Tagless Staged Interpreters for Typed Languages

Emir Pašalić*
OGI School of Science & Engineering
Oregon Health & Science University
pasalic@cse.ogi.edu

Walid Taha†
Computer Science Department
Rice University
taha@cs.rice.edu

Tim Sheard‡
OGI School of Science & Engineering
Oregon Health & Science University
sheard@cse.ogi.edu

Abstract

Multi-stage programming languages provide a convenient notation for explicitly staging programs. Staging a definitional interpreter for a domain specific language is one way of deriving an implementation that is both readable and efficient. In an untyped setting, staging an interpreter "removes a complete layer of interpretive overhead", just like partial evaluation. In a typed setting however, Hindley-Milner type systems do not allow us to exploit typing information in *the language being interpreted*. In practice, this can mean a slowdown cost by a factor of three or more.

Previously, both type specialization and tag elimination were applied to this problem. In this paper we propose an alternative approach, namely, expressing the definitional interpreter in a dependently typed programming language. We report on our experience with the issues that arise in writing such an interpreter and in designing such a language.

To demonstrate the soundness of combining staging and dependent types in a general sense, we formalize our language (called Meta-D) and prove its type safety. To formalize Meta-D, we extend Shao, Saha, Trifonov and Papaspyrou's λH language to a multi-level setting. Building on λH allows us to demonstrate type safety in a setting where the type language contains all the calculus of inductive constructions, but without having to repeat the work needed for establishing the soundness of that system.

General Terms

Languages, Performance

Keywords

Multi-stage programming, definitional interpreters, calculus of constructions, domain-specific languages.

*Supported by NSF CCR-0098126.

†Supported by NSF ITR-0113569. This work was done while the author was at Yale University.

‡Supported by NSF CCR-0098126.

ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.

1 Introduction

In recent years, substantial effort has been invested in the development of both the theory and tools for the rapid implementation of domain specific languages (DSLs) [4, 22, 40, 47, 45, 23]. DSLs are formalisms that provide their users with a notation appropriate for a specific family of tasks. A promising approach to implementing domain specific languages is to write a definitional interpreter [42] for the DSL in some meta-language, and then to stage this interpreter either manually, by adding explicit staging annotations (multi-stage programming [55, 30, 45, 50]), or by applying an automatic binding-time analysis (off-line partial evaluation [25]). The result of either of these steps is a *staged interpreter*. A staged interpreter is essentially a *translation* from a subject-language (the DSL) to a target-language[1]. If there is already a compiler for the target-language, the approach yields a simple compiler for the DSL. In addition to the performance benefit of a compiler over an interpreter, the compiler obtained by this process often retains a close syntactic connection with the original interpreter, inspiring greater confidence in its correctness.

This paper is concerned with a subtle but costly problem which can arise when *both* the subject- and the meta-language are statically typed. In particular, when the meta-language is typed, there is generally a need to introduce a "universal datatype" to represent values uniformly (see [48] for a detailed discussion). Having such a universal datatype means that we have to perform tagging and untagging operations at run time. When the subject-language is untyped, as it would be when writing an ML interpreter for Scheme, the checks are really necessary. But, when the subject-language is also statically typed, as it would be when writing an ML interpreter for ML, the extra tags are not really needed. They are only necessary to *statically type check the interpreter*. When this interpreter is staged, it inherits [29] this weakness, and generates programs that contain *superfluous tagging and untagging operations*. Early estimates of the cost of tags suggested that it produces up to a 2.6 times slowdown in the SML/NJ system [54]. More extensive studies in the MetaOCaml system show that slowdown due to tags can be as high as 10 times [21].

How can we remove the tagging overhead inherent in the use of universal types?

One recently proposed possibility is tag elimination [54, 53, 26], a transformation that was designed to remove the superfluous tags in a post-processing phase. Under this scheme, DSL implementation is divided into *three* distinct stages (rather than the traditional two). The extra stage, *tag elimination*, is distinctly different from the traditional partial evaluation (or specialization) stage. In essence, tag elimination allows us to type check the subject pro-

[1]Explicit staging in a multi-stage language usually implies that the meta-language and the target-language are the same language.

gram after it has been transformed. If it checks, superfluous tags are simply erased from the interpretation. If not, a "semantically equivalent" interface is added around the interpretation. Tag elimination, however, does not *statically* guarantee that all tags will be erased. We must run the tag elimination at runtime (in a multi-stage language).

In this paper, we study an alternative approach that does provide such a guarantee. In fact, the user never introduces these tags in the first place, because the type system of the meta-language is strong enough to avoid any need for them.

In what follows we describe the details of superfluous tags problem.

1.1 Untyped Interpreters

We begin by reviewing how one writes a simple interpreter in an untyped language.[2] For notational parsimony, we will use ML syntax but disregard types. An interpreter for a small lambda language can be defined as follows:

```
datatype exp = B of int | V of string
             | L of string * exp | A of exp * exp

fun eval e env =
 case e of
   B i     => i
 | V s     => env s
 | L (s,e) => fn v => eval e (ext env s v)
 | A (f,e) => (eval f env) (eval e env)
```

This provides a simple implementation of subject programs represented in the datatype exp. The function eval evaluates exps in an environment env that binds the free variables in the term to values.

This implementation suffers from a severe performance limitation. In particular, if we were able to inspect the result of a an interpretation, such as (eval (L("x",V "x")) env0), we would find that it is equivalent to

(fn v => eval (V "x") (ext env0 "x" v)).

This term will compute the correct result, but it contains an unexpanded recursive call to eval. This problem arises in both call-by-value and call-by-name languages, and is one of the main reasons for what is called the "layer of interpretive overhead" that degrades performance. Fortunately, this problem can be eliminated through the use of staging annotations [48].

1.2 Untyped Staged Interpreters

Staging annotations partition the program into stages. Brackets .<_>. surrounding an expression lift it to the next stage (building code). Escape .~_ drops its surrounded expression to a previous stage (splicing in already constructed code to build larger pieces of code), and should only appear within brackets. Staging annotations change the evaluation order of programs, even evaluating under lambda abstraction, and force the unfolding of the eval function at code-generation time. Thus, by just adding staging annotations to the eval function, we can change its behavior to achieve the desired operational semantics:

```
fun eval' e env =
 (case e of
   B i     => .<i>.
 | V s     => env s
 | L (s,e) => .<fn v => .~(eval' e (ext env s .<v>.))>.
 | A (f,e) => .<.~(eval' f env) .~(eval' e env))>.
```

Computing the application eval' (L("x",V "x")) env0 directly yields a term .<fn v => v>.

Now there are no leftover recursive calls to eval. Multi-stage languages come with a run annotation .!_ that allows us to execute such a code fragment. A staged interpreter can therefore be viewed as user-directed way of reflecting a subject program into a meta-program, which then can be handed over in a type safe way to the compiler of the meta-language.

1.3 Hindley-Milner Staged Interpreters

In programming languages, such as Haskell or ML, which use a Hindley-Milner type system, the above eval function (staged or unstaged) is not well-typed [48]. Each branch of the case statement has a different type, and these types cannot be reconciled.

Within a Hindley-Milner system, we can circumvent this problem by using a universal type. A universal type is a type that is rich enough to encode values of all the types that appear in the result of a function like eval. In the case above, this includes function as well as integer values. A typical definition of a universal type for this example might be:

```
datatype V = I of int | F of V -> V
```

The interpreter can then be rewritten as a well-typed program:

```
fun unF (F v) = v

fun eval e env =
 (case e of
   B i     => I i
 | V s     => env s
 | L (s,e) => F (fn v => eval e (ext env s v))
 | A (f,e) => (unF (eval f env)) (eval e env));
```

Now, when we compute (eval (L("x",V "x")) env0) we get back a value

(F (fn v => eval (V "x") (ext env0 "x" v))).

