Concoqtion: Indexed Types Now! *

Seth Fogarty
Rice University
sfogarty@cs.rice.edu

Emir Pasalic
Rice University
pasalic@cs.rice.edu

Jeremy Siek
University of Colorado
jeremy.siek@colorado.edu

Walid Taha
Rice University
taha@cs.rice.edu

Abstract

Almost twenty years after the pioneering efforts of Cardelli, the programming languages community is vigorously pursuing ways to incorporate F_ω-style indexed types into programming languages. This paper advocates Concoqtion, a practical approach to adding such highly expressive types to full-fledged programming languages. The approach is applied to MetaOCaml using the Coq proof checker to conservatively extend Hindley-Milner type inference. The implementation of MetaOCaml Concoqtion requires minimal modifications to the syntax, the type checker, and the compiler; and yields a language comparable in notation to the leading proposals. The resulting language provides unlimited expressiveness in the type system while maintaining decidability. Furthermore, programmers can take advantage of a wide range of libraries not only for the programming language but also for the indexed types. Programming in MetaOCaml Concoqtion is illustrated with small examples and a case study implementing a statically-typed domain-specific language.

1. Introduction

Eighteen years ago, Cardelli [3] argued that highly expressive *indexed types* can be based on F_ω [11] and the more expressive calculus of constructions [9]. Today, the practical potential of such expressive types is widely recognized: They can be used to statically enforce program properties such as safety of array indexing [31], type-preservation of source-to-source program transformations [4, 20, 24], type-safety of dynamically generated serializers [15], and algorithmic invariants of data-structure libraries [5, 23, 31].

Exactly how the original idea is incorporated into a language design varies dramatically from one language design to the next. Cardelli's Quest [3] draws on many different sources for inspiration, and characterizing the semantics (denotationally) became the focus. Ten years later, DML [32] and Cayenne [1] took two radically different approaches to designing languages with indexed types. DML restricts the index language to a decidable domain (Presburger integer arithmetic) and thereby maintains the decidability of the type system. In general, and especially when indexed types are added to an existing language, decidability requires a clear distinction between the computational language and the index language. In contrast, Cayenne extends what is an otherwise standard type theory with general recursion. While this renders the type theory unsound for proofs, it incorporates the key idea that a programming language is a (potentially unsound) proof language. The next wave of language designs came three years later, and continued to reflect these two approaches. Cyclone [13] extends the C programming language with special-purpose indexed types for safe multi-threading and memory management. Epigram [2] follows in the footsteps of Cayenne, expressing programs as proofs in type theory, but allowing only provably well-founded recursion so as to guarantee decidability.

Recently, the DML approach has been taken a step further in the form of Generalized Algebraic Datatypes (GADTs). GADTs aim to provide an intuitive generalization of the Algebraic Datatypes of languages such as ML and Haskell [6, 24, 30]. They have been incorporated into Haskell [21] and C# [15]. GADTs are a convenient and practical form of indexed types, as illustrated by many interesting examples in the literature, and techniques have been developed to further reduce the notational overhead of GADTs [25].

While GADTs provide a powerful tool, they have drawbacks that can have significant implications for large-scale programming with indexed types.

First, they do not provide a direct way to express functions on types. Yet for many problems, functions on types are the natural way to express dependencies between types. GADTs force the programmer to express such functions as relations.

Second, at least in the form they are used in Haskell, GADTs always require that proofs be manipulated at runtime. But for many problems, proofs need only exist during compilation.

Third, and possibly most significantly, it has not been a design goal of any of the current GADT proposals to provide the programmer with direct means to express and structure proofs. This raises two questions: First, when will standard mathematical results be available as GADT libraries, and what will the cost of developing such libraries be? Second, how readable and maintainable will these libraries be? Even if GADTs are expressive enough to develop all proofs of interest, there is a risk that they will become the C++ Template Metaprogramming of functional languages.

To systematically investigate the impact of indexed types on software engineering practice, the language design must consider the needs of the proposition language as well as the needs of the computational language. This goal can only be achieved by capitalizing on the knowledge and expertise accumulated in the proof checking community concerning the design of languages for expressing propositions and proofs.

1.1 Contributions

This paper advocates a practical approach to adding indexed types to a full-fledged programming language, and compares this approach to choices made in related languages (Section 2). The ap-

* Supported by NSF ITR-0113569 "Putting Multi-Stage Annotations to Work", Texas ATP 003604-0032-2003 "Advanced Languages Techniques for Device Drivers", and NSF SOD-0439017 "Synthesizing Device Drivers".

proach has been applied to MetaOCaml using Coq proof terms as the indexed type language: a prototype implementation is available online (Section 3).[1] Only the front-end of MetaOCaml needs to be modified, while Coq remains unchanged, thereby preserving the trustworthiness of the proof checking engine. The implementation is backward compatible with OCaml, so all existing OCaml libraries can be used.

For writing programs that take advantage of sophisticated properties of indexed types, the language compares favorably to GADTs (Section 4.1). We argue that Concoqtion allows a more natural style for programming with proofs than Haskell's GADTs, for example, by allowing the definition of index-level functions. We also show how Coq decision procedures can be used to reduce the burden of proof in Concoqtion programs.

As a case study in domain-specific language implementation, we develop a tagless staged interpreter in Concoqtion (Section 5). Tagless staged interpreters (TSI) [20] provide a semantics-based technique for rapidly implementing domain-specific languages in a way that avoids both interpretive overhead and all unnecessary runtime type checking. In particular, type checks are considered unnecessary if the static type system of the domain-specific language ensures that they will never fail at runtime. Compared to previous work [20], an immediate benefit of Concoqtion is that it distinguishes clearly between the parts of the TSI technique that involve a type-theoretic development from those that involve a computational development.

The type safety of MetaOCaml Concoqtion has been addressed in earlier theoretical work by Shao et al. on λ_H [22], and our multi-stage extension $\lambda_H\circ$ [20] for an explicitly typed core calculus. While useful as theoretical proofs of concept, these calculi were never intended to be full-fledged programming languages and lacked full-featured implementations. Technically, MetaOCaml Concoqtion's type system goes further than these works in that it combines index types with Hindley-Milner type inference [19].

2. Concoqtion

We believe that the design of a practical programming language supporting indexed types must meet four key requirements:

1. The language design should not get in the way of standard programming practice. This includes supporting computational effects when that is part of standard practice, as well as providing access to pre-existing computational libraries.
2. Type checking should be decidable.
3. The type language should provide a natural way to express properties of computational values.
4. The programmer should be allowed to express proofs, and to do so in a style that is most appropriate for expressing machine checkable proofs.

