

# Epigram Reloaded: A Standalone Typechecker for ETT

James Chapman, Thorsten Altenkirch, Conor McBride

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

## 1 INTRODUCTION

Epigram<sup>1</sup> [18, 3] is at the same time a functional programming language with dependent types and a type-driven, interactive program development system. Epigram’s type system is strong enough to express a wide range of program properties, from basic structural invariants to full specifications. The system supports the interactive development of programs exploiting types to direct the design process and helping the programmer to keep track of the constraints the evolving program has to satisfy.

Epigram interacts with the programmer in terms of a high level language of user extensible programming constructs which is *elaborated* into Epigram’s Type Theory, ETT. ETT is a rather spartan  $\lambda$ -calculus with dependent types, based on Luo’s UTT (Unified Type Theory) [12] and more broadly on Martin-Löf’s Type Theory [14]. 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 potential for optimisations due to the presence of dependent types [4].

Type correctness is built into the elaboration process, but in this paper, we shall implement a standalone typechecker for ETT in Haskell. Why do we need this?

1. We want to reload existing library components into the Epigram system without wasting time re-elaborating the program. However, to preserve safety and consistency, we should make sure that the reloaded code does typecheck.
2. Consumers of our code may not trust the elaborator but want to ensure the safety of code before running it. This is understandable—the elaborator is a big program which may easily contain bugs, while the typechecker is a small program with a precise specification which may be implemented and verified independently. We have learnt from proof carrying code [19] that we must reduce the size of our trusted code base.

### Acknowledgments

We gratefully acknowledge the support of EPSRC grant EP/C512022/1 ‘Observational Equality for Dependently Typed Programming’. We should also like to thank James McKinna, Edwin Brady and Peter Morris for many useful discussions.

<sup>1</sup>The Epigram system and its documentation is available from [www.e-pig.org](http://www.e-pig.org).

## 1.1 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 conveniences, such as the kind of ‘filling in the blanks’ we usually associate with type inference. For example, we may declare the *vectors*—lists with a specific length—as follows:

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

The elaborator fleshes out all the implicit parts of these declarations, making the hidden quantifications explicit:

$$\begin{aligned} \text{Vec} & : \Pi n : \text{Nat}. \Pi X : \star. \star \\ \text{vnil} & : \Pi X : \star. \text{Vec zero } X \\ \text{vcons} & : \Pi X : \star. \Pi n : \text{Nat}. \Pi x : X. \Pi xs : \text{Vec } n X. \text{Vec } (\text{suc } n) X \end{aligned}$$

ETT is fully explicit, and it annotates  $\Pi$ -quantifiers for hidden arguments with ‘\_’, and the corresponding applications  $f\_s$ . We shall show these annotations for clarity now, but ignore them in the implementation of the typechecker here, for ease of presentation.

For each datatype, the elaborator extends the standard overloaded recursion operator `elim`. We acquire a new specific instance in ETT, which we write postfix. In the case of vectors, if  $xs : \text{Vec } n X$ , then

$$\begin{aligned} xs \underline{\text{elim}}_{\text{Vec}} & : \Pi P : \Pi n : \text{Nat}. \Pi xs : \text{Vec } n X. \star . \\ & \quad \Pi mvn : P \text{ zero } (\text{vnil } X) . \\ & \quad \Pi mvc : \Pi n' : \text{Nat}. \Pi x : X. \Pi xs' : \text{Vec } n' X . \\ & \quad \quad \Pi h : P n' xs' . \\ & \quad \quad P (\text{suc } n') (\text{vcons } X n' x xs') . \\ & \quad P n xs \end{aligned}$$

