Chapter 6

Epigram Reloaded:

A Standalone Typechecker for ETT

James Chapman[1], Thorsten Altenkirch[1], Conor McBride[1]

Abstract Epigram, a functional programming environment with dependent types, interacts with the programmer via an extensible high level language of programming constructs which *elaborates* incrementally into Epigram's Type Theory, ETT, a rather spartan λ-calculus with dependent types, playing the rôle of a 'core language'. We implement a standalone typechecker for ETT in Haskell, allowing us to reload existing libraries into the system safely without re-elaboration.

Rather than adopting a rewriting approach to computation, we use a *glued* representation of values, pairing first-order syntax with a functional representation of its semantics, computed *lazily*. This approach separates β-*reduction* from $\beta\eta$-*conversion*. We consequently can not only allow the η-laws for λ-abstractions and pairs, but also collapse each of the unit and empty types.

6.1 INTRODUCTION

Epigram[2] [22, 5] is at the same time a functional programming language with dependent types and a type-driven, interactive program development system. Its type system is strong enough to express a wide range of program properties, from basic structural invariants to full specifications. Types assist interactive programming and help to keep track of the constraints an evolving program has to satisfy.

Epigram interacts with the programmer in an extensible high level language of programming constructs which is *elaborated* incrementally into Epigram's Type Theory, ETT. ETT is a rather spartan λ-calculus with dependent types, based on Luo's UTT (Unified Type Theory) [16] and more broadly on Martin-Löf's Type Theory [18]. It plays the rôle of a 'core language': it can be evaluated symbolically; it can also be compiled into efficient executable code, exploiting a new

[1]University of Nottingham, {jmc,txa,ctm}@cs.nott.ac.uk

[2]The Epigram system and its documentation are available from www.e-pig.org.

potential for optimisations due to the presence of dependent types [7].

Elaboration is *supposed* to generate well typed terms in ETT, but here we implement a standalone typechecker for ETT in Haskell. Why do we need this? Firstly, elaboration is expensive. We want to reload existing libraries into the system without re-elaborating their high-level source. However, to preserve safety and consistency, we should make sure that the reloaded code does typecheck.

Secondly, consumers may want to check mobile Epigram code before running it. A secure run-time system need not contain the elaborator: an ETT checker is faster, smaller and more trustworthy. McKinna suggested such a type theory for trading in 'deliverables' [23], programs paired with proofs, precisely combining computation and logic, with a single compact checker. More recent work on proof-carrying code [24] further emphasizes minimality of the 'trusted code base'.

Thirdly, as Epigram evolves, the elaborator evolves with it; ETT is much more stable. The present work provides an implementation of ETT which should accept the output of any version of the elaborator and acts as a target language reference for anyone wishing to extend or interoperate with the system.

We hope this paper will serve as a useful resource for anyone curious about how dependent typechecking can be done, especially as the approach we take is necessarily quite novel. Our treatment of evaluation in ETT takes crucial advantage of Haskell's laziness to deliver considerable flexibility in how much or little computation is done. Rather than adopting a conventional rewriting approach to computation, we use a *glued* representation of values, pairing first-order syntax with a functional representation of its semantics, computed *as required*.

This semantic approach readily separates β-*reduction* from βη-*conversion*. We support more liberal notions of 'conversion up to observation' by allowing not only the η-laws for λ-abstractions and pairs, but also identifying all elements of the unit type, 1. We further identify all elements of the empty type, O, thus making all types representing negative propositions $P \to \mathsf{O}$ *proof irrelevant*! These rules are new to Epigram—the definition [22] considers only β-equality. Adding them makes the theory more extensional, accepting more sensible programs and simplifying elaboration by allowing general solutions to more type constraints. It is also a stepping stone towards Observational Type Theory[4] based on [2]. The laws for 1 and O do not fit with Coquand and Abel's syntax-directed approach to conversion checking [1], but require a type-directed algorithm like ours.

Acknowledgments We gratefully acknowledge the support of EPSRC grant EP/C512022/1 'Observational Equality for Dependently Typed Programming'. We also thank James McKinna, Edwin Brady and Peter Morris for many useful discussions, and the anonymous referees for their helpful advice.

6.2 DEPENDENT TYPES AND TYPECHECKING

The heart of dependent type theory is the typing rule for application:

$$\frac{\Gamma \vdash f : \Pi x{:}S.\, T \qquad \Gamma \vdash s : S}{\Gamma \vdash f\, s : [x \mapsto s{:}S]T}$$

$$\frac{}{\vdash} \qquad \frac{\Gamma \vdash S : \star}{\Gamma; x\!:\!S \vdash} \qquad \frac{\Gamma \vdash s : S}{\Gamma; x \mapsto s\!:\!S \vdash}$$

FIGURE 6.1. Context validity rules $\boxed{\Gamma \vdash}$

The usual notion of function type $S \to T$ is generalised to the dependent function type $\Pi x\!:\!S.\,T$, where T may mention, hence *depend* on x. We may still write $S \to T$ if x does not appear in T. Π-types can thus indicate some relationship between the input of a function and its output. The type of our application instantiates T with the value of the argument s, by means of local definition. An immediate consequence is that *terms* now appear in the language of *types*. Moreover, we take types to be a subset of terms, with type $\star$, so that Π can also express polymorphism.

Once we have terms in types, we can express many useful properties of data. For example, consider *vector* types given by $\mathsf{Vec} : \mathsf{Nat} \to \star \to \star$, where a natural number fixes the *length* of a vector. We can now give concatenation the type

$$\textbf{vconc} : \Pi X\!:\!\star.\ \Pi m\!:\!\mathsf{Nat}.\ \Pi n\!:\!\mathsf{Nat}.\ \mathsf{Vec}\, m\, X \to \mathsf{Vec}\, n\, X \to \mathsf{Vec}\,(m+n)\, X$$

When we concatenate two vectors of length 3, we acquire a vector of length $3+3$; it would be most inconvenient if such a vector could not be used in a situation calling for a vector of length 6. That is, the arrival of terms in types brings with it, the need for *computation* in types. The computation rules for ETT do not only explain how to run programs, they play a crucial rôle in determining which types are considered the same. A key typing rule is *conversion*, which identifies the types of terms up to ETT's judgemental equality, not just syntactic equality.

