Normalization by evaluation for $\lambda^{\to 2}$

Thorsten Altenkirch[1] ⋆ and Tarmo Uustalu[2] ⋆⋆

[1] School of Computer Science and IT, University of Nottingham
Nottingham NG8 1BB, United Kingdom
txa@cs.nott.ac.uk
[2] Institute of Cybernetics, Tallinn Technical University
Akadeemia tee 21, EE-12618 Tallinn, Estonia
tarmo@cs.ioc.ee

Abstract. We show that the set-theoretic semantics for $\lambda^{\to 2}$ is complete by inverting evaluation using decision trees. This leads to an implementation of normalization by evaluation which is witnessed by the source of part of this paper being a literate Haskell script. We show the correctness of our implementation using logical relations.

1 Introduction

Which is the simplest typed λ-calculus without uninterpreted base types or type variables? We suggest that the answer should be $\lambda^{\to 2}$: simply typed lambda calculus extended by the type of booleans $\mathtt{Bool}$ with $\mathtt{True}, \mathtt{False} : \mathtt{Bool}$ and $\mathtt{If}\, t\, u_0\, u_1 : \sigma$, given $t : \mathtt{Bool}$ and $u_0, u_1 : \sigma$. The equational theory is given by the usual $\beta\eta$-equations of $\lambda^{\to}$ and the following equations concerning $\mathtt{Bool}$:

$$
\begin{aligned}
\mathtt{If}\,\mathtt{True}\, u_0\, u_1 &=_{\beta\eta} u_0 \\
\mathtt{If}\,\mathtt{False}\, u_0\, u_1 &=_{\beta\eta} u_1 \\
\mathtt{If}\, t\,\mathtt{True}\,\mathtt{False} &=_{\beta\eta} t \\
v\,(\mathtt{If}\, t\, u_0\, u_1) &=_{\beta\eta} \mathtt{If}\, t\,(v\, u_0)\,(v\, u_1)
\end{aligned}
$$

The equations are motivated by the categorical interpretation of $\mathtt{Bool}$ as a boolean object, i.e., an object Bool such that $\mathrm{Hom}(\Gamma \times \mathrm{Bool}, A) \simeq \mathrm{Hom}(\Gamma, A) \times \mathrm{Hom}(\Gamma, A)$ (naturally in Γ and A). The calculus can thus be interpreted in any cartesian closed category with Bool (using the cartesian structure to interpret contexts).

The equational theory introduces some interesting equalities. E.g., consider

$$
\begin{aligned}
\mathsf{once} &= \lambda f : \mathtt{Bool} \to \mathtt{Bool}.\, \lambda x : \mathtt{Bool}.\, f\, x \\
\mathsf{thrice} &= \lambda f : \mathtt{Bool} \to \mathtt{Bool}.\, \lambda x : \mathtt{Bool}. f\,(f\,(f\, x))
\end{aligned}
$$

⋆ partially supported by EU Framework 5 thematic networks TYPES and APPSEM II and the Estonian IT Foundation

⋆⋆ partially supported by the Estonian Science Foundation under grant No. 5567

We observe that $\mathsf{once} =_{\beta\eta} \mathsf{thrice}$. To see this, we note that, given $f : \mathtt{Bool} \to \mathtt{Bool}$, we have

$$\begin{aligned}
f\,(f\,(f\,\mathtt{True})) &=_{\beta\eta} \mathtt{If}\,(f\,\mathtt{True})\,(f\,(f\,\mathtt{True}))\,(f\,(f\,\mathtt{False}))\\
&=_{\beta\eta} \mathtt{If}\,(f\,\mathtt{True})\,\mathtt{True}\,(f\,(f\,\mathtt{False}))\\
&=_{\beta\eta} \mathtt{If}\,(f\,\mathtt{True})\,\mathtt{True}\,(\mathtt{If}\,(f\,\mathtt{False})\,(f\,\mathtt{True})\,(f\,\mathtt{False}))\\
&=_{\beta\eta} \mathtt{If}\,(f\,\mathtt{True})\,\mathtt{True}\,(\mathtt{If}\,(f\,\mathtt{False})\,\mathtt{False}\,\mathtt{False})\\
&=_{\beta\eta} \mathtt{If}\,(f\,\mathtt{True})\,\mathtt{True}\,\mathtt{False}\\
&=_{\beta\eta} f\,\mathtt{True}
\end{aligned}$$

