A Verified Staged Interpreter is a Verified Compiler

Multi-stage Programming with Dependent Types

Edwin Brady Kevin Hammond

School of Computer Science, University of St Andrews, St Andrews, Scotland.
Email: eb,kh@dcs.st-and.ac.uk

Abstract

Dependent types and multi-stage programming have both been used, separately, in programming language design and implementation. 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 the combination of these techniques. We implement an interpreter for a simply typed lambda calculus, using dependent types to guarantee correctness properties by construction. We give explicit proofs of these correctness properties, then add staging annotations to generate a translator from the interpreter. In this way, we have constructed a verified compiler from a verified staged interpreter. We illustrate the application of the technique by considering a simple staged interpreter that provides guarantees for some simple resource bound properties, as might be found in a domain specific language for real-time embedded systems.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — Interpreters, Compilers, Translator writing systems and compiler generators; D.2.4 [*Software Engineering*]: Software/Program Verification — Correctness proofs, Formal methods

General Terms Languages, Theory, Verification

Keywords Dependent types, Multi-stage programming, Partial evaluation, Domain Specific Language Implementation, Resource aware programming, Functional programming

1. Introduction

Multi-stage programming supports separation of concerns in compiler writing, by allowing automatic program generation to proceed in a series of stages. Each stage captures some new aspect of the problem space that is then reflected in subsequent stages through the program that is generated. A primary advantage of the approach is that it supports the construction of domain specific notations in a nested fashion [11]. Here, each stage allows the encapsulation of domain knowledge in a precise way, and programmers may work at different levels (corresponding to stages) according to their degree of specialisation. For example, in the domain of real-time embedded systems which we are investigating, the first stage might be a restricted notation that guaranteed bounded time and space usage, and be used by the applications programmer; the subsequent stage might be used to define these restricted notations in terms of the underlying meta-programming system, and be used by the domain expert; and the final stage would correspond to the generation of executable code, and be used by the compiler writer.

A major problem with this approach arises in ensuring that generated programs conform to the properties required by the (meta)-programmer. This problem has been explored in outline by Taha, Sheard and Pašalíc amongst others [28, 31, 30], who have produced systems that are capable of correctly preserving type information across stages. While this is a valuable contribution in reducing runtime type errors for generated programs, these approaches restrict the expressivity of their type systems. This approach is valuable in allowing the automatic verification of types, but verification of more complex properties will generally require more complex proof structures than can be supported by such frameworks. For example, the calculation of bounds on the resources used by a generated program may be essential in a real-time embedded systems setting; previous work (e.g. [34, 20]) uses multi-stage programming to generate resource correct programs, but is limited to specific resource correctness properties. We are thus motivated to consider how *arbitrary* proofs may be embedded within multi-stage programs in a homogeneous framework, in order to allow automatic verification of required program properties for domain specific languages implemented using a multi-stage approach.

1.1 Overview of our Approach

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. 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 correct implementation of a functional programming language.

We use dependent types to implement a well-typed interpreter, following and extending the ideas of Augustsson and Carlsson [4]. We are able to show *by construction* that the interpreter returns a value of the correct type and correctly evaluates well-typed terms — we take types as the prior notion representing a specification, and use the typechecker to guarantee that our program respects this specification. Dependent types ensure, by static checking, that the interpreter cannot be executed on badly formed or ill-typed code.

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.

Thus dependent types provide static guarantees of certain desired correctness properties.

We further consider the use of staging annotations [32] to control the execution order of the interpreter. Staging annotations allow code generation to be deferred until run-time, when some inputs are known. This means that we can specialise our interpreter for specific object progams — staging the interpreter yields a translator from the object language to the meta-language [14]. From here it is a small step to generating a compiler for the object language (for example using *offshoring* — passing the translated code to an external compiler [13]). The combination of dependent types and multi-stage programming therefore gives us a method for building *verified compilers*, for at least some required properties.

$$
\begin{array}{rll}
t ::= & \star_i & \text{(type universes)} \\
| & x & \text{(variable)} \\
| & (x\ :\ t) \to t & \text{(function space)} \\
| & \lambda x : t.\ t & \text{(abstraction)} \\
| & t\ t & \text{(application)} \\
| & \underline{\text{let}}\ x \mapsto t\ :\ t\ \underline{\text{in}}\ t & \text{(let binding)}
\end{array}
$$

Figure 1. The core language, TT

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

$$\frac{(x\ :\ S) \in \Gamma}{\Gamma \vdash x\ :\ S}\ \textsf{Var}$$

$$\frac{(x\ :\ S \mapsto s) \in \Gamma}{\Gamma \vdash x\ :\ S}\ \textsf{Val}$$

$$\frac{\Gamma \vdash f\ :\ (x\ :\ S) \to T \qquad \Gamma \vdash s\ :\ S}{\Gamma \vdash f\ s\ :\ T[s/x]}\ \textsf{App}$$

$$\frac{\Gamma; x\ :\ S \vdash e\ :\ T \qquad \Gamma \vdash (x\ :\ S) \to T\ :\ \star_n}{\Gamma \vdash \lambda x : S.e\ :\ (x\ :\ S) \to T}\ \textsf{Lam}$$

$$\frac{\Gamma; x\ :\ S \vdash T\ :\ \star_n \qquad \Gamma \vdash S\ :\ \star_n}{\Gamma \vdash (x\ :\ S) \to T\ :\ \star_n}\ \textsf{ForAll}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1\ :\ S \qquad \Gamma; x \mapsto e_1\ :\ S \vdash e_2\ :\ T \\ \Gamma \vdash S\ :\ \star_n \qquad \Gamma; x \mapsto e_1\ :\ S \vdash T\ :\ \star_n\end{array}}{\Gamma \vdash \underline{\text{let}}\ x\ :\ S \mapsto e_1\ \underline{\text{in}}\ e_2\ :\ T[e_1/x]}\ \textsf{Let}$$

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

Figure 2. Typing rules for TT

1.2 The Core Type Theory, TT

Our implementation language (meta-language) is a strongly normalising dependent type theory with inductive families [12], similar to Luo's UTT [22] or the Calculus of Inductive Constructions in COQ [9]. This language, which we call TT, is an enriched lambda calculus, with the usual properties of subject reduction, Church Rosser, and uniqueness of types up to conversion. The strong normalisation property (i.e. that evaluation always terminates) 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 $(x\ :\ S) \to T$ by $S \to T$ if x is not free in T.

