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 \(\lambda\)-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 \(\beta\)-reduction from \(\beta\eta\)-conversion. We consequently can not only allow the \(\eta\)-laws for \(\lambda\)-abstractions and pairs, but also collapse each of the unit and empty types.

6.1 INTRODUCTION

Epigram\(^2\) \(\cite{22,5}\) 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 \(\lambda\)-calculus with dependent types, based on Luo’s UTT (Unified Type Theory) \(\cite{16}\) and more broadly on Martin-Löf’s Type Theory \(\cite{18}\). 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 $\beta$-reduction from $\beta\eta$-conversion. We support more liberal notions of ‘conversion up to observation’ by allowing not only the $\eta$-laws for $\lambda$-abstractions and pairs, but also identifying all elements of the unit type, 1. We further identify all elements of the empty type, $\emptyset$, thus making all types representing negative propositions $P \rightarrow \emptyset$ proof irrelevant! These rules are new to Epigram—the definition [22] considers only $\beta$-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 $\emptyset$ 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:

$$\frac{\Gamma \vdash f : \Pi x : S. T \quad \Gamma \vdash s : S}{\Gamma \vdash f \, s : [x \mapsto s] \, T}$$
The usual notion of function type $S \rightarrow T$ is generalised to the dependent function type $\Pi x : S. T$, where $T$ may mention, hence depend on $x$. We may still write $S \rightarrow T$ if $x$ does not appear in $T$. $\Pi$-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 $\Pi$ 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 $\text{Vec} : \text{Nat} \rightarrow \star \rightarrow \star$, where a natural number fixes the length of a vector. We can now give concatenation the type $\text{vconc} : \Pi X : \star. \Pi m : \text{Nat}. \Pi n : \text{Nat}. \text{Vec} m X \rightarrow \text{Vec} n X \rightarrow \text{Vec} (m + n) X$

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.

$$\Gamma \vdash s : S \quad \Gamma \vdash S \simeq T : \star \quad \Gamma \vdash s : T$$

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

<table>
<thead>
<tr>
<th>Context Validity</th>
<th>Typing</th>
<th>Equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash$</td>
<td>$\Gamma \vdash t : T$</td>
<td>$\Gamma \vdash t_1 \simeq t_2 : T$</td>
</tr>
</tbody>
</table>

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 $\emptyset$, 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 $\lambda$-terms with their domain types and pairs with their range types in order to ensure that types can be
<table>
<thead>
<tr>
<th>Declared and defined variables</th>
<th>Universe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ ⊢ x : S ∈ Γ</td>
<td>Γ ⊢ Γ ⊢ x ∈ S</td>
</tr>
<tr>
<td>Conversion</td>
<td>Local definition</td>
</tr>
<tr>
<td>Γ ⊢ s : S, Γ ⊢ S ∼ T : *</td>
<td>Γ ⊢ s : S ⊢ t : T</td>
</tr>
<tr>
<td>Type formation, introduction, and elimination</td>
<td></td>
</tr>
<tr>
<td>Γ ⊢ Γ ⊢ * : *</td>
<td>Γ ⊢ Γ ⊢ () : 1</td>
</tr>
<tr>
<td>Γ ⊢ Γ ⊢ O : *</td>
<td>Γ ⊢ Γ ⊢ z : O</td>
</tr>
<tr>
<td>Γ ⊢ Γ ⊢ x : S ⊢ T : *</td>
<td>Γ ⊢ Γ ⊢ f : Π x : S. T</td>
</tr>
<tr>
<td>Γ ⊢ Γ ⊢ Π x : S. T : *</td>
<td>Γ ⊢ Γ ⊢ s : S</td>
</tr>
<tr>
<td>Γ ⊢ Γ ⊢ Π x : S. T : *</td>
<td>Γ ⊢ Γ ⊢ f s : [x ↦ s : T]</td>
</tr>
<tr>
<td>Γ ⊢ Γ ⊢ x : S ⊢ T : *</td>
<td>Γ ⊢ Γ ⊢ Γ ⊢ p : Σ x : S. T</td>
</tr>
<tr>
<td>Γ ⊢ Γ ⊢ Σ x : S. T : *</td>
<td>Γ ⊢ Γ ⊢ (s t) : Σ x : S. T</td>
</tr>
<tr>
<td>Γ ⊢ Γ ⊢ p p₀ : T</td>
<td>Γ ⊢ Γ ⊢ p p₁ : [x ↦ p p₀ : S] T</td>
</tr>
</tbody>
</table>