We advocate an approach to addressing these goals that consists of the following design choices:

1. Build the new language as an extension of a standard programming language. We refer to this language as either the *host* or *computational language*.
2. Extend the type system of the computational language with a decidable logical framework. To ensure decidability of type checking, the computational and logical languages must be kept separate. The two are tied together through *singleton types* on ground values [32].
3. Use a constructive type theory. A key advantage of this approach is that it provides a natural way for properties and proofs to live in an extension of the world of types for the computational language. It also allows the programmer to define new index types as well as functions on types.
4. Use a standard mechanical proof checking framework for the logical framework. This also means that, in addition to providing access to computational libraries, the type language will provide access to substantial libraries of proofs.

We will call this approach *Concoqtion* to suggest a particular strategy for realizing the approach, namely, by using a well-developed constructive type theory such as Coq [8]. This approach was first used by Shao et al. [22] in the context of certified binaries. In previous work [20] we argued that it is highly suited for the design not just of intermediate languages, but for programming languages as well. Understanding the significance of the particular choices made in this approach requires careful analysis of the interaction between our four requirements, and in particular, two issues:

Effects and decidability: Simplistic combinations of computational features and index types either cause type checking to be undecidable or type safety to be lost. Even if the host language is purely functional (like Haskell), allowing programs in index types would require evaluating programs during type checking, making type checking undecidable. Cayenne [1] chooses to compromise decidability, whereas Epigram [2] introduces termination analysis, changing the expressivity of the host language. If the language has other computational effects, designing a sound type system becomes substantially more involved.

Proof language, expressivity, and decidability: If the programmer does not have a way to express proofs explicitly, this means that the language design depends critically on type checking being able to build these proofs automatically. This implies that either the language of expressible properties is limited, that type checking is undecidable (for example, if the theorem proving engine is complete), or that not all valid properties can be proven. The first approach is suitable for domain-specific applications of dependent types, as is the case in DML and Cyclone. The last two approaches can be problematic if they occur in the context of large scale software development. Thus, it is essential that the programmer be able to express proofs directly. The language must also provide support for doing this in a convenient and practical manner.

Table 1 summarizes how related languages compare along the key dimensions discussed above. A full black circle indicates that the language has the specified property, a white circle indicates that it does not, and a half-circle indicates that our estimate falls somewhere in between.

3. MetaOCaml Concoqtion

We developed a conservative extension to MetaOCaml [18]. MetaOCaml is a multi-stage extension of OCaml [16]. OCaml is a call-by-value, polymorphically typed, higher-order functional language with type inference, side-effects, extensible records, and objects. The extension will be called MetaOCaml Concoqtion, or simply Concoqtion when it is clear from the context that the implementation is what is meant. Concoqtion uses the term language of the theorem prover Coq to define index types, specify index operations, represent their properties and construct proofs. Even in the presence of all OCaml-style effects, type checking in Concoqtion is decidable.

3.1 Extensions to Types

We extend the type system of MetaOCaml with five syntactic extensions: explicit universal quantification, index type expressions, a kind system, an extended form of data-types, and "prooflets".

[1] MetaOCaml Concoqtion release `308_alpha_027C-07` [7].

	Quest	DML	Cayenne	Epigram	Cyclone	MetaD	Omega	Haskell w/ GADT	ATS	RTS1	Concoqtion
Support for computational effects	●	●	◗	○	●	●	●	◗	●	●	●
Decidable type checking	●	●	○	●	●	●	○	●	●	●	●
User-defined index types	◗	○	●	●	○	●	●	◗	●	●	●
Standard property language	○	◗	●	●	◗	●	◗	○	○	○	●
Standard proof language	○	○	●	●	●	●	○	○	○	○	●
Extensive libraries (computational)	○	●	○	○	●	○	○	●	○	○	●
Extensive libraries (logical)	○	○	○	○	○	○	○	○	○	○	●

Table 1. Overview of Design Choices in Related Languages

Universal quantification. Given an OCaml type `t`, Concoqtion has an explicit universal quantifier type `forall a.t`, in the style of System F [11]. These types allow for nested quantification and more expressive notions of polymorphism than available in OCaml.

Index type expressions. A Concoqtion type `'(c)` is an *index type expression,* where `c` is a Coq term. Index type expressions can occur anywhere an OCaml type can. For example, the type `('(10), int) sized_array` is an application of a (binary) type constructor representing arrays indexed by their size to the index `'(10)` and the type `int`.

Kind system. Concoqtion extends the OCaml type system with a System F_ω-style kinds. Thus, the full syntax of the `forall` types is `forall a:k.t`, where `k` is the kind over which the variable `a` ranges. Kinds themselves are just Coq types, and are thus written as `'(c)`. All OCaml types have one kind, called `'(OCamlType)` which may be omitted from the quantifier syntax. Only `OCamlType`-kinded index type expressions can classify OCaml expressions. Quantifying over `'(OCamlType)`s produces first-class parametric polymorphism: type `forall a. '(a) -> '(a)` is the type of the identity function. In addition to `OCamlType`, there are many other kinds that can be used to classify either indices or type-constructors. Polymorphism over index types can be used to specify function invariants. For example, given an array type constructor indexed by its size, we can give the function that copies an array of size `n` the following type:

```
forall n:'(nat).
   ('(n),'a) sized_array -> ('(n),'a) sized_array
```

The kind `OCamlType` is inhabited in Coq by a set of predefined constants. Each such constant is named after the corresponding OCaml type constructor: type `int list -> bool` can be written as `'(OCaml_Arrow (OCaml_list OCaml_int) OCaml_bool)`. The two notations are treated as equivalent by the type system. The embedding of OCaml's types into Coq terms allow us to define Coq functions that map index types into OCaml types. This facility is useful, for example, when developing tagless staged interpreters (Section 5).

Type declarations. Concoqtion extends the OCaml type declarations in two ways. First, parameters of type constructors can range over any specified kind. For example, the following type synonym defines the type of square matrices of size `n`:

```
type (n:'(nat),'a) square_matrix =
  ('(n), ('(n),'a) sized_array) sized_array
```

Second, in algebraic data-type declarations, the OCaml restriction that the result type of each data-constructor must be polymorphic in the type's parameters is relaxed. For example, the OCaml type `'a list` tells us nothing about the structure the list. In Concoqtion, by varying the index parameters in the data-constructor's result type, we can say more about the structure of a value from its type. In the extreme case, this extension allows us to express *singleton* types whose runtime values are fully determined by the type of their indices.

```
type 'b:'(bool) sbool =
| T : '(true) sbool | F : '(false) sbool
```

An expression of type `'(true) sbool` is statically known to be equal to `T`. Now we can write a type which guarantees that a function implements negation as specified by the Coq function `not` on boolean indices: `forall b:'(bool). '(b) sbool -> '(not b) sbool`.