In TT, there is a hierarchy of type universes, $\star_i$, where $\star_0$ is the type of types, and $\star_n\ :\ \star_{n+1}$. We leave universe levels implicit, since they can be inferred by the machine [17]. The key typing rules which set this type system apart from more traditional simply- or polymorphically-typed λ-calculi are the App and Conv rules: App is the rule for applying dependent functions (note x may be free in T, so s may be substituted in the type of the application); and Conv is the conversion rule — two terms are convertible by the $\simeq$ relation if they have a common redex. Checking convertibility requires evaluation at compile-time, hence strong normalisation is a requirement for decidable typechecking.

For clarity of presentation, we will use the higher level EPIGRAM notation [25], which elaborates to TT. A more detailed presentation of EPIGRAM can be found in [24]; TT and its compilation scheme are detailed in [6].

1.3 Programming with 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 will define a "lists with length" (or vector) type below. We first, however, need to declare a type of natural numbers to represent such lengths:

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

Addition and multiplication can be easily defined by primitive recursion. We can now declare vectors as follows: $\mathsf{Vect}\ A\ n$ defines an inductive family of lists indexed over A, the type of vector elements, and also over n, the vector length. Note that, by construction, ϵ only targets vectors of length zero, and $x :: xs$ only targets vectors of length greater than zero:

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

Note that A and k are implicit arguments to the infix constructor $::$ — their types can be inferred from the type of Vect. When the type includes explicit length information in this way, it follows that any type-correct function over values in that type must express the invariant properties of the length. For example, we can write a bounds-safe list lookup function that gives a static guarantee that a value is never projected from an empty list. In order to do do this, we define a datatype of finite sets, which can be used to represent numbers with an upper bound:

$$\underline{\text{data}}\quad \frac{n\ :\ \mathbb{N}}{\mathsf{Fin}\ n\ :\ \star}$$

$$\underline{\text{where}}\quad \frac{}{\mathsf{f0}\ :\ \mathsf{Fin}\ (\mathsf{s}\ n)}\qquad \frac{i\ :\ \mathsf{Fin}\ n}{\mathsf{fs}\ i\ :\ \mathsf{Fin}\ (\mathsf{s}\ n)}$$

Note that there are no elements of $\mathsf{Fin}\,0$ — this would be a finite set with zero elements. The type of **lookup** below expresses statically that the bound of the index and the size of the list are the same, so there can be no run-time error:

$$\underline{\text{let}}\quad \frac{i\ :\ \mathsf{Fin}\ n \qquad xs\ :\ \mathsf{Vect}\ A\ n}{\mathbf{lookup}\ i\ xs\ :\ A}$$

$$\begin{array}{lll}
\mathbf{lookup} & i & xs \quad \Leftarrow \underline{\text{elim}}\ i \Leftarrow \underline{\text{case}}\ xs \\
\mathbf{lookup} & \mathsf{f0} & (x :: ys) \ \mapsto\ x \\
\mathbf{lookup} & (\mathsf{fs}\ j) & (x :: ys) \ \mapsto\ \mathbf{lookup}\ j\ ys
\end{array}$$

The elim and case notation invoke the primitive recursion and case analysis operators respectively on i and xs. Termination is guaranteed since these operators are the only means to inspect data. Unlike a simply typed language, we do not need to give error handling cases: the typechecker verifies that the empty vector cannot be a legal input. We can see this by observing that neither constructor of Fin targets the type Fin 0, therefore no well-typed application of **lookup** could accept a Vect A 0.

1.4 Types for Specification

By giving additional static information about the **lookup** function, we obtain a stronger guarantee of its behaviour from the typechecker. The definition itself, however, is written in the usual way — indeed, it is more concise since there is no need for error checking. When writing the type, we are really writing a specification. Then, in writing the program, we are at the same time providing a proof that the implementation meets this specification.

While we accept that such methods may not become widespread for general purpose programming in the short term, there are many important applications domains where guarantees of correctness are vital or desirable. Our own area of interest is resource-bounded safety-critical systems [16]; we aim to use the techniques we present in this paper to implement a verified compiler for a resource aware functional language. We will return to this in Section 5.

1.5 Theorem Proving

The dependent type system of TT also allows properties to be expressed directly. For example, the following heterogeneous definition of equality, due to McBride [23], is built in to TT:

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

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. For example, refl (s 0) is an instance of a proof that s 0 $=$ s 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 program below. This can be viewed as a proof that s respects equality:

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

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

We can also represent more complex properties, including logical relations:

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

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, leS (leS leO) could represent a proof of s (s 0) $\leq$ s (s (s (s 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:

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

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

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 [10, 18] 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. While it has been claimed elsewhere [8] that this implies that types cannot be erased at run-time, in fact, we are able to maintain type erasure by instead establishing a distinction between compile-time and run-time values using techniques from [6], and erasing those needed at compile-time only.

1.6 Implementation

An implementation of the staged interpreter we develop in this paper, along with an implementation of TT, is available from `http://www.dcs.st-and.ac.uk/~eb/STLC/`.

2. A Dependently Typed Interpreter

Dependent types can be used to good effect in the implementation of programming languages. One demonstration of this is Augustsson and Carlsson's well-typed interpreter in Cayenne [4, 3]. This interpreter implements the language given below:

$e ::=$	$\lambda a : T.\ e$	λ-abstraction
	$e\ e$	application
	a	variable
	$e\ op\ e$	binary operator
	n	number
	b	boolean value
	if e then e else e	boolean choice
	primrec $e\ e\ e$	primitive recursion
$op ::=$	$+ \mid * \mid < \mid = \mid > \mid$ and $\mid$ or	
$T ::=$	$\mathbb{N}$	Natural numbers
	Bool	Booleans
	$t \to t$	Function type

This is a simply typed λ-calculus with booleans and natural numbers. We call this language λ_{AC}; our implementation augments Augustsson and Carlsson's with a primitive recursion operator for natural numbers, primrec and if ... then ... else expressions.

