Dependently Typed Meta-programming

Edwin Brady and Kevin Hammond

School of Computer Science,
University of St Andrews, St Andrews, Scotland.
Email: eb,kh@dcs.st-and.ac.uk.
Tel: +44-1334-463253, Fax: +44-1334-463278

Abstract

Dependent types and multi stage programming have both been used, separately, as implementation techniques for programming languages. Each technique has its own advantages — with dependent types, we can verify aspects of interpreters and compilers such as type safety and stack invariants. Multi stage programming, on the other hand, can give the implementor access to underlying compiler technology; a staged interpreter is a translator. In this paper, we investigate how we might combine these techniques to implement a compiler for a resource-safe functional programming language for embedded systems.

1 INTRODUCTION

In this paper, we consider how the separate techniques of multi stage programming and dependently typed programming can be combined in order to implement an efficient and safe compiler for a functional programming language. Types give a program meaning; dependent type systems, in which types may be predicated on values, allow us to give a more precise type to a program and therefore to be more confident that it has the intended meaning.

We use dependent types to implement a well-typed interpreter, similar to [2]. Writing this program shows *by construction* that the interpreter returns a value of the correct type and correctly evaluates well-typed terms. Dependent types ensure, by static checking, that the interpreter cannot be executed on badly formed or ill-typed code. Thus dependent types give static guarantees of certain desired correctness properties.

We further consider the use of staging annotations [20] to control the execution order of the interpreter. Staging annotations allow code generation to be deferred until run-time. This means that we can defer code generation for the interpreter until the object program is known — staging the interpreter yields a translator from the object language to the meta-language, from where it is a small step to generating a compiler for the object language. The combination of dependent types and multi stage programming takes us towards a method for building verified compilers.

Our work follows the ideas of [18], but differs in two important respects. Firstly, we need not be concerned with side-effecting computations, as with a de-

pendently typed, staged implementation of ML — we allow *full-spectrum* dependent types, making no syntactic distinction between types and values, and using techniques of [6, 4] to establish a phase distinction between compile-time only and run-time values. Secondly, we extend the idea of a *tagless* interpreter, which exploits dependent types to remove the need for run-time tagging of value types, to a *well-typed, well-scoped* interpreter, which exploits dependent types to give static guarantees about safety and exploits multi-stage programming to generate efficient run-time code.

1.1 The Core Type Theory, TT

We use as our core language a strongly normalising dependent type theory with inductive families [9], similar to Luo's UTT [12]. This language, which we call TT, is an enriched lambda calculus, with the usual properties of subject reduction, Church Rosser, and uniqueness of types. The strong normalisation property is guaranteed by allowing only primitive recursion over strictly positive inductive datatypes. The syntax of this language is given in Figure 1, and its typing rules in Figure 2. We may also abbreviate the function space $\forall x{:}S.\,T$ by $S \to T$ if x is not free in T. There is an infinite hierarchy of predicative universes, $\star_i : \star_{i+1}$; universe levels can be left implicit and inferred by the machine, as in [10].

$t ::=$	$\star_i$	(type universes)	$\mid$	x	(variable)
$\mid$	$\forall x{:}t.\,t$	(function space)	$\mid$	D	(inductive family)
$\mid$	$\lambda x{:}t.\,t$	(abstraction)	$\mid$	c	(constructor)
$\mid$	$t\,t$	(application)	$\mid$	D**-Elim**	(elimination rule)
$\mid$	$\underline{\text{let}}\, x \mapsto t : t\, \underline{\text{in}}\, t$	(let binding)			

FIGURE 1. The core language, TT

For clarity of the presentation, we use a higher level notation similar to the EPIGRAM notation of [15], which elaborates to TT. In the rest of this section, we give a brief introduction to programming within this notation. For a more detailed presentation of EPIGRAM see [14]; TT and its compilation scheme are detailed in [4].

1.2 Inductive Families

Inductive families are simultaneously defined collections of algebraic data types which can be indexed over values as well as types. For example, we can define a "lists with length" (or vector) type; to do this we first declare a type of natural numbers to represent such lengths, using the natural deduction style notation proposed for EPIGRAM in [15]:

$$\frac{\Gamma \vdash \underline{\text{valid}}}{\Gamma \vdash \star_n \ : \ \star_{n+1}} \ \textsf{Type}$$

$$\frac{\Gamma; x \ : \ S; \Gamma' \vdash \underline{\text{valid}}}{\Gamma; x \ : \ S; \Gamma' \vdash x \ : \ S} \ \textsf{Var}$$

(Similarly for c, D, D-**Elim**)

$$\frac{\Gamma; x \mapsto s \ : \ S; \Gamma' \vdash \underline{\text{valid}}}{\Gamma; x \mapsto s \ : \ S; \Gamma' \vdash x \ : \ S} \ \textsf{Val}$$

$$\frac{\Gamma \vdash f \ : \ \forall x{:}S.T \quad \Gamma \vdash s \ : \ S}{\Gamma \vdash f\, s \ : \ \underline{\text{let}}\, x \ : \ S \mapsto s \,\underline{\text{in}}\, T} \ \textsf{App}$$

$$\frac{\Gamma; x \ : \ S \vdash e \ : \ T \quad \Gamma \vdash \forall x{:}S.T \ : \ \star_n}{\Gamma \vdash \lambda x{:}S.e \ : \ \forall x{:}S.T} \ \textsf{Lam}$$

