of manual effort. The question is only how much.

Unfortunately, using the module system has a number of significant drawbacks. First, the ML module system was hardly designed for this (lightweight) use, and the syntax overhead is rather high. Second, the module system does not mix freely with the core language – modules are not first-class values and most, if not all, uses of the module system must occur outside of core language expressions.

We'll try an example – our familiar type `ty` – to get a concrete sense of the effort involved in using the module system, and its limitations. Note that we are using actual OCAML syntax here, including its support for first-class polymorphic record fields. In addition, we have separated the record type into its own (mutually recursive) definition, because OCAML does not support nested record types.

```
module F(R: sig type 'a r end) = struct
   type ty_match = {
      int: int R.r ;
      arrow: 'a 'b. 'a ty -> 'b ty -> ('a -> 'b) R.r;
   }
   and 'a ty = ty_match -> 'a R.r
end
```

We have defined a *functor* – a function from modules to modules, `F`, which takes a single argument, the module `R`. This module, in turn, has only a single element: a type constructor `r`, whose kind is (implicitly) specified as `* -> *`. The body of `F` defines two types, familiar to us from above. The key difference is that, now, the universal quantification over type constructor `'r` has been lifted above the definitions of both types and recast as a quantification over all modules with the signature `sig type 'a r end`. This transformation, as it were, leaves us with a choice to make in formulating the type combinators in OCAML. If we want to remain faithful to the type signature we started with, where the result of each combinator – the encoding – is a higher-kinded polymorphic value, then each combinator must itself be a functor, rather than a function. Otherwise, there is no way for the argument or result of any given combinator to have a type with higher kind. Unfortunately, this would force all use of combinators to occur within the module system, not the core language, which would prohibit first-class type encodings.

An alternative is to perform a similar transformation to our type combinators as we applied to the matches. Previously, the quantification over the result type-constructor was bundled up in the definition of `ty`, and therefore hidden from the types of the combinators. Now, though, we must lift the quantification out of `ty` and over the type combinators. The result is another functor:

```
module F(R: sig type 'a r end) = struct
   val int : int F(R).ty
   val arrow : 'a F(R).ty -> 'b F(R).ty
               -> ('a -> 'b) F(R).ty
end
```

This solution, though, is really a compromise: yes, `int` and `arrow` are core-language values, but their type has already been specialized to a particular choice of `r`. So, the type encodings can be first-class, but only after they've lost their flexibility to be instantiated at any match type. This alternative, then, apparently buys us little over the previous one.

Let's take a step back and consider what is preventing us from implementing the desired solution. We can write the *expressions* we desire; we can even "prove" them to be higher-kinded by encoding them as well-typed functors. What we cannot do, however, is present that proof to OCAML's core language type checker, because the core language type checker has no support for checking or using higher-kinded polymorphic values. So, we're in an interesting bind – we have a value which is provably safe for our purposes (according to the OCAML module system) but cannot be used for our purposes (according to the OCAML core language). Well, what you do in a typed language when the type system isn't quite good enough? Following in a long tradition of seasoned programmers, you very cautiously use *unsafe cast*! We know that our encodings our higher-kinded polymorphic, so can we safely cast them between different instances of the result type `r`.

At this point you might be wondering: but isn't unsafe cast, well, unsafe? The answer is a resounding Yes! – except when it is safe. In other words, "unsafe" in this context only means that the OCAML type checker cannot prove it to be safe. But, if we prove it safe ourselves, then we can still be assured of the safety of our program. Indeed Coq does this all the time when extracting OCAML code from Coq proofs. Still, we must be very careful when using unsafe cast, so we will be methodical. First, we will present the encoding and discuss where we use the unsafe cast. Then, we will show how to use parametricity to ensure that the cast is safe.

For the encoding, we return to the examples in the preceding section, although we join the two functors presented there into one.