The typing rules for this language are given in Figure 3, with context validity defined as follows:

$$\frac{}{\vdash \underline{\mathsf{valid}}} \qquad \frac{\Gamma \vdash \underline{\mathsf{valid}}}{\Gamma, a : T \vdash \underline{\mathsf{valid}}}$$

$$\frac{}{a : T \in \Gamma, a : T} \qquad \frac{a : T \in \Gamma}{a : T \in \Gamma, b : S}$$

The interpreter we present here uses inductive families to represent well-typedness and synchronisation of type and value environments. By using inductive families, we can express explicit relationships between data structures, just like the relationship between

$$\frac{}{\Gamma \vdash n \;:\; \mathbb{N}} \qquad \frac{}{\Gamma \vdash b \;:\; \mathsf{Bool}}$$

$$\frac{}{a \;:\; t \in \Gamma \vdash a \;:\; t} \qquad \frac{\Gamma, a \;:\; s \vdash e \;:\; t}{\Gamma \vdash \lambda a \;:\; s.\; e \;:\; s \to t}$$

$$\frac{\Gamma \vdash e_1 \;:\; s \to t \qquad \Gamma \vdash e_2 \;:\; s}{\Gamma \vdash e_1\; e_2 \;:\; t}$$

$$\frac{\Gamma \vdash e_1 \;:\; \mathbb{N} \qquad \Gamma \vdash e_2 \;:\; \mathbb{N} \qquad op \in \{+ \mid *\}}{\Gamma \vdash e_1\; op\; e_2 \;:\; \mathbb{N}}$$

$$\frac{\Gamma \vdash e_1 \;:\; \mathbb{N} \qquad \Gamma \vdash e_2 \;:\; \mathbb{N} \qquad op \in \{< \mid = \mid >\}}{\Gamma \vdash e_1\; op\; e_2 \;:\; \mathsf{Bool}}$$

$$\frac{\Gamma \vdash e_1 \;:\; \mathsf{Bool} \qquad \Gamma \vdash e_2 \;:\; \mathsf{Bool} \qquad op \in \{\mathsf{and} \mid \mathsf{or}\}}{\Gamma \vdash e_1\; op\; e_2 \;:\; \mathsf{Bool}}$$

$$\frac{\Gamma \vdash x \;:\; \mathsf{Bool} \qquad t \;:\; a \qquad f \;:\; a}{\Gamma \vdash \mathsf{if}\; x\; \mathsf{then}\; t\; \mathsf{else}\; f \;:\; a}$$

$$\frac{\Gamma \vdash x \;:\; \mathbb{N} \qquad z \;:\; a \qquad s \;:\; \mathbb{N} \to a \to a}{\Gamma \vdash \mathsf{primrec}\; x\; z\; s \;:\; a}$$

Figure 3. Typing rules for λ_{AC}

a Vect and its length. In particular, the types we use express the relationship between values and their type and context membership. This means that, without having to prove *any* theorems, we have static guarantees that evaluation preserves type, and that context lookup will always succeed. Later, we will also see that we can represent raw (untyped) terms and implement typechecking in such a way as to guarantee that any well-typed term it produces is of the of the correct type. i.e. we can show soundness of the typechecking algorithm, by the types alone.

2.1 Representation

We have specified context validity, context membership and the typing rules in relation to contexts. We would now like to choose a representation for λ_{AC} terms in the meta-language which reflects this specification as directly as possible and ensures that only well-typed expressions can be built. In a traditional language some loss of information is inevitable here, but with inductive families we can specify *in the type* the relationships between these concepts.

We begin with a representation of types. These types can be reified into meta-language types with the $\mathbf{interpTy}$ function, since the dependent type system allows us to compute types:

$$\underline{\text{data}} \quad \frac{}{\mathsf{Ty} \;:\; \star} \qquad \underline{\text{where}} \quad \frac{}{\mathsf{TyNat} \;:\; \mathsf{Ty}} \qquad \frac{}{\mathsf{TyBool} \;:\; \mathsf{Ty}} \qquad \frac{s, t \;:\; \mathsf{Ty}}{\mathsf{Arrow}\; s\; t \;:\; \mathsf{Ty}}$$

$$\underline{\text{let}} \quad \frac{t \;:\; \mathsf{Ty}}{\mathbf{interpTy}\; t \;:\; \star}$$

$$\begin{array}{lll} \mathbf{interpTy} & t & \Leftarrow \underline{\text{elim}}\; t \\ \mathbf{interpTy} & \mathsf{TyNat} & \mapsto \mathbb{N} \\ \mathbf{interpTy} & \mathsf{TyBool} & \mapsto \mathsf{Bool} \\ \mathbf{interpTy}\; (\mathsf{Arrow}\; s\; t) & & \mapsto \mathbf{interpTy}\; s \to \mathbf{interpTy}\; t \end{array}$$

We represent type environments (Γ) as vectors of types, and membership of a type environment as a relation:

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

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

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

Variables are de Bruijn indexed, and represented by an element of a finite set. Using finite sets gives an explicit bound on the index which means that the variable can never refer to a value out of the scope given by the type environment. We can also view top and pop as zero and successor constructors of a natural number representing a de Bruijn index.

If we read $\mathsf{Var}\; G\; i\; t$ as $G \vdash i \;:\; t$, its intended meaning, then we see a direct correspondence with the earlier definition of context membership.

The Expr family, which represents expressions in the object language, is shown below. 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. There is no need to supply a well-typing predicate along with an expression as in previous work [4, 30], since there is no way to construct ill-typed values:

$$\underline{\text{data}} \quad \frac{G \;:\; \mathsf{Env}\; n \qquad A \;:\; \mathsf{Ty}}{\mathsf{Expr}\; G\; A \;:\; \star}$$

$$\underline{\text{where}} \quad \frac{k \;:\; \mathbb{N}}{\mathsf{enat}\; k \;:\; \mathsf{Expr}\; G\; \mathsf{TyNat}}$$

$$\frac{b \;:\; \mathsf{Bool}}{\mathsf{ebool}\; b \;:\; \mathsf{Expr}\; G\; \mathsf{TyBool}}$$

$$\frac{v \;:\; \mathsf{Var}\; G\; i\; t}{\mathsf{evar}\; v \;:\; \mathsf{Expr}\; G\; t}$$

$$\frac{f \;:\; \mathsf{Expr}\; G\; (\mathsf{Arrow}\; s\; t) \qquad a \;:\; \mathsf{Expr}\; G\; s}{\mathsf{eapp}\; f\; a \;:\; \mathsf{Expr}\; G\; t}$$

$$\frac{e \;:\; \mathsf{Expr}\; (s :: G)\; t}{\mathsf{elam}\; e \;:\; \mathsf{Expr}\; G\; (\mathsf{Arrow}\; s\; t)}$$

$$\frac{op \;:\; \mathbf{interpTy}\; A \to \mathbf{interpTy}\; B \to \mathbf{interpTy}\; C \qquad a \;:\; \mathsf{Expr}\; G\; A \qquad b \;:\; \mathsf{Expr}\; G\; B}{\mathsf{eop}\; op\; a\; b \;:\; \mathsf{Expr}\; G\; C}$$

$$\frac{x \;:\; \mathsf{Expr}\; G\; \mathsf{Bool} \qquad t, f \;:\; \mathsf{Expr}\; G\; A}{\mathsf{eif}\; x\; t\; f \;:\; \mathsf{Expr}\; G\; A}$$

$$\frac{\begin{array}{l} x \;:\; \mathsf{Expr}\; G\; \mathbb{N} \qquad z \;:\; \mathsf{Expr}\; G\; A \\ s \;:\; \mathsf{Expr}\; G\; (\mathbb{N} \to A \to A) \end{array}}{\mathsf{eprimrec}\; x\; z\; s \;:\; \mathsf{Expr}\; G\; A}$$

Again, if we read the declaration $x \;:\; \mathsf{Expr}\; G\; T$ with its intended meaning, $G \vdash x \;:\; T$, then we can read the typing rules directly from the Expr type declaration. The only minor exception is the eop constructor, which permits a more generic implementation of binary operators. Thus any correctly constructed (i.e. well-typed) instance of an Expr must be a representation of a well-typed term in λ_{AC}.