Just as we did for the untyped eval, we can stage this version of eval.

Now computing (eval (L("x",V "x")) env0) yields:

.<(F (fn v => v))>.

1.4 Problem: Superfluous Tags

Unfortunately, the result above still contains the tag F. While this may seem like minor issue in a small program like this one, the effect in a larger program will be a profusion of tagging and untagging operations. Such tags would indeed be necessary if the subject-language was untyped. But if we know that the subject-language is statically typed (for example, as a simply-typed lambda calculus) the tagging and untagging operations are really not needed. Benchmarks indicate that these tags add a 2-3 time overhead [54], sometimes as large as 3-10 times [21].

There are a number of approaches for dealing with this problem. None of these approaches, however, guarantee (at the time of writing the staged interpreter) that the tags will be eliminated before runtime. Even tag elimination, which guarantees the elimination of tags for these particular examples, requires a separate meta-theoretic proof *for each subject language* to obtain such a guarantee [54].

1.5 Contributions

In this paper we propose an alternative solution to the superfluous tags problem. Our solution is based on the use of a dependently typed multi-stage language. This work was inspired by work on writing dependently typed interpreters in Cayenne [2]. To illustrate viability of combining dependent types with staging, we have designed and implemented a prototype language we call Meta-D. We

[2] Discussing the issue of how to prove the adequacy of representations or correctness of implementations of interpreters is beyond the scope of this paper. Examples of how this can done can be found elsewhere [54].

use this language as a vehicle to investigate the issues that arise when taking this approach. We built a compiler from an interpreter, from beginning to end in Meta-D. We also report on the issues that arise in trying to develop a dependently typed programming language (as opposed to a type theory).

Meta-D features

- Basic staging operators
- Dependent types (with help for avoiding redundant typing annotations)
- Dependently typed inductive families (dependent datatypes)
- Separation between values and types (ensuring decidable type checking)
- A treatment of equality and representation types using an equality-type-like mechanism

The technical contribution of this paper is in formalizing a multi-stage language, and proving its safety under a sophisticated dependent type system. We do this by capitalizing on the recent work by Shao, Saha, Trifonov and Papaspyrou's on the TL system [44], which in turn builds on a number of recent works on typed intermediate languages [20, 7, 59, 43, 9, 57, 44].

1.6 Organization of this Paper

Section 2 shows how to take our motivating example and turn it into a tagless staged interpreter in a dependently typed setting. First, we present the syntax and semantics of a simple typed language and show how these can be implemented in a direct fashion in Meta-D. The first part of this (writing the unstaged interpreter) is similar to what has been done in Cayenne [2], but is simplified by the presence of dependent datatypes in Meta-D (see Related Work). The key observation here is that the interpreter needs to be defined over typing derivations rather than expressions. Dependently typed datatypes are needed to represent such typing derivations accurately. Next, we show how this interpreter can be easily staged. This step is exactly the same as in the untyped and in the Hindley-Milner setting.

In Section 3 we point out and address some basic practical problems that arise in the implementation of interpreters in a dependently typed programming language. First, we show how to construct the typing judgments that are consumed by the tagless interpreter. Then, we review why it is important to have a clear separation between the computational language and the type language. This motivates the need for representation types, and has an effect on the code for the tagless staged interpreter.

Section 4 presents a formalization of a core subset of Meta-D, and a formal proof of its type safety. The original work on TL used this system to type a computational language that includes basic effects, such as non-termination [44]. In this paper, we develop a multi-stage computational language, and show how essentially the same techniques can be used to verify its soundness. The key technical modifications needed are the addition of levels to typing judgments, and addressing evaluation under type binders.

Section 5 discusses related work, and Section 6 outlines directions for future work and concludes.

An extended version of the paper is available on-line as a technical report [36].

2 A Tagless Staged Interpreter

In this section we show how the example discussed in the introduction can be redeveloped in a dependently typed setting. We begin by considering a definition of the syntax and semantics (of a simply typed version) of the subject language.

2.1 Subject-Language Syntax and Semantics

Figure 1 defines the syntax, type system, and semantics of an example subject language we shall call SL. For simplicity of the development, we use de Bruijn indices for variables and binders. The semantics defines how the types of SL are mapped to their intended meaning. For example, the meaning of the type N is the set of natural numbers, while the meaning of the arrow type $t_1 \rightarrow t_2$ is the function space $[\![t_2]\!]^{[\![t_1]\!]}$. Furthermore, we map the meaning of type assignments Γ, into a product of the sets denoting the finite number of types in the assignment. Note that the semantics of programs is defined on *typing judgments*, and maps to elements of the meanings of their types. This is the standard way of defining the semantics of typed languages [56, 18, 39], and the implementation in the next section will be a direct codification of this definition.

2.2 Interpreters in Meta-D

An interpreter for SL can be simply an implementation of the definition in Figure 1. We begin by defining the datatypes that will be used to interpret the basic types (and typing environments) of SL. To define datatypes Meta-D uses an alternative notation to SML or Haskell datatype definitions. For example, to define the set of natural numbers, instead of writing

```
datatype Nat = Z | S of Nat
```

we write

```
inductive Nat : *1  = Z  : Nat | S : Nat -> Nat
```

The inductive notation is more convenient when we are defining dependent datatypes and when we wish to define not only new types but new kinds (meaning "types of types"). Now type expression and type assignments are represented as follows:

```
inductive Typ : *1  = NatT   : Typ
                    | ArrowT : Typ -> Typ -> Typ
inductive Exp : *1  = EI : Nat -> Exp
                    | EV : Nat -> Exp
                    | EL : Typ -> Exp -> Exp
                    | EA : Exp -> Exp -> Exp
inductive Env : *1  = EmptyE  : Env
                    | ExtE : Env -> Typ  -> Env
```

The `*1` in these definitions means that we are defining a new type. To implement the type judgment of SL we need a dependently typed datatype *indexed* by three parameters: a type assignment `Env`, an expression `Exp`, and a type `Typ`. We can define such a datatype as shown in Figure 2.[3] Each constructor in this datatype corresponds to one of the rules in the type system for our object language. For example, consider the rule for lambda abstraction (Lam) from Figure 1. The basic idea is to use the "judgments as types principle" [19], and so we can view the type rule as a constant combinator on judgments. This combinator takes hypothesis judgments (and their free variables) and returns the conclusion judgment. In this case the rule requires an environment Γ, two types t and t', a body e of the lambda abstraction, a judgment that $\Gamma, t \vdash e : t'$, and returns a judgment $\Gamma \vdash \lambda t.e : t \rightarrow t'$. This rule is codified directly by the following constructor

```
JL : (e1 : Env) -> (t1 : Typ) -> (t2 : Typ) ->
     (s2 : Exp) -> J(ExtE e1 t1, s2, t2) ->
       J(e1, EL t1 s2, ArrowT t1 t2).
```

In the definition of `J` we see differences between the traditional datatype definitions and inductive datatypes: each of the constructors can have dependently typed arguments and a range type `J` indexed by different indices. It is through this variability in the return

[3] For practical reasons that we will discuss in the next section, this datatype is not legal in Meta-D. We will use it in this section to explain the basic ideas before we discuss the need for so-called representation types.