[3] We are eliding the fact that OCAML and SML have different module systems – for the purposes of this discussion, they are the same.

```
module F(R: sig type 'a r end) = struct
   type ty_match = {
      int: int R.r ;
      arrow: 'a 'b. 'a ty -> 'b ty -> ('a -> 'b) R.r;
   }
   and 'a ty = ty_match -> 'a R.r

   val int : int ty
   val arrow : 'a ty -> 'b ty -> ('a -> 'b) ty
end
```

With this encoding, we can cast between values by defining a `Cast` functor:

```
module Cast(R1: sig type 'a r end)
           (R2: sig type 'a r end) :
sig
  val cast : 'a F(R1).ty -> 'a F(R2).ty
end =
struct
  let cast = Obj.magic
end
```

where `Obj.magic` is OCAML's (undocumented) unsafe cast, with type `'a -> 'b`. The signature on the `Cast` functor restricts the generality of the unsafe cast to specifically casting between different instantiations of the `ty` type.

The question, now, is whether the `cast` function is safe? The answer is "no." The problem is that, while using combinators `int` and `arrow` are the recommended methods for creating values with type `ty` (or instantiations thereof), it is not the only way. Since the `ty` type is defined transparently, we can create values inhabiting `ty` that are not valid constructor encodings, and, therefore, not safe to cast. For example,

```
fun m = "ERROR"
```

has type

```
int F(struct 'a r = string end).ty
```

but is most certainly not a valid type encoding, which, for this type, is `fun m -> m.int`. The solution, then, is twofold: first, make type-constructor `ty` abstract, which will ensure that constructors can only be created with the provided combinators, and, second, prove that all values constructed with the combinators are safe to cast.

Below is the new formulation of our GADT encoding, again using our `ty` example. The key things to notice are that `ty` is now abstract and that all universal quantification is once again hidden. The situation for the match type is somewhat different, in that we still need to parameterize using a functor. However, the functor this time simply encodes a higher-kinded type constructor, rather than a higher-kinded polymorphic value. Since no values inhabit type constructors (only types!), there is no issue of restricting a value's ability to be first class like there was with the constructor encodings.

```
type 'a ty
val int : int ty
val arrow : 'a ty -> 'b ty -> ('a -> 'b) ty

module F(R: sig type 'a r end) : sig
   type ty_match =
     int_c: int R.r ;
     arrow_c: 'a 'b. 'a ty -> 'b ty -> ('a -> 'b) R.r;

   val gmatch : 'a ty -> (ty_match -> 'a R.r)
end
```

In addition to the match type, we include a `gmatch` function which takes the abstract encoding and a match and returns the result. It can also be used (curry-style) as a cast function, which takes an opaque encoding and returns an encoding specialized to `R.r`. The function `gmatch` is implemented essentially as the identity function: it takes an `'a ty` and returns it. This makes sense, because `'a ty` is essentially equivalent to `ty_match -> 'a R.r`. We cannot prove that equivalence to the OCAML type checker, however, so `gmatch` must use `Obj.magic` to make the cast.

This new interface dispatches the first requirement for safe casting: control over the inhabitants of `'a ty`. Next, let's take a look at the implementation of the combinators to be sure that they are truly polymorphic over the result type `r`. Below is the entire implementation of the `ty` GADT.

```
module AbstractR : sig type 'a r end =
struct
  type 'a r = int
end

type ty_match = {
   int_c: int AbstractR.r;
   arrow_c: 'a 'b. 'a ty -> 'b ty
             -> ('a -> 'b) AbstractR.r;
}
and 'a ty = ty_match -> 'a AbstractR.r

let int m = m.int_c
let arrow ty1 ty2 m = m.arrow_c ty1 ty2

module F(R: sig type 'a r end) = struct
   type ty_match = {
     int_c: int R.r ;
     arrow_c: 'a 'b. 'a ty -> 'b ty -> ('a -> 'b) R.r;
   }
   let gmatch : 'a ty -> (ty_match -> 'a R.r)
              = Obj.magic
end
```

The key to ensuring the safety of `gmatch` lies in the module `AbstractR`. This module contains a single element – a type constructor `'a r` – and hides the definition of that element from the remainder of the code shown. That is, `r` is an abstract type construct. *Parametricity therefore guarantees us that any code which uses* `r` *will in fact be parametric in* `r`*'s definition*. That is, it will be a higher-order polymorphic value.