Symmetrically, we can show that $f\,(f\,(f\,\mathtt{False})) =_{\beta\eta} f\,\mathtt{False}$, and hence

$$\begin{aligned}
&\mathsf{thrice}\\
&\quad= \lambda f : \mathtt{Bool} \to \mathtt{Bool}.\,\lambda x : \mathtt{Bool}.\,f\,(f\,(f\,x))\\
&=_{\beta\eta} \lambda f : \mathtt{Bool} \to \mathtt{Bool}.\,\lambda x : \mathtt{Bool}.\,\mathtt{If}\,x\,(f\,(f\,(f\,\mathtt{True})))\,(f\,(f\,(f\,\mathtt{False})))\\
&=_{\beta\eta} \lambda f : \mathtt{Bool} \to \mathtt{Bool}.\lambda x : \mathtt{Bool}.\,\mathtt{If}\,x\,(f\,\mathtt{True})\,(f\,\mathtt{False})\\
&=_{\beta\eta} \lambda f : \mathtt{Bool} \to \mathtt{Bool}.\,\lambda x : \mathtt{Bool}.f\,x\\
&\quad= \mathsf{once}
\end{aligned}$$

It is easy to see that once and thrice are equal in the standard semantics where $\mathtt{Bool}$ is interpreted by a two-element set $\mathrm{Bool} = \{\mathrm{true}, \mathrm{false}\}$ and function types are set-theoretic function spaces. We observe that there are only four elements in $\mathrm{Bool} \to \mathrm{Bool} = \{x \mapsto \mathrm{true}, x \mapsto x, x \mapsto \neg x, x \mapsto \mathrm{false}\}$ and that for all the four $f \in \mathrm{Bool} \to \mathrm{Bool}$ we have $f^3 = f$.

May we use set-theoretic reasoning to prove equalities up to $\beta\eta$-convertibility? The answer is yes for $\lambda^{\to 2}$, because for $\lambda^{\to 2}$ we can *invert* set-theoretic *evaluation* of typed closed terms. That is: we can define a function $\mathsf{quote}^\sigma \in [\![\sigma]\!]_{\mathsf{set}} \to \mathsf{Tm}\ \sigma$ such that $t =_{\beta\eta} \mathsf{quote}^\sigma\ [\![t]\!]_{\mathsf{set}}$, for any $t \in \mathsf{Tm}\ \sigma$. Consequently, we get that, for any $t, t' \in \mathsf{Tm}\ \sigma$, $t =_{\beta\eta} t' \iff [\![t]\!]_{\mathsf{set}} = [\![t']\!]_{\mathsf{set}}$.

The existence of quote also implies that $=_{\beta\eta}$ is maximally consistent, i.e., identifying any two non-$\beta\eta$-convertible closed terms would lead to an inconsistent theory. This provides another justification for the specific choice of $=_{\beta\eta}$.

We do not analyze the normal forms here, i.e. the codomain of nf and quote here. However, the construction presented here, which is based on decision trees, leads to simple normal forms and we conjecture that this is the same set as the set of normal forms presented in [1, 5] in the case $\mathtt{Bool} = 1 + 1$.

Haskell as a poor man's Type Theory

Our construction is entirely constructive, so it can be carried out, e.g., in Martin-Löf's Type Theory, and we obtain an implementation of normalization $\mathsf{nf}^\sigma\, t = \mathsf{quote}^\sigma\ [\![t]\!]_{\mathsf{set}}$. We shall here use the functional language Haskell as a poor man's Type Theory and obtain a Haskell program to normalize terms.