$$
\begin{array}{lll}
t & \in T_1 = & N \mid t \to t \\
\Gamma & \in G_1 = & \langle\rangle \mid \Gamma, t \\
e & \in E_1 = & n \mid \lambda t.e \mid e\, e \mid \#n
\end{array}
$$

$$
\frac{}{\Gamma \vdash n : N}\,(\text{Nat}) \qquad \frac{}{\Gamma, t \vdash \#0 : t}\,(\text{Var})
$$

$$
\frac{\Gamma \vdash \#n : t}{\Gamma, t' \vdash \#(n+1) : t}\,(\text{Weak}) \qquad \frac{\Gamma, t \vdash e : t'}{\Gamma \vdash \lambda t.e : t \to t'}\,(\text{Lam})
$$

$$
\frac{\Gamma \vdash e_1 : t \to t' \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\, e_2 : t'}\,(\text{App})
$$

$$
\begin{array}{lll}
[\![N]\!] & = & \mathbb{N} \\
[\![t_1 \to t_2]\!] & = & [\![t_2]\!]^{[\![t_1]\!]} \\
[\![\langle\rangle]\!] & = & 1 \\
[\![\Gamma, t]\!] & = & [\![\Gamma]\!] \times [\![t]\!] \\
[\![\Gamma \vdash e : t]\!] & : & [\![\Gamma]\!] \to [\![t]\!] \\
[\![\Gamma \vdash n : N]\!]\rho & = & n \in \mathbb{N} \\
[\![\Gamma, t \vdash \#0 : t]\!]\rho & = & \pi_2(\rho) \\
[\![\Gamma, t' \vdash \#(n+1) : t]\!]\rho & = & [\![\Gamma \vdash \#n : t]\!](\pi_1 \rho) \\
[\![\Gamma \vdash \lambda t.e : t \to t']\!]\rho & = & x \mapsto ([\![\Gamma, t \vdash e : \tau']\!]\ (\rho, x)) \\
[\![\Gamma \vdash e_1\, e_2 : t]\!]\rho & = & [\![\Gamma \vdash e_1 : t' \to t]\!]\rho([\![\Gamma \vdash e_2 : t']\!]\rho)
\end{array}
$$

Figure 1. Semantics of SL

```
inductive J : (Env, Exp, Typ) -> *1  =
        JN : (e1:Env) -> (n :Nat) ->  J(e1,EI n,NatT)
      | JV : (e1:Env) -> (t1:Typ) -> J(ExtE e1 t1,EV Z,t1)
      | JW : (e1:Env) -> (t1:Typ) -> (t2:Typ) -> (i:Nat) -> J(e1,EV i,t1) -> J(ExtE e1 t2,EV (S i), t1)
      | JL : (e1:Env) -> (t1:Typ) -> (t2:Typ) -> (s2:Exp) -> J(ExtE e1 t1,s2,t2) -> J(e1,EL t1 s2, ArrowT t1 t2)
      | JA : (e:Env) -> (s1:Exp) -> (s2:Exp) -> (t1:Typ) -> (t2 : Typ) -> (J(e,s1,ArrowT t1 t2))  ->
             (J(e,s2,t1)) -> J(e, EA s1 s2, t2)
```

Figure 2. The typing judgment `J` (without representation types)

type of the constructors that dependent datatypes can provide more information about their values.

2.2.1 *Interpreters of Types and Judgments*

After defining judgments, we are ready to implement the interpretations. Note, however, that the type of the result of the interpretation of judgments, depends on the interpretation of SL types. This dependency is captured in the interpretation function `typEval`. Figure 3 presents the implementation of the interpretation of types `typEval`; the mapping of type assignments into Meta-D types `envEval`; and the interpretation of judgments `eval`.

The function `eval` is defined by `case` analysis on typing judgments. Computationally, this function is not significantly different from the one presented in Section 1.2. Differences include additional typing annotations, and the case analysis over typing judgments. Most importantly, writing it does not require that we use tags on the result values, because the type system allows us to specify that the return type of this function is `typEval t`. Tags are no longer needed to help us discriminate what type of value we are getting back at runtime: the type system now tells us, *statically*.

2.3 Staged Interpreters in Meta-D

Figure 4 shows a staged version of `eval`. As with Hindley-Milner types, staging is not complicated by dependent types. The staged interpreter `evalS`, returns a value of type `(code (typEval t))`. Note that the type of value assignments is also changed (see `envEvalS` in Figure 4): Rather than carrying runtime values for SL, it carries pieces of code representing the values in the variable assignment. Executing this program produces the tagless code fragments that we are interested in.

Even though the eval function never performs tagging and untagging, the interpretative overhead from traversing its input is still considerable. Judgements must be deconstructed by eval at runtime. This may require even more work than deconstructing tagged values. With staging, all these overheads are performed in the first stage, and an overhead-free term is generated for execution in a later stage.

Staging violations are prevented in a standard way by Meta-D's type system (See technical report [36]). The staging constructs are those of Davies [10] with the addition of cross-stage persistence [55]. We refer the reader to these references for further details on the nature of staging violations. Adding a run construct along the lines of previous works [51, 30] was not considered here.

Now we turn to addressing some practical questions that are unique to the dependent typing setting, including how the above-mentioned judgements are constructed.

3 Practical Concerns

Building type judgments amounts to implementing either type-checking or type inference for the language we are interpreting. Another practical concern is that types that depend on values can lead to either undecidable or unsound type checking. This happens when values contain diverging or side-effecting computations. In this section we discuss how both of these concerns are addressed in the context of Meta-D.

3.1 Constructing Typing Judgments

Requiring the user of a DSL to supply a typing judgment for each program to be interpreted is not likely to be acceptable (although it can depend on the situation). The user should be able to use the implementation by supplying only the plain text of the subject program. Therefore, the implementation needs to include at least a type checking function. This function takes a representation of a type-annotated program and produces the appropriate typing judgment, if it exists. We might even want to implement type inference, which does not require type annotations on the input. Figure 4 presents a function `typeCheck`. This function is useful for illustrating a number of features of Meta-D:

- The type of the result[4] of `typeCheck` is a dependent sum, written `[t : Typ] J(e,s,t)`. This means that the result of `typeCheck` consists of an SL type, and a typing judgment that proves that the argument expression has that particular type under a given type assignment.
- Since judgments are built from sub-judgments, a `case` (strong dependent sum elimination) construct is need to deconstruct

[4] In a pure (that is with no computational effects whatsoever) setting the result of `typeCheck` should be `option ([t : Typ] (J (e,s,t)))`, since a particular term given to `typeCheck` may not be well-typed. In the function given in this paper, we omit the `option`, to save on space (and rely on incomplete case expressions instead).

```
fun typEval (t : Typ) : *1 = case t of NatT => Nat | ArrowT t1 t1 => (typEval t1) -> (typEval t2)

fun envEval (e : Env) : *1 = case e of EmptyE  => unit |  ExtE e2 t => (envEval e2, typEval t)

fun eval (e : Env) (rho: envEval(e)) (s : Exp) (t : Typ) (j : J(e,s,t)) : (typEval t)=
  case j of
    JN  e1 n1  => n1
  | JV  e1 t1  => #2(rho)
  | JW  e1 t1 t2 i j1 => eval e1 (#1(rho)) (EV i) t1 j1
  | JL ee1 et1 et2 es2 ej1 => fn v : (typEval et1) => (eval (ExtE ee1 et1) (rho,v) es2 et2 ej1 )
  | JA e s1 s2 t1 t2 j1 j2 => (eval e rho s1 (ArrowT t1 t2) j1) (eval e rho s2 t1 j2)
```