$$\frac{\Gamma; x \ : \ S \vdash T \ : \ \star_n \quad \Gamma \vdash S \ : \ \star_n}{\Gamma \vdash \forall x{:}S.T \ : \ \star_n} \ \textsf{Forall}$$

$$\frac{\Gamma \vdash e_1 \ : \ S \quad \Gamma; x \mapsto e_1 \ : \ S \vdash e_2 \ : \ T \quad \Gamma \vdash S \ : \ \star_n \quad \Gamma; x \mapsto e_1 \ : \ S \vdash T \ : \ \star_n}{\Gamma \vdash \underline{\text{let}}\, x \ : \ S \mapsto e_1 \,\underline{\text{in}}\, e_2 \ : \ \underline{\text{let}}\, x \ : \ S \mapsto e_1 \,\underline{\text{in}}\, T} \ \textsf{Let}$$

$$\frac{\Gamma \vdash x \ : \ A \quad \Gamma \vdash A' \ : \ \star_n \quad \Gamma \vdash A \simeq A'}{\Gamma \vdash x \ : \ A'} \ \textsf{Conv}$$

FIGURE 2. Typing rules for TT

$$\underline{\text{data}} \quad \frac{}{\mathbb{N} \ : \ \star} \quad \underline{\text{where}} \quad \frac{}{0 \ : \ \mathbb{N}} \quad \frac{n \ : \ \mathbb{N}}{\mathsf{s}n \ : \ \mathbb{N}}$$

It is straightforward to define addition and multiplication by primitive recursion. Then we may make the following declaration of vectors; note that nil only targets vectors of length zero, and $\mathsf{cons}\, x\, xs$ only targets vectors of length greater than zero:

$$\underline{\text{data}} \quad \frac{A \ : \ \star \quad n \ : \ \mathbb{N}}{\mathsf{Vect}\, A\, n \ : \ \star} \quad \underline{\text{where}} \quad \frac{}{\varepsilon \ : \ \mathsf{Vect}\, A\, 0} \qquad \frac{x \ : \ A \quad xs \ : \ \mathsf{Vect}\, A\, k}{x{::}xs \ : \ \mathsf{Vect}\, A\, (\mathsf{s}k)}$$

We leave A and k as implicit arguments to the infix constructor :: — their types can be inferred from the type of Vect. When the type includes explicit length information like this, it follows that a function over that type will express the invariant properties of the length. For example, the type of the following program **vPlus**, which adds corresponding numbers in each vector, expresses the invariant that the input vectors are the same length as the output:

$$\underline{\text{let}} \quad \frac{xs, ys \ : \ \mathsf{Vect}\,\mathbb{N}\,n}{\mathbf{vPlus}\,xs\,ys \ : \ \mathsf{Vect}\,\mathbb{N}\,n}$$

$$\begin{array}{lll} \mathbf{vPlus} & xs & ys \quad \Leftarrow \underline{\text{elim}}\,xs, \underline{\text{case}}\,ys \\ \mathbf{vPlus} & \varepsilon & \varepsilon \quad \mapsto \mathsf{nil} \\ \mathbf{vPlus}\,(x :: xs) & (y :: ys) & \mapsto (x + y) :: (\mathbf{vPlus}\,xs\,ys) \end{array}$$

The elim and case notation invoke the primitive recursion and case analysis operators respectively on xs and ys; termination is guaranteed since these operators are the only means to inspect data. Unlike in a simply typed language, we do not need to give error handling cases when the lengths of the vectors do not match; the typechecker verifies that these cases are impossible.

1.3 Theorem Proving

The dependent type system of TT also allows us to express properties directly. For example, the following heterogeneous definition of equality, due to McBride [13], is built in to TT (rather than introduced as a datatype, so we omit the data keyword):

$$\frac{a \ : \ A \quad b \ : \ B}{a = b \ : \ \star} \qquad \frac{A \ : \ \star \quad a \ : \ A}{\mathsf{refl}\,a \ : \ a = a}$$

This definition introduces an infix type constructor, $=$, parametrised over two types; we can declare equality between any two types, but can only construct an instance of equality between two definitionally equal values in the same type; e.g. $\mathsf{refl}\,(\mathsf{s}\,\mathsf{0})$ is an instance of a proof that $\mathsf{s}\,\mathsf{0} = \mathsf{s}\,\mathsf{0}$. Furthermore, since equality is an ordinary datatype just like $\mathbb{N}$ and Vect, we can also write programs by case analysis on instances of equality, such as the following program which can be viewed as a proof that s respects equality:

$$\underline{\text{let}} \quad \frac{p \ : \ n = m}{\mathbf{resp\_s}\,p \ : \ (\mathsf{s}\,n) = (\mathsf{s}\,m)}$$

$$\begin{array}{lll} \mathbf{resp\_s} & p & \Leftarrow \underline{\text{case}}\,p \\ \mathbf{resp\_s}\,(\mathsf{refl}\,n) & & \mapsto \mathsf{refl}\,(\mathsf{s}\,n) \end{array}$$

We can also represent more complex properties, such as the less than or equal relation:

$$\underline{\text{data}} \quad \frac{x, y \ : \ \mathbb{N}}{x \leq y \ : \ \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{leO} \ : \ 0 \leq y} \qquad \frac{p \ : \ x \leq y}{\mathsf{leS}\,p \ : \ \mathsf{s}\,x \leq \mathsf{s}\,y}$$

Note that x and y can be left implicit, as their types (and even their values) can be inferred from the type of the relation. For example, $\mathsf{leS}\,(\mathsf{leS}\,\mathsf{leO})$ could represent a proof of $\mathsf{s}\,(\mathsf{s}\,\mathsf{0}) \leq \mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{0}))$.