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash S \simeq T : \star}{\Gamma \vdash s : T}$$

Formally, ETT is a system of inference rules for judgements of three forms

context validity	typing	equality
$\Gamma \vdash$	$\Gamma \vdash t : T$	$\Gamma \vdash t_1 \simeq t_2 : T$

We work relative to a *context* of parameters and definitions, which must have valid types and values—this is enforced by the context validity rules (figure 6.1). The empty context is valid and we may only extend it according to the two rules, introducing a parameter with a valid type or a well typed definition. In the implementation, we check each extension to the context as it happens, so we only ever work in valid contexts. In the formal presentation, we follow tradition in making context validity a precondition for each atomic typing rule.

Figure 6.2 gives the typing rules for ETT. We supply a unit type, 1, an empty type O, dependent function types $\Pi x : S.\, T$ and dependent pair types $\Sigma x : S.\, T$, abbreviated by $S \wedge T$ in the non-dependent case. We annotate λ-terms with their domain types and pairs with their range types in order to ensure that types can be

Declared and defined variables

$$\frac{\Gamma\vdash}{\Gamma\vdash x : S}\,x{:}S\in\Gamma \qquad \frac{\Gamma\vdash}{\Gamma\vdash x : S}\,x\mapsto s{:}S\in\Gamma$$

Universe

$$\frac{\Gamma\vdash}{\Gamma\vdash \star : \star}$$

Conversion

$$\frac{\Gamma\vdash s : S \quad \Gamma\vdash S\simeq T : \star}{\Gamma\vdash s : T}$$

Local definition

$$\frac{\Gamma;x\mapsto s{:}S\vdash t : T}{\Gamma\vdash [x\mapsto s{:}S]t : [x\mapsto s{:}S]T}$$

Type formation, introduction, and elimination

$$\frac{\Gamma\vdash}{\Gamma\vdash 1 : \star} \qquad \frac{\Gamma\vdash}{\Gamma\vdash \langle\rangle : 1}$$

$$\frac{\Gamma\vdash}{\Gamma\vdash \mathsf{O} : \star} \qquad \frac{\Gamma\vdash z : \mathsf{O}}{\Gamma\vdash z\text{Œ} : \Pi X{:}\star.X}$$

$$\frac{\Gamma;x{:}S\vdash T : \star}{\Gamma\vdash \Pi x{:}S.T : \star} \qquad \frac{\Gamma;x{:}S\vdash t : T}{\Gamma\vdash \lambda x{:}S.t : \Pi x{:}S.T} \qquad \frac{\begin{array}{l}\Gamma\vdash f : \Pi x{:}S.T\\ \Gamma\vdash s : S\end{array}}{\Gamma\vdash f\,s : [x\mapsto s{:}S]T}$$

$$\frac{\Gamma;x{:}S\vdash T : \star}{\Gamma\vdash \Sigma x{:}S.T : \star} \qquad \frac{\begin{array}{l}\Gamma\vdash s : S\\ \Gamma;x{:}S\vdash T : \star\\ \Gamma\vdash t : [x\mapsto s{:}S]T\end{array}}{\Gamma\vdash \langle s;t\rangle_T : \Sigma x{:}S.T} \qquad \frac{\Gamma\vdash p : \Sigma x{:}S.T}{\begin{array}{l}\Gamma\vdash p\pi_0 : S\\ \Gamma\vdash p\pi_1 : [x\mapsto p\pi_0{:}S]T\end{array}}$$

FIGURE 6.2. Typing rules $\boxed{\Gamma\vdash t : T}$

synthesised, not just checked. We write O's eliminator, Œ ('naught E'), and Σ-type projections, π_0 and π_1 postfix like application—-the eliminator for Π-types.

The equality rules (figure 6.3)[3] include β-laws which allow computations and expand definitions, but we also add η-laws and proof-irrelevance for certain types, justified by the fact that some terms are indistinguishable by observation. A proof-irrelevant type has, *as far as we can tell*, at most one element; examples are the unit type 1 and the empty type O. These rules combine to identify all inhabitants of $(A\to 1)\wedge(B\to \mathsf{O})$, for example.

Equality (hence type-) checking is decidable if all computations terminate. A carefully designed language can achieve this by executing only trusted programs in types, but we do not address this issue here. Indeed, our current implementation uses $\star : \star$ and hence admits non-termination due to Girard's paradox [11]. Here, we deliver the core functionality of typechecking. Universe stratification and positivity of inductive definitions are well established[15, 16] and orthogonal to the subject of this article.

6.3 EPIGRAM AND ITS ELABORATION

Epigram's high-level source code is *elaborated* incrementally into ETT. The elaborator produces the detailed evidence which justifies high-level programming con-

[3] We have omitted a number of trivial rules here, e.g. the rules stating that $\simeq$ is an equivalence and a number of congruence rules.

definition lookup and disposal

$$\frac{\Gamma\vdash}{\Gamma\vdash x\simeq s : S}x\mapsto s{:}S\in\Gamma \quad \frac{\Gamma\vdash s\simeq s' : S \quad \Gamma;x\mapsto s{:}S\vdash t\simeq t' : T}{\Gamma\vdash [x\mapsto s{:}S]t\simeq [x\mapsto s'{:}S]t' : [x\mapsto s{:}S]T}$$

structural rules for eliminations

$$\frac{\Gamma\vdash u\simeq u' : \mathsf{O}}{\Gamma\vdash u\,Œ\simeq u'\,Œ : \Pi x{:}\star.\,x} \quad \frac{\Gamma\vdash f\simeq f' : \Pi x{:}S.\,T \quad \Gamma\vdash s\simeq s' : S}{\Gamma\vdash f\,s\simeq f'\,s' : [x\mapsto s{:}S]T}$$

$$\frac{\Gamma\vdash p\simeq p' : \Sigma x{:}S.\,T}{\Gamma\vdash p\,\pi_0\simeq p'\,\pi_0 : S} \quad \frac{\Gamma\vdash p\simeq p' : \Sigma x{:}S.\,T}{\Gamma\vdash p\,\pi_1\simeq p'\,\pi_1 : [x\mapsto (p\,\pi_0){:}S]T}$$

