Dependent Tagless Final

Nicolas Biri

Luxembourg Institute of Science and Technology

Esch-sur-Alzette, Luxembourg

nicolas.biri@list.lu

Abstract

Tagless final embedding provides a solution to the expression problem that allows efficient code generation, thanks to multi-staged evaluation. It can, however, be a challenge to compose effectful language fragments in this embedding. This paper proposes a dependent tagless final embedding that uses dependent types to ease the composition of effects. We show that this extension preserves the multi-staging capabilities of tagless final and can help scope effects in domain-specific languages.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; *Incremental compilers*.

Keywords: dependent types, tagless final, staged compilation

ACM Reference Format:

Nicolas Biri. 2022. Dependent Tagless Final. In *Proceedings of the 2022 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM '22), January 17–18, 2022, Philadelphia, PA, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3498886.3502201>

1 Introduction

Writing efficient and type-safe embedded domain-specific languages (DSL) in functional languages is still a challenge, showing that type safety does indeed come at a high price. Fortunately, tagless approaches for DSL and staged compilation are able to address the overhead introduced by type safety and the performance of DSL interpretations, respectively.

In the tagless approach [6, 8, 9], the information ensuring that DSL expressions are well-formed is removed after compilation. This approach has two significant advantages. First,

the type-safety of the DSL is ensured at compilation time, thus avoiding runtime failure due to ill-formed expressions. Second, we do not have to keep the type information until runtime. This results in a lighter memory footprint and makes runtime more efficient. Staged computation [15, 16], or staging, allows interpreters to build terms that will be compiled at a later stage. This approach is advantageous for tagless interpreters as they usually induce code indirections that, thanks to staging, can be handled entirely at compile time.

Although one of the first papers [12] that mentioned staged compilation for DSL interpreters was based on a dependently typed language, it has been proven that tagless final and staging can both be implemented without them [6]. Dependent types may even make compilation harder, as they often blur the distinction between the type information available or required at compile-time and that available at runtime. It becomes less clear whether the type information is used at runtime or not and thus that the proposed solution is tagless.

Idris, a dependently-typed language, has recently introduced quantitative types [1, 3]. These allow developers to check during compilation if a resource (a parameter, a variable, a type) will never be used, will be used only once, or could be used freely (never, once or multiple time) at runtime. Being able to prove that types are only accessed at compile time gives us the guarantee that, despite the use of dependent types, the solution is actually tagless.

Using quantitative types in a dependently-typed language gives us at the same confidence level of confidence regarding the erasure of type information, but do dependent types have more to offer in the adoption of a tagless final approach?

We noticed that dependent types, and specifically when implemented in Idris, can ease the modularity of DSLs built using a tagless final approach. This new way of encoding the tagless final approach improves the reusability of existing DSLs and facilitates the definition of staged compilation interpreters. More precisely, we tackle the composition of polymorphic effects in a DSL, proposing a solution that allows us to add constructs that require extra type parameters to an existing language.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

PEPM '22, January 17–18, 2022, Philadelphia, PA, USA

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9188-7/22/01.

<https://doi.org/10.1145/3498886.3502201>

We can thus summarize the contributions of this paper as follows:

1. We propose an original encoding of the tagless final embedding in Idris, *dependent tagless final*, which the compiler guarantees as tagless, thanks to quantitative types.
2. We show how our approach allows us to extend our language without having to rewrite already existing parts of the language, even when the extension adds effects that were not considered in the initial design of the language.
3. We provide a staged-interpreter for dependent tagless final and illustrate the possibilities of adding rewriting rules to the staged interpreter with this implementation.

This paper is structured as follows. In Section 2, we provide an overview of Idris, focusing on the key features of the language used in this paper. Section 3 presents several examples of DSLs and exhibits the limitations of the standard tagless final encoding on those DSLs. Section 4 presents the tagless final encoding we propose in Idris and details how it bypasses the limitations of the standard encoding. Then, in Section 5, we detail the implementation of a staged interpreter for our encoding and describe how we can implement rewrite rules in this context. Before concluding (Section 7), we compare our approach with other work (Section 6).

2 A Quick Tour of Idris

While the results of this paper are relevant for any dependently-typed language, the implementation proposed is demonstrated in Idris¹. The main reasons for this choice are that Idris guarantees type erasure, and its advanced interface system facilitates the implementation of our ideas.

This section briefly introduces dependent types and the quantitative type theory implemented in Idris, and explains some details of Idris interfaces that we will take advantage of in our implementation.

2.1 Dependent Types

Dependently typed languages are known as such because types can depend on values. The most common example is the definition of a length-indexed list (usually called a *vector*). It is defined in Idris as follows:

```
data Vector : (n : Nat) -> (a : Type) -> Type where
  Nil : Vector Z a
  (::) : a -> Vector n a -> Vector (S n) a
```

The vector type takes two parameters:

- n – a Peano numeral (i.e. using two constructors: Z for 0 and $S\ n$ for the successor of n).
- a – the type of the elements in the vector.

¹<https://www.idris-lang.org>

and it has two constructors. `Nil`, the empty constructor, builds a vector of length 0 (Z), and `(::)` which prepends one element to a vector of length n , creating a vector of length $n + 1$ ($S\ n$).

However, dependent types can go further: types can also be the result of a function, as in:

```
canGetSecret: Bool -> Type
canGetSecret False = Unit
canGetSecret True = String

askSecret : (b : Bool) -> canGetSecret b
askSecret False = ()
askSecret True = "Secret"
```

Depending on the given Boolean parameter, `canGetSecret` returns either the `Unit` type or the `String` type. We can use this function in the signature of `askSecret`, and ensure that the secret can be accessed only if the given Boolean is true. Note how the b parameter of `askSecret` is used in the type result.

We can also do the opposite and use type level values in our expressions:

```
length : {n : Nat} -> (xs : Vector n a) -> Nat
length xs = n
```

Here, we use the indexed value of `Vector`, the length of the said vector as a result of the `length` function.

2.2 Type Quantities

As can be seen through these examples, dependent types blur the distinction between type-level and value-level information. Consequently, it is harder to figure out which compile-time information can be erased at runtime. If we take the `Vector` example, we might consider its `length` as a piece of compile-time information. In this case, we cannot use it in the `length` function. Linearity, or type quantities, is a way of providing developers with more control over the usage of resources. Idris defines three quantities for variables:

- 0 – for variables that are *erased* at runtime;
- 1 – for variables that are used exactly *once* at runtime;
- *Unrestricted* – for variables that can be used without restriction.

While 1 can be very useful for memory allocation and optimization, we will not consider it in this paper. However, we will make intensive use of the 0 quantity to ensure that the tags in our final tagless embedding are erased at runtime.

It is important to understand that Idris implicitly sets some variables' quantities in some declarations: a variable that is bound in a type and that is not explicitly declared has a quantity of 0 implicitly. For example, if `length` was defined with type `(xs : Vector n a) -> Nat`, n would not be explicitly declared, and thus it would not have been accessible at runtime.