**FIGURE 6.2.** Typing rules \( \Gamma ⊢ t : T \)

Synthesized, not just checked. We write \( \mathcal{O} \)'s eliminator, \( \mathcal{E} \) (‘naught E’), and \( \Sigma \)-type projections, \( \pi_0 \) and \( \pi_1 \) postfix like application—the eliminator for \( \Pi \)-types.

The equality rules (figure 6.3)\(^3\) include \( \beta \)-laws which allow computations and expand definitions, but we also add \( \eta \)-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 \( \mathcal{O} \). These rules combine to identify all inhabitants of \( (A → 1) ∧ (B → \mathcal{O}) \), 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 \( * : * \) 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-...
definition lookup and disposal

\[
\begin{align*}
\Gamma \vdash x & \simeq s : S, x \mapsto s : S \in \Gamma \\
\Gamma \vdash s \simeq s' : S & \implies \Gamma; x \mapsto s : S, t \simeq t' : T \\
\Gamma \vdash [x \mapsto s : S] t & \simeq [x \mapsto s' : S] t' \implies [x \mapsto s : S] t \simeq [x \mapsto s' : S] t'
\end{align*}
\]

structural rules for eliminations

\[
\begin{align*}
\Gamma \vdash u \simeq u' : O & \implies \Gamma; u \mapsto u' : \Pi x : \ast, x \\
\Gamma \vdash f \simeq f' : \Pi x : S, T & \implies \Gamma; s \simeq s' : S \\
\Gamma \vdash f \ s \simeq f' \ s' & \implies [x \mapsto s : S] t \simeq [x \mapsto s' : S] t'
\end{align*}
\]

\begin{tabular}{ll}
\hline
\textbf{β-rules} & \\
\hline
\end{tabular}

\[
\begin{align*}
\Gamma \vdash \lambda x : S.t : \Pi x : S, T & \implies \Gamma; s \simeq S \\
\Gamma \vdash (\lambda x : S.t) s & \simeq [s : S] t : [x \mapsto s : S] T \\
\Gamma \vdash \langle s : t \rangle T & : \Sigma x : S, T \\
\Gamma \vdash \langle s : t \rangle T \pi_0 & \simeq s : S \\
\Gamma \vdash \langle s : t \rangle T \pi_1 & \simeq t : [x \mapsto s : S] T
\end{align*}
\]

observational rules

\[
\begin{align*}
\Gamma \vdash u : 1 & \implies \Gamma; u' : 1 \\
\Gamma \vdash u \simeq u' & \implies \Gamma; \cdot \simeq \cdot : O \\
\Gamma \vdash \zeta : O & \implies \Gamma; \zeta' : O \\
\Gamma \vdash p \pi_0 & \simeq p' \pi_0 : S \\
\Gamma \vdash \ s : \Pi x : S, T & \implies \Gamma; f \ s \simeq f' \ s : T \\
\Gamma \vdash f \ s & \simeq f' \ s : T & \implies \Gamma; p \pi_1 & \simeq p' \ pi_1 : [x \mapsto (p \pi_0) : S] T
\end{align*}
\]

\[\Gamma; t \simeq t' : T\]

\[
\text{FIGURE 6.3. Equality rules}\]