Figure 3. Dependently typed tagless interpreter (without representation types)

```
fun envEvalS (e : Env) : *1 = case e of EmptyE => unit | ExtE e2 t =>
   (envEvalS e2, code (typEval t))

fun evalS (e : Env) (rho: envEvalS e) (s : Exp) (t : Typ) (j : J(e,s,t)) : (code (typEval t)) =
  case j of
    JN  e1 n1  => .<n1>.
  | JV  e1 t1  => #2(rho)
  | JW  e1 t1 t2 i j1 => evalS e1 (#1(rho)) (EV i) t1 j1
  | JL ee1 et1 et2 es2 ej1 => .<fn v:(typEval et1) => (.~(evalS (ExtE ee1 et1) (rho,.<v>.) es2 et2 ej1))>.
  | JA e s1 s2 t1 t2 j1 j2 => .<(.~(evalS e rho s1 (ArrowT t1 t2) j1)) (.~(evalS e rho s2 t1 j2))>.

fun typeCheck (e : Env) (s : Exp)  : ([t : Typ] J(e,s,t)) =
 case s of
   EI n => [t = NatT] (JN e n)
 | EV nn => (case nn of  Z => (case e of ExtE ee t2 => [t = t2](JV ee t2))
                       | S n => (case e of ExtE e2 t2 =>
                                  ((fn x : ([rt:Typ]J(e2,EV n,rt)) =>
                                        case  x of [rx : Typ]j2 => ([t = rx](JW e2  rx t2 n j2))  )
                                  (typeCheck e2 (EV n)))))
 | EL targ s2 =>
  ((fn x : ([rt : Typ](J(ExtE e targ,s2,rt))) =>
     case x of [rt : Typ] j2 => [t = ArrowT targ rt] (JL e targ rt s2 j2))
  (typeCheck (ExtE e targ) s2))
 | EA s1 s2 =>
  ((fn x1 : [rt1 : Typ](J(e,s1,rt1)) => (fn x2 : [rt2 : Typ](J(e,s2,rt2)) =>
   case x1 of [rt1 : Typ]j1 => case x2 of [rt2 : Typ]j2 =>
     (case rt1 of ArrowT tdom tcod =>
        [t = tcod] (JA e s1 s2 tdom tcod j1
                       (cast [assert rt2=tdom,J(e,s,tdom), j2])) end)))
  (typeCheck e s1) (typeCheck e s2))

fun typeAndRunNat (s : Exp) : Nat =
((fn x : ([t1 : Typ] J(EmptyE,s,t1)) => case x of [t1 : Typ] j =>
           (case t1 of NatT => eval EmptyE () s NatT j
                     | ArrowT t2 t3 => Z ))
(typeCheck EmptyE s))
```

Figure 4. Staged tagless interpreter and the function `typeCheck` (without representation types)

the results of recursive calls to `typeCheck`.

- The case for constructing application judgments illustrates an interesting point. Building a judgment for the expression `(EA s1 s2)` involves first computing the judgments for the sub-terms `s1` and `s2`. These judgments assign types `(ArrowT tdom tcod)` and `rt2` to their respective expressions. However, by definition of the inductive family `J`, in order to build the larger application judgment, `tdom` and `rt2` must be the same SL type (i.e., their `Typ` values must be equal).

  We introduce two language constructs to Meta-D to express this sort of constraints between values. First, the expression of the form `assert e1 = e2` introduces an *equality judgment*, `ID e1 e2` between values of equality types.[5]

  An elimination construct

  `cast[e1,T,e2]`

  is used to cast the expression `e2` from some type `T[v1]` to `T[v2]`, where `e1` is an equality judgment of the type `ID v1 v2`. The type checker is allowed to use the Leibniz-style equality to prove the cast correct, since `e1` is an equality judgment stating that `v1` and `v2` are equal.

  Operationally, the expression `assert e1=e2` evaluates its two subexpressions and compares them for equality. If they are indeed equal, computation proceeds If, however, the two values are not equal, the program raises an exception and terminates. The `cast` construct makes sure that its equality judgment introduced by assert is evaluated at runtime, and if the equality check succeeds, simply proceeds to evaluate its argument expression.

  An alternative to using `assert/cast` is to include equality judgments between types as part of typing judgments, and build equality proofs as a part of the `typeCheck` function.[6] This approach, while possible, proves to be verbose, and will be omitted in this paper. The `assert/cast`, however, can serve as a convenient programming shortcut and relieves the user from the effort formalizing equality at the type level and manipulating equality types.

3.2 Representation Types

Combining effects with dependent types requires care. For example, the `typeCheck` function is partial, because there are many input terms which are just not well typed in SL. Such inputs to `typeCheck` would cause runtime pattern match failures, or an equality assertion exception. We would like Meta-D to continue to have side-effects such as non-termination and exceptions. At the same time, dependently typed languages perform computations during type checking (to determine the equality of types). If we allow effectful computations to leak into the computations that are done during type checking, then we risk non-termination, or even unsoundness, at type-checking time. This goal is often described as "preserving the phase distinction" between compile time and runtime [5].

The basic approach to dealing with this problem is to allow types to only depend on other types, and not values. Disallowing any kind of such dependency, however, would not allow us to express our type checking function, as it produces a term whose type depends on the value of its argument. A standard solution to is to introduce a mechanism that allows only a limited kind of dependency between values and types. This limited dependency uses so-called singleton or representation types [60, 7, 9, 57]. The basic idea is to allow bijections on ground terms between the value and type world.

Now, we can rewrite our interpreter so that its type does not depend on runtime values, which may introduce effects into the type-checking phase. Any computation in the type checking phase can now be guaranteed to be completely effect-free. The run-time values are now forced to have representation types that reflect, in the world of values, the values of inductive kinds. In Meta-D, a special type constructor `R` is used to express this kind of dependency. For example, we can define an inductive *kind* `Nat`

```
inductive Nat : *2  = Z    : Nat | S : Nat -> Nat
```

Note that this definition is exactly the same as the one we had for the type `Nat`, except it is not classified by `*2` instead of `*1`. Once this definition is encountered, we have introduced not only the constructors for this type, but also the possibility of using the special type constructor `R`. Now we can write `R(S(S Z))` to refer to a type that has a unique inhabitant, which we also call `rep (S(S Z))`.

Figure 5 presents the implementation with representation types. Introducing this restriction on the type system requires us to turn the definition of `Exp`, `Env`, and `Typ` into definitions of kinds (again this is just a change of one character in each definition). Because these terms are now kinds, we cannot use general recursion in defining their interpretation. Therefore, we use special primitive recursion constructs provided by the type language to define these interpretations. Judgments, however, remain a type. But now, they are a type indexed by other types, not by values.

For the most part, the definition of judgments and the interpretation function do not change. We need to change judgments in the case of natural numbers by augmenting them with a representation for the value of that number. The constructor `JN` now becomes

```
JN : (e1 : Env) -> (n  : Nat) ->
     (rn : R n) ->  J(e1,EI n,NatT)
```

and the definition of `eval` is changed accordingly. The modified `eval` uses a helper function to convert a representation of a natural type to a natural number.[7]

The definition of the `typeCheck` function requires more substantial changes (Figure 5). In particular, this function now requires carrying out case analysis on types [20, 7, 59, 43, 9]. For this purpose Meta-D provides a special case construct

```
tycase x by y of C_n x_n => e_n.
```

A pattern `(C_n x_n)` matches against a value `x` of type `K`, where `K` is some inductive kind, only if we have provided a representation value `y` of type `R(x)`. Pattern matching over inductive kinds cannot be performed without the presence of a corresponding runtime value of the appropriate representation type. Inside the body of the case (e_n), the expression `rep x_n` provides a representation value for the part of the inductive constructor that `x_n` is bound to.

4 Formal Development

In this section we report our main technical result, which is type safety for a formalized core subset of Meta-D. This result shows that multi-stage programming constructs can be safely used, even when integrated with a sophisticated dependent type system such as that of TL [44]. We follow the same approach used by the developers of TL, and build a computation language $\lambda_{H\bigcirc}$ that uses TL as its type language. Integrating our formalization into the TL

[5] This feature is restricted to ground types whose value can be shown equal at runtime.

[6] Due to space limitation we omit this approach here, but define an alternative type-checking function in the accompanying technical report [36].