Instead, n is declared explicitly with $\{n : \text{Nat}\}$. The curly braces means that the parameter value can be inferred by the compiler. In this example, it means that we don't need to pass the n value explicitly to call `length` (e.g. `length [1, 2, 3]` is a valid function call). This explicit declaration allows us to set a quantity for this variable. In the specific case of `length`, n thus has an unrestricted quantity. It will enable us to use n at runtime and use it as a result of our function.

In the following sections of this paper, many functions or parameters will be preceded by a $\emptyset$ to ensure that these values can only be used at compile time. A typical example is the `const` function, which ignores its second parameter and can be written as:

```
const : a -> ( $\emptyset$  x : b) -> a
const x _ = x
```

2.3 Interfaces²

Interfaces allow the definition of functions that share a common behavior across different datatypes. For people familiar with Haskell, they have some similarities with typeclasses. An example is the `Eq` interface, which allows the definition of the equal function (`==`) for a type:

```
interface Eq a where
  (==) : (x : a) -> (y : a) -> Bool
  (/=) : (x : a) -> (y : a) -> Bool
```

Nevertheless, Idris interfaces have a set of specific features that makes them differ from typeclasses. One of the most important for this paper is that Idris interfaces offer the same possibilities as a datatype. Actually, interfaces in Idris are records with a specific resolution. As a consequence, interface parameters are not restricted to types but can also be values or other interfaces, and interfaces can be passed as parameters. These points will be explained in more detail in Section 4 when we will take advantage of them.

Compared to Haskell interfaces, one major difference is that we can define several instances of a typeclass. This has two significant consequences.

1. To distinguish the different implementations of an interface, we can use *named* implementations, which allow users to refer specifically to a typeclass. For example we can define an `FstEq` instance of `Eq` for pairs that states that two values are equal if and only if their first values are equal:

```
[FstEq] Eq a => Eq (a, b) where
  (==) (x1, x2) (y1, y2) = x1 == y1
  (/=) (x1, x2) (y1, y2) = x1 /= y1
```

2. We can pass a specific typeclass to a function when it requires an interface. If we have a function `allEq : Eq a => List a -> Bool`, and if we want to use this function

with the `FstEq` implementation of `Eq`, we can write: `allEq @{FstEq} [(1, 2), (1, 3)]`. The `@{...}` construct is a way to set the interface required by the function explicitly.

As an interface can have several parameters, it can be helpful to indicate to the compiler which parameters to choose to identify the interface implementation to be used. Idris has the *determining parameters* mechanism to do this, which is close to the functional dependencies in Haskell. The classic example is the reader monad, where we choose the monad instance thanks to the monad type, independently of the type of the read value. To reflect this, we write: `interface MonadRead r (m : Type -> Type) | m` to indicate that m should allow us to determine the implementation to use without ambiguity.

In the context of this paper, it allows us to ask the compiler to determine the interface implementation from the interpreter type.

2.4 Implementation Details

The code in this paper has been tested on a custom version of Idris2³. This version is a fork of Idris2 v0.5.1 that fixes a bug in the metaprogramming facilities offered by Idris2, used in Section 5.

The language features presented in this section are sufficient to understand the code of this paper. Other Idris details will be introduced where necessary.

3 Tagless Final and the Composition Issue

There are two main techniques available to embed a DSL in a functional language, deep and shallow embeddings [8] or, as they are now more generally called, initial and final embeddings. Both can be made tagless: we can use a type annotation, removed at compile-time, to ensure that our language expressions are well-formed.

Tagless final can be implemented in different ways. The most common implementation is an abstract structure, open for implementation (a module in CAML, a typeclass in Haskell, or a trait in Scala). The interpreters are then an implementation of this structure. In Idris, this implementation of tagless final is done through Idris interfaces. In this section, we present the traditional tagless final embedding without taking advantage of dependent types. We use an example to illustrate how this representation can limit the composition of DSLs.

3.1 Tagless Final Through Examples

As an illustration of the challenge of embedding a DSL in a strongly typed language, we will consider a set of DSL fragments that present different typing challenges and look at their compositions.

²<https://idris2.readthedocs.io/en/latest/tutorial/interfaces.html>

³<https://github.com/berewt/idris2/tree/elabVisibility>

We define three language fragments:

1. BaseExpr a minimal language of *pure* expressions (expressions that do not rely on external information)
2. Read, which gives us access to an external value and to carry over the result of an expression to another expression,
3. and Var, which introduces lambda abstractions and function applications.

The objective is to show the difficulty of composing DSLs that require different types of runtime information using tagless final languages. In the rest of the paper, *context* is used to refer to runtime information that is available to the DSLs. In Listing 1, we propose a first tagless final representation of these three DSLs without considering their composition at this stage. This representation is detailed below.

```
interface BaseExpr (0 inter : (0 tag : Type) -> Type) where
  int  : Int -> inter Int
  add  : inter Int -> inter Int -> inter Int
  mult : inter Int -> inter Int -> inter Int
  and  : inter Bool -> inter Bool -> inter Bool
  or   : inter Bool -> inter Bool -> inter Bool
  eq   : Eq a => inter a -> inter a -> inter Bool
  ifThenElse : inter Bool -> inter a -> inter a -> inter a

interface Read
  (0 inter : (0 r : Type) -> (0 tag : Type) -> Type)
  where
    read : inter r -> r
    chain : inter a b -> inter b c -> inter a c

interface Var
  (0 inter : (0 env : List (String, Type)) ->
    (0 tag : Type) -> Type) where
    lam : (lbl : String) ->
      inter ((lbl, a)::env) b -> inter env (a -> b)
    app : inter ctx (a -> b) -> inter ctx a -> inter ctx b
    get : (lbl : String) -> (prf : Elem (lbl, a) env) =>
      inter env a
```

Listing 1. Tagless final embedding for the 3 languages fragments

BaseExpr, the first language we propose, is a basic expression language for integer and boolean values. Written in tagless style, it is an interface that interpreters must implement. The expressions of this DSL do not need to interact with any context to be evaluated. Consequently, the type of the interpreters for this language only need to carry information about the type of expressions. Thus, a type that can implement BaseExpr must be of type: $(0 \text{ tag} : \text{Type}) \rightarrow \text{Type}$. tag, the type of the expression built, is declared with a 0 quantity to ensure its erasure at runtime. This type is named inter and must be erased at compile-time, ending with the $(0 \text{ inter} : (0 \text{ tag} : \text{Type}) \rightarrow \text{Type})$ type. The type of each language construct is straightforward. ifThenElse, for example, takes a condition (inter Bool), and two expressions of type (inter a), to produce an expression of type inter a.

A simple interpreter for this language holds a value of type a for an expression of type inter a. A partial implementation of this evaluator for BaseExpr is given below:

```
data EvalBase : (0 a : Type) -> Type where
  MkBase : a -> EvalBase a
```

EvalBase is the evaluator type. It is parametrised by the type of the expression it evaluates (a) and again, we assign to it a 0 quantity to ensure its erasure at compile time. MkBase is the constructor for this type, taking a value of type a as a parameter. In Idris, a constructor and a type can't have the same name, to avoid ambiguity between the constructor and the type and, by convention, constructor names are usually prefixed with Mk.