As with equality, given a proof, we can write programs by recursion over the proof. For example, we can write a safe subtraction function (i.e. the result is guaranteed to be non-negative) by primitive recursion over the proof that the second argument is less than or equal to the first:

$$\underline{\text{let}} \quad \frac{n,m : \mathbb{N} \quad p : m \leq n}{\mathbf{minus}\, n\, m\, p \; : \; \mathbb{N}}$$

$$\begin{array}{lllll} \mathbf{minus} & n & m & p & \Leftarrow \underline{\text{elim}}\, p \\ \mathbf{minus} & n & \mathsf{0} & (\mathsf{leO}\, n) & \mapsto n \\ \mathbf{minus} & (\mathsf{s}\, n) & (\mathsf{s}\, m) & (\mathsf{leS}\, p) & \mapsto \mathbf{minus}\, n\, m\, p \end{array}$$

The values for the arguments n and m are determined by the indices of leO and leS; no case analysis on the numbers themselves is required. The Curry-Howard isomorphism [8, 11] describes this correspondence between proofs and programs.

The lack of a phase distinction between types and values means that we can write proofs like this *directly* over programs; there is no need to duplicate values at the kind level. It is commonly believed [7, 19] that this lack of phase distinction means that we cannot erase types at run-time; however, we maintain type erasure by establishing a distinction between compile-time and run-time values instead [16].

2 A DEPENDENTLY TYPED INTERPRETER

Dependent types can be used to good effect in the implementation of programming languages. One demonstratation of this is Augustsson and Carlsson's well-typed interpreter in Cayenne [2]. The interpreter we present here uses inductive families to represent well-typedness and synchronisation of type and value environments. This language, given in Figure 3, augments Augustsson and Carlsson's implementation with a primitive recursion operator for natural numbers, primrec.

$e ::=$	$\lambda a : t.\, e$	λ-abstraction	$\mid$	$e\, e$	application
$\mid$	a	variable	$\mid$	$e + e$	addition
$\mid$	$e \leq e$	less than or equal	$\mid$	e and e_2	boolean and
$\mid$	n	number	$\mid$	b	boolean value
$\mid$	primrec $e\, e\, e$	primitive recursion			
$t ::=$	$\mathbb{N}$	Natural numbers	$\mid$	Bool	Booleans
$\mid$	$t \to t$	Function type			

FIGURE 3. The object language

2.1 Representation

This language can be represented as an inductive family which, by indexing over the type environment and the type of an expression, ensures that only well-typed expressions can be built.

Since the value returned by the interpreter is a type in the implementation language, we implement type environments as a vector of types. We represent type environments as vectors of types, and membership of a type environment as a relation (Figure 4). top and pop can be viewed as zero and successor constructors of a natural number representing a de Bruijn index.

$$\underline{\text{let}} \quad \frac{n : \mathbb{N}}{\mathsf{Env}\, n : \star} \quad \mathsf{Env}\, n \mapsto \mathsf{Vect} \star n$$

$$\underline{\text{data}} \quad \frac{G : \mathsf{Env}\, n \quad i : \mathsf{Fin}\, n \quad t : \star}{\mathsf{Var}\, G\, i\, t : \star}$$

$$\underline{\text{where}} \quad \frac{}{\mathsf{top} : \mathsf{Var}\,(s::G)\,\mathsf{f0}\, s} \quad \frac{v : \mathsf{Var}\, G\, i\, t}{\mathsf{pop}\, v : \mathsf{Var}\,(s::G)\,(\mathsf{fs}\, i)\, t}$$

FIGURE 4. Type environments

The Expr family, which represents expressions in the object language, is shown in Figure 5. It is indexed over the type environment in which it will be evaluated, and the type of value it represents. Therefore any well-typed instance of Expr *must* be a representation of a well-typed term; this is a static guarantee.

$$\underline{\text{data}} \quad \frac{G : \mathsf{Env}\, n \quad A : \star}{\mathsf{Expr}\, G\, A : \star} \quad \underline{\text{where}}$$

$$\frac{k : \mathbb{N}}{\mathsf{enat}\, k : \mathsf{Expr}\, G\, \mathbb{N}} \qquad \frac{b : \mathsf{Bool}}{\mathsf{ebool}\, b : \mathsf{Expr}\, G\, \mathsf{Bool}}$$

$$\frac{f : \mathsf{Expr}\, G\,(s \to t) \quad a : \mathsf{Expr}\, G\, s}{\mathsf{eapp}\, f\, a : \mathsf{Expr}\, G\, t} \qquad \frac{e : \mathsf{Expr}\,(s::G)\, t}{\mathsf{elam}\, e : \mathsf{Expr}\, G\,(s \to t)}$$

$$\frac{a, b : \mathsf{Expr}\, G\, \mathbb{N}}{\mathsf{eadd}\, a\, b : \mathsf{Expr}\, G\, \mathbb{N}} \qquad \frac{a, b : \mathsf{Expr}\, G\, \mathbb{N}}{\mathsf{ele}\, a\, b : \mathsf{Expr}\, G\, \mathsf{Bool}}$$

$$\frac{a, b : \mathsf{Expr}\, G\, \mathsf{Bool}}{\mathsf{eand}\, a\, b : \mathsf{Expr}\, G\, \mathsf{Bool}} \qquad \frac{v : \mathsf{Var}\, G\, i\, t}{\mathsf{evar}\, v : \mathsf{Expr}\, G\, t}$$

$$\frac{x : \mathsf{Expr}\, G\, \mathbb{N} \quad z : \mathsf{Expr}\, G\, A \quad s : \mathsf{Expr}\, G\,(\mathbb{N} \to A \to A)}{\mathsf{eprimrec}\, x\, z\, s : \mathsf{Expr}\, G\, A}$$