A final extension to data-constructor declarations allows the programmer to declare *locally quantified* type variables. For example, consider the type `listl` of lists whose first parameter is a natural number index indicating its length:

```
type ('n:'(nat),'a) listl =
| Nil : ('(0),'a) listl
| Cons of let 'm:'(nat) in 'a * ('(m),'a) listl
        : ('(m+1),'a) listl
```

The declaration of the data-constructor `Cons` uses a locally quantified variable `m` of kind `nat` and states that given some natural number `m`, an element of type `a`, and a list of length `m`, `Cons` produces a list of length `m+1`.

Prooflets. Concoqtion extends OCaml declarations with the notion of *prooflets*. A prooflet is a Coq Vernacular script (the same language used to interact with the theorem prover) delimited by the keywords `coq` and `end`. Any declarations, definitions or Coq proofs written in the prooflet are added to the Coq environment and visible in the following index type expressions. The most common use of sections is to add definitions of new index types (see an example in Section 5.3). By issuing commands to Coq in the prooflet, the programmer can import any standard or separately compiled Coq library.

Prooflets also allow the programmer to state properties of indices as Coq theorems and then prove them. For example, one might wish to prove that for any type constructor `f` over natural numbers `'(f (m+n))` is equal to `'(f (n+m))`.

The proofs of this and similar properties can be constructed using tactics:

```
coq
Require Arith.
  Lemma comm_eq :
```

```
    forall m n:nat (f: nat->OCamlType),
                   (f(m+n))=(f(n+m)).
    intros; eauto with arith. Qed.
end
```

After stating the lemma comm_eq in prooflets, Coq goes into proof mode. Issuing the tactic `intros; eauto with arith` proves the lemma. At the Vernacular command `Qed`, Coq checks and accepts the theorem, which is then available in the rest of the Concoqtion program as an index-type function named `comm_eq`.

3.2 Extensions to Expressions

Concoqtion extends the syntax of OCaml expressions with appropriate introduction and elimination constructs for the OCaml type extensions described above.

Type abstraction and application. The `forall` types are introduced and instantiated in System F style, using explicit type abstraction and application: `/\a.e` is an expression with type `forall a.t`, where `a` is a variable that may appear in index expressions in `t`; `e .|t|` is an expression of type `t'[a := t]`, where `e` is an expression of type `forall a.t'`. The type variable may also be annotated with a kind, as in `/\n:'(nat). e`, in which case it introduces a kinded forall type `forall n:'(nat).t`.

By analogy to OCaml's function declaration syntax, there is syntactic sugar for writing type abstractions in a let-definition. To distinguish them from expression variables, type variables appearing in let declarations are surrounded with type-application braces:

```
let id .|a| (x:'(a)) = x
```

This notation is syntactic sugar for:

```
let id = /\a. fun (x:'(a)) -> x
```

Data-constructors and pattern matching. Data-constructors that have locally quantified type variables must be fully type-applied in all their type arguments, then applied to any expression arguments they may require. For example, the following function takes an integer and adds it twice to a `listl` increasing its length by two:

```
let add_twice .|m:'(nat)| x xs =
    Cons .|'(1+m)| (x, Cons .|'(m)| (x, xs))
```

Concoqtion has an extended form of `match` expressions datatypes whose indices may vary for each constructor.

```
let rec app .|m:'(nat), n:'(nat)|
  (l1 : ('(m), _) listl)  (l2 : ('(n), _) listl)
  : ('(m+n), _) listl =
 match l1 as ('i:'(nat),'a:'(OCamlType)) listl
          in ('(i+n),'(a)) listl with
 | Nil -> l2
 | Cons .| m2:'(nat)| (x,xs) ->
     Cons .| '(m2+n) | (x, app .|'(m2),'(n)| xs l2)
```

An extended match expression requires two additions. First is a *type pattern*, introduced by the keyword `as`. The type pattern `('i:'(nat),'a:'(OCamlType)) listl` binds the type variables `i` and `a` in the scope of the rest of the match. A type `t`, following the keyword `in` is a *result type annotation*, which may contain free type variables bound by the type pattern. When type-checking the match expression, the type of the discriminated expression `l1` is matched against the type pattern, obtaining a substitution for the type variables. Applying this substitution to the result type annotation gives the result of the whole `match` expression. In each branch of the case, the type pattern is first matched against the type computed for the constructor pattern, obtaining a type substitution for that branch. The type of the body of the branch then must be precisely the result type annotation to which this substitution is applied. This allows each branch to have a different type depending on the types of the indices of the constructor in the branch.

For example, in the `Nil` case, `i` is replaced by `'(0)`, allowing the branch expression to be a list of type `('(0+n),'(a)) listl`. In the `Cons` case, `i` is replaced by `'(1+m2)`. This means that the type of the branch expression must be `('((1+m2)+n),'(a)) listl`. The type computed for the branch expression is `('(1+(m2+n)),'(a)) listl`. By expanding the Coq definitions of `+` the Concoqtion type checker determines that the two types are equal, and accepts the match.

If the type of the discriminated expression is simple enough, the type pattern may be omitted. In particular, this is the case when the parameters of the type are comprised entirely of variables (`'(i)`) and constant index type expressions (`|'(0)|`) In this situation, the Concoqtion type checker can infer the particular substitution binding the type variables to more specific types in each branch. The restriction on the discriminated expression's type is necessary to make computing this substitution decidable – in all other cases the programmer must use the more general type-pattern notation. In practice we find that many functions in Concoqtion can be written using this simpler syntax. Let consider a simple example of omitted type patterns by writing a `zip` function on lists with length.

```
let rec zip .|n:'(nat)|
  (l1:('(n),'a) listl) (l2:('(n),'b) listl)
  : (('(n),('a * 'b)) listl) =
 match (l1,l2) in ('(n),'a*'b) listl with
 | Nil,Nil -> Nil
 | Cons .|i:'(nat)| (x,xs), Cons .|j:'(nat)| (y,ys) ->
     Cons .|'(j)| ((x,y), zip .|'(i)| xs ys)
```

The type of the expression `(l1,l2)` is a pair of lists of length `n`. In the first branch, Concoqtion infers that `n` must be equal to zero, substituting `0` for `n` in the result type annotation when checking the right-hand side.

In the next case, the pattern has the type `('(S i),'a) listl * ('(S j),'b) listl` where the sub-lists `xs` and `ys` are have lengths `'(i)` and `'(j)` respectively. The Concoqtion type checker, concludes that since both `'(S i)` and `'(S j)` must be equal to `n`, `i` and `j` must be equal. This allows us to apply `zip` to xs and ys although the variables representing their length are different.