Next, we define the types `ty_match` and `ty` in terms of `AbstractR.r` and then define the constructors `int` and `arrow` in terms of the `ty_match` record type. Finally, we define the functor `F` which creates versions of `ty_match` and `gmatch` based on a user-specified type constructor `R.r`.

There is one last detail to attend to. The astute OCAML programmer will note that OCAML record types are generative – that is, if two record types are structurally identical, and differ only by name, they are still *different* types. Therefore, even though the type `ty` is parametric in `r`, each instantiation is in fact a different type, which raises the question whether their representation differs in any way. The answer, according to the OCAML manual (Leroy et al. 2008), Section 18.3, is that they are not. This sections discusses the representation of OCAML values for use in interfacing with C. To quote the particular section of relevance:

> 18.3.2 Tuples and records ... Records are also represented by zero-tagged blocks. The ordering of labels in the record type declaration determines the layout of the record fields: the value associated to the label declared first is stored in field 0 of the block, the value associated to the label declared next goes in field 1, and so on. ...

Notice that the representation of records depends only on the types and labels of the fields, and not on the particular declared record type.

```
type ('a,'b) sum = Left of 'a | Right of 'b

type 'r ty_rep

val int   : int ty_rep
val unit  : unit ty_rep
val tuple : 'a ty_rep -> 'b ty_rep -> ('a * 'b) ty_rep
val sum   : 'a ty_rep -> 'b ty_rep -> ('a,'b) sum ty_rep
val list  : 'a ty_rep -> 'a list ty_rep

module type Result = sig type 'r tycon end
module MakeTys(R : Result) : sig
  type 'r result = 'r R.tycon
  type type_case = {
    int_c   : int result;
    unit_c  : unit result;
    tuple_c : 'a ty_rep -> 'b ty_rep -> ('a * 'b) result;
    sum_c   : 'a ty_rep -> 'b ty_rep -> ('a,'b) sum result;
    list_c  : 'a ty_rep -> 'a list result;
  }
  val gmatch : 'a ty_rep -> type_case -> 'a result
end
```

Figure 1. Interface to Type module

7. OCaml Examples

An encoding is only valuable so long as it is usable. In this section, we present some extended examples demonstrating the usability of GADTs. However, because recursing over a GADT element will normally require polymorphic recursion, we begin with a simple demonstration of how to use polymorphic recursion in OCAML, before getting to the examples.

7.1 Polymorphic Recursion

Polymorphic recursion means recursion within a polymorphic function for which the recursive occurrences of the function are instantiated at different types than that of surrounding invocation. Hinze (2000) presents a data structure for perfectly balanced, binary leaf trees, which uses so-called *nested* types (Bird and Meertens 1998):

```
type 'a perfect = Zerop of 'a
                | Succp of ('a * 'a) perfect
```

Now, if we want to recurse on such a datatype we'll need to use polymorphic recursion, because the recursive reference to `perfect` in the `Succp` branch instantiates the datatype argument at a type other than `'a`. So, we declare a record with a single, polymorphic field, whose type is the signature of the function we wish to write. Then, we create a recursive *record* with the field set to the recursive function we are writing. The recursive call then goes through the record (`pcount.v`), rather than through a function name.

```
type pcount_sig = {v: 'a. 'a perfect -> int}
let rec pcount ={v= function
    Zerop(x) -> 0
  | Succp(x) -> 1 + pcount.v x}
```

That's it. This function will count the depth of the perfect tree.

7.2 Example: Run-time Type Encodings

As our first example, we'll extend the `ty` GADT from Section 4 and present two generic functions: an S-expression printer, and a query function, which makes use of the printer, demonstrating the flexibility of the encoding.