β-rules

$$\frac{\Gamma\vdash \lambda x{:}S.\,t : \Pi x{:}S.\,T \quad \Gamma\vdash s : S}{\Gamma\vdash (\lambda x{:}S.\,t)\,s\simeq [x\mapsto s{:}S]t : [x\mapsto s{:}S]T}$$

$$\frac{\Gamma\vdash \langle s;t\rangle_T : \Sigma x{:}S.\,T}{\Gamma\vdash \langle s;t\rangle_T\,\pi_0\simeq s : S} \quad \frac{\Gamma\vdash \langle s;t\rangle_T : \Sigma x{:}S.\,T}{\Gamma\vdash \langle s;t\rangle_T\,\pi_1\simeq t : [x\mapsto s{:}S]T}$$

observational rules

$$\frac{\Gamma\vdash u : \mathsf{1} \quad \Gamma\vdash u' : \mathsf{1}}{\Gamma\vdash u\simeq u' : \mathsf{1}} \quad \frac{\Gamma\vdash z : \mathsf{O} \quad \Gamma\vdash z' : \mathsf{O}}{\Gamma\vdash z\simeq z' : \mathsf{O}}$$

$$\frac{\Gamma;x{:}S\vdash f\,x\simeq f'\,x : T}{\Gamma\vdash f\simeq f' : \Pi x{:}S.\,T} \quad \frac{\begin{array}{c}\Gamma\vdash p\,\pi_0\simeq p'\,\pi_0 : S\\ \Gamma\vdash p\,\pi_1\simeq p'\,\pi_1 : [x\mapsto (p\,\pi_0){:}S]T\end{array}}{\Gamma\vdash p\simeq p' : \Sigma x{:}S.\,T}$$

FIGURE 6.3. Equality rules $\boxed{\Gamma\vdash t\simeq t' : T}$

veniences, such as the kind of 'filling in the blanks' we usually associate with type inference. For example, we may declare Nat and Vec as follows:

$$\underline{\text{data}}\ \frac{}{\mathsf{Nat} : \star}\quad \underline{\text{where}}\ \frac{}{\mathsf{zero} : \mathsf{Nat}}\quad \frac{n : \mathsf{Nat}}{\mathsf{suc}\,n : \mathsf{Nat}}$$

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat}\,;\ X : \star}{\mathsf{Vec}\,n\,X : \star}\quad \underline{\text{where}}\ \frac{}{\mathsf{vnil} : \mathsf{Vec}\,\mathsf{zero}\,X}\quad \frac{x : X\,;\ xs : \mathsf{Vec}\,n\,X}{\mathsf{vcons}\,x\,xs : \mathsf{Vec}\,(\mathsf{suc}\,n)\,X}$$

The elaborator fleshes out the implicit parts of programs. Elaboration makes hidden quantifiers and their instances explicit. The above yields:

$\mathsf{Nat} : \star$	$\mathsf{Vec} : \Pi n{:}\mathsf{Nat}.\,\Pi X{:}\star.\,\star$
$\mathsf{zero} : \mathsf{Nat}$	$\mathsf{vnil} : \Pi X{:}\star.\,\mathsf{Vec}\,\mathsf{zero}\,X$
$\mathsf{suc} : \mathsf{Nat}\to\mathsf{Nat}$	$\mathsf{vcons} : \Pi X{:}\star.\,\Pi n{:}\mathsf{Nat}.\ X\to\mathsf{Vec}\,n\,X\to\mathsf{Vec}\,(\mathsf{suc}\,n)\,X$

For each datatype, the elaborator overloads the operator elim (postfix in ETT) with the standard induction principle. For $n : \mathsf{Nat}$ and $xs : \mathsf{Vec}\,n\,X$, we acquire

$n\ \underline{\mathrm{elim}}_{\mathsf{Nat}}$:	$xs\ \underline{\mathrm{elim}}_{\mathsf{Vec}}$:
$\quad\Pi P : \mathsf{Nat}\to\star.$	$\quad\Pi P : \Pi n{:}\mathsf{Nat}.\,\Pi xs{:}\mathsf{Vec}\,n\,X.\ \star\ .$
$\quad P\,\mathsf{zero}\to$	$\quad P\,\mathsf{zero}\,(\mathsf{vnil}\_X)\to$
$\quad(\Pi n'{:}\mathsf{Nat}.$	$\quad(\Pi n'{:}\mathsf{Nat}.\,\Pi x{:}X.\,\Pi xs'{:}\mathsf{Vec}\,n'\,X.$
$\quad\ P\,n'\to P\,(\mathsf{suc}\,n'))\to$	$\quad\ P\,n'\,xs'\to P\,(\mathsf{suc}\,n')\,(\mathsf{vcons}\,X\,n'\,x\,xs'))\to$
$\quad P\,n$	$\quad P\,n\,xs$

These types are read as schemes for constructing structurally recursive programs. Epigram has no hard-wired notion of pattern matching—rather, if you invoke an eliminator via the 'by' construct $\Leftarrow$, the elaborator reads off the appropriate patterns from its type. If we have an appropriate definition of $+$, we can define concatenation for vectors using <u>elim</u> (prefix in Epigram source) as follows:

$$\underline{\text{let}}\ \frac{x, y\ :\ \mathsf{Nat}}{x+y\ :\ \mathsf{Nat}} \qquad \begin{array}{l} x+y\ \Leftarrow\ \underline{\text{elim}}\,x \\ \quad \mathsf{zero}\ +y\ \Rightarrow y \\ \quad \mathsf{suc}\,x'+y\ \Rightarrow \mathsf{suc}\,(x'+y) \end{array}$$

$$\underline{\text{let}}\ \frac{xs\ :\ \mathsf{Vec}\,m\,X\ ;\ \ ys\ :\ \mathsf{Vec}\,n\,X}{\mathbf{vconc}\,xs\,ys : \mathsf{Vec}\,(m+n)\,X} \qquad \begin{array}{l} \mathbf{vconc}\,xs\,ys\ \Leftarrow\ \underline{\text{elim}}\,xs \\ \quad \mathbf{vconc}\ \mathsf{vnil}\qquad\quad ys \Rightarrow ys \\ \quad \mathbf{vconc}\ (\mathsf{vcons}\,x\,xs')\ ys \Rightarrow \mathsf{vcons}\,x\,(\mathbf{vconc}\,xs'\,ys) \end{array}$$