As an example, the factorial function could be defined by primitive recursion as follows:

$$\mathbf{fact} \;=\; \lambda x : \mathbb{N}.\; \mathsf{primrec}\; x\; 1\; (\lambda k : \mathbb{N}.\; \lambda ih : \mathbb{N}.\; (k + 1) \;\times\; ih)$$

The representation of this as an Expr is:

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

$$\begin{array}{llll}
\mathbf{interp} & x & ve & \Leftarrow \underline{\mathsf{elim}}\ x \\
\mathbf{interp} & (\mathsf{enat}\ k) & ve & \mapsto k \\
\mathbf{interp} & (\mathsf{ebool}\ b) & ve & \mapsto b \\
\mathbf{interp} & (\mathsf{evar}\ v) & ve & \mapsto \mathbf{envLookup}\ v\ ve \\
\mathbf{interp} & (\mathsf{eapp}\ f\ a) & ve & \mapsto (\mathbf{interp}\ f\ ve)\ (\mathbf{interp}\ a\ ve) \\
\mathbf{interp} & (\mathsf{elam}_s\ e) & ve & \mapsto \lambda a : \mathbf{interpTy}\ s.\ \mathbf{interp}\ e\ (\mathsf{extend}\ a\ ve) \\
\mathbf{interp} & (\mathsf{eop}\ op\ a\ b) & ve & \mapsto op\ (\mathbf{interp}\ a\ ve)\ (\mathbf{interp}\ b\ ve) \\
\mathbf{interp} & (\mathsf{eif}\ x\ t\ f) & ve & \mapsto \mathbf{runif}\ (\mathbf{interp}\ x\ ve)\ (\mathbf{interp}\ t\ ve)\ (\mathbf{interp}\ f\ ve) \\
\mathbf{interp} & (\mathsf{eprimrec}\ x\ z\ s) & ve & \mapsto \mathbf{primrec}\ (\mathbf{interp}\ x\ ve)\ (\mathbf{interp}\ z\ ve)\ (\mathbf{interp}\ s\ ve)
\end{array}$$

$$\underline{\text{let}}\quad \frac{x\ :\ \mathsf{Bool} \qquad t, f\ :\ A}{\mathbf{runif}\ x\ t\ f\ :\ A}$$

$$\begin{array}{llll}
\mathbf{runif} & x & t\ f & \Leftarrow \underline{\mathsf{case}}\ x \\
\mathbf{runif}\ \mathsf{true} & t & f & \mapsto t \\
\mathbf{runif}\ \mathsf{false} & t & f & \mapsto f
\end{array}$$

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

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

Figure 4. The interpreter

$$\begin{array}{l}
\mathbf{fact} = \mathsf{elam}(\mathsf{eprimrec}\ \mathsf{top}\ (\mathsf{enat}\ (\mathsf{s}\ 0)) \\
\qquad (\mathsf{elam}\ (\mathsf{elam} \\
\qquad\quad (\mathsf{eop}\ \mathbf{mult}\ (\mathsf{eop}\ \mathbf{plus} \\
\qquad\qquad (\mathsf{evar}\ (\mathsf{pop}\ \mathsf{top}))\ (\mathsf{enat}\ (\mathsf{s}\ 0)))\ (\mathsf{evar}\ \mathsf{top})))))
\end{array}$$

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 define a type which synchronises the value environment 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 and a lookup function are as follows:

$$\underline{\text{data}}\quad \frac{G\ :\ \mathsf{Env}\ n}{\mathsf{ValEnv}\ G\ :\ \star}\qquad \underline{\text{where}}\quad \frac{}{\mathsf{empty}\ :\ \mathsf{ValEnv}\ \epsilon}$$

$$\frac{t\ :\ \mathbf{interpTy}\ T \qquad r\ :\ \mathsf{ValEnv}\ G}{\mathsf{extend}\ t\ r\ :\ \mathsf{ValEnv}\ (T :: G)}$$

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

$$\begin{array}{llll}
\mathbf{envLookup} & v & ve & \Leftarrow \underline{\mathsf{elim}}\ v \\
\mathbf{envLookup} & \mathsf{top} & (\mathsf{extend}\ t\ r) & \mapsto t \\
\mathbf{envLookup}\ (\mathsf{pop}\ v) & & (\mathsf{extend}\ t\ r) & \mapsto \mathbf{envLookup}\ v\ r
\end{array}$$

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 possible inputs are well-typed and that the output will be of the correct type.

2.2 The Interpreter

The implementation of the interpreter is shown in Figure 4. **interp** is written by structural recursion over the input expression x. It returns a semantic representation, as a TT term, of the input. 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 [5] in that we construct a semantic representation of the term to be interpreted. It is also a *tagless* interpreter; i.e., there is no need to tag the return value with its type, since the the type is known from the index of the input.

Evaluating the **fact** function defined above gives, as expected, the meta-level implementation of factorial by primitive recursion[1]:

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

3. Dependent Types and Staging

In [33], Taha states that "a staged interpreter is a translator". His 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. 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. A major advantage over Taha's approach is that dependent types allow us to maintain the compile-time correctness guarantees of Section 2 automatically. It follows that, by construction, our object programs are well-typed, well-scoped and terminating.

3.1 Extensions to TT

To add staging constructs to TT, we begin by extending 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. $\Gamma \vdash_n x\ :\ A$ states that x has type A at level n. We extend TT with the following expression forms:

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

- $'e$ quotes an expression e, deferring its execution until the next stage. This lifts the expression from level n to level $n + 1$

[1] In fact **primrec**, **mult** and **plus** are also unfolded, but we omit these details for clarity.

- $\langle e \rangle$ is the "code" type of a quoted term.
- $!e$ evaluates a quoted expression e. Since this executes code, it is only valid at level 0, where e has no free variables.
- $\sim e$ 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 5. These typing rules additionally ensure staging correctness — in particular, a value bound in a later stage cannot be used in an earlier stage. To guarantee this, context entries are annotated with the level where they are bound and checked at the point of use, with updated Var and Val rules, and code can only be executed if it has no free variables. This is a more restrictive solution than strictly necessary — we could for example use environment classifiers [35] to ensure that all free variables have an originating environment — but is sufficient for many programs including those we present in this paper.