FIGURE 5. Interpreter type declaration

An an example, multiplication could be defined by primitive recursion as follows:

$$\mathbf{mult} = \lambda x{:}\mathbb{N}.\, \lambda y{:}\mathbb{N}.\, \mathsf{primrec}\, x\, 0\, (\lambda k{:}\mathbb{N}.\, \lambda ih{:}\mathbb{N}.\, y + ih)$$

The representation of this as an Expr is:

$$\begin{aligned}\textbf{mult} = {} & (\mathsf{elam}\,(\mathsf{elam}\,(\mathsf{eprimrec}\,(\mathsf{pop}\,\mathsf{top})\,(\mathsf{enat}\,0)\\ & \qquad (\mathsf{elam}\,(\mathsf{elam}\,(\mathsf{eadd}\,(\mathsf{evar}\,(\mathsf{pop}\,\mathsf{pop}\,\mathsf{top}))\,(\mathsf{evar}\,\mathsf{top})))))))\end{aligned}$$

The interpreter has a value environment in which to look up the values of variables. Since variables in the environment may have different types, using a Vect is not appropriate. Instead, we synchronise it with the type environment; each value in the value environment gets its type from the corresponding entry in the type environment. The declaration of the value environment is given in figure 6, along with a lookup function.

$$\underline{\text{data}} \quad \frac{G : \mathsf{Env}\,n}{\mathsf{ValEnv}\,G : \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{empty} : \mathsf{ValEnv}\,\varepsilon} \qquad \frac{t : T \quad r : \mathsf{ValEnv}\,G}{\mathsf{extend}\,t\,r : \mathsf{ValEnv}\,(T::G)}$$

$$\underline{\text{let}} \quad \frac{v : \mathsf{Var}\,G\,i\,T \quad ve : \mathsf{ValEnv}\,G}{\textbf{envLookup}\,v\,ve : T}$$

$$\begin{aligned}&\textbf{envLookup} \quad v \quad ve \quad \Leftarrow \underline{\text{elim}}\,v\\ &\textbf{envLookup} \quad \mathsf{top} \quad (\mathsf{extend}\,t\,r) \quad \mapsto t\\ &\textbf{envLookup}\,(\mathsf{pop}\,v)\,(\mathsf{extend}\,t\,r) \quad \mapsto \textbf{envLookup}\,v\,r\end{aligned}$$

FIGURE 6. Value environments

The type environment helps to ensure that we can only represent well-typed terms. The value environment is used with the interpreter to ensure that any value we project out of the environment has the correct type. The invariants on **envLookup** guarantee that we can never project a non-existent value out of the environment. It is not possible to project a value from the empty environment; such an operation would be ill-typed. This means that there is no need for any error checking in the interpreter; we know *by construction* that all input is well-typed and the output will be of the correct type.

2.2 The Interpreter

The implementation of the interpreter is shown in Figure 7. **interp** is written by structural recursion over the input expression x. It returns a semantic representation, as a TT term, of the input expression. So, for example, the interpretation of a λ-abstraction (elam) is a TT function which implements that λ-abstraction. Interpretation of an application then simply applies the function to the interpretation of its argument. Note that in the case for elam, we use the implicit argument s to establish the input type of the function. This approach is similar to normalisation by evaluation [3] in that we construct a semantic representation of the term to be

interpreted, but there is no reification back to the object language here. This is a *tagless* interpreter; i.e., there is no need to tag the return value with its type, since the input depends on its type.

$$\underline{\text{let}} \quad \frac{x : \mathsf{Expr}\, G\, T \qquad ve : \mathsf{ValEnv}\, G}{\mathbf{interp}\, x\, ve : T}$$

$$\begin{array}{lcll}
\mathbf{interp} & x & ve & \Leftarrow \underline{\mathrm{elim}}\, x \\
\mathbf{interp} & (\mathsf{enat}\, k) & ve & \mapsto k \\
\mathbf{interp} & (\mathsf{ebool}\, b) & ve & \mapsto b \\
\mathbf{interp} & (\mathsf{eapp} f\, a) & ve & \mapsto (\mathbf{interp}\, f\, ve)\, (\mathbf{interp}\, a\, ve) \\
\mathbf{interp} & (\mathsf{elam}_s\, e) & ve & \mapsto \lambda a : s.\, \mathbf{interp}\, e\, (\mathsf{extend}\, a\, ve) \\
\mathbf{interp} & (\mathsf{eadd}\, a\, b) & ve & \mapsto \mathbf{plus}\, (\mathbf{interp}\, a\, ve)\, (\mathbf{interp}\, b\, ve) \\
\mathbf{interp} & (\mathsf{ele}\, a\, b) & ve & \mapsto \mathbf{le}\, (\mathbf{interp}\, a\, ve)\, (\mathbf{interp}\, b\, ve) \\
\mathbf{interp} & (\mathsf{eand}\, a\, b) & ve & \mapsto \mathbf{and}\, (\mathbf{interp}\, a\, ve)\, (\mathbf{interp}\, b\, ve) \\
\mathbf{interp} & (\mathsf{evar}\, v) & ve & \mapsto \mathbf{envLookup}\, v\, ve \\
\mathbf{interp} & (\mathsf{eprimrec}\, x\, z\, s) & ve & \mapsto \mathbf{primrec}\, (\mathbf{interp}\, x\, ve)\, (\mathbf{interp}\, z\, ve)\, (\mathbf{interp}\, s\, ve)
\end{array}$$