The elaborator then generates this lump of ETT, inferring the 'P' argument to xs $\underline{\text{elim}}_{\mathsf{Vec}}$ and constructing the other two from the branches of the program.

$$\begin{array}{l} \mathbf{vconc} \mapsto \lambda X\!:\!\star.\,\lambda m\!:\!\mathsf{Nat}.\,\lambda n\!:\!\mathsf{Nat}.\,\lambda xs\!:\!\mathsf{Vec}\,m\,X.\,\lambda ys\!:\!\mathsf{Vec}\,n\,X. \\ \qquad xs\ \underline{\text{elim}}_{\mathsf{Vec}}\,(\lambda m\!:\!\mathsf{Nat}.\,\lambda xs\!:\!\mathsf{Vec}\,m\,X.\,\Pi n\!:\!\mathsf{Nat}.\ \mathsf{Vec}\,n\,X \to \mathsf{Vec}\,(m+n)\,X) \\ \qquad\quad (\lambda n\!:\!\mathsf{Nat}.\,\lambda ys\!:\!\mathsf{Vec}\,n\,X.\,ys) \\ \qquad\quad (\lambda m'\!:\!\mathsf{Nat}.\,\lambda x\!:\!X.\,\lambda xs'\!:\!\mathsf{Vec}\,m'\,X.\ \lambda h\!:\!\Pi n\!:\!\mathsf{Nat}.\ \mathsf{Vec}\,n\,X \to \mathsf{Vec}\,(m'+n)\,X. \\ \qquad\quad\ \lambda n\!:\!\mathsf{Nat}.\,\lambda ys\!:\!\mathsf{Vec}\,n\,X.\,\mathsf{vcons}\,X\,(m'+n)\,x\,(h\,n\,ys)) \\ \qquad\quad n\,ys \end{array}$$

The elaborator works even harder in more complex situations, like this:

$$\underline{\text{let}}\ \frac{xs\ :\ \mathsf{Vec}\,(\mathsf{suc}\,n)\,X}{\mathbf{vtail}\,xs\ :\ \mathsf{Vec}\,n\,X} \qquad \begin{array}{l} \mathbf{vtail}\,xs\ \Leftarrow\ \underline{\text{elim}}\,xs \\ \quad \mathbf{vtail}\,(\mathsf{vcons}\,x\,xs') \Rightarrow xs' \end{array}$$

Here, the unification on lengths which eliminates the vnil case and specialises the vcons case rests on a <u>noConfusion</u> theorem—constructors disjoint and injective—proven by the elaborator for each datatype, and on the <u>subst</u> operator—replacing equal with equal. These techniques are detailed in [19, 20], but their effect is to deliver a large dull term which justifies the dependent case analysis.

$$\begin{array}{l} \mathbf{vtail} \mapsto \lambda n\!:\!\mathsf{Nat}.\,\lambda X\!:\!\star.\,\lambda xs\!:\!\mathsf{Vec}\,(\mathsf{suc}\,n)\,X.\,xs\ \underline{\text{elim}}_{\mathsf{Vec}} \\ \qquad (\lambda m\!:\!\mathsf{Nat}.\,\lambda ys\!:\!\mathsf{Vec}\,m\,X.\,\Pi n\!:\!\mathsf{Nat}.\,\Pi xs\!:\!\mathsf{Vec}\,(\mathsf{suc}\,n)\,X.\,\Pi q\!:\!m=\mathsf{suc}\,n.\,\Pi q'\!:\!ys=xs.\,\mathsf{Vec}\,n\,X) \\ \qquad (\lambda n\!:\!\mathsf{Nat}.\,\lambda xs\!:\!\mathsf{Vec}\,(\mathsf{suc}\,n)\,X.\,\lambda q\!:\!\mathsf{zero}=\mathsf{suc}\,n.\,\lambda q'\!:\!\mathsf{vnil}=xs.\,q\ \underline{\text{noConfusion}}_{\mathsf{Nat}}\,(\mathsf{Vec}\,n\,X)) \\ \qquad (\lambda n'\!:\!\mathsf{Nat}.\,\lambda x\!:\!X.\,\lambda xs'\!:\!\mathsf{Vec}\,n'. \\ \qquad\ \lambda h\!:\!\Pi n\!:\!\mathsf{Nat}.\,\Pi xs\!:\!\mathsf{Vec}\,(\mathsf{suc}\,n)\,X.\,\Pi q\!:\!n'=\mathsf{suc}\,n.\,\Pi q'\!:\!xs'=xs.\,\mathsf{Vec}\,n\,X. \\ \qquad\ \lambda n\!:\!\mathsf{Nat}.\,\lambda xs\!:\!\mathsf{Vec}\,(\mathsf{suc}\,n)\,X.\,\lambda q\!:\!\mathsf{suc}\,n'=\mathsf{suc}\,n.\,\lambda q'\!:\!\mathsf{vcons}\,X\,n'\,x\,xs'=xs. \\ \qquad\ q\ \underline{\text{noConfusion}}_{\mathsf{Nat}}\,(\mathsf{Vec}\,n\,X) \\ \qquad\quad (\lambda q\!:\!n'=n.\,q\ \underline{\text{subst}} \\ \qquad\qquad (\lambda n\!:\!\mathsf{Nat}.\,\Pi xs'\!:\!\mathsf{Vec}\,n'\,X.\,\Pi h\!:\!\Pi n\!:\!\mathsf{Nat}.\,\Pi xs\!:\!\mathsf{Vec}\,(\mathsf{suc}\,n)\,X.\,\Pi q\!:\!n'=\mathsf{suc}\,n.\,\Pi q'\!:\!xs'=xs.\,\mathsf{Vec}\,n\,X. \\ \qquad\qquad\ \Pi xs\!:\!\mathsf{Vec}\,(\mathsf{suc}\,n)\,X.\,\Pi q'\!:\!\mathsf{vcons}\,X\,n'\,x\,xs'=xs.\,\mathsf{Vec}\,n\,X) \\ \qquad\qquad (\lambda xs'\!:\!\mathsf{Vec}\,n'\,X.\,\lambda h\!:\!\Pi n\!:\!\mathsf{Nat}.\,\Pi xs\!:\!\mathsf{Vec}\,(\mathsf{suc}\,n)\,X.\,\Pi q\!:\!n'=\mathsf{suc}\,n.\,\Pi q'\!:\!xs'=xs.\,\mathsf{Vec}\,n\,X. \\ \qquad\qquad\ \lambda xs\!:\!\mathsf{Vec}\,(\mathsf{suc}\,n')\,X.\,\lambda q'\!:\!\mathsf{vcons}\,X\,n'\,x\,xs'=xs.\,q'\ \underline{\text{subst}}\,(\lambda xs\!:\!\mathsf{Vec}\,(\mathsf{suc}\,n')\,X.\,\mathsf{Vec}\,n'\,X)\,xs') \\ \qquad\qquad xs'\,h\,xs\,q')) \\ \qquad (\mathsf{suc}\,n)\,xs\,(\mathsf{refl}\,\mathsf{Nat}\,(\mathsf{suc}\,n))\,(\mathsf{refl}\,(\mathsf{Vec}\,(\mathsf{suc}\,n)\,X)\,xs) \end{array}$$