You can see this type as a scheme for constructing programs by simultaneous ‘pattern matching’ on a vector and its length. Epigram has no hard-wired notion of pattern matching—rather, if you invoke an eliminator (Eg. `elim xs`) via the ‘by’ construct  $\Leftarrow$ , the elaborator reads off the appropriate patterns from the type of the eliminator. If we have an appropriate definition of  $+$ , we can define  $++$  for vectors using `elim` as follows:

$$\begin{aligned} \underline{\text{let}} \frac{x, y : \text{Nat}}{x + y : \text{Nat}} & \quad x + y \Leftarrow \underline{\text{elim}} x \\ & \quad \text{zero} + y \Rightarrow y \\ & \quad (\text{suc } x') + y \Rightarrow \text{suc } (x' + y) \\ \\ \underline{\text{let}} \frac{xs : \text{Vec } m X ; ys : \text{Vec } n X}{xs ++ ys : \text{Vec } (m + n) X} & \quad xs ++ ys \Leftarrow \underline{\text{elim}} xs \\ & \quad \text{vnil} ++ ys \Rightarrow ys \\ & \quad (\text{vcons } x xs') ++ ys \Rightarrow \text{vcons } x (xs' ++ ys) \end{aligned}$$

Given this definition, what the elaborator actually generates for  $(++)$  is this rather less readable lump of ETT:

$$\begin{aligned}
(++)\mapsto & \lambda X : \star. \lambda m : \text{Nat}. \lambda n : \text{Nat}. \lambda xs : \text{Vec } m \ X. \lambda ys : \text{Vec } n \ X. \\
& xs \text{elim}_{\text{Vec}} (\lambda m : \text{Nat}. \lambda xs : \text{Vec } m \ X. \Pi n : \text{Nat}. \Pi ys : \text{Vec } n \ X. \text{Vec } (m+n) \ X) \\
& (\lambda n : \text{Nat}. \lambda ys : \text{Vec } n \ X. ys) \\
& (\lambda m' : \text{Nat}. \lambda x : X. \lambda xs' : \text{Vec } m' \ X. \\
& \lambda h : \Pi n : \text{Nat}. \Pi ys : \text{Vec } n \ X. \text{Vec } (m'+n) \ X. \\
& \lambda n : \text{Nat}. \lambda ys : \text{Vec } n \ X. \text{vcons } X \_ (m'+n) \ x (h n ys)) \\
& n \ ys
\end{aligned}$$

The elaborator works even harder when the situation is more complex. In the following program, we need to apply  $\text{elim}_{\text{Vec}}$  not for a vector of any length, but only for *nonempty* vectors. Correspondingly, we only get a  $\text{vcons}$  case— $\text{vnil}$  is impossible because zero is not  $(\text{suc } n)$ .

$$\text{let } \frac{xs : \text{Vec } (\text{suc } n) \ X}{\text{vtail } xs : \text{Vec } n \ X} \quad \text{vtail } xs \Leftarrow \text{elim } xs \\
\text{vtail } (\text{vcons } x \ xs') \Rightarrow xs'$$

All of the hard work is done behind the scenes. The first-order unification on lengths which eliminates the  $\text{vnil}$  case and specialises the  $\text{vcons}$  case rests on standard noConfusion theorem—constructors are disjoint and injective—proven by the elaborator for each datatype, and on the subst operator—replacing equal with equal. These techniques and constructions are detailed in [15, 16]. The effect is to deliver a large but dull term in ETT which provides a checkable explanation for the apparent ‘magic’ of dependent case analysis.