Haskell hasn't got dependent types (in particular inductive families), hence the Haskell types we are using are only approximations of their type-theoretic correspondents. E.g., the type-theory type $\mathsf{Tm}_\Gamma\ \sigma$ contains all welltyped terms of type σ in context Γ, but its Haskell counterpart `Tm` contains all untyped terms. Similarly, the set-theoretic denotation of a type $\sigma :\to \tau$ is given by $[\![\sigma :\to \tau]\!]_{\mathsf{set}} = [\![\sigma]\!]_{\mathsf{set}} \to [\![\tau]\!]_{\mathsf{set}}$ but its Haskell implementation is given by a recursive type `El` with a constructor `SLam` $\in \mathsf{Ty} \to (\mathtt{El} \to \mathtt{El}) \to \mathtt{El}$.

We believe that this informal use of Type Theory is an effective way to arrive at functional programs which are correct by construction. However, we hope that in the future we can go further and bridge the gap between informal type theoretic reasoning and the actual implementation by using a dependently typed programming language. such as the Epigram system, which is currently being developed by Conor McBride [12].

Related work

Inverting evaluation to achieve normalization by evaluation (NBE, aka. reduction-free normalization) was pioneered in [6] for simply typed lambda calculus with type variables and a non-standard semantics; a categorical account in terms of presheaves was given in [2]; this was extended to System F in [3, 4]; see [9] for a recent survey on NBE. The completeness of the set-theoretic model in the presence of coproducts has been shown in [8] and our case arises as a special case when there are no type variables. Normalization procedures for typed λ-calculus with coproducts can be found in [10, 11] using rewriting techniques and [1, 5] using NBE and sheaf theory. Both approaches allow type variables but do not handle the empty type. Here we present a much simpler construction for closed types using the simplest possible semantics of first-order simply typed λ-calculi—the set-theoretic one—and also provide a concrete implementation of quote and nf, whose correctness we show in detail.

2 Implementation of $\lambda^{\to 2}$

The source of Sections 2 and 3 of this paper is a literate Haskell script implementing normalization for $\lambda^{\to 2}$ and is available from

`http://www.cs.nott.ac.uk/~txa/publ/Nbe2.lhs`

We start by introducing types $\mathsf{Ty} \in \star$, variables $\mathsf{Var} \in \star$, typing contexts $\mathsf{Con} \in \star$ and untyped terms $\mathtt{Tm} \in \star$ of the object language by the following Haskell datatype definitions:

```
data Ty = Bool | Ty :-> Ty
        deriving (Show, Eq)

type Var = String

type Con = [ (Var, Ty) ]
```

```
data Tm = Var Var
        | TTrue | TFalse | If Tm Tm Tm
        | Lam Ty String Tm | App Tm Tm
        deriving (Show, Eq)
```

We view these recursive definitions as inductive definitions, i.e., we do not consider infinite terms. All the functions we define are total wrt. their precise type-theoretic types.

Implementing typed terms $\mathsf{Tm} \in \mathsf{Con} \to \mathsf{Ty} \to \star$ would take inductive families, which we cannot use in Haskell. But we can implement type inference $\mathsf{infer} \in \mathsf{Con} \to \mathtt{Tm} \to \mathsf{Maybe}\ \mathsf{Ty}$ (where $\mathsf{Maybe}\ X = 1 + X$ as usual):

```
infer :: Con -> Tm -> Maybe Ty
infer gamma (Var x) =
    do sigma <- lookup x gamma
       Just sigma
infer gamma TTrue  = Just Bool
infer gamma TFalse = Just Bool
infer gamma (If t u0 u1) =
    do Bool <- infer gamma t
       sigma0 <- infer gamma u0
       sigma1 <- infer gamma u1
       if sigma0 == sigma1 then Just sigma0 else Nothing
infer gamma (Lam sigma x t) =
    do tau <- infer ((x, sigma) : gamma) t
       Just (sigma :-> tau)
infer gamma (App t u) =
    do (sigma :-> tau) <- infer gamma t
       sigma' <- infer gamma u
       if sigma == sigma' then Just tau else Nothing
```

This implementation is correct in the sense that $t \in \mathsf{Tm}_\Gamma\ \sigma$ iff $\mathsf{infer}_\Gamma\ t = \mathsf{Just}\ \sigma$.