Figure 1 shows the interface to the `Types` module. Figure 2 shows a client of the module which implements a generic querying function. The query specifies a path from the root of the data structure to a particular element, where each path component is an integer specifying the nth subcomponent. In the figure, we show a base case, where the query has ended, and the cases for tuples and lists. Notice the use of `gen_to_string` in the base case, which demonstrates the convenience of the Scott encoding. If we had chosen the Church encoding, either `gen_query` and `gen_to_string` would need to be defined together, and paired in their definitions, or `gen_query` would need to return as its result the both the real result *and* the type representation from which the result was derived. Interestingly, it is just this sort of added convenience that motivated dependency-style generic Haskell (Löh et al. 2003).

```
module Q = Type.MakeTys(struct type 'a r = 'a -> string)

type to_string_sig = v: 'a. 'a ty_rep -> a' -> string
let rec gen_to_string_r ={v= fun ty q x ->
  Q.gmatch ty {Q.
       tuple_c = (fun tya tyb (a,b) ->
           "(tuple " ^ gen_to_string.v tya a ^ " " ^
                       gen_to_string.v tyb b ^ ")"
       ...
  }
}
let gen_to_string = gen_to_string_r.v

type query = string list
type query_sig = {v: 'a. 'a ty_rep -> query -> a' -> string}
let rec gen_query_r ={v= fun ty q x ->
  match q with
    [] -> gen_to_string ty x
  | n::qs ->
     let x_to_string = Q.gmatch ty {Q.
       ...
       tuple_c = (fun tya tyb (a,b) ->
         match n with
           1 -> gen_query.v tya a qs
         | 2 -> gen_query.v tyb b qs
         | _ -> throw (Failure "query")
       list_c = (fun ty_elt xs ->
         gen_query.v ty_elt (List.nth xs n) qs);
     } in
     x_to_string x
```

Figure 2. Code fragment implementing generic query

```
type ('a,'b) eq
type zero
type 'a succ
type ('m,'n) plus

type one = zero succ
val plus_zero_x : ('a, (zero, 'a) plus) eq
val plus_succ_comm : (('x,'y) plus succ,
                      ('x succ, 'y) plus) eq
...
```

Figure 3. Interface of natural-number arithmetic module `NatArith`

7.3 Example: Indexed Lists

For our second example, we present a GADT encoding lists with statically tracked length. Since OCAML does not natively provide support for natural numbers at the type level, we provide a unary encoding of natural numbers through abstract types in an natural-number arithmetic module, shown in Figure 3. The module includes an equational theory of the natural numbers, which we elide.

Using the `NatArith` module we can declare indexed lists as shown in Figure 4. The interface follows the pattern described in Section 6, with two differences. First, we have changed the

```
open NatArith

type ('r,'i) ilist

val nil       : ('a, zero) ilist
val cons      : 'a -> ('a, 'm) ilist
                -> ('a, 'm succ) ilist
val coerce    : ('m,'n) eq -> ('a,'m) ilist
                -> ('a,'n) ilist

module type Result = sig type ('a,'i,'s) tycon end

module MakeTys(R : Result) :
sig
  type ('a,'i,'s) result = ('a,'i,'s) R.tycon
  type ('a,'s) ilist_case = {
      nil_c   : ('a, zero, 's) result;
      cons_c  : 'i. 'a -> ('a,'i) ilist
                     -> ('a,'i succ, 's) result;
    }
  val gmatch : ('a,'i) ilist -> ('a,'s) ilist_case
               -> ('a,'i,'s) result
end
```

Figure 4. Interface of IndexedList module

```
module App_match = struct
  type ('a, 'i, 'c) tycon = ('a,('i, 'c) plus) ilist
end
module Am = MakeTys(App_match)

type ('a,'j) psig2 = {v: 'i.('a,'i) ilist -> ('a,'j) ilist
                              -> ('a,('i,'j) plus) ilist}
let rec iappendr ={v=fun l1 l2 ->
  Am.gmatch l1 {Am.
    nil_c  = (coerce plus_zero_x l2);
    cons_c = (x xs =
       let xs2  = cons x (iappendr.v xs l2) in
       coerce plus_succ_commute xs2);
  }
}
let iappend = iappendr.v
```

Figure 5. Interface and implementation of append for indexed lists. The function `plus_succ_comm` is a lemma that `plus` and `succ` commute.

result type constructor to take three arguments, instead of one: the type of the list element, the length of the list and another variable (`'s`) whose value will be inferred by the type checker. This added variable allows the result to depend on a type from the environment, which would otherwise be impossible. Second, we add a `coerce` function, which provides a way to coerce a list whose length is expressed with type-level natural number `'m` to one with length `'n`, based on a proof that `'m` and `'n` are equivalent. This function is needed to convince the type checker of type equivalence over lists – its implementation is essentially the identity function.