$$\begin{aligned}
\text{vtail} \mapsto & \lambda n : \text{Nat}. \lambda X : \star. \lambda xs : \text{Vec } (\text{suc } n) \ X. xs \text{elim}_{\text{Vec}} \\
& (\lambda m : \text{Nat}. \lambda ys : \text{Vec } m \ X. \Pi n : \text{Nat}. \Pi xs : \text{Vec } (\text{suc } n) \ X. \Pi q : m = \text{suc } n. \Pi q' : ys = xs. \text{Vec } n \ X) \\
& (\lambda n : \text{Nat}. \lambda xs : \text{Vec } (\text{suc } n) \ X. \lambda q : \text{zero} = \text{suc } n. \lambda q' : \text{vnil} = xs. q \text{noConfusion}_{\text{Nat}} (\text{Vec } n \ X)) \\
& (\lambda n' : \text{Nat}. \lambda x : X. \lambda xs' : \text{Vec } n' \ X. \\
& \lambda h : \Pi n : \text{Nat}. \Pi xs : \text{Vec } (\text{suc } n) \ X. \Pi q : n' = \text{suc } n. \Pi q' : xs' = xs. \text{Vec } n \ X. \\
& \lambda n : \text{Nat}. \lambda xs : \text{Vec } (\text{suc } n) \ X. \lambda q : \text{suc } n' = \text{suc } n. \lambda q' : \text{vcons } X \ n' \ x \ xs' = xs. \\
& q \text{noConfusion}_{\text{Nat}} (\text{Vec } n \ X) \\
& (\lambda q : n' = n. q \text{subst} \\
& (\lambda n : \text{Nat}. \Pi xs' : \text{Vec } n' \ X. \Pi h : \Pi n : \text{Nat}. \Pi xs : \text{Vec } (\text{suc } n) \ X. \Pi q : n' = \text{suc } n. \Pi q' : xs' = xs. \text{Vec } n \ X. \\
& \Pi xs : \text{Vec } (\text{suc } n) \ X. \Pi q' : \text{vcons } X \ n' \ x \ xs' = xs. \text{Vec } n \ X) \\
& (\lambda xs' : \text{Vec } n' \ X. \lambda h : \Pi n : \text{Nat}. \Pi xs : \text{Vec } (\text{suc } n) \ X. \Pi q : n' = \text{suc } n. \Pi q' : xs' = xs. \text{Vec } n \ X. \\
& \lambda xs : \text{Vec } (\text{suc } n) \ X. \lambda q' : \text{vcons } X \ n' \ x \ xs' = xs. q' \text{subst} (\lambda xs : \text{Vec } (\text{suc } n) \ X. \text{Vec } n' \ X) \ xs') \\
& xs' \ h \ xs \ q')) \\
& (\text{suc } n) \ xs \ (\text{refl } \_ . \text{Nat} \_ . (\text{suc } n)) \ (\text{refl } \_ . (\text{Vec } (\text{suc } n) \ X) \_ . xs)
\end{aligned}$$

## 1.2 Checking dependent types

A type checker for a language with dependent types has to decide symbolic conversion. E.g. to check that the definition of  $++$  above is well typed we may want to check that the type of the left hand side  $(\text{vcons } x \ xs') ++ ys : \text{Vec } ((\text{suc } m) + n) \ X$  agrees with the type of the right hand side  $\text{vcons } x \ (xs' ++ ys) : \text{Vec } (\text{suc } (m+n)) \ X$ . Hence the typechecker has to exploit that  $(\text{suc } m) + n \simeq \text{suc } (m+n)$ .

We specify our type system by judgements which are inductively defined by derivation rules. Apart from the usual judgement  $\Gamma \vdash t : T$  expressing that  $t$  has

type  $T$  given the typing assumptions in  $\Gamma$  we also introduce the judgements  $\Gamma \vdash$  of context validity and equality  $\Gamma \vdash t \simeq t' : T$ . The typing judgement also covers the case  $\Gamma \vdash T : \star$  expressing that  $T$  is a valid type wrt  $\Gamma$ . Typing and equality are connected via the conversion rule:

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

This rule allows us to use equalities, like the one above, during type checking. The conversion rule interacts nicely with the dependent application rule

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

which instantiates the codomain of a dependent function using local definitions  $[x \mapsto s : S]T$ .

The equality rules include  $\beta$ -rules which allow us to carry out computations and replace names by their definitions. Apart from these purely computational rules we add  $\eta$ -rules and rules for proof-irrelevant types, which are justified by the fact that some terms cannot be distinguished by any observation. A proof-irrelevant type is a type which, observationally, has at most one element; examples are the unit type  $1$  and the empty type  $0$ .