Merely *checking* all these details is much simpler than inferring them in the first place. Reloading ETT involves none of the complexity of implicit syntax handling or dependent pattern matching. Meanwhile, our observational equality rules help the elaborator by allowing more type constraints to have general solutions.

6.4 ETT SYNTAX IN HASKELL

We now implement ETT in Haskell. We first represent its syntax.

```
data Term  = R Reference                -- free variable (carries definition)
           | V Int                      -- bound variable (de Bruijn index)
           | Pi Type Scope              -- Πx:S. T
           | Si Type Scope              -- Σx:S. T
           | L Type Scope               -- λx:S. t
           | P Term (Term, Scope)       -- ⟨s;t⟩_T
           | Term :$ Elim Term          -- elimination form
           | C Const                    -- constant
           | Let (Term, Type) Scope     -- [x ↦ s:S]t
type Type  = Term   -- types are just a subset of terms
data Scope = (:.){ adv :: String, bdy :: Term }
data Elim t = A t | P0 | P1 | OE        -- − t, − π0, − π1, − Œ
data Const = Star | One | Void | Zero   -- ⋆, 1, ⟨⟩, O
```

As in [21], we explicitly separate free variables from bound, using a de Bruijn index [13] representation for the latter. Each time we bind a variable, the indices shift by one; we wrap up the term in scope of the new bound variable in the datatype Scope. This distinction helps to avoid silly mistakes, supports useful overloading and allows us to cache a string used only for display-name generation.

Correspondingly a λ-term carries a Type for its domain and a Scope for its body. Σ and Π types are represented similarly. Pairs P Term (Term, Scope) carry the range of their Σ-type—you cannot guess this from the type of the second projection, which gives only its instance for the value of the first projection.

We gather the constants in Const. We also collect the elimination forms Term :$ Elim Term, so that we can define their computational behaviour in one place. Elim is an instance of Functor in the obvious way. By way of example, the 'twice' function, $\lambda X\!:\!\star.\ \lambda f\!:\!X \to X.\ \lambda x\!:\!X.\ f\ (f\ x)$ becomes the following:

```
twice = L (C Star) ("X" :.  L (Pi (V 0) ("x" :. V 1)) ("f" :.
          L (V 1) ("x" :.  V 1 :$ A (V 1 :$ A (V 0))  )))
```

In section 6.6, we shall equip this syntax with a semantics, introducing the type Value which pairs these first-order terms with a functional representation of Scopes. We exploit this semantics in the free variables R Reference, which include both parameters and global definitions. A Reference carries its Name but also caches its type, and in the case of a definition, its value.

```
type Reference = Name := Typed Object
data Typed x   = (:∈){ trm :: x, typ :: Value }
data Object    = Para | Defn Value
```

It is easy to extend Object with tagged constructor objects and Elim with datatype eliminators which switch on the tags—constructing their types is explained in [21].

6.4.1 Navigation under binders

The operations $/\!\!/$ and $\backslash\!\!\backslash$ provide a means to navigate into and out of binders.

$(/\!\!/) :: \mathsf{Scope} \rightarrow \mathsf{Value} \rightarrow \mathsf{Term}$
  -- instantiates the bound variable of a Scope with a Value
$(\backslash\!\!\backslash) :: (\mathsf{Name}, \mathsf{String}) \rightarrow \mathsf{Term} \rightarrow \mathsf{Scope}$
  -- binds a variable free in a Term to make a Scope

Namespace management uses the techniques of [21]. Names are backward lists of Strings, resembling long names in module systems.

```
type Name = BList String
data BList x = B0 | BList x :◁ x deriving Eq
```

Our work is always relative to a root name: we define a Checking monad which combines the threading of this root and the handling of errors. For this presentation we limit ourselves to Maybe for errors.

```
newtype Checking x = MkChecking{ runChecking :: Name → Maybe x}
instance Monad Checking where
  return x = MkChecking $ λ_ → return x
  MkChecking f >>= g = MkChecking $ λname → do
    a ← f name
    runChecking (g a) name
```

User name choices never interfere with machine Name choices. Moreover, we ensure that different tasks never choose clashing names by locally extending the root name of each subtask with a different suffix.

```
(⊙ ) :: String → Checking x → Checking x
name ⊙ (MkChecking f) = MkChecking $ λroot → f (root :◁ name)
root :: Checking Name
root = MkChecking return
```

Whether we really need to or not, we uniformly give every subcomputation a distinct local name, trivially guaranteeing the absence of name clashes. In particular, we can use $x \odot$ root to generate a fresh name for a fresh variable if we ensure that x is distinct from the other local names.