[7] In practice, we see no fundamental reason to distinguish the two. Identifying them, however, requires the addition of some special support for syntactic sugar for this particular representation type.

```
inductive nat : *1  = zero : nat | succ : (nat -> nat)
inductive Nat : *2  = Z    : Nat | S : (Nat -> Nat)

inductive Typ : *2 = ArrowT : Typ -> Typ -> Typ  | NatT   : Typ

inductive Exp : *2  = EI : Nat -> Exp | EV : Nat -> Exp
                    | EL : Typ -> Exp -> Exp  | EA : Exp -> Exp -> Exp

inductive Env : *2  = EmptyE  : Env  | ExtE : Env -> Typ  -> Env

inductive J : (Env, Exp, Typ) -> *1  =
  JN :(e1 : Env) -> (n  : Nat) -> (rn : R n) ->  J(e1,EI n,NatT)
| JV :(e1 : Env) -> (t1  : Typ) -> J(ExtE e1 t1,EV Z,t1)
| JW :(e1 : Env) -> (t1 : Typ) -> (t2 : Typ) -> (i : Nat) -> J(e1,EV i,t1) -> J(ExtE e1 t2,EV (S i), t1)
| JL :(e1 : Env) -> (t1 : Typ) -> (t2 : Typ) -> (s2 : Exp) -> J(ExtE e1 t1,s2,t2) -> J(e1,EL t1 s2,ArrowT t1 t2)
| JA :(e : Env)-> (s1 : Exp)-> (s2 : Exp)-> (t1 : Typ)-> (t2 : Typ) ->
      J(e,s1,ArrowT t1 t2)-> J(e,s2,t1)-> J(e,EA s1 s2,t2)


val typEval : Typ -> *1 =
    primrec Typ nat (fn c : *1 => fn  d : *1 => c -> d)

val envEval : Env -> *1 =
   primrec Env unit (fn r : *1 => fn t : Typ => (r,typEval  t)

fun cast (n : Nat)  (rn : R(n))  : nat = tycase n by rn of Z => zero
                                                         | S n2  => succ (cast n2 (rep n2))

fun eval (e : Env) (rho: envEval e) (s : Exp) (t : Typ) (j : J(e,s,t)) : (typEval t) =
 case j of
  JN  e1 n1 rn1 => cast n1 rn1
| JV  e1 t1  => #2(rho)
| JW  e1 t1 t2 i j1 => eval e1 (#1(rho)) (EV i) t1 j1
| JL ee1 et1 et2 es2 ej1 =>fn v : (typEval et1) => (eval (ExtE ee1 et1) (rho,v) es2 et2 ej1)
| JA e s1 s2 t1 t2 j1 j2 => (eval e rho s1 (ArrowT t1 t2) j1) (eval e rho s2 t1 j2)

fun typeCheck (e : Env) (re: R(e)) (s : Exp)  (rs: R(s))  : ([t : Typ] (R(t),J(e,s,t))) =
tycase s by rs of
  EI n => [t = NatT] (NatT',(JN e n (rep n)))
| EV n =>
 (tycase n by (rep n) of Z => (tycase e by re of ExtE ee  t2 => [t = t2](rep t2, JV ee t2))
                      | S n => (tycase e by re of ExtE (e2) (t2) =>
                                  ((fn x : ([t:Typ] (R(t), J(e2,EV n,t))) =>
                                     case x of [rx : Typ]j2 => ([t = rx]
                                                                (#1 j2, JW e2 rx t2 n (#2 j2)))
                                  (typeCheck e2 (rep e2) (EV n) (rep (EV n)))))))
| EL targ s2 =>
 ((fn x : ([t : Typ](R(t),(J(ExtE e targ,s2,t)))) =>
      case x of [t : Typ] j2 => [t = ArrowT targ t] (rep (ArrowT targ (#1 t))), (JL e targ t s2 (#2 j2)) )
 (typeCheck (ExtE e targ) (rep (ExtE e targ)) s2 (rep s2)))
| EA s1 s2 =>
 ((fn x1 : [t1 : Typ](R(t1),(J(e,s1,t1))) => (fn x2 : [t2 : Typ](R(t2),(J(e,s2,t2))) =>
    case x1 of [t1 : Typ]j1 => case x2 of [t2 : Typ]j2 =>
      (tycase t1 by (#1 (j1)) of
       ArrowT tdom tcod =>
        [t = tcod] (rep tcod, (JA e s1 s2 tdom tcod j1
                              (cast [assert t2=tdom,J(e,s,tdom),j2]))) end)))
 (typeCheck e (rep e) s1 (rep s1)) (typeCheck e (rep e) s2 (rep s2)))
```

Figure 5. Tagless interpreter with representation types in MetaD

$$
\begin{array}{lll}
\text{inductive } \Omega^{\bigcirc} : \text{Kind} & ::= & \text{snat} : \text{Nat} \to \Omega^{\bigcirc} \\
 & \mid \text{sbool} & : \text{Bool} \to \Omega^{\bigcirc} \\
 & \mid \twoheadrightarrow & : \Omega^{\bigcirc} \to \Omega^{\bigcirc} \to \Omega^{\bigcirc} \\
 & \mid \text{tup} & : \text{Nat} \to (\text{Nat} \to \Omega^{\bigcirc}) \to \Omega^{\bigcirc} \\
 & \mid \forall_k & : \Pi k : \text{Kind}.(k \to \Omega^{\bigcirc}) \to \Omega^{\bigcirc} \\
 & \mid \exists_k & : \Pi k : \text{Kind}.(k \to \Omega^{\bigcirc}) \to \Omega^{\bigcirc} \\
 & \mid \forall_{KS} & : \Pi k : \text{KScheme}.(k \to \Omega^{\bigcirc}) \to \Omega^{\bigcirc} \\
 & \mid \exists_{KS} & : \Pi k : \text{KScheme}.(k \to \Omega^{\bigcirc}) \to \Omega^{\bigcirc} \\
 & \mid \bigcirc & : \Omega^{\bigcirc} \to \Omega^{\bigcirc} \\
 & \mid \text{EQ} & : \text{Nat} \to \text{Nat} \to \Omega^{\bigcirc}
\end{array}
$$

Figure 6. The TL definition of the types of $\lambda_{H\bigcirc}$

$$
\begin{array}{rcl}
X & \equiv & \text{type variables of TL} \\
A & \equiv & \text{type expressions of TL} \\
exp^0 \in E^0 & ::= & x \mid n \mid \mathsf{tt} \mid \mathsf{ff} \mid f^0 \mid \mathsf{fix}\, x : A.f^0 \mid e_1^0\; e_1^0 \mid e^0[A] \mid \llparenthesis X = A_1, e^0 : A_2 \rrparenthesis \\
 & \mid & \text{open } e^0 \text{ as } X, x \;\mathsf{in}\; e^0 \mid (e_0^0, \ldots e_{n-1}^0) \mid \mathsf{sel}\ [A](e_1^0, e_2^0) \mid e_1^0 \oplus e_2^0 \mid \mathsf{if}\ [A_1, A_2](e^0, X_1.e_1^0, X_2.e_2^0) \mid \langle e^1 \rangle \\
 & \mid & \text{assert } e_1^0 : A_1 = e_2^0 : A_2 \mid \text{ cast } (e_1^0, A, e_2^0) \\
f^n & ::= & \Lambda X : A.e^n \mid \lambda x : A.e^n \\
exp^{n+} & ::= & x \mid m \mid \mathsf{tt} \mid \mathsf{ff} \mid f^{n+} \mid \mathsf{fix}\, x : A.f^{n+1} \mid e^{n+}\; e^{n+} \mid e^{n+}\; [A] \mid \llparenthesis X = A_1, e^{n+} : A_2 \rrparenthesis \mid \text{open } e^{n+} \text{ as } X, x \;\mathsf{in}\; e^{n+} \\
 & \mid & (e_0^{n+}, \ldots, e_{m-1}^{n+1}) \mid \mathsf{sel}\ [A](e_1^{n+}, e_2^{n+}) \mid e_1^{n+} \oplus e_2^{n+} \mid \langle e^{n++} \rangle \mid \tilde{\ } e^n \mid \mathsf{if}\ [A_1, A_2](e^{n+}, X_1.e_1^{n+}, X_2.e_2^{n+}) \\
 & \mid & \text{assert } e_1^{+} : A_1 = e_2^{+} : A_2 \mid \text{ cast } (e_1^{+}, A, e_2^{+}) \\
v^0 \in V^0 & ::= & n \mid \mathsf{tt} \mid \mathsf{ff} \mid f^0 \mid \mathsf{fix}\, x : A.f^0 \mid \llparenthesis X = A_1, v^0 : A_2 \rrparenthesis \mid (v_1^0, \ldots, v_{m-1}^0) \mid \langle v^1 \rangle \\
 & \mid & \text{assert } v^0 : A = v^0 : B \\
v^{n+1} \in V^{n+1} & ::= & E^n
\end{array}
$$