An implementation of the BaseExpr interface is now required for the interpreter. It is defined as follows:

```
BaseExpr EvalBase where
  int x = MkBase x
  add (MkBase x) (MkBase y) = MkBase (x + y)
  eq (MkBase x) (MkBase y) = MkBase (x == y)
  ifThenElse (MkBase True) (MkBase t) _ = MkBase t
  ifThenElse (MkBase False) _ (MkBase f) = MkBase f
```

This implementation simply unwraps the values in MkBase and perform the expected computation.

We continue with Read, which allows an expression to access an external value (read); and chain expressions (chain), passing the result of a first expression as an external value to the second. The main interest here is to allow our expression to obtain external information. Aside from the expression's return type, we also need to know the value type that can be accessed. This leads us to the following type for the read interpreters: $(0 \text{ r} : \text{Type}) \rightarrow (\text{tag} : \text{Type}) \rightarrow \text{Type}$. tag, as previously, is the type of the expression, and r the type of value that can be accessed from the context. Both are quantified as 0 to ensure their erasure. The read function type is then inter r r: the interpreter will emit a value of the stored type. Similarly, the context passing is easy to model in chain: inter a b -> inter b c -> inter a c. Given an expression that can read a value of type a to produce a value of type b, and an expression that can read a value of type b to produce a value of type c, we can define an expression that produce a value of type c if it can read a value of type a.

A minimal evaluation interpreter is a function (or, for more convenience a wrapper around a function). Its definition, along with its implementation of the Read interface, is the following:

```
data EvalRead : (0 r : Type) -> (0 tag : Type) -> Type where
  MkRead : (r -> tag) -> EvalRead r tag

Read EvalRead where
  read = MkRead id
  chain (MkRead f) (MkRead g) = MkRead (g . f)
```

Finally, we propose a language that introduces variables, à la de Bruijn. De Bruijn variables are variables that are accessed by their indices in the environment. We propose a

slightly customized version: we name the variables and ask the compiler to find their index by its name (Idris has a way to declare *auto* parameters, provided by the compiler if such values can be built from the constructors and hints).

To embed this language, we need to store the list of available variables in the context. The peculiarity of this language is that we also need to access an external context, to which we also need to add information. For classical de Bruijn variables, the usual representation for this list, in a language without dependent types, is a nested list of pairs. With dependent types, we can represent it a bit more easily as a list. As we want to access variables by their name, this list is a `List (String, Type)`. We must also embed the type of the expressions, and end up with this type for the evaluators: `( $\emptyset$  env : List (String, Type) -> ( $\emptyset$  tag : Type) -> Type)`. The language fragment proposes three constructs:

- `lam` – the lambda abstraction, takes the variable name (`lbl : String`); and an expression that can access this variable (`inter ((lbl, a)::env) b`, to create an expression `inter env (a -> b)`.
- `app` – apply a value to a lambda.
- `get` – the accessor of variables, given a variable name (`lbl : String`), if the compiler can build a proof showing that the variable is in the env (`Elem (lbl, a) env`), we can provide an expression of type `a`.

To build an evaluator for `Var`, we first need to define a datatype for the environment.

```
data Env : List (String, Type) -> Type where
  Empty : Env []
  Cons : a -> Env env -> Env ((lbl, a)::env)
```

The `Empty` environment is an environment with no variables, and we use `Cons` to add values to the environment. We can now continue with the evaluation:

```
data EvalVar : ( $\emptyset$  env : List (String, Type)) ->
  ( $\emptyset$  tag : Type) -> Type where
  MkEvalVar : (Env ts -> a) -> EvalVar ts a

Var EvalVar where
  lam lbl (MkEvalVar g) =
    MkEvalVar $ \env, x => g (Cons x env)
  app (MkEvalVar f) (MkEvalVar x) =
    MkEvalVar $ \env => f env (x env)
  get lbl {prf} = MkEvalVar $ \env => (go prf env)
  where
    go : Elem (l, b) e -> Env e -> b
    go Here (Cons x _) = x
    go (There later) (Cons _ xs) = go later xs
```

We have shown that our three languages can be embedded using tagless final. We have also provided an evaluator for each of them. Now we will see how these language fragments can be composed.

3.2 Tagless Final Composition

If we try to compose the three languages as they are defined in Listing 1, we will incur the problem of the interpreter type. Our three interfaces have different interpreter types that must be unified as a single one.

If we consider only the first two languages, `BaseExpr` and `Read`, we can still find a way to type expressions that rely on them. `EvalRead`, which we defined in the previous subsection, is a valid candidate.

Suppose we want to type the following expression: `chain (eq read (int 2)) (ifThenElse read (int 4) (int 0))`. It leads to the following signature:

```
testBaseRead : {inter : ( $\emptyset$  r : Type) ->
  ( $\emptyset$  tag : Type) -> Type} ->
  ({ $\emptyset$  r : Type} -> BaseExpr (inter r)) =>
  Read inter ->
  inter Int Int

testBaseRead =
  chain (eq read (int 2)) (ifThenElse read (int 4) (int 0))
```

The trick here is to require an implementation of `BaseExpr` for each value of `r`. If we consider `EvalRead`, the evaluator we have built for `Read`, we can implement `BaseExpr` for this type (we show below only a subset of the definition):

```
BaseExpr (EvalRead r) where
  int x = MkRead $ const x
  add (MkRead f) (MkRead g) = MkRead $ \x => f x + g x
  eq (MkRead f) (MkRead g) = MkRead $ \x => f x == g x
  ifThenElse (MkRead c) (MkRead t) (MkRead f) =
    MkRead $ \x => if c x then t x else f x
```

When `Var` is added, things become a little more complicated. The `Var` and `Read` interpreters both have a type annotation to give information about their context (the variables list for `Var`, the type of the provided value for `Read`). To unify the three languages, the interpreter should be both:

- `( $\emptyset$  env : List (String, Type)) -> ( $\emptyset$  r : Type) -> ( $\emptyset$  tag : Type) -> Type,`
- `and ( $\emptyset$  r : Type) -> ( $\emptyset$  env : List (String, Type)) -> ( $\emptyset$  tag : Type) -> Type.`

While the two types are equivalent, we cannot unify them and the earlier method to define an interface for all the parameter values cannot work.

The solution is usually to define upfront a type that embeds the contextual information required by all the interpreters and then define all the interfaces for this context type (as in [6]). The example contains three language fragments. For the three fragments, the interpreters must be parametrised by the return type of expressions. `BaseExpr` do not have further requirement, but an interpreter for `Read` must be parametrised by the type of the value to read, and `Var` must be parametrised by the accessible variables list.

As a consequence, to unify the requirements for these three types, interface must require the interpreters to be of type (or any permutation of these three parameters): $(\emptyset \text{ env} : \text{List} (\text{String}, \text{Type})) \rightarrow (\emptyset \text{ r} : \text{Type}) \rightarrow (\emptyset \text{ tag} : \text{Type}) \rightarrow \text{Type}$.