3.3 Implementation

The Concoqtion compiler extends the full MetaOCaml compiler, which itself extends the OCaml 3.08 compiler through a set of patches that add support for multi-stage programming [29]. An important design feature of the MetaOCaml implementation is that it modifies only the *front end* of the compiler. The same approach was used with Concoqtion: the Concoqtion type-checker produces the same intermediate representation that the OCaml type-checker does, erases all extra type-related annotations, and then invokes the unmodified OCaml back-end compiler to produce an executable program.

Concoqtion uses a stand-alone implementation of the Coq theorem prover as a library accessed by the type-checker. Because Coq and the OCaml compiler are both implemented in OCaml it was possible to compile and link the two together, allowing them to share the same runtime and address space. Thus Coq is a single component of the Concoqtion type-checker. The type-checker acts as a user in a theorem proving session: it issues commands to the Coq infrastructure and queries its global state about constants and theorems.

Coq itself consists of a small secure kernel that provides syntax, reduction, and type- and convertibility- checking of a core Calculus of Inductive Constructions. Around this secure layer are numerous libraries of the theorem prover itself, including support for parsing, interactive theorem proving, management for compilation, and access to libraries of theorems and definitions. To process prooflets, Concoqtion uses the outer theorem prover layer, passing control to the internal Coq interpreter for the Vernacular proof scripts. The Concoqtion type-checker limits itself to a small, well-defined interface to the Coq kernel. No patches or changes to the Coq implementation are needed.

The unification algorithm in the Concoqtion type-checker uses the convertibility checker of the Coq kernel to compare index type expressions for equality. Most of the OCaml type-checker code is unchanged: it does not interact directly with Coq, continuing to rely instead on a modified OCaml unification algorithm. The OCaml unification algorithm is modified to convert between the Coq and OCaml representations of types on the fly, and to perform kind checking when necessary. This way the interaction between Coq and OCaml is isolated to a relatively few places in the OCaml type-checker.

Whenever a new unifiable variable binding is discovered by the OCaml type-checker, this information is communicated to the Coq kernel as a new definitional equality. This allows Coq convertibility checkers and evaluators to use the equalities between OCaml type variables discovered by the OCaml unification engine. The Concoqtion language extensions do impose some additional syntactic burden of type annotations, but the type system of Concoqtion uses the Hindley-Milner inference to propagate some (though not all) redundant annotations.

Supporting separate compilation in Concoqtion requires maintaining a consistent Coq state across compilation boundaries. This is accomplished by ensuring that each Concoqtion compilation unit gives rise to a compiled Coq theory which can be loaded when type-checking other compilation units, or even from a stand-alone Coq application. Prooflets and `OCamlType` constants are organized into Coq modules: the programmer can refer to `OCaml_` constants and theorems with the same naming discipline as in OCaml modules.

4. Programming with Index Types

In this section, we use small examples to compare programming in Concoqtion to programming with GADTs. First, we illustrate the utility of index-level functions on the append example from Section 3.2. We compare Concoqtion and GADTs using the this example. Next, we show how Coq theorems about index types can be used in Concoqtion to type-check more programs. Finally, we demonstrate the features of Concoqtion designed to ease the creation of Coq proofs in Concoqtion programs by harnessing the power of Coq tactics and decision procedures.

4.1 Concoqtion Data-types vs. GADTs

While Concoqtion's extension to algebraic data-types requires data constructors to be type-applied to their parameters, Haskell and GADT languages implicitly reconstruct these type applications using an inference algorithm [21]. However, the inference algorithm that automatically constructs these applications is undecidable in the presence of type-level functions, which are therefore not available in Haskell. Instead, functions over indices must be encoded relationally: a type function from `f : A -> B` is represented by a GADT `R` whose indices are drawn from both `A` and `B`. The values of type `(R x y)` are witnesses that `x = f y`, and need to be manipulated explicitly by Haskell programs. In Concoqtion, the explicit type application of data constructors is the price we pay to allow type-level functions over indices. Below, we compare using Concoqtion's type functions and the Haskell's relational style in programming with index types.

Consider the Concoqtion data-type `listl` for lists of length `n` (Section 3.1). The type constructor `listl` takes two parameters: the first, an index of kind `'(nat)`, is the length of the list; the second is the type of the list element. Appending two lists of length `m` and `n`, respectively, results in a list of length `m+n`. This invariant is captured in the type of the concatenation function:

```
app : forall m,n : '(nat). ('(m),'a) listl
       -> ('(n),'a) listl ->  ('(m+n),'a) listl
```

```
data Z
data S x

data ListN n a where
   Nil  :: ListN Z a
   Cons :: a -> ListN m a -> ListN (S m) a

data Sum m n s where
   SumZ :: Sum Z n n
   SumS :: (Sum a n r) -> Sum (S a) n (S r)

data PlusLenL m n a where
   PP :: (Sum m n sum) -> (ListN sum a) -> PlusLenL m n a

app :: ListN m a -> ListN n a -> PlusLenL m n a
app Nil ys = PP SumZ ys
app (Cons x xs) ys =
    case app xs ys of
    PP sum rest -> PP (SumS sum) (Cons x rest)
```

Figure 1. Lists with length in Haskell

Let us compare the Concoqtion implementation of `app` (Section 3.2) to a similar implementation in Haskell using GADTs [21] (Figure 1, following an example of Sheard's [23]). The data-type `ListN` plays the same role as `listl` in Concoqtion, except that the numeric length indices are encoded as Haskell types built up of type constructors `Z` and `S`. Aside from surface syntactic differences with Concoqtion, in the sub-index `m` in the constructor `Cons` is quantified implicitly in Haskell. Similarly, when constructing values with `Cons` in Haskell, the type application is implicitly reconstructed by the type-checker.

The Concoqtion type of `app` directly expresses the fact that the length of two appended lists is the sum of their length: `('(m+n),'a) listl`. In Haskell, however, we have no way of directly writing down the type index `m+n`. Instead, we need to supply a proof that an index `s` is the sum of `m` and `n`. This proof is encoded in the auxiliary data-type `Sum m n s`: if we can construct a value of type `Sum m n s`, than we have a proof that `m+n = s`. When referring to a list of length `m+n`, we need to define a completely new Haskell data-type: `PlusLenL m n a`.

```
data PlusLenL m n a where
  PP :: (Sum m n sum) -> (ListN sum a) -> PlusLenL m n a
```

This type, `PlusLenL m n a`, bundles up witness that there exists some index `s`, such that `m + n` are equal to `s` together with a list of type `ListN s a`.

Both the Concoqtion and Haskell examples use a kind of GADT for representing lists which are computational data. In Haskell, the programmer is also forced to use GADTs to encode propositions as relations between indices. In Concoqtion, on the other hand, we are free to use GADT-like notation for list values, for which GADTs are well suited, but use the more natural and concise notation of Coq for indices and their properties.