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:

\[
\begin{align*}
data & & \text{Nat} & : & \ast & \\
& & \text{zero} & : & \text{Nat} & \\
& & \text{succ} & : & \text{Nat} & \rightarrow \text{Nat} \\
data & & \text{Vec} & : & \Pi n : \text{Nat}. \Pi X : \ast. \\
& & \text{vnil} & : & \Pi X : \ast. \text{Vec} n X & \\
& & \text{vcons} & : & \Pi X : \ast. \Pi n : \text{Nat}. X \rightarrow \text{Vec} n X \rightarrow \text{Vec (suc n) X}
\end{align*}
\]

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

\[
\begin{align*}
\text{Nat} & : \ast \\
\text{zero} : \text{Nat} & \\
\text{succ} : \text{Nat} \rightarrow \text{Nat} & \\
\text{Vec} & : \Pi n : \text{Nat}. \Pi X : \ast. \\
\text{vnil} & : \Pi X : \ast. \text{Vec} n X \\
\text{vcons} & : \Pi X : \ast. \Pi n : \text{Nat}. X \rightarrow \text{Vec} n X \rightarrow \text{Vec (suc n) X}
\end{align*}
\]

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

\[
\begin{align*}
n \ \text{elim}_{\text{Nat}} : & \\
\Pi P : \text{Nat} \rightarrow \ast. & \\
P \text{zero} & \rightarrow P \text{zero} (\text{vnil} X) \\
(P \text{elim}_{\text{Nat}} : & \\
\Pi P : \Pi n : \text{Nat}. \Pi X : \ast. \text{Vec} n X. \ast. & \\
P \text{zero} & \rightarrow P \text{zero} (\text{vnil} X) \\
(P \text{elim}_{\text{Vec}} : & \\
\Pi P : \Pi n : \text{Nat}. \Pi X : \ast. \text{Vec} n X. & \\
P n' & \rightarrow P (\text{succ} n') \rightarrow P (\text{succ} n') (\text{vcons} X n' x x s') \\
(P n & \rightarrow P n x x s)
\end{align*}
\]

83
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 \texttt{elim} (prefix in Epigram source) as follows:

\[
\begin{align*}
\text{let} & \quad x, y : \text{Nat} \\
\text{let} & \quad x + y : \text{Nat} \quad \text{such that} \quad x + y \leftarrow \text{elim} x \\
\end{align*}
\]

\[
\begin{align*}
\text{zero} & \quad + y \Rightarrow y \\
\text{suc} \quad x' + y \Rightarrow \text{suc} (x' + y) \\
\end{align*}
\]

\[
\begin{align*}
\text{let} & \quad x : \text{Vec} m X; \quad y : \text{Vec} n X \\
\text{vconc} x; y : \text{Vec} (m+n) X \quad \text{vconc} \; x; y \leftarrow \text{elim} \; x \\
\text{vconc} \; \text{vnil} \quad y s \Rightarrow y s \\
\text{vconc} \; (\text{vcons} \; x; y s) \quad y s \Rightarrow \text{vcons} \; (\text{vconc} \; x; y s) \\
\end{align*}
\]

The elaborator then generates this lump of ETT, inferring the ‘\( P \)’ argument to \( \text{elim}_{\text{Vec}} \) and constructing the other two from the branches of the program.

\[
\begin{align*}
vconc & \quad \leftarrow \lambda x : \text{Vec} \; m X. \lambda \eta : \text{Nat}. \; \lambda x : \text{Vec} \; m X. \lambda y : \text{Vec} \; n X. \\
vconc \; (x; y) & \quad \rightarrow \lambda \eta : \text{Nat}. \; \lambda x : \text{Vec} \; m X. \Pi_{\text{Nat}}. \; \text{Vec} \; n X \rightarrow \text{Vec} \; (m+n) X \\
(\lambda \eta : \text{Nat}. \; \lambda x : \text{Vec} \; m X. \; \text{elim} \; (\lambda m : \text{Nat}. \; \lambda x : \text{Vec} \; m X. \; \Pi_{\text{Nat}}. \; \text{Vec} \; n X \rightarrow \text{Vec} \; (m+n) X)) \\
(\lambda \eta : \text{Nat}. \; \lambda x : \text{Vec} \; m X. \; \lambda \eta : \text{Nat}. \; \lambda x : \text{Vec} \; m X. \; \lambda y : \text{Vec} \; n X. \; \lambda \eta : \text{Nat}. \; \lambda x : \text{Vec} \; m X. \; \text{vconc} \; x; y s \rightarrow \text{vconc} \; x; y s) \\
\end{align*}
\]