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

$$\frac{\vdash_0 e \ : \ \langle A \rangle}{\Gamma \vdash_0 !e \ : \ A} \ \textsf{Eval} \qquad \frac{\Gamma \vdash_n e \ : \ \langle A \rangle}{\Gamma \vdash_{n+1} \sim e \ : \ A} \ \textsf{Escape}$$

$$\frac{x :_n S \in \Gamma \qquad n \leq m}{\Gamma \vdash_m x \ : \ S} \ \textsf{Var}$$

$$\frac{x \mapsto s :_n S \in \Gamma \qquad n \leq m}{\Gamma \vdash_m x \ : \ S} \ \textsf{Val}$$

Figure 5. Typing rules for staging constructs

The two basic notions of equivalence are:

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

The distinction between the escape and evaluation constructs is that escape must be enclosed by quotation brackets and evaluation applies only to closed terms — the purpose of escape is to allow reduction inside a code building context. Extending an implementation of TT with these constructs is fairly straightforward; the one aspect needing care is conversion of quoted terms:

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

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:

$$\begin{array}{l} \underline{\text{let}} \quad \dfrac{n, m \ : \ \mathbb{N}}{\mathbf{power}\ n\ m \ : \ \mathbb{N}} \\ \quad \mathbf{power} \quad n \quad m \ \Leftarrow \underline{\text{elim}}\ n \\ \quad \mathbf{power} \quad 0 \quad m \ \mapsto \ \mathsf{s}\,0 \\ \quad \mathbf{power}\ (\mathsf{s}\ k)\ m \ \mapsto \ m \ \times \ (\mathbf{power}\ k\ m) \end{array}$$

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.:

$$\begin{array}{l} \mathbf{power2} \ \mapsto \ \mathbf{power}\ (\mathsf{s}\ (\mathsf{s}\,0)) \\ \mathbf{power4} \ \mapsto \ \mathbf{power}\ (\mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\,0)))) \end{array}$$

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 without staging annotations, 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:

$$\begin{array}{l} \underline{\text{let}} \quad \dfrac{n \ : \ \mathbb{N} \qquad m \ : \ \langle \mathbb{N} \rangle}{\mathbf{power}'\ n\ m \ : \ \langle \mathbb{N} \rangle} \\ \quad \mathbf{power}' \quad n \quad m \ \Leftarrow \underline{\text{elim}}\ n \\ \quad \mathbf{power}' \quad 0 \quad m \ \mapsto \ \text{'}(\mathsf{s}\,0) \\ \quad \mathbf{power}'\ (\mathsf{s}\ k)\ m \ \mapsto \ \text{'}(\sim m \ \times \ \sim(\mathbf{power}'\ k\ m)) \end{array}$$

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

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

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

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

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

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. This allows us to remove the interpretation overhead and thus generate a *translator* 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. We will now consider how to apply this technique to the interpreter of Section 2.2. Figure 6 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

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

$$\begin{array}{llll}
\mathbf{interp} & x & ve & \Leftarrow \underline{\text{elim}}\ x \\
\mathbf{interp} & (\mathsf{enat}\ k) & ve & \mapsto\ 'k \\
\mathbf{interp} & (\mathsf{ebool}\ b) & ve & \mapsto\ 'b \\
\mathbf{interp} & (\mathsf{evar}\ v) & ve & \mapsto\ \mathbf{envLookup}\ v\ ve \\
\mathbf{interp} & (\mathsf{eapp}\ f\ a) & ve & \mapsto\ '(\sim(\mathbf{interp}\ f\ ve)\ \sim(\mathbf{interp}\ a\ ve)) \\
\mathbf{interp} & (\mathsf{elam}_s\ e) & ve & \mapsto\ '(\lambda a : \mathbf{interpTy}\ s.\ \sim(\mathbf{interp}\ e\ (\mathsf{extend}\ a\ ve))) \\
\mathbf{interp} & (\mathsf{eop}\ op\ a\ b) & ve & \mapsto\ '(op\ \sim(\mathbf{interp}\ a\ ve)\ \sim(\mathbf{interp}\ b\ ve)) \\
\mathbf{interp} & (\mathsf{eif}\ x\ t\ f) & ve & \mapsto\ '(\mathbf{runif}\ \sim(\mathbf{interp}\ x\ ve)\ \sim(\mathbf{interp}\ t\ ve)\ \sim(\mathbf{interp}\ f\ ve)) \\
\mathbf{interp} & (\mathsf{eprimrec}\ x\ z\ s) & ve & \mapsto\ '(\mathbf{primrec}\ \sim(\mathbf{interp}\ x\ ve)\ \sim(\mathbf{interp}\ z\ ve)\ \sim(\mathbf{interp}\ s\ ve))
\end{array}$$

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

$$\begin{array}{lll}
\mathbf{runif} & n\ \ t\ f & \Leftarrow \underline{\text{case}}\ n \\
\mathbf{runif}\ \mathsf{true} & t\ f & \mapsto\ t \\
\mathbf{runif}\ \mathsf{false} & t\ f & \mapsto\ f
\end{array}
\qquad
\begin{array}{lll}
\mathbf{primrec} & n\ \ z\ s & \Leftarrow \underline{\text{elim}}\ n \\
\mathbf{primrec} & 0\ \ z\ s & \mapsto\ z \\
\mathbf{primrec}\ (\mathsf{s}\ k) & z\ s & \mapsto\ s\ k\ (\mathbf{primrec}\ k\ z\ s)
\end{array}$$

Figure 6. The staged interpreter

only once. We stage **envLookup** as follows, storing *code* in the value environment:

$$\underline{\text{data}}\quad \frac{G\ :\ \mathsf{Env}\ n}{\mathsf{ValEnv}\ G\ :\ \star}\quad \underline{\text{where}}\quad \frac{}{\mathsf{empty}\ :\ \mathsf{ValEnv}\ \epsilon}$$

$$\frac{t\ :\ \langle \mathbf{interpTy}\ T\rangle \qquad r\ :\ \mathsf{ValEnv}\ G}{\mathsf{extend}\ t\ r\ :\ \mathsf{ValEnv}\ (T :: G)}$$

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

$$\begin{array}{lllll}
\mathbf{envLookup} & v & ve & \Leftarrow \underline{\text{elim}}\ v \\
\mathbf{envLookup} & \mathsf{top} & (\mathsf{extend}\ t\ r) & \mapsto\ t \\
\mathbf{envLookup}\ (\mathsf{pop}\ v) & & (\mathsf{extend}\ t\ r) & \mapsto\ \mathbf{envLookup}\ v\ r
\end{array}$$

Evaluation of the **fact** function gives a quoted representation of the meta-level implementation of factorial, with **primrec**, **mult** and **plus** *not* inlined:

$$\mathbf{fact} = '(\lambda x : \mathbb{N}.\mathbf{primrec} x (\mathsf{s0}) (\lambda k, ih : \mathbb{N}.\mathbf{mult} (\mathbf{plus} k (\mathsf{s0})) ih))$$

We have now deferred the partial evaluation until run-time — rather than incurring interpretation overhead each time we wish to run this object language function, there is now a single compilation overhead, with code generated for the quoted function at run-time.