4.2 Using Proofs and Casts

Suppose we wish to call a function in Haskell that took a list of length `m+n`, `(PlusLenL m n a)` but all we have is a list of length `n+m`, `(PlusLenL n m a)`. To use the value available, the programmer needs to explicitly prove that addition is commutative by providing a function of type `Sum m n s->Sum n m s`. Such a function can indeed be built by recursively deconstructing a witness value of type `Sum m n s` and building another of type `Sum n m s`.

What about Concoqtion? Again, suppose we had a value `x` of type `('(m+n),'a) listl`, and what we really need is a value of type `('(n+m),'a) listl`. Somehow, we must use the fact that addition is commutative to convert between the two types. These two types are *not* implicitly convertible (modulo Coq reduction relations) to each other: we will have to prove them equal and use that proof to *cast* from one type to another. To do this we use the type-safe `cast` function, which works for any two types we can prove equal in Coq:

```
forall a,b. forall p:'(a=b). '(a) -> '(b)
```

First, we prove that lists of equal lengths are equal:

```
coq
  Require Import Arith.
  Lemma lemma1 : forall elem, forall m n, (m = n) ->
        ((OCaml_listl m elem) = (OCaml_listl n elem))
   intros; eauto. Qed.
end
```

Next, can combine `lemma1` with a standard Coq library theorem `plus_comm` to obtain the following function:

```
let comm .|a , m:'(nat), n:'(nat)|
    (x:('(m+n),'(a)) listl) : ('(n+m),'(a)) listl =
   cast
     .| '(OCaml_listl (m+n) a),  '(OCaml_listl (n+m) a),
        '(lemma1 a (m+n) (n+m) (plus_comm m n)) | x
```

Note also that, since these proofs and properties live entirely in the logical language, they are erased by the Concoqtion compiler and incur no runtime overhead.

4.3 Using Built-in Decision Procedures

In writing the function `comm` in DML [32], the `cast` would not be necessary, as the equivalence `m+n = n+m` would be proven automatically by a Presburger arithmetic decision procedure that is built into the DML type-checker. In Concoqtion the added burden of proof construction can be reduced by using Coq decision procedures. For example, if the commutativity of addition were not predefined, the `comm` function could prove it on the fly:

```
let comm .|a, m:'(nat), n:'(nat)|
    (x:('(m+n),'(a)) listl) : ('(n+m),'(a)) listl =
   cast .| _^(eauto), _^(eauto), _^(omega;eauto)| x
```

Here we use an alternate form of index expression type, written `^( [ goal| ] script)`. The omitted goal argument specifies a proposition (in Concoqtion this is a *kind*), in this case `('(m+n),_)listl = ('(n+m),_) listl`. This annotation can sometimes be inferred from the context. The `script` argument is a Coq proof script which instructs the theorem prover to use a particular decision procedure or tactic. The above `^(omega;eauto)` instructs Coq to use the Presburger arithmetic decision procedure `omega` together with the standard propositional manipulation package `eauto`.

Concoqtion's advantage over languages with built-in decision procedures lies in support for greater logical expressiveness and graceful degradation. Sometimes the proof that is needed cannot be constructed by any one decision procedure, but can be obtained by applying several such procedures, with minimal, but *necessary* guidance by the user. For example, if the DML-style type-checker had to show that `x*x+2x+1=(x+1)*(x+1)`, it would fail; in Concoqtion this can be proven in Coq as a theorem, added to the standard set of Coq simplification rules, and used by `eauto`.

5. Tagless Staged Interpreters

Indexed types can play a powerful role in the implementation of domain-specific languages (DSLs) [12, 26]. In contrast to C++ Templates and Template Haskell, a multi-stage language like MetaOCaml allows the programmer to implement DSLs as staged interpreters: translators from the DSL to MetaOCaml programs that are both free from the overhead of deconstructing the DSL syntax and statically guaranteed to be type safe [10]. A staged interpreter in MetaOCaml can completely eliminate the interpretive overheard for a language with very simple type structure [26]. But for virtually any domain-specific language with non-trivial types, we encounter the problem of Jones optimality for staged interpreters [14, 27].

Tagless staged interpreters (TSI) provide a superior approach to addressing the superfluous tags that prevent Jones optimality. In particular, when using a multi-stage language with sufficiently expressive indexed types we can statically guarantee that staged interpreters are type and do not produce any unnecessary runtime tags in the resulting computation [20].

Writing a tagless interpreter requires a bit more work than writing tagged one, but we argue that this added work structures the process of implementing a DSL. The tagless staged interpreter approach requires the programmer to follow a sequence of steps that produces semantically well-motivated artifacts one would expect to see in any careful language design. To leverage index types towards a more efficient implementation, the implementation must reflect more of the design information. After a concise overview of the development of a tagless staged interpreter, we summarize the key steps in applying this method to developing an interpreter for the simply-typed λ calculus.

5.1 Overview

Developing a tagless interpreter proceeds first by developing an interpreter that lacks Jones optimality, and thus will have unnecessary tags in the result. This builds the framework and reference for the tagless interpreter:

1. Define a "throw-away" universal domain of tagged values (`val`), representing the results of DSL programs. This step may include definitions of auxiliary types, such as runtime environments (`env`).

2. Define an abstract syntax type `exp` for the DSL.

3. Write an interpreter of type `eval0 : exp -> env -> val`.

Building the tagless interpreter itself requires the following steps:

1. Define a index datatype for types (`typ`) of the DSL. An index datatype is a datatype that lives strictly on the level of types. Because the result type of the tagless interpreter will ultimately depend on this datatype, it must be an index. In Concoqtion, index datatypes are Coq definitions of inductive sets. In this step we also need to define index datatypes for any environments (`tenv`) or stores that are needed to describe well-typed terms.

2. Define the interpretation of DSL types

```
evalT : typ -> OCamlType
```

This is a function at the level of types that maps (syntactic) DSL types to their meaning (Concoqtion types).

3. Define a indexed datatype for the typing derivations of well-typed DSL terms `('e:'(tenv),'t:'(typ)) expr`. A value of this indexed type is a proof that the DSL expression `expr` has the (DSL) type `'t` in the typing environment `'e`.

4. Define a *partial* type-checking function

```
check_expr : exp -> env -> ('(e),'(t)) expr
```

5. Define the tagless interpreter as a function of the following type:

```
eval_expr : forall e:'(tenv) t:'(typ).
        ('(e),'(t)) expr -> '(evalEnv e) -> '(evalT t)
```

6. Stage the interpreter using standard methods [26, 29].

In the rest of this section, we follow the steps above using the simply typed λ-calculus as a small prototype DSL.