Figure 7. Syntax of $\lambda_{H\bigcirc}$

$$
\begin{array}{ll}
\langle\rangle|_n & = \langle\rangle \\
\Delta, X : A^m|_n & = \begin{cases} \Delta|_n, X : A & m = n \\ \Delta|_n & \text{otherwise} \end{cases}
\end{array}
$$

$$
\frac{\Delta \vdash^n \Gamma \quad (x : A^m) \in \Gamma \quad m \leq n}{\Delta; \Gamma \vdash^n x : A}
\qquad
\frac{\Delta \vdash^n \Gamma\ \text{OK}}{\Delta; \Gamma \vdash^n m : \mathsf{snat}\ \ m}
\qquad
\frac{\Delta \vdash^n \Gamma\ \text{OK}}{\Delta; \Gamma \vdash^n \mathsf{tt} : \mathsf{sbool}\ \ \text{True}}
\qquad
\frac{\Delta \vdash^n \Gamma\ \text{OK}}{\Delta, \Gamma \vdash^n \mathsf{ff} : \mathsf{sbool}\ \ \text{False}}
$$

$$
\frac{\begin{array}{c}\Delta|_n \vdash A : \Omega^{\bigcirc} \\ \Delta; \Gamma, x : A^n \vdash^n f : A\end{array}}{\Delta; \Gamma \vdash^n (\mathsf{fix}\, x : A.f) \ :\ A}
\qquad
\frac{\Delta|_n \vdash A_1 : \Omega^{\bigcirc} \quad \Delta; \Gamma, x : A_1^n \vdash^n e : A_2}{\Delta; \Gamma \vdash^n (\lambda x : A_1.e) : A_1 \twoheadrightarrow A_2}
\qquad
\frac{\begin{array}{c}\Delta; \Gamma \vdash^n e_1 : A_1 \twoheadrightarrow A_2 \\ \Delta; \Gamma \vdash^n e_2 : A_1\end{array}}{\Delta; \Gamma \vdash^n e_1\; e_2 \ :\ A_2}
$$

$$
\frac{\begin{array}{c}\Delta|_n \vdash B : s \\ \Delta, X : B^n; \Gamma \vdash^n f : A\end{array}}{\Delta; \Gamma \vdash^n (\Lambda X : B.f) \ :\ \forall_s X : B.A}
\qquad
\frac{\begin{array}{c}\Delta|_n \vdash A : B \\ \Delta, \Gamma \vdash^n e : \forall_s X : B.A_2\end{array}}{\Delta; \Gamma \vdash^n e[A] \ :\ A_2[X := A]}
\qquad
\frac{\Delta; \Gamma \vdash^{n+1} e : A}{\Delta; \Gamma \vdash^n \langle e \rangle : \bigcirc A}
\qquad
\frac{\Delta; \Gamma \vdash^n e : \bigcirc A}{\Delta; \Gamma \vdash^{n+1} \tilde{\ } e : A}
$$

$$
\frac{\begin{array}{c}\Delta; \Gamma \vdash^n e_1 : \mathsf{snat}\ \ A_1 \\ \Delta; \Gamma \vdash^n e_2 : \mathsf{snat}\ \ A_2\end{array}}{\Delta; \Gamma \vdash^n e_1 \oplus e_2 : \mathsf{snat}\ \ (A_1 \hat{\oplus} A_2)}
\qquad
\frac{\begin{array}{c}\Delta|_n \vdash A : B \\ \Delta|_n \vdash B : s \\ \Delta; \Gamma \vdash^n e : A[X := A_1]\end{array}}{\Delta; \Gamma \vdash^n \llparenthesis X = A_1, e : A \rrparenthesis : \exists_s X : B.A}
\qquad
\frac{\begin{array}{c}\Delta, \Gamma \vdash^n e : \exists_s X' : B.A_1 \\ \Delta|_n \vdash A_2 : \Omega^{\bigcirc} \\ \Delta, X : B; \Gamma, x : A_1[X' := X] \vdash^n e_2 : A_2\end{array}}{\Delta; \Gamma \vdash^n \text{open } e_1 \text{ as } X, x \;\mathsf{in}\; e_2 : A_2}\ X \notin \Delta
$$

$$
\frac{\begin{array}{c}\Delta; \Gamma \vdash^n e_1 : \mathsf{tup}\, A_3\ B \\ \Delta; \Gamma \vdash^n e_2 : \mathsf{snat}\ \ A_2 \\ \Delta|_n \vdash A : \mathsf{LT}\, A_2\, A_3\end{array}}{\Delta; \Gamma \vdash^n \mathsf{sel}\ [A](e_1, e_2) : B\, A_2}
\qquad
\frac{\begin{array}{cc}\Delta|_n \vdash B : \text{Bool} \to Kind & \Delta; \Gamma \vdash^n e : \mathsf{sbool}\ \ A_3 \\ \Delta|_n \vdash A : B\, A_3 & \Delta, X_1 : B\ \mathsf{true}; \Gamma \vdash e_1 : A_2 \\ \Delta|_n \vdash A_2 : \Omega^{\bigcirc} & \Delta, X_2 : B\ \mathsf{false}; \Gamma \vdash e_2 : A_2\end{array}}{\Delta; \Gamma \vdash^n \mathsf{if}\ [B, A](e, X_1.e_1, X_2.e_2) : A_2}
$$

$$
\frac{0 \leq i < m.\ \Delta, \Gamma \vdash^n e_i : A_i}{\Delta; \Gamma \vdash^n (e_0, \ldots, e_{m-1}) : \mathsf{tup}\ \hat{m}\ (\mathsf{nth}\ [A_0, \ldots, A_{m-1}])}
\qquad
\frac{\Delta; \Gamma \vdash^n e : A_1 \quad A_1 = A_2 \quad \Delta|_n \vdash A_2 : \Omega^{\bigcirc}}{\Delta; \Gamma \vdash^n e : A_2}
$$

$$
\frac{\Delta; \Gamma \vdash^n e_1 : \text{snat}\, A \quad \Delta; \Gamma \vdash^n e_2 : \text{snat}\, B}{\Delta; \Gamma \vdash^n \ \text{assert } e_1 : A = e_2 : B : \text{ID}\, A\, B}
\qquad
\frac{\Delta; \Gamma \vdash^n e_1 : \text{ID}\, A\, B \quad \Delta; \Gamma \vdash^n e_2 : \text{snat}\, A}{\Delta; \Gamma \vdash^n \ \text{cast } (e_1, B, e_2) : \text{snat}\, B}
$$

Figure 8. Type system of $\lambda_{H\bigcirc}$

framework gave us significant practical advantages in formal development of $\lambda_{H\bigcirc}$:

- Important meta-theoretic properties of the type language we use, TL, have already been proven [44]. Since we do not change anything about the type language itself, all these results (e.g., the Church-Rosser property of the type language, decidable equality on type terms) are easily reused in our proofs.
- $\lambda_{H\bigcirc}$ is based on the computational language λ_H [44]. We have tried to make the difference between these two languages as small as possible. As a result, the proof of type safety of $\lambda_{H\bigcirc}$ is very similar to the type safety proof for λ_H. Again, we were able to reuse certain lemmata and techniques developed for λ_H to our own proof.