6.5 CHECKING TYPES

In this section, we shall show how to synthesise the types of expressions and check that they are correct. Typechecking makes essential use of the semantics of terms. We defer our implementation of this semantics until section 6.6: here we indicate our *requirements* for our representation of Values.

The typing rules are realized by three functions infer, synth and check. Firstly, infer infers the type of its argument in a syntax-directed manner.

```
infer :: Term → Checking Value
```

Secondly, synth calls infer to check that its argument has a type and, *safe in this knowledge*, returns both its value and the inferred type.

```
synth :: Term → Checking (Typed Value)
synth t = do
  ty ← "ty" ⊙ infer t
  return (val t :∈ ty)
val :: Term → Value      -- must only be used with well-typed terms
syn :: Value → Term      -- recovers the syntax from a Value
```

Note that "ty" ⊙ infer *t* performs the inference in the namespace extended by "ty" ensuring that name choices made by infer *t* are local to the new namespace. Thirdly, check takes a Value representing a *required* type and a Term. It synthesises the value and type of the latter, then checks that types coincide, in accordance with the conversion rule.

```
check :: Value → Term → Checking Value
check ty t = do
  (tv :∈ sty) ← "sy" ⊙ synth t
  "eq" ⊙ areEqual ((ty, sty) :∈ vStar)
  return tv
```

Type checking will require us to ask the following questions about values:

```
areEqual :: Typed (Value, Value) → Checking ()
isZero :: Value → Checking ()
isPi, isSi :: Value → Checking (Value, ScoVal)
```

We have just seen that we need to check when types are equal. We also need to determine whether a type matches the right pattern for a given elimination form, extracting the components in the case of Π- and Σ-types. The ScoVal type gives the semantics of Scopes, with val and syn correspondingly overloaded, as we shall see in section 6.6.

In order to synthesise types, we shall need to construct values from checked components returned by infer, synth and check, isPi and isSi. We thus define 'smart constructors' which assemble Values from the semantic counterparts of the corrresponding Term constructors.

```
vStar, vAbsurd :: Value
vStar    = val (C Star)
vAbsurd = val (Pi (C Star) ("T" :. V 0))
vPi, vSi  :: Value → ScoVal → Value
vLet      :: Typed Value → ScoVal → Value
vdefn     :: Typed (Name, Value) → Value
vpara     :: (Typed Name) → Value
```

6.5.1 Implementing the Typing Rules

We will now define infer in accordance with the typing rules from figure 6.2. We match on the syntax of the term and in each case implement the rule with the corresponding conclusion, performing the checks in the hypotheses, then constructing the type from checked components. The base cases are easy: references cache their types and constants have constant types—we just give the case for $\star$.

```
infer (R (_:=(_:∈ ty))) = return ty
infer (C Star) = return vStar
```

The case for bound variables V *i* never arises. We always work with *closed* terms, instantiating a bound variable as we enter its Scope, abstracting it when we leave. Local definition is a case in point:

```
infer (Let (s,sty) t) = do
  styv ← "sty" ⊙ check vStar sty
  sv   ← "s"   ⊙ check styv s
  x    ← "x"   ⊙ root
  ttyv ← "tty" ⊙ infer (t//vdefn ((x,sv) :∈ styv))
  return (vLet (sv :∈ styv) (val ((x,adv t)\\syn ttyv)))
```

We check that *ty* is a type and that *s* inhabits it. The rules achieve this indirectly via context validity at each leaf of the typing derivation; we perform the check once, before vdefn creates the reference value which realises the extension of the context. The new variable gets its fresh name from "x" ⊙ root, and the corresponding value is used to instantiate the bound variable of *t*. Once we have *t*'s type, *ttyv*, we use vLet to build the type of the whole thing from checked components. Values do not support the (\\) operation, so we abstract *x* from the syntax of *ttyv*, then generate a semantic scope with val. Checking a Π-type requires a similar journey under a binder, but the resulting type is a simple $\star$.

```
infer (Pi dom ran) = do
  domv ← "dom" ⊙ check vStar dom
  x    ← "x"   ⊙ root
  _    ← "ran" ⊙ check vStar (ran//vpara (x :∈ domv))
  return vStar
```

We check that *dom* is a type, then create a fresh variable and instantiate the range, ensuring that it also is a type. Checking a Σ-type works the same way. Meanwhile, to typecheck a λ, we must use the type inferred under the binder to generate the Π-type of the function, abstracting a scope from its syntax as we did for Let.

```
infer (L dom t) = do
  domv ← "dom" ⊙ check vStar dom
  x    ← "x"   ⊙ root
  ranv ← "ran" ⊙ infer (t//vpara (x :∈ domv))
  return (vPi domv (val ((x,adv t)\\syn ranv)))
```

To infer the type of an application we check that the 'function' actually has a Π-type, revealing the domain type for which to check the argument. If all is well we let-bind the return type, corresponding to the rule exactly.

```
infer (f :$ A a) = do
  fty ← "f" ⊙ infer f
  (dom, ran) ← isPi fty
  av ← "a" ⊙ check dom a
  return (vLet (av :∈ dom) ran)
```

Here is how we infer the type of pairs:

```
infer (P s (t, ran)) = do
  tys@(sv :∈ domv) ← "s"   ⊙ synth s
  x                ← "x"   ⊙ root
  _                ← "ran" ⊙ check vStar (ran // vpara (x :∈ domv))
  _                ← "t"   ⊙ check (vLet tys (val ran)) t
  return (vSi domv (val ran))
```