In Figure 5, we show the list-append function. Each case is coded as normal, but with an added coercion. In the case of `nil` we need to prove that $0 + j$ is equivalent to j, which we do with an axiom provided in the `NatArith` module (`plus_zero_x`). The case of `cons` requires us to prove that $Succ(i + j)$ – the result of cons'ing after an append – is equivalent to $Succ(i) + j$, which is the declared result of the `cons` case. We achieve this with the axiom that plus and succ commute.

8. Related Work

There is a lot of work on encoding various datatypes into various high-level languages. Bohm and Berarducci (1985) is a classic work on a Church-style encoding of datatypes (called *term algebras*) into System F. M.Abadi et al. (1993) is a short note on encoding Scott numerals in System F. However, they do not address the issue of how to *use* the encodings without recursion. Also, as folklore would have it, early versions of Coq used a Church-like encoding for inductive datatypes in which each piece of data was interpreted as a pair consisting of the data itself and a value computed for that data. These inductive datatypes were in fact more powerful than GADTs.

Berarducci (Bohm and Berarducci 1985).

An alternative path of research has been to create formalisms for natively integrating GADTs into existing languages. Cheney and Hinze (2003) investigated their integration into a Haskell-like setting, while, concurrently, Xi et al. (2003) investigated their integration into an ML-like setting. While Cheney and Hinze's formalism is somewhat more general than Xi et al.'s, we focused on the latter because of our interest in an encoding appropriate for OCAML. Sulzmann and Wang (2004) also investigate integration of GADT functionality within the setting of Haskell.

In addition to this general work, there has been a great deal of work on encoding *particular* GADTs[4]. Pfenning and Lee (1991) present one of the first encodings of a GADT for typed abstract syntax trees. Carette et al. (2007) present related results, focusing on tagless, staged interpretation. They include implementations in Haskell and ML. However, their ML encoding is strictly limited to the module system. Weirich presents a number of encodings related to polytypic programming (Weirich 2001, 2006). In essence, these are Church-style encodings of a `type` GADT. She also presents implementations in Haskell. Hinze (2004), inspired by Weirich's Haskell encoding, shows how the encoding can be realized in Haskell using only type classes. He offers two different encodings, which offer a tradeoff between convenience and flexibility. Interestingly, although Hinze does not note this himself, his first encoding uses the Scott encoding, while his second uses the Church encoding.

Of particular relevance to this paper are encodings of generic programing in ML, which all are essentially encodings of the `type` GADT in ML. The essential reference is Yang's work (Yang 1998), in which he showed how to encode the `type` GADT entirely in ML's module system. However, because of the difficulty of programming with the module system, he also shows an alternative encoding based on projection/injection functions. Karvonen (2007) generalizes Yang's results. However, his work is still limited to ML's module system. Finally, Fernández et al. (2008) combine Yeng's and Hinze's work, to encode the `type` GADT in OCAML. This pearl is a generalization of that encoding to any GADT and provides a more flexible mechanism for instantiating match result types based on the safe use of OCAML's unsafe cast.

Related ideas have also appeared in the object-oriented world. We mention only two, although there are likely many more. Buchlovsky and Thielecke (2005) provide a type-theoretic account of the visitor pattern, which is quite similar to both the Scott and Church encodings. Kennedy and Russo (2005) discuss how GADTs can be encoded in C#, and propose language extensions which would make the encoding more efficient.

For readers looking for more leads on GADTs, we recommend Tim Sheard's home page (Sheard), which lists a large collection of relevant works. Finally, for readers intrigued by the use of indexed-lists in OCAML, but put off by the need to manually prove equalities between integer expressions, we recommend Con-

[4] That is, paradigms which can be captured with GADTs.

coqtion, which presents a simple and elegant integration of OCAML with Coq (Fogarty et al. 2007). Along the same lines, the Omega language aims to provide language/compiler support for paradigms of this sort (Sheard).