A detailed proof of the type safety of $\lambda_{H\bigcirc}$ is presented in an extended technical report [36].

Figure 6 defines $\lambda_{H\bigcirc}$ computational types, and is the first step needed to integrate $\lambda_{H\bigcirc}$ into the TL framework. The syntax of the computational language $\lambda_{H\bigcirc}$ is given in Figure 7. The language $\lambda_{H\bigcirc}$ contains recursion and staging constructs. It contains two predefined representation types: naturals and booleans. The `if` construct, as in λ_H, provides for propagating proof information into branches (analogous to the `tycase` construct of MetaD); full implementation of inductive datatypes in the style of MetaD is left for future work. Since arbitrary dependent types are prohibited in $\lambda_{H\bigcirc}$, we use universal and existential quantification to express dependencies of values on types and kinds. For example, the identity function on naturals is expressed in $\lambda_{H\bigcirc}$ as follows:

$$(\Lambda n : Nat.\lambda x : \text{snat}\ n.x) : \forall n : Nat.\text{snat}\ n \to \text{snat}\ n$$

In $\lambda_{H\bigcirc}$, we also formalize the `assert/cast` construct, which requires extending the language of computational types with equality judgment types. Similarly, we add the appropriate constructs to the syntax of $\lambda_{H\bigcirc}$.

To be able to define the small-step semantics for a staged language, we had to define the syntax of $\lambda_{H\bigcirc}$ in terms of level-indexed families of expressions and values [48]. The typing judgment (Figure 8), as well as the type assignments, of $\lambda_{H\bigcirc}$ has also been appropriately extended with level annotations [55]. A level-annotation erasure function $(\cdot\ |_n)$ is used to convert $\lambda_{H\bigcirc}$ typing assignments into a form required by the typing judgment of TL[44]. This interface then allows us to reuse the original TL typing judgment.

Due to lack of space we do not show all the definitions for the small-step semantics of $\lambda_{H\bigcirc}$. These, together with proofs of the relevant theorems, are included in a companion technical report [36]. Here, we list the most important theorems.

LEMMA 1 (PROGRESS). *If* $\Delta;\Gamma^+ \vdash^n e^n : A$, *then* $e^n \in V^n$ *or* $\exists e'.e \longmapsto e'$.

Proof is by structural induction on $e^n \in E^n$, and then by examination of cases of the typing judgment.

LEMMA 2 (SUBJECT REDUCTION). $\forall n.$ *if* $\Delta,\Gamma \vdash^n e : A$ *and* $e \to e'$, *then* $\Delta,\Gamma \vdash^n e' : A$.
Proof is by cases of possible reductions $e \to e'$.

THEOREM 1 (TYPE SAFETY). *If* $\Delta;\Gamma^+ \vdash^n e^n : A$ *then* $e \overset{n}{\longmapsto}{}^* v^n$, *and* $\Delta,\Gamma^+ \vdash^n v : A$, *or* $e \Uparrow$.
Proof uses subject reduction (Lemma 2) and progress (Lemma 1) lemmas and follows Wright and Felleisen's syntactic technique [58].

5 Related Work

Barendregt [3] is a good high-level introduction to the theory of dependent type systems. There are a number of other references to (strictly terminating) functional programming in dependent type theory literature [32, 31, 6].

Cayenne is a dependently typed programming language [1]. In essence, it is a direct combination of a dependent type theory with (potentially) non-terminating recursion. It has in fact been used to implement an (unstaged) interpreter similar to the one discussed in this paper [2]. The work presented here extends the work done in Cayenne in three respects: First, Cayenne allows types to depend on values, and thus, does not ensure that type checking terminates. Second, Cayenne does not support dependent datatypes (like `J(e,s,t)`), and so, writing an interpreter involves the use of a separate proof object to encode the information carried by `J(e,s,t)`), which is mostly just threaded through the program. The number of parameters passed to both the Meta-D and Cayenne implementation of the eval function is the same, but using dependent datatypes in Meta-D allows direct analogy with the standard definition of the semantics over typing judgments rather than raw terms. Third, Cayenne does not provide explicit support for staging, an essential component for achieving the performance results that can be achieved using tagless staged interpreters.

Xi and Pfenning study a number of different practical approaches to introducing dependent types into programming languages [59, 60]. Their work concentrates on limiting the expressivity of the dependent types, and thus limiting the constraints that need to be solved to Presburger arithmetic problems. Singleton types seem to have been first used by Xi in the context of DML [60]. The idea was later used in a number of works that further developed the idea of representation types and intensional type analysis.

Logical frameworks [19, 37] use dependent types as a basis for proof systems. While this is related to our work, logical frameworks alone are not sufficient for our purposes, as we are interested in computational programming languages that have effects such as non-termination. It is only with the recent work of Shao, Saha, Trifonov and Papaspyrou that we have a generic framework for safely integrating a computation base language, with a rich dependent type system, without losing decidability (or soundness) of type-checking.

Dybjer extensively studies the semantics of inductive sets and families [11, 12, 13, 14, 16] and simultaneous inductive- recursive definitions [15]. TL uses only the former (in the type level), and we also use them at the value level (`J(e,s,t)`). The Coq proof assistant provides fairly extensive support for both kinds of definitions [17, 34, 35]. In the future, it will be interesting to explore the integration of the second of these techniques into programming languages.

One interesting problem is whether self-interpretation is possible in a given programming language. This is possible with simply-typed languages [54]. It is not clear, however, that it can be done in a dependently typed language [38]. Exploring this problem is interesting future work.

Finally, staged type inference [46] can also be used as a means of obtaining programs without tags. Of the techniques discussed in this paper, it is probably closest in spirit to tag elimination. In fact, in a multi-stage setting tag elimination is applied at runtime and is nothing but a non-standard type analysis. Key differences are that in the staged type inference system the code type that is used does not reflect any type information, and type information can only be determined by dynamic type checking. More importantly, the success and failure of staged type inference can depend on whether the value in the code type has undergone simplification, and it is easy to return a value that tells us (at runtime, in the language) whether this dynamic inference succeeded or not. Tag elimination, on the other hand, works on code that has an explicit static type. Additionally, by using carefully crafted "fall-back plan" projection/embedding pairs, runtime tag elimination is guaranteed to always have the same denotational semantics (but certainly not operational semantics) independently of the test of the code being analyzed and any simplifications that may be done to the subject program [54].

6 Conclusions and Future Work

In this paper we have shown how a dependently typed programming language can be used to express a staged interpreter that completely circumvents the need for runtime tagging and untagging operations associated with universal datatypes. In doing so we have highlighted two key practical issues that arise when trying to develop staged interpreters in a dependently typed language. First, the need for functions that build the representations of typing judgments that the interpretation function *should* be defined over. And second, the need for representation types to avoid polluting the type language with the impure terms of the computational language. To demonstrate that staging constructs and dependent types can be safely combined, we formalize our language as a multi-stage computational language typed by Shao, Saha, Trifonov, and Papaspyrou's TL system. This allows us to prove type safety in a fairly straightforward manner, and without having to duplicate the work done for the TL system.

A practical concern about using dependent types for writing interpreters is that such systems do not have decidable type *inference*, which some view as a highly-valued feature for any typed language. We did not find that the annotations were a burden, and some simple tricks in the implementation were enough to avoid the need for redundant annotations.