Evaluation of types $[\![-]\!] \in \mathsf{Ty} \to \star$ is again an inductive family, which we cannot implement in Haskell, and the workaround is to have all $[\![\sigma]\!]$ coalesced into one metalanguage type el (of untyped elements) much the same way as all $\mathsf{Tm}_\Gamma\ \sigma$ appear coalesced in `Tm`. We use a type class Sem to state what we require of such a coalesced type el:

```
class Sem el  where
    true  :: el
    false :: el
    xif :: el -> el -> el -> el
    lam :: Ty -> (el -> el) -> el
    app :: el -> el -> el
```

Evaluation of types $[\![-]\!] \in \mathsf{Ty} \to \star$ naturally induces evaluation of contexts $[\![-]\!] \in \mathsf{Con} \to \star$, defined by

$$\frac{}{[] \in [\![[]]\!]} \qquad \frac{\rho \in [\![\Gamma]\!] \quad d \in [\![\sigma]\!]}{(x, d) : \rho \in [\![(x, \sigma) : \Gamma]\!]}$$

We write $\mathsf{Tm}\ \Gamma = [\![\Gamma]\!]_{\mathsf{syn}}$ for the set of closed substitutions, which arises as in instance when using $[\![\sigma]\!]_{\mathsf{syn}} = \mathsf{Tm}_{[]}\sigma$ as the interpretation of types.

In the Haskell code we approximate evaluation of contexts by `Env`:

```
type Env el = [ (Var, el) ]
```

Given $t \in \mathsf{Tm}_\Gamma\,\sigma$ we define the evaluation of terms $[\![t]\!] \in [\![\Gamma]\!] \to [\![\sigma]\!]$. In Haskell this is implemented as eval:

```
eval :: Sem el => Env el -> Tm -> el
eval rho (Var x) = d
    where (Just d) = lookup x rho
eval rho TTrue = true
eval rho TFalse =  false
eval rho (If t u0 u1) =
    xif (eval rho t) (eval rho u0) (eval rho u1)
eval rho (Lam sigma x t) =
    lam sigma (\ d -> eval ((x, d) : rho) t)
eval rho (App t u) = app (eval rho t) (eval rho u)
```

The standard set-theoretic semantics is given by

$$\begin{aligned} [\![\mathtt{Bool}]\!]_{\mathsf{set}} &= \mathtt{Bool} \\ [\![\sigma :\to \tau]\!]_{\mathsf{set}} &= [\![\sigma]\!]_{\mathsf{set}} \to [\![\tau]\!]_{\mathsf{set}} \end{aligned}$$

This can be represented in Haskell as an instance of Sem:

```
data El = STrue | SFalse | SLam Ty (El -> El)

instance Sem El where
    true  = STrue
    false = SFalse
    xif STrue  d _ = d
    xif SFalse _ d = d
    lam = SLam
    app (SLam _ f) d = f d
```

Since sets form a cartesian closed category with a boolean object, the set-theoretic semantics validates all $\beta\eta$-equalities. This is to say that $[\![-]\!]_{\mathsf{set}}$ is equationally sound:

Proposition 2.1 (Soundness). *If $\rho \in [\![\Gamma]\!]$ and $t =_{\beta\eta} t' \in \mathsf{Tm}_\Gamma\ \sigma$, then $[\![t]\!]_{\mathsf{set}}\ \rho = [\![t']\!]_{\mathsf{set}}\ \rho$.*

Since all the sets we consider are finite, semantic equality can be implemented in Haskell, by making use of the function $\mathsf{enum} \in (\sigma \in \mathsf{Ty}) \to \mathsf{Tree}\ [\![\sigma]\!]$, which we will provide later:

```
instance Eq El where
    STrue  == STrue  = True
    SFalse == SFalse = True
    (SLam sigma f) == (SLam _ f') =
        and [f d == f' d | d <- flatten (enum sigma)]
    _ == _ = False
```