4. A Verified Compiler

Our choice of data type for the object language means that the implementation of the interpreter is simultaneously a proof of the desired properties. By using full-spectrum dependent types, 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.

In this section, we will give some desired properties of our staged interpreter, and give explicit proofs of these properties. The principal advantage of our method for compiler construction is that these proofs are extremely simple — the use of inductive families and our choice of invariants on these families means that the properties in which we are interested are already checked by the (meta-language) typechecker.

$$\underline{\text{data}}\quad \frac{n\ :\ \mathbb{N}}{\mathsf{Raw}\ n\ :\ \star}$$

$$\underline{\text{where}}\quad \frac{k\ :\ \mathbb{N}}{\mathsf{rnat}\ k\ :\ \mathsf{Raw}\ n} \qquad \frac{b\ :\ \mathsf{Bool}}{\mathsf{rbool}\ b\ :\ \mathsf{Raw}\ n}$$

$$\frac{i\ :\ \mathsf{Fin}\ n}{\mathsf{rvar}\ i\ :\ \mathsf{Raw}\ n} \qquad \frac{f, a\ :\ \mathsf{Raw}\ n}{\mathsf{rapp}\ f\ a\ :\ \mathsf{Raw}\ n}$$

$$\frac{e\ :\ \mathsf{Raw}\ (\mathsf{s}\ n)}{\mathsf{rlam}\ e\ :\ \mathsf{Raw}\ n}$$

$$\frac{op\ :\ \mathbf{interpTy}\ A \to \mathbf{interpTy}\ B \to \mathbf{interpTy}\ C \qquad a, b\ :\ \mathsf{Raw}\ n}{\mathsf{rop}\ op\ a\ b\ :\ \mathsf{Raw}\ n}$$

$$\frac{x, t, f\ :\ \mathsf{Raw}\ n}{\mathsf{rif}\ x\ t\ f\ :\ \mathsf{Raw}\ n} \qquad \frac{x, z, s\ :\ \mathsf{Raw}\ n}{\mathsf{rprimrec}\ x\ z\ s\ :\ \mathsf{Raw}\ n}$$

$$\underline{\text{data}}\quad \frac{G\ :\ \mathsf{Env}\ n \qquad r\ :\ \mathsf{Raw}\ n \qquad T\ :\ \mathsf{Ty}}{\mathsf{Checked}\ G\ r\ T\ :\ \star}$$

$$\underline{\text{where}}\quad \frac{e\ :\ \mathsf{Expr}\ G\ T}{\mathsf{ok}\ e\ :\ \mathsf{Checked}\ G\ r\ T} \qquad \frac{}{\mathsf{error}\ :\ \mathsf{Checked}\ G\ r\ T}$$

$$\underline{\text{let}}\quad \frac{G\ :\ \mathsf{Env}\ n \qquad r\ :\ \mathsf{Raw}\ n \qquad T\ :\ \mathsf{Ty}}{\mathbf{check}\ G\ r\ T\ :\ \mathsf{Checked}\ G\ r\ T}$$

Figure 7. Checking Raw Terms

Let us first complete our prototype implementation with a datatype for raw terms (we assume the existence of a parser which generates these well-scoped terms) and a typechecker. The **check** function (declared in Figure 7) takes a raw term and its intended type and returns a Checked structure, which either contains a well-typed term, or an error. We omit the definition of **check**; its structure follows that of the typechecker example in [25].

To begin, let us define a correspondence between the definition of λ_{AC} and our representation, Raw, Ty and Expr.

Definition 1 (Representation). *If* $\Gamma \vdash t : T$ *then:*

- $[\![\Gamma]\!]$ *is the representation of the context (with* n *entries) as an* $\mathsf{Env}\ n$
- $[\![T]\!]$ *is the representation of the type as a* Ty.
- $[\![t]\!]$ *is the representation of the term as an* $\mathsf{Expr}\ [\![\Gamma]\!]\ [\![T]\!]$.
- $\textsc{raw}(t)$ *is the representation of the term as raw syntax,* $\mathsf{Raw}\ n$.

Since dependent types encode many correctness properties directly within the program, the substantial part of any compiler correctness proof is in checking that the implementation faithfully models the semantics. We can show that our representation is a complete and faithful representation of λ_{AC} by its correspondence with the typing rules.

Lemma 2 (Soundness of representation). *If* $e : \mathsf{Expr}\ [\![\Gamma]\!]\ [\![T]\!]$, *then there is a term* t *such that* $[\![t]\!] = e$ *and* $\Gamma \vdash t : T$.

Proof. Each constructor of Expr directly corresponds to a typing rule (Figure 3), by reading $x : \mathsf{Expr}\ G\ T$ as $G \vdash x : T$. □

Lemma 3 (Completeness of representation). *If* $\Gamma \vdash t : T$, *then there is a term* $e : \mathsf{Expr}\ [\![\Gamma]\!]\ [\![T]\!]$, *such that* $[\![t]\!] = e$.

Proof. Each typing rule (Figure 3), directly corresponds to a constructor of Expr by reading $x : \mathsf{Expr}\ G\ T$ as $G \vdash x : T$. □

For the correctness of our implementation, we verify that the typechecker is a sound and complete implementation of the typing rules, and that evaluation preserves type.

Theorem 4 (Soundness of typechecking function). *If* $\mathbf{check}\ [\![\Gamma]\!]\ \textsc{raw}(t)\ [\![T]\!] \rightsquigarrow \mathsf{ok}\ e$ *then* $\Gamma \vdash t : T$, *where* $e : \mathsf{Expr}\ [\![\Gamma]\!]\ [\![T]\!]$.

Proof. By the type of the $\mathsf{Checked}$ view, and the $\mathbf{check}$ function, no e can be constructed which does not have the type $\mathsf{Expr}\ [\![\Gamma]\!]\ [\![T]\!]$. Then $\Gamma \vdash t : T$ by Lemma 2. □

Although we can show soundness using types alone, completeness requires some extra work, involving the details of the $\mathbf{check}$ function.