In carrying out this work we developed a deeper appreciation for the subtleties involved in both dependently typed programming and in the implementation of type checkers for dependently typed languages. Our current implementation is a prototype system that we have made available online [27]. Our next step is to study the integration of such a dependently typed language into a practical implementation of multi-stage programming, such as MetaOCaml [28]. We have also found that there a lot of opportunities in the context of dependently typed languages that we would like to explore in the future. Examples include syntactically lighter-support for representation types, formalizing some simple tricks that we have used in our implementation to help alleviate the need for redundant type annotations. We are also interested in exploring the use of dependent types to reflect the resource needs of generated programs [8, 24, 52].

7 References

[1] Lennart Augustsson. Cayenne – a language with dependent types. Available from `http://www.cs.chalmers.se/ augustss/cayenne`.

[2] Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. In *Workshop on Dependent Types in Programming*, Gothenburg, 1999. Available online from `www.cs.chalmers.se/~augustss/cayenne/interp.ps`.

[3] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, 1991.

[4] Jon Bentley. Little languages. *CACM*, 29(8):711–721, 1986.

[5] Luca Cardelli. Phase distinctions in type theory. Unpublished manuscript available online from author's homepage, 1988.

[6] Thierry Coquand, Bengt Nordström, Jan M. Smith, and Bjorne von Sydow. Type theory and programming. *Bulletin of the European Association for Theoretical Computer Science*, 52:203–228, February 1994. Columns: Logic in Computer Science.

[7] Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP-99)*, volume 34.9 of *ACM Sigplan Notices*, pages 233–248, N.Y., September 27–29 1999. ACM Press.

[8] Karl Crary and Stephanie Weirich. Resource bound certification. In *the Symposium on Principles of Programming Languages (POPL '00)*, pages 184–198, N.Y., January 19–21 2000. ACM Press.

[9] Karl Crary, Stephanie Weirich, and J. Gregory Morrisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming (ICFP) , Baltimore, Maryland, USA*, pages 301–312, 1998.

[10] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *the Symposium on Principles of Programming Languages (POPL '96)*, pages 258–270, St. Petersburg Beach, 1996.

[11] Peter Dybjer. Inductively defined sets in Martin-Löf's set theory. In A. Avron, R. Harper, F. Honsell, I. Mason, and G. Plotkin, editors, *Workshop on General Logic*, February 1987.

[12] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Preliminary Proc. of 1st Int. Workshop on Logical Frameworks, Antibes, France, 7–11 May 1990*, pages 213–230. 1990. ftp://ftp.inria.fr/INRIA/Projects/coq/types/Proceedings/book90.ps.Z.

[13] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[14] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

[15] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, 2000.

[16] Peter Dybjer and Anton Setzer. Finite axiomatizations of inductive and inductive-recursive definitions. In R. Backhouse and T. Sheard, editors, *Informal Proc. of Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998*. Dept. of Computing Science, Chalmers Univ. of Techn., and Göteborg Univ., June 1998. Electronic version available at http://wsinwp01.win.tue.nl:1234/WGPProceedings/.

[17] Eduardo Giménez. A tutorial on recursive types in Coq. Technical Report TR-0221, INRIA Rocquencourt, May 1998.

[18] Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.

[19] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings Symposium on Logic in Computer Science*, pages 194–204, Washington, 1987. IEEE Computer Society Press. The conference was held at Cornell University, Ithaca, New York.

[20] Robert Harper and Greg Morrisett. Compiling polymorphism using intentional type analysis. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.*, pages 130–141, New York, NY, January 1995. ACM.

[21] Liwen Huang and Walid Taha. A practical implementation of tag elimination. In preperation.

[22] Paul Hudak. Building domain specific embedded languages. *ACM Computing Surveys*, 28A:(electronic), December 1996.

[23] 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.

[24] R.J.M. Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Guy L. Steele Jr, editor, *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, volume 23, St Petersburg, Florida, 1996. ACM Press.

[25] Neil D. Jones, Carsten K Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[26] Henning Makholm. On Jones-optimal specialization for strongly typed languages. In *[49]*, pages 129–148, 2000.

[27] Meta-D: A dependently typed multi-stage language. Available online from http://www.cse.ogi.edu/ pasalic/metad/, 2002.

[28] MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from http://cs-www.cs.yale.edu/homes/taha/MetaOCaml/, 2001.

[29] Torben Mogensen. Inherited limits. In *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 189–202. Springer-Verlag, 1999.

[30] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[31] Bengt Nordström. Programming in constructive set theory: Some examples. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture, Portsmouth, NH*, pages 141–154, New York, 1981. ACM.

[32] Bengt Nordström, Kent Peterson, and Jan M. Smith. *Programming in Martin-Lof's Type Theory*, volume 7 of *International Series of Monographs on Computer Science*. Oxford University Press, New York, NY, 1990. Currently available online from first authors homepage.

[33] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from ftp://cse.ogi.edu/pub/tech-reports/README.html.

[34] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. Research report RR 92-49, Laboratoire de l'Informatique du Parallélisme, Ecole normale supérieure de Lyon, December 1992.

[35] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. F. Groote, editors, *Proc. of 1st Int. Conf. on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, The Netherlands, 16–18 March 1993*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, Berlin, 1993.

[36] Emir Pašalić, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages (formal development). Technical Report 02-006, OGI, 2002. Available from [33].

[37] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[38] Frank Pfenning and Peter Lee. LEAP: A language with eval and polymorphism. In Josep Díaz and Fernando Orejas, editors, *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development,*, volume 352 of *Lecture Notes in Computer Science*, pages 345–359. Springer-Verlag, 1989.

[39] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, Mass., 1991.

[40] Calton Pu, Andrew Black, Crispin Cowan, and Jonathan Walpole. Microlanguages for operating system specialization. In *Proceedings of the Workshop on Domain-Specific Languages*, Paris, 1997.

[41] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, pages 717–740. ACM, 1972.

[42] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4), 1998. (Reprinted from the proceedings of the 25th ACM National Conference (1972) [41]).

[43] Zhong Shao and Andrew W. Appel. A type-based compiler for standard ML. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 116–129, La Jolla, California, 18–21 June 1995.

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

[45] Tim Sheard, Zine El-Abidine Benaissa, and Emir Pašalić. DSL implementation using staging and monads. In *Second Conference on Domain-Specific Languages (DSL'99)*, Austin, Texas, 1999. USEUNIX.

[46] Mark Shields, Tim Sheard, and Simon L. Peyton Jones. Dynamic typing through staged type inference. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 289–302, 1998.

[47] Yannis Smaragdakis and Don Batory. DiSTiL: A transformation library for data structures. In *USENIX Conference on Domain-Specific Languages*, October 1997.

[48] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [33].

[49] Walid Taha, editor. *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, Montréal, 2000. Springer-Verlag.

[50] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Maniplation (PEPM)*, Boston, 2000. ACM Press.

[51] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety.

In *25th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 918–929, Aalborg, 1998.

[52] Walid Taha, Paul Hudak, and Zhanyong Wan. Directions in functional programming for real(-time) applications. In *the International Workshop on Embedded Software (ES '01)*, volume 221 of *Lecture Notes in Computer Science*, pages 185–203, Lake Tahoe, 2001. Springer-Verlag.

[53] Walid Taha and Henning Makholm. Tag elimination – or – type specialisation is a type-indexed effect. In Subtyping and Dependent Types in Programming, APPSEM Workshop. INRIA technical report, 2000.

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

[55] 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.

[56] Robert D. Tennent. *Semantics of Programming Languages*. Prentice Hall, New York, 1991.

[57] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 82–93, N.Y., September 18–21 2000. ACM Press.

[58] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[59] 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.

[60] Howgwei Xi and Frank Pfenning. Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, January 1999. ACM.