Equality checking and hence type checking is decidable, if all computations terminate. While a carefully designed language can achieve this goal by only executing trusted programs on type level we will not address this issue here. Indeed, our current implementation of Epigram uses  $\star : \star$  (Type is a Type) and hence contains non-terminating terms due to Girard's paradox [7]. This article is concerned with type checking core functionality. Stratification of universes and well-formedness of inductive definitions are well established [11, 12] and orthogonal to the subject of this article.

### 1.3 Related Work

Type checking algorithms for dependent types are at the core of implementations of systems like LEGO [13], COQ [5] or AGDA [6]. Coquand [8] presents a simple algorithm which decides  $\beta\eta$ -equality for  $\Pi$ -types; in joint work with Abel [1] this has recently been extended to include the  $\beta\eta$ -equality for  $\Sigma$ -types. Coquand's and Abel's algorithms are shape-directed and it is not obvious how to extend this approach to include the  $\eta$ -rule for  $1$  or proof-irrelevance for  $0$ . The algorithm we present here is type-directed and hence can deal with those situations.

## 2 EPIGRAM'S TYPE THEORY IMPLEMENTED

We present here the formal definition of ETT together with a sketch of our implementation of a type checker for it in Haskell.

## 2.1 Syntax

We start with the definition of the syntax and its representation in Haskell. Note that in a dependent type theory, types are just a subset of terms.

The following constants are terms:

- ★ the type of types
- 1 the unit type
- ⟨ ⟩ the only element of the unit type
- the empty type;

The variable  $x$  is a term;

If  $S, T, f, s, t$  and  $p$  are terms, so are the following:

- $\Pi x : S. T$  Dependent function space
- $\lambda x : S. T$  Lambda abstraction
- $f s$  Function application
- $\Sigma x : S. T$  Dependent product space
- $\langle s ; t \rangle_T$  Pair
- $p \pi_0$  First projection
- $p \pi_1$  Second projection
- $z \mathbb{E}$  Zero Elimination, or ‘naught E’ if you prefer.
- $[x \mapsto s : S]t$  local definition

Our Haskell representation of terms:

```
data Term = R Reference           -- Free variable
          | V Int                 -- Bound variable
          | D Quant Term Scope    --  $\Pi$ -type or  $\Sigma$ -type
          | L Term Scope          --  $\lambda$ -abstraction
          | P Term Term Scope     -- Pair
          | C Con                 -- Constant
          | Term :$ (Elim Term)   -- Elimination
          | (Term : $\in$  Term) :! Scope -- Local definition (let)
deriving Show
```

where  $:\in$  is just syntactic sugar for a pair indicating that it is a term or value together with its type:

```
data a : $\in$  b = a : $\in$  b
```

Our representation of free variables (*R Reference*), which caters for parameters and

global definitions, carries more than the just the name in the term. It also carries the type in the case of a parameter, and the actual value of the term being defined and its corresponding type in the case of a global definition.

```
type Reference = Name := (Object :∈ Value)
data Object    = Para | Defn Value
```

For bound variables, we use a de Bruijn index [10] representation (*Var Int*). As in [17], each time we shift indices, we wrap up the term in scope of the new bound variable in the datatype *Scope*. This distinction helps to avoid silly mistakes and allows us to cache a string to be used only for display-name generation.

```
data Scope = String :. Term
```

Correspondingly a  $\lambda$ -term carries a *Term* for its domain and a *Scope* for its body.  $\Sigma$  and  $\Pi$  types are represented similarly, (*D Quant Term Scope*) where

```
data Quant = Sig | Pi deriving (Show, Eq)
```

Pairs (*P Term Term Scope*) carry the range of their  $\Sigma$ -type—you cannot guess this given only the particular instance for a specific element of the domain.