Using on the same function we can also print elements of `El`:

```
instance Show El where
    show STrue  = "STrue"
    show SFalse = "SFalse"
    show (SLam sigma f) =
        "SLam " ++ (show sigma) ++ " " ++
        (show [ (d, f d) | d <- flatten (enum sigma) ])
```

The equational theory of the calculus itself gives rise to another semantics—the free semantics, or typed terms up to $\beta\eta$-convertibility. This can be approximated by the following Haskell code, which uses a redundancy-avoiding version `if'` of `If` which produces a shorter but $\beta\eta$-equivalent term:

```
if' :: Tm -> Tm -> Tm -> Tm
if' t TTrue TFalse = t
if' t u0 u1 = if u0 == u1 then u0 else If t u0 u1

instance Sem Tm where
    true  = TTrue
    false = TFalse
    xif = if'
    lam sigma f = Lam sigma "x" (f (Var "x"))
    app = App
```

We also observe that the use of a fixed variable is only justified by the fact that our algorithm uses at most one bound variable at the time. A correct dependently typed version of the free semantics requires the use of presheaves to ensure that the argument to `Lam` is stable under renaming. We refrain from presenting the details here. It is well known that this semantics is equationally sound.

3 Implementation of quote

We now proceed to implementing $\mathsf{quote} \in (\sigma \in \mathsf{Ty}) \to [\![\sigma]\!]_{\mathsf{set}} \to \mathsf{Tm}\ \sigma$.

To define $\mathsf{quote}^{\sigma\to\tau}$ we use enum^{σ}, which generates a decision tree whose leaves are all the elements of $[\![\sigma]\!]$, and $\mathsf{questions}^{\sigma}$, which generates the list of questions, i.e. elements of $[\![\sigma]\!] \to [\![\mathtt{Bool}]\!]$, based on answers to whom an element of $[\![\sigma]\!]$ can be looked up in the tree enum^{σ}. (Since our decision trees are perfectly balanced and we use the same list questions along each branch of a tree, we separate the questions labelling from the tree.)

Decision trees $\mathsf{Tree} \in \mathsf{Ty} \to \star$ are provided by

```
data Tree a = Val a | Choice (Tree a) (Tree a) deriving (Show, Eq)
```

We will exploit the fact that Tree is a monad

```
instance Monad Tree where
    return = Val
    (Val a) >>= h = h a
    (Choice l r) >>= h = Choice (l >>= h) (r >>= h)
```

(`return` and `>>=` are Haskell for the unit resp. the bind or Kleisli extension operation of a monad) and hence a functor

```
instance Functor Tree where
    fmap h ds = ds >>= return . h
```

(`fmap` is Haskell for the action of a functor on morphisms).

It is convenient to use the function flatten which calculates the list of leaves of a given tree:

```
flatten :: Tree a -> [ a ]
flatten (Val a) = [ a ]
flatten (Choice l r) = (flatten l) ++ (flatten r)
```

We implement enum^{σ} and $\mathsf{questions}^{\sigma}$ by mutual induction on $\sigma \in \mathsf{Ty}$. The precise typings of the functions are $\mathsf{enum} \in (\sigma \in \mathsf{Ty}) \to \mathsf{Tree}\ [\![\sigma]\!]$ and $\mathsf{questions} \in (\sigma \in \mathsf{Ty}) \to [[\![\sigma]\!] \to [\![\mathtt{Bool}]\!]]$. As usual, Haskell cannot express those subtleties due to its lack of dependent types, but we can declare

```
enum :: Sem el => Ty -> Tree el
questions :: Sem el => Ty -> [ el -> el ]
```

The base case is straightforward: A boolean is true or false and to know which one it is it suffices to know it.

```
enum Bool = Choice (Val true) (Val false)

questions Bool = [ \ b -> b ]
```