9. Conclusion

We have seen how GADTs may be encoded in polymorphic lambda calculus using a typed version of the Scott encoding of inductive datatypes. We have also seen how OCAML's module system can be used to implement this encoding, even though the term language of OCAML lacks first-class polymorphism. We hope that this encoding will make GADTs more accessible to OCAML programmers, and lead to native implementation of GADTs in OCAML. If that goal is achieved, the need for this encoding will disappear, much like Scott-encoded data themselves, in their applications.

Acknowledgments. Thanks to David Walker on comments on an earlier draft, and Stephanie Weirich for several references to related work.

References

Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Proceedings 4th Int. Conf. on Mathematics of Program Construction, MPC'98, Marstrand, Sweden, 15–17 June 1998*, volume 1422, pages 52–67. Springer-Verlag, Berlin, 1998. URL `http://citeseer.ist.psu.edu/bird98nested.html`.

C. Bohm and A. Berarducci. Automatic synthesis of typed lambda - programs on term algebras. *Theoretical Computer Science*, 39(2-3): 135–153, August 1985. ISSN 0304-3975.

Peter Buchlovsky and Hayo Thielecke. A type-theoretic reconstruction of the visitor pattern. In *21st Conference on Mathematical Foundations of Programming Semantics*, 2005. URL `http://www.cl.cam.ac.uk/~pb368/mfps-visitors.pdf`.

Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated. pages 222–238. 2007.

James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

H. Curry, J. Hindley, and J. Seldin. *Combinatory Logic*, volume 2. North-Holland Publishing Company, 1972.

Mary Fernández, Kathleen Fisher, J. Nathan Foster, Michael Greenberg, and Yitzhak Mandelbaum. A generic programming toolkit for PADS/ML: First-class upgrades for third-party developers. In *PADL*, 2008.

Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: indexed types now! In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 112–121, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-620-2. doi: http://dx.doi.org/10.1145/1244381.1244400. URL `http://dx.doi.org/10.1145/1244381.1244400`.

J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1990.

Ralf Hinze. Generics for the masses. In *ICFP*, 2004.

Ralf Hinze. Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 10(03):305–317, 2000. doi: http://dx.doi.org/10.1017/S0956796800003701. URL `http://dx.doi.org/10.1017/S0956796800003701`.

Vesa A.J. Karvonen. Generics for the working ml'er. In *ML Workshop*, 2007.

Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 21–40, New York, NY, USA, 2005. ACM Press. ISBN 1595930310. doi: http://dx.doi.org/10.1145/1094811.1094814. URL `http://dx.doi.org/10.1145/1094811.1094814`.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The objective caml system, release 3.11, documentation and user's manual, 2008.

Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style generic haskell. *SIGPLAN Not.*, 38(9):141–152, September 2003. ISSN 0362-1340. doi: http://dx.doi.org/10.1145/944746.944719. URL `http://dx.doi.org/10.1145/944746.944719`.

M.Abadi, L.Cardelli, and G.Plotkin. Types for the Scott Numerals. Available from `http://lucacardelli.name`, 1993.

Frank Pfenning and Peter Lee. Metacircularity in the polymorphic lambda-calculus. *Theor. Comput. Sci.*, 89(1):137–159, 1991.

B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

Tim Sheard. Tim Sheard's home page. `http://web.cecs.pdx.edu/ sheard/`.

A. Stump. Directly Reflective Meta-Programming. *Higher-Order and Symbolic Computation*, 2008. To appear, available online from the journal's web site.

Martin Sulzmann and Meng Wang. A systematic translation of guarded recursive data types to existential types. Technical Report TR22/04, The National University of Singapore, 2004.

Stephanie Weirich. Type-safe run-time polytypic programming. *J. Funct. Program.*, 16(6):681–710, 2006.

Stephanie Weirich. Encoding intensional type analysis. In David Sands, editor, *ESOP*, volume 2028 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2001. ISBN 3-540-41862-8.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.

Zhe Yang. Encoding types in ML-like languages. In *ICFP*, 1998.