The canonical constants are as follows:

```
data Con = Star
         | One
         | Void
         | Zero
deriving Show
```

All the elimination forms are written postfix (*Term* :\$ (*Elim Term*)), where

```
data Elim t = A t -- application
         | PO -- first projection
         | PI -- second projection
         | OE -- ‘naught E’, delivering anything from Zero
deriving Show
```

It is straightforward to add the standard eliminators for inductive datatypes, but we omit them here for reasons of space.

## 2.2 Checking types and equality

ETT consists of three judgement forms and a set of rules for deriving them.

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

$$\boxed{\frac{}{\vdash} \quad \frac{\Gamma \vdash S : \star}{\Gamma; x : S \vdash} \quad \frac{\Gamma \vdash s : S}{\Gamma; x \mapsto s : S \vdash}}$$

**FIGURE 1.** Context validity rules  $\boxed{\Gamma \vdash}$

The rules for context validity (figure 1) ensure that the context can only be extended by a well-typed variables. The empty context is valid and we may only extend it according to the two extension rules, introducing a typing assumptions or a local definition, so we need only consider valid contexts. In the implementation, the checking of contexts is done implicitly.

The rules for typing (figure 2) are the central part of our type theory. In the implementation they are realized by two functions, one for *type synthesis*

$synth :: Term \rightarrow Checking (Value \in Value)$

and one for *type checking*

$check :: (Term \in Value) \rightarrow Checking Value$

Here *Checking* is a monad which combines the threading of name spaces and the handling of error messages. We use the type *Value* to represent terms together with a semantic representation of their weak head normal form.

*synth* takes a term and synthesizes its type according to the typing rules. It necessitates that  $\lambda$ -abstractions and pairs are annotated by their type's domains and ranges respectively. It is clear from the rules that introduce these constructions that without the annotations we would be unable to synthesize their types.

*check* can be implemented using *synth* and *equate* which checks conversion:

```

check (tm :∈ ty) = do
  (tmval :∈ synthty) ← synth tm
  equate ((ty, synthty) :∈ star)
  return tmval
  
```

*check* also realizes the conversion rule (see figure 2) by calling *equate*:

Declared and defined variables		Universe
$\frac{\Gamma \vdash}{\Gamma \vdash x : S} x : S \in \Gamma$	$\frac{\Gamma \vdash}{\Gamma \vdash x : S} x \mapsto s : S \in \Gamma$	$\frac{\Gamma \vdash}{\Gamma \vdash \star : \star}$
Conversion		Local definition
$\frac{\Gamma \vdash s : S \quad \Gamma \vdash S \simeq T : \star}{\Gamma \vdash s : T}$		$\frac{\Gamma; x \mapsto s : S \vdash t : T}{\Gamma \vdash [x \mapsto s : S]t : [x \mapsto s : S]T}$
Type formation,	introduction,	and elimination
$\frac{\Gamma \vdash}{1 : \star}$	$\frac{\Gamma \vdash}{\Gamma \vdash \langle \rangle : 1}$	
$\frac{\Gamma \vdash}{\Gamma \vdash 0 : \star}$		$\frac{\Gamma \vdash z : 0}{\Gamma \vdash z \mathbf{CE} : \Pi X : \star. X}$
$\frac{\Gamma; x : S \vdash T : \star}{\Gamma \vdash \Pi x : S. T : \star}$	$\frac{\Gamma; x : S \vdash t : T}{\Gamma \vdash \lambda x : S. t : \Pi x : S. T}$	$\frac{\Gamma \vdash f : \Pi x : S. T \quad \Gamma \vdash s : S}{\Gamma \vdash f s : [x \mapsto s : S]T}$
$\frac{\Gamma; x : S \vdash T : \star}{\Gamma \vdash \Sigma x : S. T : \star}$	$\frac{\Gamma \vdash s : S \quad \Gamma \vdash t : [x \mapsto s : S]T}{\Gamma \vdash \langle s; t \rangle_T : \Sigma x : S. T}$	$\frac{\Gamma \vdash p : \Sigma x : S. T}{\Gamma \vdash p \pi_0 : S}$ $\Gamma \vdash p \pi_1 : [x \mapsto p \pi_0 : S]T$