The implementation of $\mathsf{enum}^{\sigma:\to\tau}$ and $\mathsf{questions}^{\sigma:\to\tau}$ proceeds from the idea that a function is determined by its graph: to know a function it suffices to know its value on all possible argument values. The main idea in the implementation of $\mathsf{enum}^{\sigma:\to\tau}$ is therefore to start with enum^{τ} and to duplicate the tree for each question in $\mathsf{questions}^{\sigma}$ using the bind of Tree:

```
enum (sigma :-> tau) =
    fmap (lam sigma) (mkEnum (questions sigma) (enum tau))

mkEnum :: Sem el => [ el -> el ] -> Tree el -> Tree (el -> el)
mkEnum [] es = fmap (\ e -> \ d -> e) es
mkEnum (q : qs) es = (mkEnum qs es) >>= \ f1 ->
                     (mkEnum qs es) >>= \ f2 ->
                     return (\ d -> xif (q d) (f1 d) (f2 d))
```

$\mathsf{questions}^{\sigma:\to\tau}$ produces the appropriate questions by enumerating σ and using questions from τ:

```
questions (sigma :-> tau) =
    [ \ f -> q (app f d) | d <- flatten (enum sigma),
                           q <- questions tau ]
```

As an example, the enumeration and questions for `Bool` $:\to$ `Bool` return:

```
Choice
   (Choice
       (Val (lam Bool (\ d -> xif d true  true)))
       (Val (lam Bool (\ d -> xif d true  false))))
   (Choice
       (Val (lam Bool (\ d -> xif d false true )))
       (Val (lam Bool (\ d -> xif d false false))))
```

resp.

```
(\ f -> app f true :
   (\ f -> app f false :
       []))
```

We can look up an element in the decision tree for a type by answering all the questions, this is realized by the function find below. To define the domain of find precisely we define a relation between lists of questions and trees of answers $\diamond \subseteq [a] \times \mathsf{Tree}\ b$ inductively:

$$\frac{}{[] \diamond (\mathsf{Val}\ t)} \qquad \frac{as \diamond l \quad as \diamond r}{a : as \diamond \mathsf{Choice}\ l\ r}$$

Now given $as \in [[\![\texttt{Bool}]\!]]$, $ts \in \mathsf{Tree}\,[\![\sigma]\!]$, s.t. $as \diamond ts$ we obtain $\mathsf{find}\ as\ ts \in [\![\sigma]\!]$, implemented in Haskell:

```
find :: Sem el => [ el ] -> Tree el -> el
find [] (Val t) = t
find (a : as) (Choice l r) = xif a (find as l) (find as r)
```

We are now ready to implement $\mathsf{quote}^\sigma \in [\![\sigma]\!]_{\mathsf{set}} \to \mathsf{Tm}\ \sigma$, with Haskell typing

```
quote :: Ty -> El -> Tm
```

by induction on $\sigma \in \mathsf{Ty}$. As usual, the base case is easy:

```
quote Bool STrue  = TTrue
quote Bool SFalse = TFalse
```

$\mathsf{quote}^{\sigma:\to\tau}$ is more interesting: Our strategy is to map $\mathsf{quote}^\tau \circ f$ to the set-theoretic enum^τ and to then build a tree of `If` expressions by using the syntactic $\mathsf{questions}^\sigma$ in conjunction with the syntactic find:

```
quote (sigma :-> tau) (SLam _ f) =
    lam sigma (\ t -> find [ q t | q <- questions sigma ]
                           (fmap (quote tau . f) (enum sigma)))
```

(Notice that in Haskell it is inferred automatically which semantics is meant where.)