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

\[
\begin{align*}
\text{let} & \quad x : \text{Vec} \; (\text{suc} \; n) X \\
\text{vtail} \; x : \text{Vec} \; n X \quad \text{vtail} \; x s \leftarrow \text{elim} \; x \quad \text{vtail} \; (\text{vcons} \; x; y s) \Rightarrow x s' \\
\end{align*}
\]

Here, the unification on lengths which eliminates the \text{vnil} case and specialises the \text{vcons} case rests on a noConfusion theorem—constructors disjoint and injective—proven by the elaborator for each datatype, and on the subst 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.

\[
\begin{align*}
v\text{tail} & \quad \leftarrow \lambda x : \text{Nat}. \; \lambda x : \text{Vec} \; (\text{suc} \; n) X \; \text{elim} \; x \; (\text{vconc} \; x; y s) \\
(\lambda m : \text{Nat}. \; \lambda x : \text{Vec} \; m X. \; \text{elim} \; x \; (\text{vconc} \; x; y s)) \\
(\lambda \eta : \text{Nat}. \; \lambda x : \text{Vec} \; (\text{suc} \; n) X. \; \text{vconc} \; x; y s \rightarrow \text{vconc} \; x; y s) \\
(\lambda m : \text{Nat}. \; \lambda x : \text{Vec} \; m X. \; \lambda \eta : \text{Nat}. \; \lambda x : \text{Vec} \; m X. \; \lambda \eta : \text{Nat}. \; \lambda x : \text{Vec} \; m X. \; \text{vconc} \; x; y s \rightarrow \text{vconc} \; x; y s) \\
\end{align*}
\]

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.

84
6.4 ETT SYNTAX IN HASKELL

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

```haskell
data Term
    = R Reference -- free variable (carries definition)
    | V Int        -- bound variable (de Bruijn index)
    | Pi Type Scope -- Π \( x : S . T \)
    | Si Type Scope -- Σ \( x : S . T \)
    | L Type Scope -- λ \( x : S . t \)
    | P Term (Term, Scope) -- \( \langle s ; t \rangle \)
    | Term :$ Elim Term -- elimination form
    | C Const       -- constant
    | Let (Term, Type) Scope -- \( [x \mapsto s : S] t \)

type Type = Term -- types are just a subset of terms

data Scope = (::,)
    { adv :: String, bdy :: Term }

data Elim t = A t | P0 | P1 | OE
    -- \( \langle x \mapsto s : S \rangle t \)

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, \( λ \ X : ⋆ . λ f : X \to X . λ x : X . f ( f x ) \) becomes the following:

```haskell
  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.

```haskell
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 \ provide a means to navigate into and out of binders.

\( (/) \) :: Scope \( \rightarrow \) Value \( \rightarrow \) Term

-- instantiates the bound variable of a Scope with a Value

\( (\backslash) \) :: (Name, String) \( \rightarrow \) Term \( \rightarrow \) 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.

\textbf{type} Name = BList String

\textbf{data} BList \( x = B0 \mid \text{BList } x \bowtie x \) 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.

\textbf{newtype} Checking \( x = \text{MkChecking } \{ \text{runChecking :: Name } \rightarrow \text{Maybe } x \} \)

\textbf{instance} Monad Checking \textbf{where}

\textbf{return} \( x = \text{MkChecking } \lambda \_ \rightarrow \text{return } x \)

\textbf{MkChecking} \( f \gg g = \text{MkChecking } \lambda name \rightarrow \text{do }\)

\( a \leftarrow f \text{name} \)

\( \text{runChecking } (g a) \text{name} \)

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.

\( (\circ) \) :: String \( \rightarrow \) Checking \( x \rightarrow \) Checking \( x \)

\( name \circ (\text{MkChecking } f) = \text{MkChecking } \lambda root \rightarrow f (\text{root } \bowtie name) \)

\( \text{root} :: \text{Checking Name} \)

\( \text{root} = \text{MkChecking return} \)

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 \( x \circ \text{root} \) 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.

\[
\text{synth} :: \text{Term} \rightarrow \text{Checking (Typed Value)}
\]

\[
\text{synth } t = \begin{array}{l}
\text{ty} \leftarrow "t.y" \odot \text{infer } t \\
\text{return } (\text{val } t : \in \text{ty})
\end{array}
\]

\[
\text{val} :: \text{Term} \rightarrow \text{Value} \quad -- \text{must only be used with well-typed terms}
\]

\[
\text{syn} :: \text{Value} \rightarrow \text{Term} \quad -- \text{recovers the syntax from a Value}
\]

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

\[
\text{check} :: \text{Value} \rightarrow \text{Term} \rightarrow \text{Checking Value}
\]

\[
\text{check } ty t = \begin{array}{l}
(\text{tv} : \in \text{sty}) \leftarrow "s.y" \odot \text{synth } t \\
"s\circ q" \odot \text{areEqual } ((ty, sty) : \in \text{vStar})
\end{array}
\]

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

\[
\text{areEqual} :: \text{Typed } (\text{Value, Value}) \rightarrow \text{Checking ()}
\]

\[
\text{isZero} :: \text{Value} \rightarrow \text{Checking ()}
\]

\[
\text{isPi, isSi} :: \text{Value} \rightarrow \text{Checking } (\text{Value, ScoVal})
\]

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 \Pi- and \Sigma-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 corresponding Term constructors.

\[
\text{vStar, vAbsurd} :: \text{Value}
\]

\[
\text{vStar} = \text{val } (C \text{ Star})
\]

\[
\text{vAbsurd} = \text{val } (\Pi \text{ (C Star) } ("T" :: V 0))
\]

\[
\text{vPi, vSi} :: \text{Value} \rightarrow \text{ScoVal} \rightarrow \text{Value}
\]

\[
\text{vLet} :: \text{Typed Value} \rightarrow \text{ScoVal} \rightarrow \text{Value}
\]

\[
\text{vdefn} :: \text{Typed } (\text{Name, Value}) \rightarrow \text{Value}
\]

\[
\text{vpara} :: (\text{Typed Name}) \rightarrow \text{Value}
\]

87
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 `⋆`.

\[
\text{infer } (R (\_:= (\_\in ty))) = \text{return } ty
\]
\[
\text{infer } (C \text{ Star}) = \text{return } v\text{Star}
\]

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:

\[
\text{infer } (\text{Let } (s, sty) t) = \text{do}
\]
\[
styv \leftarrow "sty" \circ \text{check } v\text{Star } sty
\]
\[
sv \leftarrow "s" \circ \text{check } styv s
\]
\[
x \leftarrow "x" \circ \text{root}
\]
\[
ttyv \leftarrow "ttv" \circ \text{infer } (t/vdefn ((x,sv) \in styv))
\]
\[
\text{return } (v\text{Let } (sv: \in styv) (\text{val } ((x, advt) \| syn ttyv)))
\]

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 "\(x\)" \circ 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 \(Π\) 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 `⋆`.

\[
\text{infer } (Π \text{ dom ran}) = \text{do}
\]
\[
domv \leftarrow "dom" \circ \text{check } v\text{Star } dom
\]
\[
x \leftarrow "x" \circ \text{root}
\]
\[
ranv \leftarrow "ran" \circ \text{infer } (t/vpara (x: \in domv))
\]
\[
\text{return } v\text{Star}
\]

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`.

\[
\text{infer } (L \text{ dom t}) = \text{do}
\]
\[
domv \leftarrow "dom" \circ \text{check } v\text{Star } dom
\]
\[
x \leftarrow "x" \circ \text{root}
\]
\[
ranv \leftarrow "ran" \circ \text{infer } (t/vpara (x: \in domv))
\]
\[
\text{return } (vΠ \text{ domv } (\text{val } ((x, advt) \| syn ranv)))
\]
To infer the type of an application we check that the `function` actually has a $\Pi$-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.