**FIGURE 2. Typing rules**  $\boxed{\Gamma \vdash t : T}$

$equate :: ((Value, Value) : \in Value) \rightarrow Checking ()$

$equate$  determines whether two well-typed terms are convertible following the rules<sup>2</sup> given in figure 3. It uses the supplied type to implement the observational rules for  $\Pi$ ,  $\Sigma$ ,  $1$  and  $0$  by a type-directed process. These rules are equivalent to the usual  $\eta$ -rules for  $\Pi$ ,  $\Sigma$  and  $1$ , together with proof irrelevance for  $0$ . We maintain values in a semantic weak-head-normal form, using a *glued* representation exploiting ideas from [9]:

```

data  $t : \Downarrow w = t : \Downarrow w$ 
type  $Value = Term : \Downarrow WhnV$ 
 $syn :: (t : \Downarrow w) \rightarrow t$ 
 $syn (t : \Downarrow \_ ) = t$ 
 $sem :: (t : \Downarrow w) \rightarrow w$ 

```

<sup>2</sup>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.

<p>definition lookup and disposal</p> $\frac{\Gamma \vdash}{\Gamma \vdash x \simeq s : S} x \mapsto s : S \in \Gamma \quad \frac{\Gamma \vdash s \simeq s' : S \quad \Gamma; x \mapsto s : S \vdash t \simeq t' : T}{\Gamma \vdash [x \mapsto s : S]t \simeq [x \mapsto s' : S]t' : [x \mapsto s : S]T}$ <p>structural rules for eliminations</p> $\frac{\Gamma \vdash u \simeq u' : O}{\Gamma \vdash u \mathbb{E} \simeq u' \mathbb{E} : \Pi x : \star. x} \quad \frac{\Gamma \vdash f \simeq f' : \Pi x : S. T \quad \Gamma \vdash s \simeq s' : S}{\Gamma \vdash fs \simeq f's' : [x \mapsto s : S]T}$ $\frac{\Gamma \vdash p \simeq p' : \Sigma x : S. T}{\Gamma \vdash p\pi_0 \simeq p'\pi_0 : S} \quad \frac{\Gamma \vdash p \simeq p' : \Sigma x : S. T}{\Gamma \vdash p\pi_1 \simeq p'\pi_1 : [x \mapsto (p\pi_0) : S]T}$ <p><math>\beta</math>-rules</p> $\frac{\Gamma \vdash \lambda x : S. t : \Pi x : S. T \quad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x : S. t)s \simeq [x \mapsto s : S]t : [x \mapsto s : S]T}$ $\frac{\Gamma \vdash \langle s; t \rangle_T : \Sigma x : S. T \quad \Gamma \vdash \langle s; t \rangle_T : \Sigma x : S. T}{\Gamma \vdash \langle s; t \rangle_T \pi_0 \simeq s : S} \quad \frac{\Gamma \vdash \langle s; t \rangle_T : \Sigma x : S. T}{\Gamma \vdash \langle s; t \rangle_T \pi_1 \simeq t : [x \mapsto s : S]T}$ <p>observational rules</p> $\frac{\Gamma \vdash u : 1 \quad \Gamma \vdash u' : 1}{\Gamma \vdash u \simeq u' : 1} \quad \frac{\Gamma \vdash z : O \quad \Gamma \vdash z' : O}{\Gamma \vdash z \simeq z' : O}$ $\frac{\Gamma; x : S \vdash fx \simeq f'x : T \quad \Gamma \vdash p\pi_0 \simeq p'\pi_0 : S}{\Gamma \vdash f \simeq f' : \Pi x : S. T} \quad \frac{\Gamma \vdash p\pi_1 \simeq p'\pi_1 : [x \mapsto (p\pi_0) : S]T}{\Gamma \vdash p \simeq p' : \Sigma x : S. T}$
---