As already discussed in the introduction, we implement normalization $\mathsf{nf} \in (\sigma \in \mathsf{Ty}) \to \mathsf{Tm}\ \sigma \to \mathsf{Tm}\ \sigma$ by

```
nf :: Ty -> Tm -> Tm
nf sigma t = quote sigma (eval [] t)
```

Since we can infer types, we can implement $\mathsf{nf}' \in \mathtt{Tm} \to \mathsf{Maybe}\ (\Sigma_{\sigma\in\mathsf{Ty}}\mathsf{Tm}\ \sigma)$:

```
nf' :: Tm -> Maybe (Ty, Tm)
nf' t = do sigma <- infer [] t
           Just (sigma, nf sigma t)
```

We test our implementation with the example from the introduction:

```
b2b = Bool :-> Bool
once   = Lam b2b "f" (Lam Bool "x" (App (Var "f") (Var "x")))
twice  = Lam b2b "f" (Lam Bool "x" (App (Var "f")
                                     (App (Var "f") (Var "x"))))
thrice = Lam b2b "f"
            (Lam Bool "x" (App (Var "f")
                               (App (Var "f")
                                    (App (Var "f") (Var "x")))))
```

and convince ourselves that $(\mathsf{nf}'\ \mathsf{once} == \mathsf{nf}'\ \mathsf{thrice}) = \mathsf{true}$ but $(\mathsf{nf}'\ \mathsf{once} == \mathsf{nf}'\ \mathsf{twice}) = \mathsf{false}$. Since semantic equality is decidable we do not actually have to construct the normal forms to decide convertibility.

Since testing can only reveal the presence of errors we shall use the rest of this paper to prove that quote and hence nf behave correctly.

4 Correctness of quote

The main tool in our proof will be a notion of logical relations, a standard tool for the characterization of definable elements in models of typed lambda calculi since the pioneering work of Plotkin [13].

Let us agree to abbreviate $\mathsf{Tm}_{[]}\ \sigma$ by $\mathsf{Tm}\ \sigma$ and $[\![t]\!]_{\mathsf{set}}[]$ by $[\![t]\!]_{\mathsf{set}}$.

Definition 4.1 (Logical Relations). *We define a family of relations $\mathsf{R}^\sigma \subseteq \mathsf{Tm}\ \sigma \times [\![\sigma]\!]_{\mathsf{set}}$ by induction on $\sigma \in \mathsf{Ty}$ as follows:*

- $t\mathsf{R}^{\mathsf{Bool}}b$ *iff* $t =_{\beta\eta} \mathtt{True}$ *and* $b = \mathrm{true}$ *or* $t =_{\beta\eta} \mathtt{False}$ *and* $b = \mathrm{false}$*;*
- $t\mathsf{R}^{\sigma:\to\tau}f$ *iff, for all* u, d*,* $u\mathsf{R}^\sigma d$ *implies* $\mathtt{App}\ t\ u\mathsf{R}^\tau f\ d$*.*

Note that R is not indexed by contexts, logical relations only relate closed terms.

We extend logical relations to contexts: Given $\Gamma \in \mathsf{Con}$ we define $\mathsf{R}^{\Gamma} \subseteq \mathsf{Tm}\Gamma \times [\![\Gamma]\!]$ by:

$$\frac{}{[]\mathsf{R}^{[]}[]} \qquad \frac{\rho \mathsf{R}^{\Gamma} \rho' \quad d \mathsf{R}^{\sigma} d'}{(x,d) : \rho \ \mathsf{R}^{(x,\sigma):\Gamma} \ (x,d') : \rho'}$$

Logical relations are invariant under $\beta\eta$-equality.

Lemma 4.2. *If* $t \mathsf{R}^{\sigma} d$ *and* $t =_{\beta\eta} t'$, *then* $t' \mathsf{R}^{\sigma} d$.

Logical relations obey the following Fundamental Theorem, a kind of soundness theorem for logical relations.

Lemma 4.3 (Fundamental Theorem of Logical Relations). *If* $\theta \mathsf{R}^{\Gamma} \rho$ *and* $t \in \mathsf{Tm}_{\Gamma}\ \sigma$, *then* $[t]\ \theta \mathsf{R}^{\sigma} [\![t]\!]_{\mathsf{set}}\ \rho$. *In particular, if* $t \in \mathsf{Tm}\ \sigma$, *then* $t \mathsf{R}^{\sigma} [\![t]\!]_{\mathsf{set}}$.