```plaintext
tyf ← "f" ⊙ infer f
(dom, ran) ← isPi tyf
av ← "a" ⊙ check dom a
return (\Let (av :∈ dom) ran)
```

Here is how we infer the type of pairs:

```plaintext
tys@ (sv :∈ domv) ← "s" ⊙ synth s
x ← "x" ⊙ root
ran ← "ran" ⊙ check \Star (ran // vpara (x :∈ domv))
− ← "−" ⊙ check (\Let tys (val ran)) t
return (\Let domv (val ran))
```

First, we ensure that $s$ is well typed yielding the domain of the $\Sigma$-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 $\Sigma$-type. Meanwhile, projections are straightforward.

```plaintext
tyP ← "p" ⊙ infer p
(dom, _) ← isSi tyP
return dom
```

```plaintext
tyP ← "p" ⊙ infer p
(dom, ran) ← isSi tyP
return (\Let (val (p : $P$ P0)) :∈ dom) ran
```

Finally, eliminating the empty type always yields absurdity!

```plaintext
tyz ← "z" ⊙ infer z
isZero tyz
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.

```plaintext
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. Whnf\$s are given as follows:

\[
data \text{Whnf} = \begin{array}{l}
\text{WR Reference (BList (Elim Value))} & \text{-- Spine} \\
\text{WPi Value ScoVal} | \text{WSi Value ScoVal} & \text{-- Π-type, Σ-type} \\
\text{WL ScoVal} | \text{WP Value Value} & \text{-- λ-abstraction, pair} \\
\text{WC Const} & \text{-- Constant}
\end{array}
\]

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:

\[
\text{class Eliminable } t \text{ where } \\
(\$\$) :: t → (Elim Value) → t
\]

\[
\text{instance Eliminable Value where } \\
t$\$e = \begin{array}{l}
\text{(syn } t : \$\$ fmap syn } e ) \\
\text{(sem } t \$\$ e )
\end{array}
\]

\[
\text{instance Eliminable Whnf where } \\
\text{WL } (\_ : \$\$ f ) \$\$ A v = f v & \text{-- β-reduction by Haskell application} \\
\text{WP } x _\_ \$\$ P0 = \text{sem } x & \text{-- projections} \\
\text{WP } _\_ y \$\$ P1 = \text{sem } y \\
\text{WR } x \_ e \$\$ e = \text{WR } x (\_ es e ) & \text{-- inert computations}
\]

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 \textit{closed} terms for bound variables.

\[
\text{type Env} = \begin{array}{l}
\text{BList Value}
\end{array}
\]

\[
bproj :: \text{BList } x \rightarrow \text{Int } → x
\]

\[
\text{class Close } t \text{ where } \\
\text{close} :: t → \text{Env } → \text{Int } → t & \text{-- the Int is the first bound variable to replace}
\]

\[
\text{class Close } t ⇒ \text{Whnv } t w | t → w \text{ where } \\
\text{whnv} :: t → \text{Env } → w \\
\text{eval} :: t → \text{Env } → \text{Glued } t w \\
\text{eval } t γ = (\text{close } t γ 0 : \$\$ (\text{whnv } t γ ) \\
\text{val} :: t → \text{Glued } t w \\
\text{val } t = t : \$\$ \text{whnv } t B0
\]

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.

\[
\text{instance Close Scope where } \\
\text{close } (s : : t) γ i = s : : \text{close } t γ (i + 1) & \text{-- start γ further out}
\]
instance Whnv Scope (Value → Whnf) where
  whnv (\_: t) \gamma = \lambda x \to whnv t (\gamma \otimes 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) \gamma = sem (bproj \gamma i)
  whnv (Pi d r) \gamma = WPI (eval d \gamma) (eval r \gamma)
  whnv (Si d r) \gamma = WSI (eval d \gamma) (eval r \gamma)