$$\underline{\text{let}} \quad \frac{n : \mathbb{N} \qquad z : A \qquad s : \mathbb{N} \to A \to A}{\mathbf{primrec}\, n\, z\, s : A}$$

$$\begin{array}{lcll}
\mathbf{primrec} & n & z\, s & \Leftarrow \underline{\mathrm{elim}}\, n \\
\mathbf{primrec} & 0 & z\, s & \mapsto z \\
\mathbf{primrec} & (\mathsf{s}\, k) & z\, s & \mapsto s\, k\, (\mathbf{primrec}\, k\, z\, s)
\end{array}$$

FIGURE 7. The interpreter

Evaluating the **mult** function defined above gives, as expected, the meta-level implementation of multiplication by primitive recursion:

$$\mathbf{mult} = \lambda x, y : \mathbb{N}.\, \mathbf{primrec}\, x\, 0\, (\lambda k, ih : \mathbb{N}.\, \mathbf{plus}\, y\, ih)$$

2.3 Compiling

If we compile this program, we get an executable interpreter for the object language, with all the associated overheads that interpretation brings— in particular, we still have a representation of the object program as data, which is analysed at run-time. However, if we were to partially evaluate the interpretation of a known object program, we would get a representation of the program *in the meta-language*, as with **mult** above. But in general we do not know anything about the object program at compile-time; multi stage programming allows us to defer the partial evaluation until run-time, when the object program *is* known.

3 DEPENDENT TYPES AND STAGING

In Taha's introduction to multi-stage programming [21] he states that "a staged interpreter is a translator". The idea is to defer code generation for staged fragments until run-time; in the case of an interpreter, the program generates a meta-language implementation of the object program, eliminating the interpretation overhead. Taha implements such an interpreter in MetaOCaml, a multi-stage variant of OCaml. This loses the compile-time guarantees of the dependently typed version of Section 2; there is no statically checkable guarantee that the object program is well-typed or even well-scoped.

In this section, we apply Taha's staging technique to the TT language and show how we can modify our dependently typed interpreter to take advantage of staging annotations.

3.1 Extensions to TT

To make TT multi-stage, we extend it with the notion of *levels*. Level 0 is the run-time execution level; higher levels contain deferred code. We index the typing judgment with the level n, i.e. $\Gamma \vdash nx : A$ states that x has type A at level n. We extend TT with the following expression forms (summarised in Figure 8):

- $'e$, which quotes an expression e, deferring its execution until the next stage. This lifts the expression from level n to level $n+1$
- $\langle e \rangle$, which is the "code" type of a quoted term.
- $!e$, which evaluates a quoted expresion e. Since this executes code, it is only valid at level 0.
- $\sim e$, which splices (escapes) a quoted term into a lower level. This moves an expression from level $n+1$ to level n; n cannot be level 0, since this would imply execution.

The typing rules are given in Figure 9.

$$
\begin{array}{lllll}
e ::= & \ldots & & & \\
 & | \;\; 'e & \text{(Quoted term)} & | \;\; \langle e \rangle & \text{(Code type)} \\
 & | \;\; !e & \text{(Evaluate term)} & | \;\; \sim e & \text{(Escape term)}
\end{array}
$$

FIGURE 8. Syntax extensions for staging constructs

The two basic notions of equivalence are:

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash !('e) = e} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash \sim('e) = e}$$

$$\frac{\Gamma \vdash_n e \,:\, A}{\Gamma \vdash_n {}'e \,:\, \langle A \rangle}\ \mathsf{Quote} \qquad \frac{\Gamma \vdash_{n+1} A \,:\, \star}{\Gamma \vdash_n \langle A \rangle \,:\, \star}\ \mathsf{Code}$$

$$\frac{\Gamma \vdash_0 e \,:\, \langle A \rangle}{\Gamma \vdash_0 !e \,:\, A}\ \mathsf{Eval} \qquad \frac{\Gamma \vdash_{n+} e \,:\, \langle A \rangle}{\Gamma \vdash_{n+} {\sim}e \,:\, A}\ \mathsf{Escape}$$

FIGURE 9. Typing rules for staging constructs

The distinction between the escape and evaluation constructs is that escape must be enclosed by quotation brackets. Extending an implementation of TT with these constructs is fairly straightforward; the one aspect needing care is conversion of quoted terms:

$$\frac{\Gamma \vdash x, y \,:\, A \quad \Gamma \vdash x \simeq y}{\Gamma \vdash {}'x \simeq {}'y}$$

This rule states that quoted terms are definitionally equal if the values they quote are definitionally equal. An implementation of conversion which simply compares normal forms would not implement this rule correctly, so should be extended accordingly.