5.2 An Interpreter with Tags: `eval0`

The untyped DSL expressions are represented, using de Bruijn indices, with the data-type `exp`. The data-type `val` represents the universal domain of values, containing integers and functions returned by interpreting source programs. For simplicity we represent runtime environments as lists of values.

```
type val = I of int | F of val -> val
type env = val list
type exp = P_Var of int | P_Abs of exp
         | P_App of exp * exp | P_Const of int
```

The tagged interpreter is a function mapping object language expressions `exp` to values `val`:

```
let rec eval0 exp env =
match expr with
| P_Var i -> lookup i env
| P_Const n -> I n
| P_Abs e -> F (fun v -> eval e (v::env))
| P_App(e1,e2) ->
   match eval env e1 with F f -> f (eval env e2)
```

5.3 Types and their Semantics: `typ` and `tenv`

In our example the set of DSL types is an index type defined in Coq. For the simply typed λ-calculus we define the domain `typ` of types, and the domain `tenv` of type assignments

We next provide two Coq functions interpreting object language types and type assignments. The `evalT : typ -> OCamlType` function interprets types by mapping `T_Int` to the OCaml integers and arrows, `T_Arr t1 t2`, to OCaml functions `'(evalT t1) -> '(evalT t2)`. The type assignment interpretation function maps object language type assignments to the OCaml types for runtime environments, which are represented as nested tuples with element types corresponding the variables types in the type assignment.

```
coq
Inductive typ : Set :=
|T_Int  : typ | T_Arr : typ -> typ -> typ.
Inductive tenv: Set :=
| Empty : tenv | Ext   : tenv -> typ -> tenv.

Fixpoint evalT (t:typ)  : OCamlType :=
 match t with
 | T_Int => OCaml_int
 | T_Arr dom cod =>  OCaml_Arr (evalT dom)  (evalT cod)
 end.

Notation "A -*- B" :=
  OCaml_Tuple (OCaml_List_cons A
               (OCaml_List_cons B OCaml_List_nil)).

Fixpoint evalEnv (e:tenv) : OCamlType :=
 match e with
 | Empty => OCaml_unit
 | Ext env t => (evalEnv env) -*- (evalT t) end.
end
```

5.4 Well-typed Expressions: `expr`

Next, we define a Concoqtion data-type that represent typing derivations for *well-typed* expressions. Unlike `typ` and `tenv`, which were purely static (index-level) entities, the typing derivations are defined as Concoqtion data-types. The type constructor `expr` takes two indices: the object-type of the term it represents, and the environment that assigns types to its free variables. To represent well-typed object language variables, we use an auxiliary type constructor `jvar`. These two type constructors correspond to the two judgment forms in Figure 2: each data constructor implements one eponymous derivation rule.

$$
\begin{array}{rcl}
\tau & ::= & \mathrm{int} \mid \tau \to \tau \\
\Gamma & ::= & \cdot \mid \Gamma, \tau \\
e & ::= & \overline{n} \mid \lambda\tau.e \mid e\ e
\end{array}
$$

$$
\frac{}{\Gamma, \tau \vdash 0 : \tau}(\texttt{JV\_Z}) \qquad \frac{\Gamma \vdash n : \tau}{\Gamma, \tau' \vdash (n+1) : \tau}(\texttt{JV\_S})
$$

$$
\frac{\Gamma \vdash n : \tau}{\Gamma \vdash \overline{n} : \tau}(\texttt{Var}) \qquad \frac{\Gamma, \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda e.\tau_1 \to \tau_2}(\texttt{Abs}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau}(\texttt{App})
$$

Figure 2. Type system of the object language

```
type ('e:'(tenv),'t:'(typ)) jvar =
| JV_Z of let 'e1:'(tenv) 't1:'(typ) in unit
     : ('(Ext e1 t1),'(t1)) jvar
| JV_S of let 'e:'(tenv) 't1:'(typ) 't2:'(typ) in
          ('(e),'(t2)) jvar : ('(Ext e t1),'(t2)) jvar

type ('e:'(tenv),'t:'(typ)) expr =
| Const of int : ('e,'(T_Int)) expr
| Var of  ('e,'t) jvar : ('e,'t) expr
| Abs of let 'e:'(tenv) 'dom:'(typ) 'cod:'(typ) in
       ('(Ext e dom),'(cod)) expr
     : ('(e),'(T_Arr dom cod)) expr
| App of let 'e:'(tenv) 'dom:'(typ) 'cod:'(typ) in
       ('(e),'(T_Arr dom cod)) expr * ('(e),'(dom)) expr
     : ('(e),'(cod)) expr
```

In addition to representing typing derivations of DSL programs, we have to have some way of constructing them out of the untyped representation of object language programs.

We define a partial function `check_expr` that takes an `exp` as its input and constructs a typing derivation. In addition to the untyped terms, `check_expr` needs to be able to record types of free variables while it is checking under λ-abstractions. However, the types for the free variables cannot be known before running the type checker. To construct and compare these types, we define two singleton types `'(e:'(tenv)) r_tenv` and `'(t:'(typ)) r_typ`. Similarly, the precise type index for the result of `check_expr` cannot be known before running the type checker. Here, we hide the exact type index with an existential:

```
val check_expr : forall e:'(tenv).
              '(e) r_tenv -> preexp -> '(e) some_expr
```

In Concoqtion, we can encode the needed existential types using extended data-types. This is a standard trick, where an existential type $\exists a.t$ is encoded as a data-type with a locally quantified type variable a of a data-constructor, where a does not occur in its return type:

```
type pretyp = ST of let 't:'(typ) in '(t) r_typ
type ('e:'(tenv)) some_jvar =
| SV of let 't:'(typ) in '(t) r_typ * ('e,'(t)) jvar
type ('env:'(tenv)) some_expr =
| SE of let 't:'(typ) in '(t) r_typ * ('env,'(t)) expr
```

Due to lack of space, we will show only the most interesting case of `check_exp` in Figure 3. First, the application operator and operand are checked recursively to construct their own judgments. Second, we open the two existential packages. Note that we know nothing in particular about the actual type indices of the

```
let rec check_expr .|e:'(tenv)| (re:'(e) r_tenv) pe : '(e) some_expr =
  match pe in '(e) some_expr with
  | P_App(pe1,pe2)  ->
    match (check_expr .|'(e)| re pe1, check_expr .|'(e)| re pe2) in '(e) some_expr with
    | SE .|ratorenv:'(tenv), trator:'(typ)| (trator,jtrator),
      SE .|randenv:'(tenv), trand:'(typ)  | (trand,jtrand)  ->
      (match trator in '(e) some_expr with
       | RArr .|tdom:'(typ),tcod:'(typ)| (rdom,rcod) ->
         let tarr = RArr .|'(trand),'(tcod)| (trand,rcod) in
         let p1 = comp_typ .|'(trator),'(T_Arr trand tcod)| trator tarr in
         let v  = cast_eq_typ .|'(e),'(trator),'(T_Arr trand tcod)| p1 jtrator in
         SE .|'(e),'(tcod)| (rcod, (App .|'(e),'(trand),'(tcod)| (v,jtrand))))
```