Theorem 5 (Completeness of typechecking function). *If* $\Gamma \vdash t : T$ *then* $\mathbf{check}\ [\![\Gamma]\!]\ \textsc{raw}(t)\ [\![T]\!] \rightsquigarrow \mathsf{ok}\ e$, *where* $e : \mathsf{Expr}\ [\![\Gamma]\!]\ [\![T]\!]$.

Proof Sketch. By Lemma 3, there exists a term $e : \mathsf{Expr}\ [\![\Gamma]\!]\ [\![T]\!]$. The totality of the $\mathbf{check}$ function ensures that it will produce either a term $\mathsf{ok}\ e$, where e represents a term of the correct type (by Theorem 4), or error.

To show that $\mathbf{check}$ produces a well-typed term where it exists, we define a forgetful map operation $|e|$, where $e : \mathsf{Expr}\ G\ T$, and show by induction over $|e|$ that $\mathbf{check}\ G\ |e|\ T \rightsquigarrow \mathsf{ok}\ e$.

□

Theorem 6 (Subject Reduction). *If* $\mathbf{interp}\ s\ ve \rightsquigarrow t$ *and* $s : \mathsf{Expr}\ G\ T$ *then* $t : T$.

Proof. The return type of $\mathbf{interp}$ ensures that the meta-level value returned has the type required by its representation. By typechecking in TT, $s : \mathsf{Expr}\ G\ T$ ensures that $t : T$. Furthermore, since $\mathbf{interp}$ is total, evaluation will always return a value. □

In addition, because the implementation language is strongly normalising (i.e., evaluation always terminates with a constructor headed value), we know that variable lookup will always succeed, and the interpreter will always terminate correctly. Therefore, for any object program, the interpreter will always produce a meta-level representation of that program. We have verified these properties for the unstaged implementation — since the staging annotations guarantee that types are preserved between stages, and these properties are shown by the program's type, we can be sure that these correctness properties are preserved after staging the interpreter. By verifying the properties for the interpreter, and adding staging annotations, we have verified the properties for the resulting compiler.

4.1 Termination and Side Effects

Since we have used a strongly normalising language to implement a staged interpreter for λ_{AC}, we can be sure of strong properties such as termination of λ_{AC}. For many domain specific languages, guaranteed termination is important. For example Hume [16] is separated into two layers, a co-ordination layer which directs the flow of data, and a computation layer which processes data. If we desire strong static guarantees about programs in the computation layer, we may also require computations to be total.

From our perspective, the primary benefits of a strongly terminating language, in which $\perp$ is not a value, are that proofs are total and typechecking is decidable. In a full-spectrum dependently typed language, where values can appear in types and vice versa, the termination property is essential for decidability of typechecking. Furthermore, without it, it is possible to construct a proof of any proposition by creating a non-terminating function $\mathbf{bottom} = \mathbf{bottom}$, which can have any type we like.

As far as proofs and typechecking are concerned, totality is vital. However, the resulting limitation is that our programs must also not possess side-effects, such as I/O, since these effects could appear in types. This precludes the implementation of a Turing complete language in TT as it stands. Meta-D [30] avoids this problem by restricting the type language. We prefer not to make the same restrictions since a key part of our method is that proofs are easily expressible within the source language. Our proposed solution to this problem is as for I/O in Haskell, treating partiality as a monad. In collaboration with Capretta, Altenkirch [1] and Uustalu [36] are developing such a method for introducing partiality into a dependently typed language without losing decidable typechecking, and this work transfers neatly to our setting.

5. Dependent Types for Resource Analysis

We have previously considered the use of dependent types to provide static guarantees of resource properties for functional programs [7]. The idea is to predicate each user-defined type on a natural number, representing the size of values in that type; each function then returns both a value and its size plus a proof that the size satisfies a given predicate. This is represented by the following Size type:

$$\underline{\text{data}} \quad \frac{A : \mathbb{N} \to \star \qquad P : (n : \mathbb{N}) \to A\ n \to \star}{\mathsf{Size}\ A\ P\ :\ \star}$$

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

For example, we can implement a size-safe list append function by predicating a List type on a natural number, and implementing $\mathbf{append}$ as follows:

$$\underline{\text{data}} \quad \frac{A \;:\; \mathbb{N} \to \star}{\mathsf{List_S}\ A \;:\; \mathbb{N} \to \star}$$

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

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

$$\begin{array}{l}\mathbf{append}\quad \mathsf{nil_S}\quad ys \;\mapsto\; \mathsf{size}\ ys\ (\mathsf{refl}\ ysn)\\ \mathbf{append}\ (\mathsf{cons_S}\ x\ xs)\ ys\\ \qquad\mapsto\; \underline{\text{let}}\ (\mathsf{size}\ val\ p) \;=\; \mathbf{append}\ xs\ ys\ \underline{\text{in}}\\ \qquad\qquad \mathsf{size}\ (\mathsf{cons_S}\ x\ val)\ (\mathbf{resp\_s}\ p)\end{array}$$

We have not yet considered in detail how to execute these programs. It is important when constructing a resource analysis that the cost proofs do not interfere with the actual program execution costs, otherwise the costs will no longer be valid! While the techniques of [6] will mitigate this problem, they will not completely remove this dynamic cost information. A further (potentially error-prone) pass is required in the implementation in order to translate Size structures into simple values. A staged interpreter for a resource-aware functional language would, however, allow us to remove the cost information in a principled way.

The verified interpreter presented above can now be extended to include size properties as well as type properties. To do this, we extend the type environment to include *size* variables as well as λ-bound variables, as shown in Figure 8. Since type environments now contain size information, we can embed size properties in types and proofs of those properties within programs, as required. We are continuing to develop this idea for a full size aware language, using staging to generate TT code for execution via Hume [16].

$$\underline{\text{data}} \quad \frac{v_n, v_s \;:\; \mathbb{N}}{\mathsf{TyEnv}\ v_n\ v_s \;:\; \star}$$

$$\underline{\text{where}} \quad \frac{}{\mathsf{empty} \;:\; \mathsf{TyEnv}\ 0\ 0}$$

$$\frac{G \;:\; \mathsf{TyEnv}\ v_n\ v_s}{\mathsf{sizeExtend}\ G \;:\; \mathsf{TyEnv}\ v_n\ (\mathsf{s}\ v_s)}$$

$$\frac{t \;:\; \mathsf{Ty}\ v_s \qquad G \;:\; \mathsf{TyEnv}\ v_n\ v_s}{\mathsf{typeExtend}\ t\ G \;:\; \mathsf{TyEnv}\ (\mathsf{s}\ v_n)\ v_s}$$

Figure 8. Type environments with size variables

6. Related Work

Our work brings together the related areas of dependently typed programming [2] and multi-stage programming [32], building on the idea of a tagless staged interpreter [30, 29]. One important difference in our work is the use of *full-spectrum dependent types* [26]; 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 originally developed in Brady's PhD thesis [6].