3.2 Example — power

A classic example of the benefit of multi-stage programming is the **power** function, which returns the nth power of m. We can define this as follows:

$$\underline{\mathrm{let}} \quad \frac{n, m \,:\, \mathbb{N}}{\mathbf{power}\, n\, m \,:\, \mathbb{N}}$$

$$\begin{aligned}
&\mathbf{power} \quad n \quad m \;\Leftarrow \underline{\mathrm{elim}}\, n \\
&\mathbf{power} \quad \mathsf{0} \quad m \;\mapsto \mathsf{s0} \\
&\mathbf{power}\, (\mathsf{s}\, k)\, m \;\mapsto m \times (\mathbf{power}\, k\, m)
\end{aligned}$$

This is a generic power function, in that it can be used to compute a value raised to any non-negative exponent. However, there is a price to pay for genericity; if the function is often used with the *same* value of n, our program still has to perform the work associated with this input on every application.

We could, at compile time, create specialised instances of this function for commonly used inputs, e.g.:

$$\begin{aligned}
&\mathbf{power2} \mapsto \mathbf{power}\, (\mathsf{s}\, (\mathsf{s0})) \\
&\mathbf{power4} \mapsto \mathbf{power}\, (\mathsf{s}\, (\mathsf{s}\, (\mathsf{s}\, (\mathsf{s0}))))
\end{aligned}$$

These can be partially evaluated at compile-time, but this requires knowing in advance what the specialised inputs are. What if we do not know until run-time? We do not have a way, in the language as it stands, of partially evaluating at run-time to create these specialised instances on dynamic data.

Staging annotations, combined with run-time code generation, give us a method for creating these specialised instances dynamically.

$$\underline{\text{let}} \quad \frac{n \,:\, \mathbb{N} \qquad m \,:\, \langle \mathbb{N} \rangle}{\mathbf{power'}\, n\, m \,:\, \langle \mathbb{N} \rangle}$$

$$\begin{array}{l} \mathbf{power'} \quad n \quad m \Leftarrow \underline{\text{elim}}\, n \\ \mathbf{power'} \quad \mathsf{0} \quad m \;\mapsto\; \text{'}(\mathsf{s}\,\mathsf{0}) \\ \mathbf{power'}\,(\mathsf{s}\,k)\, m \;\mapsto\; \text{'}(m \times \sim(\mathbf{power'}\, k\, m)) \end{array}$$

Now, reading an input at run-time, we dynamically generate a specialised **power** function; for example, with the input $\mathsf{s}\,(\mathsf{s}\,\mathsf{0})$:

$$\mathbf{power'}\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{0})) = \lambda m{:}\langle \mathbb{N} \rangle .\, \text{'}(m \times m \times (\mathsf{s}\,\mathsf{0}))$$

We run such generated code with the ! construct. For example, **power2** can now be defined as follows:

$$\mathbf{power2} = !(\text{'}\lambda m{:}\mathbb{N}.\ \sim(\mathbf{power'}\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{0}))\,\text{'}m))$$

The ! construct compiles and executes its argument; at run-time, this will generate code for a specialised power of two function. This behaves in the same way as our earlier unstaged definition of **power2** but with the partial evaluation deferred until run-time.

Although this is a somewhat artificial example, it illustrates the difference between specialising functions at compile-time and run-time. This staging technique has greater benefit where we have a large, commonly used structure which is nevertheless not known until run-time — for example, a syntax tree for a program which is to be interpreted.

3.3 A Staged Interpreter

We observed in section 2.3 that partial evaluation of an object program yields a representation of the program in the meta-language; effectively this is a translated version of the object program. If we can defer this partial evaluation until run-time, then we do not need to know the object program until run-time, removing the interpretation overhead and generating a compiler for the object language. In the previous section, we saw that staging a program allowed us to defer partial evaluation of the **power** function until run-time; let us now see how we can apply this technique to the interpreter of section 2.2.

Figure 10 gives a staged version of the interpreter. The basic definition is as before, but with staging annotations added to denote where code generation should be deferred until run-time.

Note that there are no staging annotations on the call to **envLookup**. This is because we would also like to partially evaluate this function — thus projection of variables from the value environment is done only once. We stage **envLookup** as follows:

$$\underline{\text{let}} \quad \frac{x : \mathsf{Expr}\, G\, T \quad ve : \mathsf{ValEnv}\, G}{\mathbf{interp}\, x\, ve : \langle T \rangle}$$

$$\begin{array}{lllll}
\mathbf{interp} & x & ve & \Leftarrow \underline{\text{elim}}\, x \\
\mathbf{interp} & (\mathsf{enat}\, k) & ve & \mapsto \text{'}k \\
\mathbf{interp} & (\mathsf{ebool}\, b) & ve & \mapsto \text{'}b \\
\mathbf{interp} & (\mathsf{eapp}\, f\, a) & ve & \mapsto \text{'}(\sim(\mathbf{interp}\, f\, ve)\ \sim(\mathbf{interp}\, a\, ve)) \\
\mathbf{interp} & (\mathsf{elam}_s\, e) & ve & \mapsto \text{'}(\lambda a : s.\ \sim(\mathbf{interp}\, e\, (\mathsf{extend}\, a\, ve))) \\
\mathbf{interp} & (\mathsf{eadd}\, a\, b) & ve & \mapsto \text{'}(\mathbf{plus}\ \sim(\mathbf{interp}\, a\, ve)\ \sim(\mathbf{interp}\, b\, ve)) \\
\mathbf{interp} & (\mathsf{ele}\, a\, b) & ve & \mapsto \text{'}(\mathbf{le}\ \sim(\mathbf{interp}\, a\, ve)\ \sim(\mathbf{interp}\, b\, ve)) \\
\mathbf{interp} & (\mathsf{eand}\, a\, b) & ve & \mapsto \text{'}(\mathbf{and}\ \sim(\mathbf{interp}\, a\, ve)\ \sim(\mathbf{interp}\, b\, ve)) \\
\mathbf{interp} & (\mathsf{evar}\, v) & ve & \mapsto \mathbf{envLookup}\, v\, ve \\
\mathbf{interp} & (\mathsf{eprimrec}\, x\, z\, s) & ve & \mapsto \text{'}(\mathbf{primrec}\, (\mathbf{interp}\, x\, ve)\, (\mathbf{interp}\, z\, ve)\, (\mathbf{interp}\, s\, ve))
\end{array}$$