In Listing 2, we show the result of this approach.

```
interface BaseExpr
  (∅ inter : (∅ env : List (String, Type)) →
    (∅ r : Type) →
    (∅ tag : Type) → Type) where
  int  : Int → inter env r Int
  add  : inter env r Int → inter env r Int →
    inter env r Int
  mult : inter env r Int → inter env r Int →
    inter env r Int
  and  : inter env r Bool → inter env r Bool →
    inter env r Bool
  or   : inter env r Bool → inter env r Bool →
    inter env r Bool
  eq   : Eq a => inter env r a → inter env r a →
    inter env r Bool
  ifThenElse : inter env r Bool →
    inter env r a → inter env r a →
    inter env r a

interface Read (∅ inter : (∅ env : List (String, Type)) →
  (∅ r : Type) →
  (∅ tag : Type) → Type) where
  read : inter env r r
  chain : inter env a b → inter env b c → inter env a c

interface Var (∅ inter : (∅ env : List (String, Type)) →
  (∅ r : Type) →
  (∅ tag : Type) → Type) where
  lam : (lbl : String) →
    inter ((lbl, a)::env) r b → inter env r (a → b)
  app : inter ctx r (a → b) → inter ctx r a →
    inter ctx r b
  get : (lbl : String) → (prf : Elem (lbl, a) env) =>
    inter env r a
```

Listing 2. Unified Tagless final embedding

The interfaces are identical to the one in Listing 1, except that all of them are now defined for interpreters of the aforementioned unified type.

As the three interpreters have the same parameters, they can now be composed seamlessly. It is, however, a significant restriction to the expression problem. Given a base language, we cannot ensure it can be extended if it relies on type parameters that were not included up front.

The following section presents a solution based on dependent types to solve this issue and achieve a genuinely modular tagless final embedding.

4 Dependent Tagless Final

The previous section showed that tagless final, as encoded in non-dependently typed language, is hard to compose when the language constructs have different context requirements.

How do dependent types help in this context? The main idea of our implementation of tagless final in Idris is that an interpreter is contextful and that we can describe the

behavior of this context through interfaces, just as we do for the interpreter itself. The interpreter type is the following:

```
Inter : Type → Type
Inter context = (∅ ctx : context) → (∅ tag : Type) → Type
```

The first parameter of our interpreter is its context type. The context is the type that gathers all the information that the interpreters can access. The idea is similar to that provided by indexed monads [11]: each construct of our DSL can interact with the context and modify it. tag is the return type of our expression.

This representation resembles what we initially proposed for Read and Var in Listing 1. The main difference is that instead of a concrete type, we have a polymorphic type for our context. This will profoundly improve the expressivity we have available to define our context.

In the following parts of this section, we will see how we can use this interpreter type to define the different language fragments we present in the last section and how it helps their composition.

4.1 Dependent Tagless Final

We propose a new encoding for the tagless final approach, dependent tagless final, that fully embraces the use of dependent types. The encoding of the three language fragments, according to this approach, is given in Listing 3. Below, we detail this encoding and explain the implementation choices.

The general principle of the encoding is that all your DSLs interact with a context that can provide and store information at runtime. Each DSL that relies on information of this context must provide a type level interface (ReadContext and VarContext in the example) that explains how it can manipulates this context. These interfaces should be implemented by the *context type* that will carry around contextual information at runtime.

Then we provide the actual interface of the DSL, which is parametrised by the context type (context), the interpreter type (inter) and, if it relies on any contextual information, the type level interface that manipulates the context (r in Read and (v in Var). We can now detail the definition of each DSL.

BaseExpr does not need any information from the context. Consequently, it does not have a type-level requirements interface. Its final tagless encoding is quite close to those in the previous section, except that the interpreter must be an Inter type (defined at the beginning of this section). However, the interpreter must fully determine the interface. For this reason, we use inter as a determining parameter for the interface. The function signatures carry the context over, with no interaction.

```

interface BaseExpr context
  (0 inter : Inter context) | inter where
  int : {0 ctx : context} -> Int -> inter ctx Int
  add : {0 ctx : context} ->
    inter ctx Int -> inter ctx Int -> inter ctx Int
  mult : {0 ctx : context} ->
    inter ctx Int -> inter ctx Int -> inter ctx Int
  and : {0 ctx : context} ->
    inter ctx Bool -> inter ctx Bool -> inter ctx Bool
  or : {0 ctx : context} ->
    inter ctx Bool -> inter ctx Bool -> inter ctx Bool
  eq : {0 ctx : context} ->
    Eq a => inter ctx a -> inter ctx a -> inter ctx Bool
  ifThenElse : {0 ctx : context} -> inter ctx Bool ->
    inter ctx a -> inter ctx a ->
    inter ctx a

interface ReadContext context where
  0 getRead : context -> Type
  0 setRead : Type -> context -> context
  0 getSetId : {0 ctx : context} ->
    getRead (setRead a ctx) = a

interface Read context
  (0 inter : Inter context)
  (0 r : ReadContext context) | inter where
  read : {0 ctx : context} -> inter ctx (getRead @{r} ctx)
  chain : {0 ctx : context} ->
    inter ctx a -> inter (setRead @{r} a ctx) b ->
    inter ctx b

interface VarContext context where
  0 getEnv : context -> List (String, Type)
  0 addVar : (String, Type) -> context -> context
  0 getAddCons : {0 ctx : context} ->
    getEnv (addVar v ctx) = v :: getEnv ctx

interface Var context
  (0 inter : Inter context)
  (0 v : VarContext context) | inter where
  lam : {0 ctx : context} -> (lbl : String) ->
    inter (addVar @{v} (lbl, a) ctx) b ->
    inter ctx (a -> b)
  app : {0 ctx : context} ->
    inter ctx (a -> b) -> inter ctx a -> inter ctx b
  get : {0 ctx : context} -> (lbl : String) ->
    (prf : Elem (lbl, a) (getEnv @{v} ctx)) =>
    inter ctx a

```

Listing 3. Dependent Tagless final embedding

Read is the first interface in our examples to introduce interaction with the context. For the read function, we must get the type of the provided value from the context; for the chain function, we must create a new context with a new read value for the second parameter.

ReadContext provides the two context manipulation functions we need to bring these requirements to life. The two functions are given a 0 quantity to specify that they are only available at compile time.

With these type-level requirements available, we can now type the Read functions. read, given a context ctx, will provide a value of type getRead ctx; for chain, we pass the read value from the function's first parameter to the second using setRead. As Idris allows multiple implementations for the type-level requirements, the specific ReadContext implementation used to implement the Read typeclass in getRead and setRead is explicitly passed with @{r}.

ReadContext includes a third function, getSetId, a proof that if we get the read value type after we set it, the result is the type we have just set. This proof gives us some guarantees about the implementation of ReadContext; its role will be explained later in this section.