Figure 3. Type-checking application.

two well-typed subexpressions. We must explicitly check that the operator is a function and that the operand's type matches its domain. To do this, we use the function `comp_typ` which takes two singleton representations of object language types `'(t1) r_typ` and `'(t2) r_typ` and compares them, either returning a value of type `('(t1),'(t2)) eq_typ` or raising an exception (see Appendix A for definition). This value is a runtime representation of the proof that `t1` and `t2` are equal, and can be used as an argument to the function `cast_eq_typ` to cast from any type containing `t1` to the same type where `t1` is replaced by `t2`. Finally, we apply the cast to put the operator judgment into the form `('(e),'(T_Arr trand tcod)) expr` and construct the typing derivation for the application.

5.5 The Tagless Interpreter: `evalExp`

The tagless interpreter is a function `eval`, parameterized by a type `t` and a type assignment `e`, that takes an object language expression of type `t` under `e`, a runtime environment of type `'(evalEnv e)`, and produces a value of type `'(evalT t)`.

```
forall e:'(tenv). forall t:'(typ).
   ('(e), '(t)) expr -> '(evalEnv e) -> '(evalT t)
```

Figure 4 gives the implementation of the interpreter. First, we define an auxiliary function `lookupVar` that looks up a well-typed variable index in a runtime environment whose type is `'(evalEnv e)`. The interpreter, `evalExp` uses Concoqtion's extended `match` statement to deconstruct the well-typed object language expression. Note that each the return types of each branch of this match expression vary according to the type indices of each data-constructor.

5.6 The Tagless Staged Interpreter

The final step is to stage the interpreter using Concoqtion's Multi-stage programming constructs: program fragments of type `('c, 'a) code` code are constructed using brackets, `.<e>.` which delays the evaluation of the expression `e` of type `'a` until runtime. The first parameter `'c` is an *environment classifier* [28], required for type-safe runtime execution of code fragments. Except for Concoqtion and MetaOCaml, no other multi-stage language has this form of type safety. Inside brackets the programmer can force an expression `e` to be evaluated immediately with the escape construct `.~e`, causing its result, a piece of code, to be "spliced" into the context of the escape. Once constructed, a value of type `(c,t) code` can be executed using the run annotation (`.!e`) to produce a value of type `t`.

Staging the interpreter involves changing its type to return a code value. Moreover, we need to change the interpretation of the type assignments to produce tuples whose elements are of type code. This removes variable lookup overhead from the runtime of the generated program. Note that `evalEnvS` takes an extra parameter, a type `cls`. This parameter used to represent the environment classifier needed to construct a `code` type, and is simply passed along,

```
coq
Fixpoint evalEnvS (cls:OCamlType) (e:tenv) : OCamlType :=
 match e with
 | Empty => OCaml_unit
 | Ext env t =>
     (evalEnvS cls env) -*- (OCaml_code cls (evalT t))
 end.
end
```

The staged version of `evalExp` is shown in Figure 5. `lookupVar` is omitted as it differs only in its type annotations. Aside from the change in type annotations, the only difference between `evalExp` and `evalExpS` is the addition of brackets and escapes. Let us examine one case in more detail.

The abstraction case deconstructs an `Abs` node of type `('(e),'(evalT T_Arr tdom tcod)) expr`. The interpreter immediately constructs a piece of code containing the function abstraction `.< fun v -> ... >.`. The body of this function is constructed and spliced in by a recursive call to `evalExpS` with a runtime environment that is extended with the code value `.<v>.` containing the parameter. Thus, any time the corresponding variable is evaluated in the body, the piece of code containing the parameter `v` will be looked up and spliced in place.

5.7 Discussion

The Tagless Staged Interpreters technique was first described using the language MetaD [20]. We outline the crucial difference between the MetaD and the Concoqtion TSI implementations. While MetaD supports the separation of computational and type languages in principle, it uses the same inductive family facility for both type-level indices and for computational data-types. This renders the separation between indices and programs difficult to perceive. Further, the semantics of functions defined over these inductive structures is different in that only the awkward primitive recursion operators is allowed in type-level functions, and unrestricted recursion is allowed in computational functions. In Concoqtion the language of indices and the language of programs are completely separate: prooflets and index type expressions ensure that the semantics distinctions between indices and programs are syntactically visible.

6. Conclusions and Future Work

We have presented Concoqtion, an approach to designing programming languages with indexed types. We argue that this approach can have significant benefits over GADTs. The approach was applied to MetaOCaml, extending it with highly expressive indexed types provided by the Coq proof checker. Small examples and a case study in tagless staged interpreters are used to illustrate programming in the language.

Naturally, the most important direction for future work is building more applications in MetaOCaml Concoqtion so as to better

```
let rec lookupVar .|e:'(tenv),t:'(typ)| (j:('(e),'(t)) jvar)  (env: '(evalEnv e)) :  '(evalT t) =
  match j in '(evalT t) with
  | JV_Z .|e:'(tenv),t:'(typ)| _ -> snd env
  | JV_S .|e2:'(tenv),t1:'(typ),t2:'(typ)| v' -> lookupVar .|'(e2),'(t2)| v'  (fst env)

let rec evalExp .|e:'(tenv),t:'(typ)| (j: ('(e),'(t)) expr) (env:'(evalEnv e)) : '(evalT t) =
  match j in '(evalT t) with
  | Const n  -> n
  | Var   v -> lookupVar .|'(e),'(t)| v env
  | Abs .|e:'(tenv),td:'(typ),tc:'(typ)| body -> (fun v -> evalExp .|'(Ext e td),'(tc)| body (env,v))
  | App .|e:'(tenv), td:'(typ),tc:'(typ)| (rator,rand) ->
            (evalExp .|'(e),'(T_Arr td tc)| rator env) (evalExp .|'(e),'(td)| rand env)
```

Figure 4. The tagless interpreter

```
let rec evalExpS .|c,e:'(tenv),t:'(typ)| (j: ('(e),'(t)) expr) (env:'(evalEnvS c e)) : ('(c), '(evalT t)) code =
  match j in ('(c), '(evalT t)) code with
  | Const n  -> .< n  >.
  | Var   v -> lookupVarS .|'(c),'(e),'(t)| v env
  | Abs .|e:'(tenv),td:'(typ),tc:'(typ)| body ->
        .< fun v -> .~(evalExpS .|'(c), '(Ext e td),'(tc)| body (env,.< v >.)) >.
  | App .|e:'(tenv), td:'(typ),tc:'(typ)| (rator,rand) ->
        .< .~(evalExpS .|'(c), '(e),'(T_Arr td tc)| rator env) .~(evalExpS .|'(c), '(e),'(td)| rand env) >.
```

Figure 5. Tagless staged interpreter `evalExpS`.

understand the impact of using indexed types. Simultaneously, we wish to address several engineering issues, such as the integration of the OCaml and Coq parsers. This will allow us to improve the concrete syntax of MetaOCaml Concoqtion.