$$\underline{\text{let}} \quad \frac{n : \mathbb{N} \quad z : A \quad s : \mathbb{N} \to A \to A}{\mathbf{primrec}\, n\, z\, s : \langle A \rangle}$$

$$\begin{array}{llll}
\mathbf{primrec} & n & z\, s & \Leftarrow \underline{\text{elim}}\, n \\
\mathbf{primrec} & 0 & z\, s & \mapsto \text{'}z \\
\mathbf{primrec} & (\mathsf{s}\, k) & z\, s & \mapsto \text{'}(s\, k\ \sim(\mathbf{primrec}\, k\, z\, s))
\end{array}$$

FIGURE 10. The staged interpreter

$$\underline{\text{let}} \quad \frac{v : \mathsf{Var}\, G\, i\, T \quad ve : \mathsf{ValEnv}\, G}{\mathbf{envLookup}\, v\, ve : \langle T \rangle}$$

$$\begin{array}{llll}
\mathbf{envLookup} & v & ve & \Leftarrow \underline{\text{elim}}\, v \\
\mathbf{envLookup} & \mathsf{top} & (\mathsf{extend}\, t\, r) & \mapsto \text{'}t \\
\mathbf{envLookup} & (\mathsf{pop}\, v) & (\mathsf{extend}\, t\, r) & \mapsto \mathbf{envLookup}\, v\, r
\end{array}$$

Evaluation of the **mult** function gives a quoted representation of the meta-level implementation of multiplication:

$$\mathbf{mult} = \text{'}(\lambda x, y : \mathbb{N}.\ \mathbf{primrec}\, x\, 0\, (\lambda k, ih : \mathbb{N}.\ \mathbf{plus}\, y\, ih))$$

So what have we gained here that we did not have when partially evaluating as before? The result is the same, with the addition of the staging annotations. However, we have deferred the partial evaluation until run-time. Rather than an interpretation overhead every time we want to run this object language function, we have a single compilation overhead, with code generated for the quoted function at run-time. From here, it is a small step to implement a compiler which outputs machine code for this program via the meta-language compiler.

3.4 Towards Verified Compilers

We have implemented the data type for the object language in such a way as to be able to verify properties of the interpreter simply by writing it. By using full-

spectrum dependent types to implement the interpreter, we have a guarantee that input to the interpreter is well-scoped and well-typed, and that the output will be the correct type. Our representation of the object language, in particular the invariants which it satisfies, means that we know the properties that the interpreter satisfies *by construction* rather than by post-hoc theorem proving.

Staging lets us take this a step further — from the representation of the object language and the staging of the interpreter, we generate a correct compiler, by construction. This is on the condition that the compiler for the meta-language generates correct code, but this only has to be shown once. Proving correctness of the compiler for the meta-language thus gives correctness properties of any language we implement in this way.

4 DEPENDENT TYPES FOR RESOURCE ANALYSIS

We have previously considered the use of dependent types to give static guarantees of resource properties of functional programs [5]. Obtaining accurate information about the run-time time and space behaviour of computer software is important in a number of areas. One of the most significant of these is embedded systems. Embedded systems are becoming an increasingly important application area: today, more than 98% of *all* processors are used in embedded systems and the number of processors employed in such systems is increasing year on year.

The basic idea behind our use of dependent types in resource aware functional programming is to predicate each user defined type on a natural number; each function then returns a value combined with its size and a proof that the size satisfies a given predicate, as represented by the following Size type:

$$\underline{\text{data}} \quad \frac{A \;:\; \mathbb{N} \to \star \qquad P \;:\; \forall n{:}\mathbb{N}.\, A\, n \to \star}{\mathsf{Size}\, A\, P \;:\; \star}$$

$$\underline{\text{where}} \quad \frac{val \;:\; A\, n \qquad p \;:\; P\, n\, val}{\mathsf{size}\, val\, p \;:\; \mathsf{Size}\, A\, P}$$