We can now apply the same reasoning to the Var typeclass. Var needs two context manipulations: to obtain the list of available variables and to add a variable to this list. Once again, we also include proof, getAddCons. It proves that addVar and getEnv interact as intended: getEnv after addVar returns the newly added variable and the old ones. As for getSetId, this proof ensures that our context implementation is valid. It can be used to rewrite terms that rely on the Var interface. We explain at the end of this section when it is needed. We do not need to remove variables from the list because this is done automatically thanks to variable scope and applications. We group these requirements in the VarContext interface, and use it in the definition of Var. The lam function, takes advantage of addVar to express that its content has access to the variable introduced via the lambda. get uses getEnv to obtain the list of available variables and find the one that should be returned.

4.2 Dependent Tagless Final in Action

Simply having a definition of the interfaces is not sufficient. We now illustrate how to provide an implementation of these interfaces and use them to define expressions.

First, a context type needs to be defined that is able to implement the ReadContext and the VarContext typeclasses. A natural candidate is a pair type, which will contain the read type and variables environment. This type leads to the following definitions:

```

[ReadPair] ReadContext (Type, List (String, Type)) where
  getRead (a, env) = a
  setRead r (a, env) = (r, env)
  getSetId {ctx = (a, env)} = Refl

```

```

[VarPair] VarContext (Type, List (String, Type)) where
  getEnv (a, env) = env
  addVar var (a, env) = (a, var :: env)
  getAddCons {ctx = (a, env)} = Refl

```

ReadPair and VarPair define names for our implementation. This will allow us to refer explicitly to these implementations in the definition of Read and Var.

Ref1 is a standard library constructor for the $a = b$ type. It can be seen as a way to ask the compiler to check if the equality given in the type holds.

Defining the context is the hardest part. Once done, the evaluation interpreter can then be defined, along with its interface implementations, as given in Listing 4.

```
-- We define a type alias for context type
PairCtx : Type
PairCtx = (Type, List (String, Type))
-- The context that will be carried around at runtime
data Context : (0 ctx : PairCtx) -> Type where
  Ctx : a -> Env env -> Context (a, env)
-- The evaluation interpreter
data Eval : (ctx : PairCtx) -> (tag : Type) -> Type where
  MKEval : (Context ctx -> a) -> Eval ctx a

-- Implementation of the DSLs interfaces
Read PairCtx Eval ReadPair where
  read = MKEval $ \(Ctx r e) => r
  chain (MKEval f) (MKEval g) =
    MKEval $ \(Ctx x env) => g (Ctx (f (Ctx x env)) env)
Var PairCtx Eval VarPair where
  lam lbl (MKEval f) = MKEval $ \(Ctx x env), y =>
    f $ Ctx x (Cons y env)
  app (MKEval f) (MKEval g) = MKEval $ \x => f x (g x)
  get lbl = MKEval $ \(Ctx r e) => (lookup prf e)
  where
    lookup : Elem (label, ty) vars -> Env vars -> ty
    lookup Here (Cons x xs) = x
    lookup (There later) (Cons x xs) = lookup later xs
BaseExpr PairCtx Eval where
  int x = MKEval $ const x
  add (MKEval f) (MKEval g) = MKEval $ \x => f x + g x
  eq (MKEval f) (MKEval g) = MKEval $ \x => f x == g x
  mult (MKEval f) (MKEval g) = MKEval $ \x => f x * g x
  and (MKEval f) (MKEval g) = MKEval $ \x => f x && g x
  or (MKEval f) (MKEval g) = MKEval $ \x => f x || g x
  ifThenElse (MKEval c) (MKEval t) (MKEval f) =
    MKEval $ \x => if c x then t x else f x
```

Listing 4. Implementation of dependent tagless final interfaces

The final step is to build expressions using our interfaces. Let us try to add a variable and add its value to the read value to test how the different fragments can be composed:

```
testExpr :
  Read PairCtx inter ReadPair =>
  Var PairCtx inter VarPair =>
  BaseExpr PairCtx inter =>
  inter (Int, env) (Int -> Int)
testExpr = lam "x" (add (get "x") (add read (int 5)))
```

and using the Eval interpreter on this expression is as easy as:

```
testExpr' : Eval (Int, env) (Int -> Int)
testExpr' = testExpr
```

We obtain an expression that is open for interpretation. However, we had to set a context type in this example. One can consider whether we could build expressions that would be

valid for any context as soon as it implements the necessary interfaces. While this method may work, it requires some proof machinery. When the context is abstract, the compiler cannot reduce type-level expressions that rely on functions of the context interfaces. Consequently, we must guide the compiler to find this proof. That is where the getSetId and getAddCons play a role: they allow us to rewrite our terms to obtain the expected ones. This rewriting is out of the scope of this paper.

5 Partial Interpretation and Staging

The tagless final approach makes defining staged interpreters easier [5]. In this section, we show that dependent tagless final preserves this advantage. After explaining how Idris handles metaprogramming, we show how we can build a staged evaluation for the dependent tagless final embedding presented in Section 4. We continue by presenting a rewrite rule that removes an unused effect from expressions to illustrate how dependent tagless final enable the design of such rules.

5.1 Staged Compilation of Dependent Tagless final

5.1.1 Staging in Idris. Idris has an internal language used during the compilation, TTImp, that can be used for metaprogramming. It has a quasi-quotation and an unquoting mechanism to ease the definition of TTImp expressions:

- We can obtain a TTImp from Idris code using a specific pattern: $\sim(\text{expr})$;
- we can inject existing TTImp value in a larger expression with the prefix $\sim$.

Here is an example of the combined use of the two constructs.

```
addTTImp : (x : TTImp) -> (y : TTImp) -> TTImp
addTTImp x y =  $\sim(\sim x + \sim y)$ 
```

From this example, we can see that TTImp is untyped. Fortunately, Idris provides another mechanism for metaprogramming, Elab. Elab is of type $\text{Type} \rightarrow \text{Type}$, and its parameter is the type of the expression that is embedded in Elab.

Idris also provides a check function to build an Elab from TTImp, asserting its type (the assertion will be checked when the Elab is compiled). We can thus move from TTImp to Elab quite easily.

Elab offers much more than typing TTImp expressions. Among others, it provides applicative and monad-like functions. It means that we can embed Idris code into an Elab with pure, and we can use do-notation to manipulate Elab and their content.

We can write an equivalent of the previous addTTImp example in Elab as follows:

```
addElab : (x : Elab Int) -> (y : Elab Int) -> Elab Int
addElab x y = do x' <- x; y' <- y; pure $ x' + y'
```


Once we have a value of type `Elab a`, there are two ways to create a value of type `a` from it. We can annotate the function that produces a value of type `Elab a` with `%macro` and use it as a function of type `a`. We can also use the directive `%runElab` on an `Elab a` value to compile it and obtain a `a` value. Both compute value `a` at compile-time, which means we can safely erase the `Elab` value at runtime. Here is an example with the two approaches and the embedding of `TTImp`, which shows how they can be used and combined at runtime:

```
%macro
twoMacro : Elab Int
twoMacro = addElab (pure 1) (pure 1)
twoRunElab : Int
twoRunElab = %runElab addElab (pure 1) (pure 1)
twoTTImp : Int
twoTTImp = %runElab check (addTTImp `(1) `(1))
combined : Int -- 6
combined = twoMacro + twoRunElab + twoTTImp
```

5.1.2 Staging Interpreters. Consequently, to enable staged compilation for dependent tagless final, one needs to provide an interpreter that can produce an `Elab` for expressions of the language.