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 $(P \;:\; \star) \to (P \to P) \to P$) 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 [2]. By adding staging annotations to a logically sound type theory we aim to implement compilers with verifiable static guarantees.

Our approach to verification is to annotate data structures with their invariants. Another possibility with a dependent type system is to pair a program with a proof of its specification. This is the approach taken by Leroy [21], who is developing and certifying a compiler in COQ. Hutton and Wright [19] also discuss compiler correctness, but providing an external proof of the required properties. McKinna and Wright [27] use EPIGRAM to give an explicit proof of compiler correctness in the implementation language. These methods concentrate on the back-end; our approach, assuming a correct compiler for the meta-language, allows us to concentrate on the semantics of the object language.

We have not yet addressed the implementation techniques required of our multi-stage language. Our current prototype implementation is based on normalisation by evaluation [5]. The MetaOCaml compiler implements the ! (eval) construct by linking the compiler in with the executable and representing code as an abstract syntax tree[2]. A more practical implementation technique may be to exploit the relationship with Grégoire and Leroy's compiled strong reduction [15] — one way of understanding staging annotations is that they direct when to reduce under binders. Grégoire and Leroy's technique, in which they extend run-time values with a representation of free variables, may provide an efficient method for constructing code at run-time.

7. Conclusions and Further Work

This paper has investigated the construction of verified compilers for domain specific languages using a combination of dependent types and multi-stage programming. While the use of dependent types to expose software properties means that some initial work must be done in order to properly express these properties in the types, there is a significant benefit in improving the ease of subsequent verification, and in providing support for automation. Moreover, once constructed, a single dependently typed framework may be reused in either a general or domain specific manner.

We have taken great care to annotate the representation of the interpreter data structures in such a way as to guarantee that an implementation of the interpreter is both total and preserves the type. The payoffs are that the implementation is straightforward, no error checking is required, and that the typechecker guarantees the properties we specify. The rationale behind working so hard to choose the right representation is that data is static and code is dynamic — if we choose to annotate the static construct with more information, we do less work dynamically. This greatly simplifies program verification — the only work we have to do is to show that the data structure we use accurately models the requirements.

This paper has considered only fairly simple properties. In a real domain specific language, such as our Hume target [16], we will need to express both more complex invariants and more complex properties than the simple size metrics used here. in such a setting, full spectrum dependent types will be invaluable in allowing us to express the relationship between the language and its properties precisely. Such properties might include: guaranteed termination; time, space and power usage for some implementation; or the correctness of specific program transformations. Adding staging

[2] Walid Taha, pers. comm.

annotations to such a domain specific interpreter will give a meta-language implementation, without explicit proofs, but preserving the guarantees in the interpreter's type.

While partial evaluation of a strongly normalising program can yield a semantic representation of an object program in the meta language without any need for adding staging annotations, we do get an important further benefit from adding these annotations — namely, that we obtain a *machine code* representation of the object program, in addition 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.

Acknowledgements

This work is generously supported by EPSRC grant EP/C001346/1 and by EU Framework VI IST-510255 (EmBounded). We would like to thank Walid Taha and James McKinna, and the anonymous referees for their helpful comments.

References

[1] T. Altenkirch. Stop thinking about bottoms when writing programs, 2006. Talk at BCTCS 2006.

[2] T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter. `http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf`, 2005. Draft.

[3] L. Augustsson. Cayenne - a language with dependent types. In *Proc. 1998 International Conf. on Functional Programming (ICFP '98)*, pages 239–250, 1998.

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

[5] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In R. Vemuri, editor, *Proc. 1991 IEEE Symp. on Logic in Comp. Sci.*, pages 203–211, 1991.

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

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

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

[9] Coq Development Team. The Coq proof assistant — reference manual. `http://coq.inria.fr/`, 2001.

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

[11] K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In *Domain Specific Program Genearation 2004*, volume 3016 of *LNCS*. Springer, 2004.

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

[13] J. Eckhardt, R. Kaibachev, E. Pašalíc, K. Swadi, and W. Taha. Implicitly heterogeneous multi-stage programming. In *Proc. 2005 Conf. on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *LNCS*. Springer, 2005.

[14] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4), 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.

[15] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proc. 2002 International Conf. on Functional Programming (ICFP 2002)*, pages 235–246, 2002.

[16] K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

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

[18] 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.

[19] G. Hutton and J. Wright. Compiling exceptions correctly. In *Mathematics of Program Construction*, volume 3125 of *LNCS*. Springer, 2004.

[20] O. Kiselyov, K. Swahi, and W. Taha. A methodology for generating verified combinatorial circuits. In *Fourth International Conference on Embedded Software*, pages 249–258. ACM, 2004.

[21] X. Leroy. Formal certification of a compiler back-end. In *Principles of Programming Languages 2006*, pages 42–54. ACM Press, 2006.

[22] Z. Luo. *Computation and Reasoning – A Type Theory for Computer Science*. Intl. Series of Monographs on Comp. Sci. OUP, 1994.

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

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

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

[26] J. McKinna. Why dependent types matter. In *Proc. ACM Symp. on Principles of Programming Languages (POPL 2006)*, pages 1–1, 2006.

[27] J. McKinna and J. Wright. A type-correct, stack-safe, provably correct expression compiler in Epigram. *Journal of Functional Programming*, 2006. To appear.

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

[29] E. Pašalíc. *The Role of Type-Equality in Meta-programming*. PhD thesis, OGI School of Science and Engineering, 2004.

[30] E. Pašalíc, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *Proc. 2002 International Conf. on Functional Programming (ICFP 2002)*. ACM, 2002.

[31] T. Sheard and S. Peyton-Jones. Template meta-programming for haskell. In *Proc. 2002 ACM Haskell workshop*, pages 1–16, 2002.

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

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

[34] W. Taha, S. Ellner, and H. Xi. Generating heap-bounded programs in a functional setting. In *Third International Conference on Embedded Software*, volume 2855 of *LNCS*. Springer, 2003.

[35] W. Taha and M. F. Nielsen. Environment classifiers. In *Proc. ACM Symp. on Principles of Programming Languages (POPL 2003)*, pages 26–37, 2003.

[36] T. Uustalu. Partiality is an effect, 2004. Talk at Dagstuhl workshop on Dependently Typed Progamming.