For example, we can implement a size-safe list append function by predicating a List type on a natural number, and implementing **append** as follows, where **resp_s** is a proof that the successor constructor on natural numbers respects equality:

$$\underline{\text{data}} \quad \frac{A \;:\; \mathbb{N} \to \star}{\mathsf{List_S}\, A \;:\; \mathbb{N} \to \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{nil_S} \;:\; \mathsf{List_S}\, A\, 0} \qquad \frac{x \;:\; A\, xn \qquad xs \;:\; \mathsf{List_S}\, A\, xsn}{\mathsf{cons_S}\, x\, xs \;:\; \mathsf{List_S}\, A\, (\mathsf{s}\, xsn)}$$

$$\underline{\text{let}} \quad \frac{xs \;:\; \mathsf{List_S}\, A\, xsn \qquad ys \;:\; \mathsf{List_S}\, A\, ysn}{\mathbf{append}\, xs\, ys \;:\; (\mathcal{S}\,(n,v) \;:\; \mathsf{List_S}\, A.\, n = xsn + ysn)}$$

$$\begin{aligned}
&\mathbf{append} \quad \mathsf{nil_S} \quad ys \;\mapsto\; \mathsf{size}\, ys\, (\mathsf{refl}\, ysn)\\
&\mathbf{append}\, (\mathsf{cons_S}\, x\, xs)\, ys \;\mapsto\; \underline{\text{let}}\, (\mathsf{size}\, val\, p) \;=\; \mathbf{append}\, xs\, ys\, \underline{\text{in}}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{size}\, (\mathsf{cons_S}\, x\, val)\, (\mathbf{resp_s}\, p)
\end{aligned}$$

However, we have not yet considered in detail how we might execute these programs. It is important when constructing such a resource analysis that the proofs do not interfere with the execution costs of the program, otherwise the costs are no longer valid! While we can use some of the techniques of [4] to mitigate the problem, these will not completely remove the cost information. A further (potentially error-prone) pass is required in the implementation to translate Size structures into simple values.

A staged interpreter for a resource aware functional language would allow us to remove the cost information in a principled way. We extend the Size type above to represent code as well as data, and build a staged translator for the resulting language. In this way, we verify the resource properties with the dependent type system, and construct an efficient compiled program through multi stage programming. In future work, we plan to use the techniques we have described in this paper to implement a verified compiler for a resource-safe functional programming language.

5 RELATED WORK

Our work brings together the related areas of dependently typed programming [1] and multi-stage programming [20], following the ideas of [18, 17]. One important difference in our work is the use of full-spectrum dependent types; there is no syntactic distinction between types and terms. An advantage of this is that it becomes easier to express properties of a program in the type, without the need to duplicate values at the kind level. We do not need to maintain a phase distinction between types and values; instead, we maintain a phase distinction between compile-time and run-time values using techniques of [4, 16].

Furthermore, our language does not have effects such as non-termination. This is important if we want our programs to have verifiable static guarantees; non-termination (via a fixpoint combinator with type $\forall P : \star.\ (P \to P) \to P$) introduces an inconsistency into our programs which means that proofs of properties in the program can no longer be trusted. Further examples of programming in this way, including the rationale, are presented in [1]. By adding staging annotations to a logically sound type theory we aim to implement compilers with verifiable static guarantees.

6 CONCLUSION

We have looked at how dependently typed programming and multi stage programming can be used as independent techniques for various aspects of the implementation of a functional programming language. Each has its own advantages, and we have observed that the two techniques can be combined effectively.

While partial evaluation of a strongly normalising program can give a semantic representation of an object program in the meta language without any need for

adding staging annotations, we do get an important futher benefit from adding these annotations — namely, that we get a *machine code* representation of the object program, further to the *meta-language code* provided by partial evaluation. By combining the two techniques of dependently typed programming and multi stage programming, we can implement an efficient compiler for a resource aware functional language with strong static guarantees. This work takes us a step closer to a general method for implementing verified compilers.

ACKNOWLEDGEMENTS

This work is generously supported by EPSRC grants GR/R70545 and EP/C001346/1 and by EU Framework VI IST-510255 (EmBounded).

REFERENCES

[1] T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter, 2005. Submitted for publication.

[2] L. Augustsson and M. Carlsson. An exercise in dependent types: A well-typed interpreter. `http://www.cs.chalmers.se/~augustss/cayenne/`, 1999.

[3] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society Press, 1991.

[4] E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

[5] E. Brady and K. Hammond. A dependently typed framework for static analysis of program execution costs. In *Implementation of Functional Languages 2005*. Springer, 2006.

[6] E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs 2003*, volume 3085, pages 115–129. Springer, 2004.

[7] L. Cardelli. Phase distinctions in type theory. Manuscript, 1988.

[8] H. B. Curry and R. Feys. *Combinatory Logic, volume 1*. North Holland, 1958.

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

[10] R. Harper and R. Pollack. Type checking with universes. *Theoretical Computer Science*, 89(1):107–136, 1991.

[11] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B.Curry: Essays on combinatory logic, lambda calculus and formalism*. Academic Press, 1980. A reprint of an unpublished manuscript from 1969.

[12] Z. Luo. *Computation and Reasoning – A Type Theory for Computer Science*. International Series of Monographs on Computer Science. OUP, 1994.

[13] C. McBride. *Dependently Typed Functional Programs and their proofs*. PhD thesis, University of Edinburgh, May 2000.

[14] C. McBride. Epigram: Practical programming with dependent types. Lecture Notes, International Summer School on Advanced Functional Programming, 2004.

[15] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[16] J. McKinna and E. Brady. Phase distinctions in the compilation of Epigram, 2005. Draft.

[17] E. Pasalic. *The Role of Type-Equality in Meta-programming*. PhD thesis.

[18] E. Pasalic, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *International Conference on Functional Programming*. ACM, 2002.

[19] T. Sheard, J. Hook, and N. Linger. GADTs + extensible kinds = dependent programming, 2005.

[20] W. Taha. *Multi-stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[21] W. Taha. A gentle introduction to multi-stage programming, 2003. Available from `http://www.cs.rice.edu/~taha/publications/journal/dspg04a.pdf`.