Here, we reuse the `(Type, List (String, Type))` context and the interface declarations defined in the previous section. With these definitions, the datatype of our interpreter is the following:

```
data ElabInter : Inter (Type, List (String, Type)) where
  MkElab : (elab : Context ctx -> Elab b) ->
    ElabInter ctx b
```

The implementation of the DSL interfaces is quite similar to the one of the standard (non-staged) evaluator. As for previous interpreters, `BaseExpr` is the most straightforward implementation. It follows the implementation of the `Eval` interpreter, with some metaprogramming boilerplate in between. We provide as an illustration a subset of the implementation:

```
BaseExpr (Type, List (String, Type)) ElabInter where
  int x = MkElab $ const (pure x)
  add (MkElab f) (MkElab g) = MkElab $ \ctx => do
    x <- f ctx
    y <- g ctx
    pure $ x + y
  eq (MkElab f) (MkElab g) = MkElab $ \ctx => do
    x <- f ctx
    y <- g ctx
    pure $ x == y
  ifThenElse (MkElab c) (MkElab t) (MkElab f) =
    MkElab $ \ctx => do
      x <- c ctx
      y <- t ctx
      z <- f ctx
      pure $ if x then y else z
```

`Read` for `ElabInter` is also really close to the value for `Elab`. Again, the main difference is that we have to handle the metaprogramming part:

```
Read (Type, List (String, Type)) ElabInter ReadPair where
  read = MkElab $ \ (Ctx r env) => pure r
  chain (MkElab f) (MkElab g) = MkElab $ \ (Ctx r env) => do
    r' <- f (Ctx r env)
    g (Ctx r' env)
```

The `Var` instance of `ElabInter` requires further explanation. An important point here is that we can benefit from staging to resolve the variables. The lookup, function called by `get` to resolve the variable, performs the resolution outside the `Elab` and include the resolved value in the final `Elab`. The resolution of the variable index is made when we compute the `Elab` value (at compile-time) rather than at runtime. The definition of `lam` is also of interest: it uses `lambda`, an `Elab` function that introduces a lambda abstraction into `Elab`.

```
Var (Type, List (String, Type)) ElabInter VarPair where
  lam x (MkElab body) = MkElab $ \ (Ctx r env) =>
    lambda _ $ \v => body $ Ctx r (Cons v env)
  app (MkElab f) (MkElab x) = MkElab $ \ctx => do
    f' <- f ctx
    x' <- x ctx
    pure $ f' x'
  get x = MkElab $ \ (Ctx r env) => lookup prf env
  where
    lookup : Elem (x, a) env -> Env env -> Elab a
    lookup Here (Cons x xs) = pure x
    lookup (There later) (Cons x xs) = lookup later xs
```

To evaluate the quality of the generated code, we can look back at the `testExpr` expression we have defined at the end of the previous section, interpret it with `ElabInter` and look at the generated code.

```
getElab : Context ctx -> ElabInter ctx a -> Elab a
getElab c (MkElab f) = f c
```

```
elabTestExpr : Int -> Int
elabTestExpr = %runElab (getElab (Ctx 5 Empty) testExpr)
```

`Idris` allows us to check the representation of a term in its REPL via the `:printdef` command. Here is the result:

```
Staged> :printdef elabTestExpr
Staged.elabTestExpr : Int
elabTestExpr = \v => v + (5 + 5)
```

All the function calls and the resolution of the interfaces had occurred at compile-time and we obtained the raw computation as a final result.

However, we could not reduce the `5 + 5` subexpression, while both the `read` value and the constant are available at compile time. The issue is that `add` subexpressions may rely on a lambda abstraction, which we may not resolve at compile time. While it is not a big deal for a single addition, reducing complex expressions that rely on values provided

at compile-time, in general, is an important improvement for code generation. If we could be sure that the subexpressions do not rely on lambda, we should compute them. Fortunately, we can use tagless final rewriting capability and a few dependent types to achieve this.

5.2 Term Rewriting in Dependent Tagless Final

One advantage of tagless final is that it eases efficient code generation by combining staging and term rewriting. To illustrate the rewriting capabilities, we show here how we can switch to an evaluator that does not support lambda abstraction for suitable subexpressions of our language to improve the quality of the generated code.

We start with a staged compilation interpreter for Read and BaseExpr only (we exclude Var). Let's introduce its type first before we discuss it:

```
data ReadElab : (0 r : Type) -> (0 tag : Type) -> Type where
  MkRead : (r -> (a, Elab a)) -> ReadElab r a
-- an accessor for elab
getReadElab : r -> ReadElab r a -> Elab a
getReadElab x (MkRead f) = snd (f x)
```

We build at the same time an `a` and an `Elab a`, to be able to inject the computed value into an `Elab` when we will have to compose sub expressions. The instance of `BaseExpr` illustrates this idea:

```
BaseExpr Type ReadElab where
  int k = MkRead $ \r => (k, check (ttInt k))
  add (MkRead x) (MkRead y) = MkRead $ \r => let
    res = fst (x r) + fst (y r)
    in (res, check (ttInt res))
  eq (MkRead x) (MkRead y) = MkRead $ \r => let
    res = fst (x r) == fst (y r)
    in (res, check (ttBool res))
  ifThenElse (MkRead c) (MkRead t) (MkRead f) =
    MkRead $ \r => if fst (c r) then t r else f r
```

In `int`, we show how we build constants: we return them, along with the `Elab`, as intended (`ttInt` is a custom helper to build a `TTImp` for an integer value). In `add` and other operators, we apply the context to the subexpressions, discard the computed `Elab`, build a new result with the newly computed value, and inject it into a new `Elab`. The computations are performed outside of the `Elab`, and we inject their result into it when needed.

We can check that it's working as intended on a small arithmetic example, that we try to interpret with the original `ElabInter` and with `ReadElab`:

```
testAddEq : BaseExpr context inter => inter ctx Bool
-- (4 + 3) == (3 * 2)
testAddEq = eq (add (int 4) (int 3)) (mult (int 3) (int 2))
elabAddEq : Bool
elabAddEq = %runElab getElab (Ctx () Empty) testAddEq
readElabAddEq : Bool
readElabAddEq = %runElab getReadElab () testAddEq
```

If we compare the code generated by the `ElabInter` and the `ReadElab` interpreters, we obtain the following results:

```
Staged> :printdef elabAddEq
Staged.elabAddEq : Bool
elabAddEq = (4 + 3) == (3 * 2)
Staged> :printdef readElabAddEq
Staged.elabAddEq : Bool
elabAddEq = False
```

Even if the type of `ReadElab` is erased at runtime, we can produce a specialized version of `Elab` for `Int` and `Bool` expressions built with our DSL.