**FIGURE 3. Equality rules**  $\boxed{\Gamma \vdash t \simeq t' : T}$

*sem*  $(\_ : \Downarrow w) = w$

Here *WhnV* is the semantic representation of weak head normal forms:

**data** *WhnV* = *WR Reference* [*Elim Value*] -- *Spine*  
 | *WL ScopeV* --  $\lambda$ -abstraction  
 | *WP Value Value* -- *Pair*  
 | *WC Con* -- *Constant*  
 | *WD Quant Value ScopeV* -- *Dependent type*  
**deriving** *Show*

*ScopeV* is the real workhorse of this definition representing terms depending on a variable as a pair of a *Scope* and a Haskell function:

**type** *ScopeV* = *Scope* : $\Downarrow$  (*Value*  $\rightarrow$  *WhnV*)

We implement elimination for *Value*, incorporating all the  $\beta$ -rules, by

$$(\$ \$) :: \text{Value} \rightarrow \text{Elim Value} \rightarrow \text{Value}$$

Our *Checking* monad can deliver a fresh variable in a given type via this convenient higher-order function:

$$\text{forFreshIn} :: \text{Value} \rightarrow (\text{Value} \rightarrow \text{Checking } ()) \rightarrow \text{Checking } ()$$

Together, this equipment gives us an easy implementation of the type-directed observational rules.

$$\begin{aligned} \text{equate } ((f1, f2) : \in (\_ : \Downarrow \text{WD Pi dom ran})) \\ &= \text{forFreshIn dom } \$ \lambda x \rightarrow \text{equate } ((f1 \$ \$ x, f2 \$ \$ x) : \in \text{sem ran } x) \\ \text{equate } ((p1, p2) : \in (\_ : \Downarrow \text{WD Sig dom ran})) \\ &= \text{equate } ((p1 \$ \$ P0, p2 \$ \$ P0) : \in \text{dom}) \gg \\ &\quad \text{equate } ((p1 \$ \$ P1, p2 \$ \$ P1) : \in \text{sem ran } (p1 \$ \$ P0)) \\ \text{equate } (\_ : \in (\_ : \Downarrow \text{WC One})) &= \text{return } () \\ \text{equate } (\_ : \in (\_ : \Downarrow \text{WC Zero})) &= \text{return } () \end{aligned}$$

Comparison proceeds structurally at types without observational rules.

### 3 CONCLUSIONS AND FURTHER WORK

The main deliverable of our work is an independent type-checker for ETT which plays an important rôle in the overall architecture of the Epigram system. We have adressed a number of challenges in implementing a stronger conversion incorporating observational rules. This is important at present for our implementation of elaboration and it will play a vital rôle in future, for our project of implementing an Observational Type Theory, which is computationally well behaved but allows observational reasoning, see [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, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.
- [4] Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

- [5] L'Équipe Coq. The Coq Proof Assistant Reference Manual. <http://pauillac.inria.fr/coq/doc/main.html>, 2001.
- [6] Catarina Coquand and Thierry Coquand. Structured Type Theory. In *Workshop on Logical Frameworks and Metalanguages*, 1999.
- [7] Thierry Coquand. An analysis of Girard's paradox. In *Proceedings of the First IEEE Symposium on Logic in Computer Science, Cambridge, Massachusetts*, pages 227–236, 1986.
- [8] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.
- [9] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(1):75–94, 1997.
- [10] Nicolas G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicæ*, 34:381–392, 1972.
- [11] Robert Harper and Randy Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.
- [12] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.
- [13] Zhaohui Luo and Robert Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.
- [14] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis-Napoli, 1984.
- [15] 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.
- [16] Conor McBride, Healdene Goguen, and James McKinna. A Few Constructions on Constructors. In *Types for Proofs and Programs, Paris, 2004*, LNCS. Springer-Verlag, 2005. accepted; to appear.
- [17] 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.
- [18] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.
- [19] 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.