First, we ensure that s is well typed yielding the domain of the Σ-type. Next, we check that the supplied range *ran* is a type in the context extended with the parameter of the domain type. Then we check t in the appropriately let bound range. We then deliver the Σ-type. Meanwhile, projections are straightforward.

```
infer (p :$ P0) = do              infer (p :$ P1) = do
  pty      ← "p" ⊙ infer p          pty        ← "p" ⊙ infer p
  (dom, _) ← isSi pty               (dom, ran) ← isSi pty
  return dom                        return (vLet ((val (p :$ P0)) :∈ dom) ran)
```

Finally, eliminating the empty type always yields absurdity!

```
infer (z :$ OE) = do
  zty ← "z" ⊙ infer z
  isZero zty
  return vAbsurd
```

6.6 FROM SYNTAX TO SEMANTICS

We shall now give a definition of Value which satisfies the requirements of our checker. Other definitions are certainly possible, but this one has the merit of allowing considerable control over which computations happen.

```
data Glued t w = (:⇓){syn :: t, sem :: w}
type Value     = Glued Term  Whnf
type ScoVal    = Glued Scope (Value → Whnf)
```

A Value glues a Term to a functional representation of its weak head normal form (Whnf). The semantic counterpart of a Scope is a ScoVal, which affixes a Haskell function, delivering the meaning of the scope with its bound variable instantiated.

Just as in 'normalisation-by-evaluation' [6], the behaviour of scopes (for Π and Σ, not just λ) is delivered by the implementation language, but if we want to read a Value, we just project its syntax. Whnfs are given as follows:

```
data Whnf = WR Reference (BList (Elim Value))          -- Spine
          | WPi Value ScoVal | WSi Value ScoVal        -- Π-type, Σ-type
          | WL ScoVal        | WP Value Value          -- λ-abstraction, pair
          | WC Const                                   -- Constant
```

The only elimination forms we need to represent are those which operate on an inert parameter, hence we pack them together, with the WR constructor. Bound variables do not occur, except within the Scope part of a ScoVal. We drop the type annotations on λ-abstractions and pairs as they have no operational use. With this definition, operations such as isPi, isSi and isZero can be implemented directly by pattern matching on Whnf. Meanwhile, the computational behaviour of Values is given by the overloaded $$ operator:

```
class Eliminable t where
  ($$) :: t → (Elim Value) → t
instance Eliminable Value where
  t $$ e = (syn t :$ fmap syn e) :⇓ (sem t $$ e)
instance Eliminable Whnf where
  WL (_ :⇓ f) $$ A v = f v            -- β-reduction by Haskell application
  WP x _      $$ P0  = sem x           -- projections
  WP _ y      $$ P1  = sem y
  WR x es     $$ e   = WR x (es ⊲ e)   -- inert computations
```

We shall now use $$ to deliver the function eval which makes values from checked syntax. This too is overloaded, and its syntactic aspect relies on the availability of substitution of *closed* terms for bound variables.

```
type Env = BList Value
bproj :: BList x → Int → x
class Close t where
  close :: t → Env → Int → t   -- the Int is the first bound variable to replace
class Close t ⇒ Whnv t w | t → w where
  whnv :: t → Env → w
  eval :: t → Env → Glued t w
  eval t γ = (close t γ 0) :⇓ (whnv t γ)
  val :: t → Glued t w
  val t = t :⇓ whnv t B0
```

We export val, for closed terms, to the typechecker. However, eval and whnv, defined mutually, thread an environment γ explaining the bound variables. By separating Scope from Term, we can say how to go under a binder once, for all.

```
instance Close Scope where
  close (s :. t) γ i = s :. close t γ (i + 1)      -- start γ further out
```

```
instance Whnv Scope (Value → Whnf) where
  whnv (_:. t) γ = λx → whnv t (γ :◁x)    -- extend the environment
```

Meanwhile, whnv for Term traverses the syntax, delivering the semantics.

```
instance Whnv Term Whnf where
  whnv (R (_:= (Defn v :∈ _))) _ = sem v
  whnv (R r)                   _ = WR r B0
  whnv (V i)                   γ = sem (bproj γ i)
  whnv (Pi d r)                γ = WPi (eval d γ) (eval r γ)
  whnv (Si d r)                γ = WSi (eval d γ) (eval r γ)
  whnv (L _ r)                 γ = WL (eval r γ)
  whnv (P x (y,_))             γ = WP (eval x γ) (eval y γ)
  whnv (t :$ e)                γ = whnv t γ $$ fmap (`eval`γ) e
  whnv (C c)                   _ = WC c
  whnv (Let (t,_) s)           γ = whnv s γ (eval t γ)
```

Defined free variables are expanded; parameters gain an empty spine; γ explains bound variables. We interpret (:\$) with (\$\$). Lets directly exploit the their bodies' functional meaning. Everything else is structural.

The close operation just substitutes the environment for the bound variables, without further evaluation. The Int counts the binders crossed, hence the number of variables which should stay bound. We give only the interesting cases:

```
instance Close Term where
  close t      B0 _ = t
  close (V j) γ   i = if j < i then V j else syn (bproj γ (j − i))
```

6.7 CHECKING EQUALITY

Our equality algorithm does 'on-the-fly' η-expansion on weak-head β-normal forms, directed by their types. The observational rules for elements of Π and Σ-types perform the η-expansion to yield η-long normal forms at ground type ($\star$,1 or *Zero*). We now define areEqual skipping the structural cases for constant types, WPi, WSi, and going straight to the interaction between the the observational rules and checking equality on spines.