As expected, while our original elaborator interpreter keeps the AST untouched, `ReadElab` computed the expression value. We continue with the definition of `Read`. Our context will be a single type, as we do not need any other information, and it leads to the following code:

```
[ReadType] ReadContext Type where
  getRead = id
  setRead = const
  getSetId = Refl
Read Type ReadElab ReadType where
  read = MkRead $ \x => (x, pure x)
  chain (MkRead f) (MkRead g) = MkRead $ \r => g $ fst $ f r
```

Now that we have an interpreter that evaluates subterms that do not rely on `Var`, we need to use it where it is needed. As we may want to keep interpretation open, we want to be able to perform this transformation on any interpreter that shares the same context as `ElabInter` and `ReadElab`.

We can summarize it as follows: we have two interpreters that operate on different contexts, one being the projection of the other. We want to embed the interpreter that relies on the projected context into the one that operates on the initial one.

The issue here is that this embedding requires a relation to know, given a projected context, what was the original context. This relation is defined a type that proves that a context is a projection of another context. Here is how it is done for `(Type, List (String, Type))` and `Type`:

```
data Rel : (Type, List (String, Type)) -> Type -> Type where
  MkRel : Rel (r, env) r
```

Given a source context `contextSrc` and a target context `contextTgt`, our relation has a type `contextSrc -> contextTgt -> Type`. Given this relation and two interpreters (respectively for `contextSrc` and `contextTgt`), we can define an `Embed` interface and its implementation for `ElabInter` and `ReadElab` as follows:

```
interface Embed
  (0 src : Inter contextSrc) (0 tgt : Inter contextTgt)
  (0 rel : contextSrc -> contextTgt -> Type) where
  embed : tgt ctxT a -> (0 prf : rel ctx ctxT) => src ctx a
```

```
Embed ElabInter ReadElab Rel where
  embed (MkRead f) {prf = MkRel} =
    MkElab $ \ (Ctx r env) => snd (f r)
```

Now we need to define an interpreter that allows us to jump between the two interpretations, and to choose whether we delegate the expressions to the lower interpreter or not. It leads us to the following definition:

```
data Delegate :
  (0 src : Inter contextSrc) ->
  (0 tgt : Inter contextTgt) ->
  (prj : contextSrc -> contextTgt) ->
  (rel : contextSrc -> contextTgt -> Type) ->
  Inter contextSrc where
NoDelegate :
  {0 ctx : contextSrc} -> {0 src : Inter contextSrc} ->
  src ctx a -> Delegate src tgt prj rel ctx a
WithDelegate :
  {0 ctx : contextSrc} -> {0 tgt : Inter contextTgt} ->
  tgt (prj ctx) a -> (prf : rel ctx (prj ctx)) ->
  Delegate src tgt prj rel ctx a
```

Delegate takes four parameters: the two interpreters, the projection function `prj` that projects the source context to the target context and the relation between the two contexts. `NoDelegate` embeds an expression of the source interpreter. In contrast, `WithDelegate` embeds an expression of the target interpreter on the projected context and a proof to get back the original context from its projection.

The underlying idea is to use `WithDelegate` as much as possible. When we hit a construct that cannot be embedded in the target interpreter, we switch to `NoDelegate`. The switch is performed thanks to the following function:

```
asSrc : Embed src tgt rel =>
  Delegate src tgt prj rel ctx a -> src ctx a
asSrc (NoDelegate x) = x
asSrc (WithDelegate x prf) = embed {rel} x
```

Now that we have all the generic pieces to embed a language into another one, we can go back to the initial optimization issue and define the `Delegate` instances that will compute the constant values at compile time. These instances are given in Listing 5. The key points of these definitions are the following: `int`, in `BaseExpr`, and `read`, in `Read`, both introduce a delegation of the computation to `ReadElab`. On the contrary, each function of `Var` switches back to `ElabInter`, as `ElabInter` is the only interpreter that provides the `Var` interface. We stick to `ReadElab` as far as possible for all the other functions, switching back to `ElabInter` if it is required to compute one of the subexpressions.

Now, the interpreter provides now a reduced evaluation of our test expression:

```
getDelegate : Context ctx ->
  Delegate ElabInter ReadElab
  (getRead @{ReadPair}) Rel ctx a -> Elab a
```

```
liftDelegate2 : Embed src tgt rel =>
  (src ctx a -> src ctx b -> src ctx c) ->
  (tgt (prj ctx) a -> tgt (prj ctx) b -> tgt (prj ctx) c) ->
  Delegate src tgt prj rel ctx a ->
  Delegate src tgt prj rel ctx b ->
  Delegate src tgt prj rel ctx c
liftDelegate2 f f' (NoDelegate x) y =
  NoDelegate (f x (asSrc y))
liftDelegate2 f f' x (NoDelegate y) =
  NoDelegate (f (asSrc x) y)
liftDelegate2 f f' (WithDelegate x z)
  (WithDelegate y prf) =
  WithDelegate (f' x y) z
```

```
Embed src tgt Rel =>
BaseExpr PairCtx src => BaseExpr Type tgt =>
BaseExpr PairCtx
  (Delegate src tgt (getRead @{ReadPair}) Rel) where
  int k {ctx = (r, env)} = WithDelegate (int k) MkRel
  add x y = liftDelegate2 add add x y
  mult x y = liftDelegate2 mult mult x y
  eq x y = liftDelegate2 eq eq x y
  and x y = liftDelegate2 and and x y
  or x y = liftDelegate2 or or x y
  ifThenElse (NoDelegate c) t f =
    NoDelegate $ ifThenElse c (asSrc t) (asSrc f)
  ifThenElse c (NoDelegate t) f =
    NoDelegate $ ifThenElse (asSrc c) t (asSrc f)
  ifThenElse c t (NoDelegate f) =
    NoDelegate $ ifThenElse (asSrc c) (asSrc t) f
  ifThenElse (WithDelegate c prf)
    (WithDelegate t _)
    (WithDelegate f _) =
    WithDelegate (ifThenElse c t f) prf
```

```
Embed src tgt Rel => Read PairCtx src ReadPair =>
Read Type tgt ReadType => Read PairCtx
  (Delegate src tgt (getRead @{ReadPair}) Rel)
  ReadPair where
  read {ctx = (r, env)} = WithDelegate read MkRel
  chain (NoDelegate x) y = NoDelegate (chain x (asSrc y))
  chain x (NoDelegate y) = NoDelegate (chain (asSrc x) y)
  chain (WithDelegate x _)
    (WithDelegate y _) {ctx = (r, env)} =
    WithDelegate (chain x y) MkRel
```

```
Embed src tgt Rel => Var PairCtx src VarPair =>
Var PairCtx (Delegate src tgt (getRead @{ReadPair}) Rel)
  VarPair where
  lam lbl body = NoDelegate (lam lbl (asSrc body))
  app x y = NoDelegate (app (asSrc x) (asSrc y))
  get lbl = NoDelegate (get lbl)
```

Listing 5. Delegation instances for constant computation

```
getDelegate c x = getElab c (asSrc x)
delegateTestExpr : Int -> Int
delegateTestExpr =
  %runElab getDelegate (Ctx 5 Empty) testExpr

Staged> :printdef delegateTestExpr
\v => v + 10
```

We were able to define the whole transformation without having to specify the interpreters explicitly. We provided a way to embed an interpreter in another one with a broader context, generically.