  whnv (L _ r) \gamma = WL (eval r \gamma)
  whnv (P x (y, _)) \gamma = WP (eval x \gamma) (eval y \gamma)
  whnv (t:$ e) \gamma = whnv t \gamma \$ fmap ('eval' \gamma) e
  whnv (C c) _ = WC c
  whnv (Let (t, _ s)) \gamma = whnv s \gamma (eval t \gamma)

Defined free variables are expanded; parameters gain an empty spine; \gamma 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 i) \gamma i = if j < i then V j else syn (bproj \gamma (j - i))

6.7 CHECKING EQUALITY

Our equality algorithm does 'on-the-fly' \eta-expansion on weak-head \beta-normal forms, directed by their types. The observational rules for elements of \Pi and \Sigma-types perform the \eta-expansion to yield \eta-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 \Omega (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 ← "\"\" ∘ root
  let v = vpara (x :\_ dom)
  "\ran" ∘ areEqual ((f \$ A v, f \$ A v) :\_ :\_ Let (v :\_ dom) ran)
areEqual \( ((p, q) : (\_ : \downarrow WSi \; \text{dom \; ran})) = \text{do} \)

"fst" \( \circ \) areEqual \( ((p \# P0, q \# P0) : \text{dom}) \)

"snd" \( \circ \) areEqual \( ((p \# P1, q \# P1) : (\text{vLet} \; (p \# P0 : \text{dom} \; \text{ran})) \)

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 \( ((\_ : \downarrow \; \text{WR} \; r1 \; @ (\_ := (\_ : \in \; \text{ty})) \; as, (\_ : \downarrow \; \text{WR} \; r2 \; bs) : \in \; \_)) = \text{spineEq} \; (as, bs) \) where

spineEq :: (\text{Elim \; Value}, \text{Elim \; Value}) \rightarrow \text{Checking \; Value}

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

spineEq (B0, B0) = \text{guard} \; (r1 \equiv r2) \gg \text{return} \; \text{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 \( (as \times A \; a, bs \times A \; b) = \text{do} \)

sty ← spineEq \( (as, bs) \)

\( (\text{dom, ran}) \leftarrow \text{isPi} \; \text{sty} \)

"eqargs" \( \circ \) areEqual \( ((a, b) : \in \; \text{dom}) \)

return \( (\text{vLet} \; (a : \in \; \text{dom} \; \text{ran})) \)

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 \( (as \times P0, bs \times P0) = \text{do} \)

sty ← spineEq \( (as, bs) \)

\( (\text{dom, \_}) \leftarrow \text{isSi} \; \text{sty} \)

return \( \text{dom} \)

spineEq \( (as \times P1, bs \times P1) = \text{do} \)

sty ← spineEq \( (as, bs) \)

\( (\text{dom, ran}) \leftarrow \text{isSi} \; \text{sty} \)

return \( (\text{vLet} \; ((\text{spine} \; (as \times P0)) : \in \; \text{dom} \; \text{ran})) \)

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

spineEq \( (as \times OE, bs \times OE) = \text{return} \; \text{vAbsurd} \)

spine :: (\text{Elim \; Value}) \rightarrow \text{Value}

spine B0 = \text{val} \; (R \; r1)

spine \( (es \times e) = \text{spine} \; r1 \; es \; \$ \; e \)

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 Co-
quand’s simple algorithm with βη-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 λx.t with some variable-headed (or ‘neutral’) f, which gets expanded to λx.fx. Also, when comparing ⟨s,t⟩ with neutral p, the latter expands to ⟨p π₀,p π₁⟩. 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 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