Finally, for the purposes of programming low-level applications using low-level types, we would like to investigate ways to improve support for reasoning about OCaml primitive types. Leroy has already formalized many such types in Coq [17]. These libraries can be imported directly into Concoqtion. What remains to be done is to connect them with the computational language, possibly through special syntax for literals.

References

[1] Lennart Augustsson. Cayenne — a language with dependent types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 239–250. ACM, June 1999.

[2] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2003.

[3] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 431–507. Springer-Verlag, New York, 1991.

[4] Chiyan Chen and Hongwei Xi. Implementing typeful program transformations. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics Based Program Manipulation*, pages 20–28. ACM Press, June 2003.

[5] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 66–77, New York, NY, USA, 2005. ACM Press.

[6] James Cheney and Ralf Hinze. First-class phantom types. Technical Report 1901, Cornell University, 2003.

[7] The MetaOCaml Concoqtion web site. Online at `http://www.metaocaml.org/concoqtion`, July 2006.

[8] The Coq. The coq proof assistant : Reference manual : Version 7.2, February 2002.

[9] Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. In *EUROCAL85*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184, Berlin, 1986. Springer-Verlag.

[10] Krzysztof Czarnecki1, John O'Donnell, Jörg Striegnitz, and Walid Taha. Dsl implementation in metaocaml, template haskell, and c++. In Batory, Consel, Lengauer, and Odersky, editors, *Dagstuhl Workshop on Domain-specific Program Generation*, LNCS. 2004. To appear.

[11] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse de doctorat d'etat, University of Paris VII, 1972.

[12] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.

[13] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10–15, 2002, Monterey, California, USA*, Berkeley, CA, USA, 2002. USENIX.

[14] N. D. Jones, P. Sestoft, and H. Sondergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. Technical Report DIKU Report 87/08, University of Copenhagen, Denmark, 1987.

[15] Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*. ACM Press, 2005.

[16] Xavier Leroy. Objective Caml, 2000. Available from `http://caml.inria.fr/ocaml/`.

[17] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54, New York, NY, USA, 2006. ACM Press.

[18] MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.metaocaml.org/`, 2004.

[19] Emir Pasalic, Jeremy Siek, and Walid Taha. Concoqtion: Mixing dependent types and Hindley-Milner type inference (extended version). Technical report, Rice University, 2006. Available from http://www.metaocaml.org/concoqtion.

[20] Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *the International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002. ACM.

[21] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. submitted to ICFP 2006, April 2006.

[22] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. *ACM SIGPLAN Notices*, 31(1):217–232, January 2002.

[23] Tim Sheard. Languages of the future. *ACM SIGPLAN Notices*, 39(12):119–132, 2004.

[24] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, Cork, Ireland, July 2004.

[25] Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, December 2005.

[26] Walid Taha. A gentle introduction to multi-stage programming. In Don Batory, Charles Consel, Christian Lengauer, and Martin Odersky, editors, *Domain-specific Program Generation*, LNCS. 2004.

[27] Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In Olivier Danvy and Andrzej Filinksi, editors, *Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 257–275, 2001.

[28] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.

[29] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, 1997. ACM Press.

[30] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 224–235, New York, NY, USA, 2003. ACM Press.

[31] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 249–257, Montreal, Canada, 17–19 June 1998.

[32] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of Programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 214–227, New York, NY, USA, 1999. ACM Press.

A. Type-checker auxiliary definitions

```
(* The Checking *)
type ('t:'(typ)) r_typ =
        | RInt :'(T_Int) r_typ
        | RArr of let 'tdom:'(typ) 'tcod:'(typ) in
                          '(tdom) r_typ * '(tcod) r_typ
                        : '(T_Arr tdom tcod) r_typ

type ('e:'(tenv)) r_tenv =
        | Empty : '(Empty) r_tenv
        | Ext of let 'e1:'(tenv) 't:'(typ) in
                           '(e1) r_tenv * '(t) r_typ
                       : '(Ext e1 t) r_tenv

type ('t1:'(typ),'t2:'(typ)) eq_typ =
        Refl_typ of let 'z:'(typ) in unit : ('(z),'(z)) eq_typ

let cast_eq_typ .|f:'(typ -> OCamlType)| .|t1:'(typ),t2:'(typ)|
                (p:('(t1),'(t2)) eq_typ) :'(f t1) -> '(f t2) =
  match p as ('t1:'(typ),'t2:'(typ)) eq_typ
             : '(f t1) -> '(f t2) with
        | Refl_typ .|z:'(typ)| () -> fun x -> x

let combine_arr .|a:'(typ),b:'(typ),c:'(typ),d:'(typ)|
        (x:('(a),'(b)) eq_typ) (y:('(c),'(d)) eq_typ)
                     : ('(T_Arr a c),'(T_Arr b d)) eq_typ =
  match (x,y) as ('a:'(typ),'b:'(typ)) eq_typ
              * ('c:'(typ),'d:'(typ)) eq_typ
            : ('(T_Arr a c),'(T_Arr b d)) eq_typ with
        | Refl_typ .|z1:'(typ)| (), Refl_typ .|z2:'(typ)| () ->
      Refl_typ .|'(T_Arr z1 z2)| ()

let rec comp_typ .|t1:'(typ),t2:'(typ)|
      (t1:'(t1) r_typ) (t2:'(t2) r_typ):(('(t1),'(t2)) eq_typ)  =
      match (t1,t2) in (('(t1),'(t2)) eq_typ)  with
   | RInt, RInt ->  (Refl_typ .|'(T_Int)| ())
   | (RArr .|tdom1:'(typ),tcod1:'(typ)| (rdom1,rcod1),
      RArr .|tdom2:'(typ),tcod2:'(typ)| (rdom2,rcod2))->
           let  p1 = comp_typ .|'(tdom1),'(tdom2)| rdom1 rdom2 in
           let  p2 = comp_typ .|'(tcod1),'(tcod2)| rcod1 rcod2 in
           combine_arr .|'(tdom1),'(tdom2),'(tcod1),'(tcod2)| p1 p2
```