6 Related Work

From a language design perspective, our work is directly related to tagless final embedding [6], as mentioned in Section 3. As explained in Section 3, one limitation we faced

initially with tagless final and that we address with the dependent type version is the possibility of composing languages with different context requirements. In the following sections, we proved that switching to the dependently typed representation does not sacrifice the staging and term rewriting facilities offered by the original implementation of tagless final (and brilliantly illustrated in [10, 14]). While our use of dependent types is mostly to ease the composition of language fragments; and build generic transformation primitives, it could be interesting to investigate how dependent types could help optimize the generated code.

In [13], the specific problem of language composition in the case of lambda abstraction is studied. The authors point out the difficulty of managing lambda abstractions when they are composed with other effects. We did not go as far in the interaction of lambda with other effects in this paper. While the authors use a tagless initial encoding, this encoding can be connected to the tagless final representation [8]. It could be interesting to explore how dependently tagless final can adopt the effect handlers solution they proposed.

The flexibility we obtain with dependent tagless final can also be compared with effects systems and their parametric (indexed) monad implementation [11]. Parametric monads provides an effect interface that enables the decoration of a monad with effects. There is a similarity in attaching a context to the monadic computation to store and get information. Parametric effects provide a way to extend an existing monad with effects, at a cost of an significant typeclass machinery to manage the context manipulation. In comparison, the use of dependent types drastically reduces the implementation complexity. Closer to the dependent type world, we can compare our approach to implementing effects using Idris dependent types [2]. The effect approach in Idris provides a nice way to control resource usage and is also obtained by decorating the initial computation with contextual elements. In both cases, dependent tagless final borrows the idea of decorating the computation with a context that provides contextual information during the computation. We do differ on the final objective, however. The focus on embedded DSL has different requirements; we do not have to deal with effects introduction and removal, but we need to provide a rewriting mechanism that is not central for effects.

Regarding staging, we benefit a lot from work performed on elaborator reflection in Idris [7], which has been reworked and improved with the release of the second version of Idris. The elaborator reflection was initially designed for proof elaboration but it provides a great toolbox for metaprogramming. It can elegantly be mixed with the tagless final style. The staging compilation solution we presented take advantage of some of the partial evaluation facilities offered in Idris2. However, we did not take advantage of the %spec directive in Idris2, which implements the partial evaluation

for dependent types proposed in [4]. Doing so might have allowed us to provide an even closer representation for the naive and staged interpreters.

7 Conclusion

We have explained how the traditional tagless final embedding using interfaces can restrict the composition of different effects and proposed a solution to this problem relying on dependent types. We have then shown that the introduction of dependents type does not affect evaluation and staged computation. The solution adopts the tagless final approach at the value level and type level, allowing us to compose type-level requirements as we compose type-level constructs.

Our solution also eases the definition of rewriting rules that reason about the context requirements and that allows us to reason on a selected subset of effects for appropriate subexpressions. We have illustrated how these rewriting rules can improve the quality of the generated code via an example. Introducing a distinction between the context and the language constructs facilitates the definition of compilation stages that manipulates the context. Defining interpreters for an abstract context still requires type and proof manipulations that make the approach less suitable for mainstream development. Future work may explore dealing with these proof requirements to facilitate the definition of genuinely parametric expressions in our language.

Acknowledgments

The author would like to thank the Idris' discord community for their help on the technical part and William E. Byrd for constructive criticism of the manuscript.

References

- [1] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (2018), 56–65. <https://doi.org/10.1145/3209108.3209189>
- [2] Edwin Brady. 2015. Resource-Dependent Algebraic Effects. *Trends in Functional Programming* (2015), 18–33.
- [3] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. *35th European Conference on Object-Oriented Programming (ECOOP 2021)* 194 (2021), 9:1–9:26. <https://doi.org/10.4230/LIPIcs.ECOOP.2021.9>
- [4] Edwin C. Brady and Kevin Hammond. 2010. Scrapping Your Inefficient Engine: Using Partial Evaluation to Improve Domain-Specific Language Implementation. *SIGPLAN Not.* 45, 9 (Sept. 2010), 297–308. <https://doi.org/10.1145/1932681.1863587>
- [5] Jacques Carette and Oleg Kiselyov. 2011. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Science of Computer Programming* 76, 5 (2011), 349–375. <https://doi.org/10.1016/j.scico.2008.09.008> Special Issue on Generative Programming and Component Engineering (Selected Papers from GPCE 2004/2005).
- [6] Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543. <https://doi.org/10.1017/S0956796809007205>

- [7] David Christiansen and Edwin Brady. 2016. Elaborator Reflection: Extending Idris in Idris. *SIGPLAN Not.* 51, 9 (Sept. 2016), 284–297. <https://doi.org/10.1145/3022670.2951932>
- [8] Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). *SIGPLAN Not.* 49, 9 (Aug. 2014), 339–347. <https://doi.org/10.1145/2692915.2628138>
- [9] Oleg Kiselyov. 2012. Typed Tagless Final Interpreters. *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures* (2012), 130–174. https://doi.org/10.1007/978-3-642-32202-0_3
- [10] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. *SIGPLAN Not.* 52, 1 (Jan. 2017), 285–299. <https://doi.org/10.1145/3093333.3009880>
- [11] Dominic Orchard and Tomas Petricek. 2014. Embedding Effect Systems in Haskell. *SIGPLAN Not.* 49, 12 (Sept. 2014), 13–24. <https://doi.org/10.1145/2775050.2633368>
- [12] Emir Pašalić, Walid Taha, and Tim Sheard. 2002. Tagless Staged Interpreters for Typed Languages. *SIGPLAN Not.* 37, 9 (Sept. 2002), 218–229. <https://doi.org/10.1145/583852.581499>
- [13] Casper Bach Poulsen, Cas van der Rest, and Tom Schrijvers. 2021. Staged Effects and Handlers for Modular Languages with Abstraction. *Workshop on Partial Evaluation and Program Manipulation (PEPM)* (2021).
- [14] Amir Shaikhha and Lionel Parreaux. 2019. Finally, a Polymorphic Linear Algebra Language (Pearl). *33rd European Conference on Object-Oriented Programming (ECOOP 2019)* 134 (2019), 25:1–25:29. <https://doi.org/10.4230/LIPIcs.ECOOP.2019.25>
- [15] Tim Sheard, Zine-el-abidine Benaissa, and Emir Pasalic. 2000. DSL Implementation Using Staging and Monads. *Proceedings of the 2nd Conference on Domain-Specific Languages* (2000), 81–94. <https://doi.org/10.1145/331960.331975>
- [16] Walid Taha. 2004. A gentle introduction to multi-stage programming. *Domain-Specific Program Generation* (2004), 30–50.