We do not need to look at elements of type 1 to know that they are equal to $\langle\rangle$. Elements of O (hypothetical, of course) are also equal. We compare functions by applying them to a fresh parameter and pairs by comparing their projections.

```
areEqual :: Typed (Value, Value) → Checking ()
areEqual (_ :∈ (_ :⇓ WC One )) = return ()
areEqual (_ :∈ (_ :⇓ WC Zero)) = return ()
areEqual ((f, g) :∈ (_ :⇓ WPi dom ran)) = do
  x ← "x" ⊙ root
  let v = vpara (x :∈ dom)
  "ran" ⊙ areEqual ((f $$ A v, f $$ A v) :∈ vLet (v :∈ dom) ran)
```

```
areEqual ((p,q) :∈ (_ :⇓ WSi dom ran)) = do
  "fst" ⊙ areEqual ((p $$ P0, q $$ P0) :∈ dom)
  "snd" ⊙ areEqual ((p $$ P1, q $$ P1) :∈ (vLet (p $$ P0 :∈ dom) ran))
```

For ground terms of types other than 1 and O, we can only have inert references with spines, which we compare in accordance with the structural rules. We rebuild the type of a spine as we process it, in order to compare its components correctly.

```
areEqual ((_ :⇓ WR r1@(_ := (_ :∈ ty)) as, _ :⇓ WR r2 bs) :∈ _) =
  spineEq (as, bs) where
    spineEq :: (Elim Value, Elim Value) → Checking Value
```

We peel eliminators until we reach the reference, whose type we pass back.

```
spineEq (B0, B0) = guard (r1 ≡ r2) ≫ return ty
```

For applications, we check that preceding spines are equal and analyse the Π-type they deliver; we then confirm that the arguments are equal elements of its domain and pass on the instantiated range.

```
spineEq (as :◁ A a, bs :◁ A b) = do
  sty ← spineEq (as, bs)
  (dom, ran) ← isPi sty
  "eqargs" ⊙ areEqual ((a, b) :∈ dom)
  return (vLet (a :∈ dom) ran)
```

For like projections from pairs we analyse the Σ-type from the preceding spines and pass on the appropriate component, instantiated if need be.

```
spineEq (as :◁ P0, bs :◁ P0) = do
  sty ← spineEq (as, bs)
  (dom, _) ← isSi sty
  return dom
spineEq (as :◁ P1, bs :◁ P1) = do
  sty ← spineEq (as, bs)
  (dom, ran) ← isSi sty
  return (vLet ((spine (as :◁ P0)) :∈ dom) ran)
```

For ‘naught E’, we need look no further!

```
spineEq (as :◁ OE, bs :◁ OE) = return vAbsurd
spine :: (Elim Value) → Value
spine B0 = val (R r1)
spine (es :◁ e) = spine r1 es $$ e
```

6.8 RELATED WORK

Type checking algorithms for dependent types are at the core of systems like Lego [17] and Coq [9] (which have only β-equality) and Agda [10], for which Coquand's simple algorithm with $\beta\eta$-equality for Π-types [12] forms the core; he and Abel have recently extended this to Σ-types [1]. Our more liberal equality makes it easy to import developments from these systems, but harder to export to them.

Coquand's and Abel's algorithms are syntax-directed: comparison proceeds structurally on β-normal forms, except when comparing $\lambda x.\, t$ with some variable-headed (or 'neutral') f, which gets expanded to $\lambda x.\, f\, x$. Also, when comparing $\langle s,t\rangle$ with neutral p, the latter expands to $\langle p\,\pi_0, p\,\pi_1\rangle$. Leaving two neutral functions or pairs unexpanded cannot make them appear different, so this 'tit-for-tat' η-expansion suffices. However, there is no such syntactic cue for $\mathsf{1}$ or O: apparently distinct neutral terms can be equal, if they have a proof-irrelevant type.

We have taken type-directed η-expansion from normalisation-by-evaluation [6, 3], fusing it with the conversion check. Our whnv is untyped and lazy, but compilation in the manner of Gregoire and Leroy [14] would certainly pay off for heavy type-level computations, especially if enhanced by Brady's optimisations [7, 8].

6.9 CONCLUSIONS AND FURTHER WORK

The main deliverable of our work is a standalone typechecker for ETT which plays an important rôle in the overall architecture of Epigram. We have addressed a number of challenges in implementing a stronger conversion incorporating observational rules. These simplify elaboration and will play a vital rôle in our project to implement Observational Type Theory[4] whose equality judgement remains decidable, but which supports *reasoning* up to observation as in [2].

REFERENCES

[1] Andreas Abel and Thierry Coquand. Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. In *Typed Lambda Calculus and Applications*, pages 23–38, 2005.

[2] Thorsten Altenkirch. Extensional equality in intensional type theory. In *LICS 99*, 1999.

[3] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In David Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, LNCS 953, pages 182–199, 1995.

[4] Thorsten Altenkirch and Conor McBride. Towards observational type theory. Manuscript, available online, February 2006.

[5] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[6] Ulrich Berger and Helmut 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 Science Press, Los Alamitos, 1991.

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

[8] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferrucio Damiani, editors, *Types for Proofs and Programs, Torino, 2003*, volume 3085 of *LNCS*, pages 115–129. Springer-Verlag, 2004.

[9] L'Équipe Coq. The Coq Proof Assistant Reference Manual. http://pauillac.inria.fr/coq/doc/main.html, 2001.

[10] Catarina Coquand and Thierry Coquand. Structured Type Theory. In *Workshop on Logical Frameworks and Metalanguages*, 1999.

[11] Thierry Coquand. An analysis of Girard's paradox. In *Proceedings of the First IEEE Symposium on Logic in Computer Science, Cambridge, Massachussetts*, pages 227–236, 1986.

[12] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.

[13] Nicolas G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicæ*, 34:381–392, 1972.

[14] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.

[15] Robert Harper and Randy Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

[16] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.

[17] Zhaohui Luo and Robert Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, LFCS, 1992.

[18] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.

[19] Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*, volume 2277 of *LNCS*. Springer-Verlag, 2002.

[20] Conor McBride, Healfdene Goguen, and James McKinna. A Few Constructions on Constructors. In *Types for Proofs and Programs, Paris, 2004*, LNCS. Springer-Verlag, 2005. accepted; to appear.

[21] Conor McBride and James McKinna. Functional Pearl: I am not a Number: I am a Free Variable. In Henrik Nilsson, editor, *Proceedings of the ACM SIGPLAN Haskell Workshop 2004, Snowbird, Utah*. ACM, 2004.

[22] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.

[23] James McKinna. *Deliverables: A Categorical Approach to Program Development in Type Theory*. PhD thesis, LFCS, 1992.

[24] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, January 1997.