The main result required to see that quote is correct is the following lemma:

Lemma 4.4 (Main Lemma). *If* $t \mathsf{R}^{\sigma} d$, *then* $t =_{\beta\eta} \mathsf{quote}^{\sigma}\ d$.

The proof of this lemma is the subject of the next section.

By correctness of quote we mean that it inverts set-theoretic evaluation of typed closed terms.

Theorem 4.5 (Main Theorem). *If* $t \in \mathsf{Tm}\ \sigma$, *then* $t =_{\beta\eta} \mathsf{quote}^{\sigma}\ [\![t]\!]_{\mathsf{set}}$.

Proof. Immediate from the Fundamental Theorem and the Main Lemma. □

The (constructive) existence and correctness of quote has a number of straightforward important consequences.

Corollary 4.6 (Completeness). *If* $t, t' \in \mathsf{Tm}\ \sigma$, *then* $[\![t]\!]_{\mathsf{set}} = [\![t']\!]_{\mathsf{set}}$ *implies* $t =_{\beta\eta} t'$.

Proof. Immediate from the Main Theorem. □

From soundness (Proposition 2.1) and completeness together we get that $=_{\beta\eta}$ is decidable: checking whether $t =_{\beta\eta} t'$ reduces to checking whether $[\![t]\!]_{\mathsf{set}} = [\![t']\!]_{\mathsf{set}}$, which is decidable as $[\![-]\!]_{\mathsf{set}}$ is computable and equality in finite sets is decidable.

Corollary 4.7. *If* $t, t' \in \mathsf{Tm}\ \sigma$, *then* $t =_{\beta\eta} t'$ *iff* $\mathsf{quote}^{\sigma}\ [\![t]\!]_{\mathsf{set}} = \mathsf{quote}^{\sigma}\ [\![t']\!]_{\mathsf{set}}$.

Proof. Immediate from soundness (Proposition 2.1) and the Main Theorem. □

This corollary shows that $\mathsf{nf}^{\sigma} = \mathsf{quote}^{\sigma} \circ [\![-]\!]_{\mathsf{set}} : \mathsf{Tm}\ \sigma \to \mathsf{Tm}\ \sigma$ indeed makes sense as normalization function: apart from just delivering, for any given typed closed term, some $\beta\eta$-equal term, it is actually guaranteed to deliver the same term for t, t', if t, t' are $\beta\eta$-equal (morally, this is Church-Rosser for reduction-free normalization).

Note that although we only stated completeness and normalization for typed closed terms above, these trivially extend to all typed terms as opens terms can always be closed up by lambda-abstractions and this preserves $\beta\eta$-equality.

Corollary 4.8. *If* $t, t' \in \mathsf{Tm}\ \sigma$ *and* $[C]\ [(\mathtt{x}, t)] =_{\beta\eta} [C]\ [(\mathtt{x}, t')]$ *for every* $C : \mathsf{Tm}_{[(\mathtt{x},\sigma)]}\ \mathtt{Bool}$*, then* $t =_{\beta\eta} t'$*. Or, contrapositively, and more concretely, if* $t, t' \in \mathsf{Tm}\ (\sigma_1 :\rightarrow \ldots :\rightarrow \sigma_n :\rightarrow \mathtt{Bool})$ *and* $t \neq_{\beta\eta} t'$*, then there exist* $u_1 \in \mathsf{Tm}\ \sigma_1$*,* $\ldots u_n \in \mathsf{Tm}\ \sigma_n$ *such that*

$$\mathsf{nf}^{\mathtt{Bool}}\ (\mathtt{App}\ (\ldots(\mathtt{App}\ t\ u_1)\ \ldots)\ u_n) \neq \mathsf{nf}^{\mathtt{Bool}}\ (\mathtt{App}\ (\ldots\ (\mathtt{App}\ t'\ u_1)\ \ldots)\ u_n)$$

Proof. This corollary does not follow from the statement of the Main Theorem, but it follows from its proof. □

Corollary 4.9 (Maximal consistency). *If* $t, t' \in \mathsf{Tm}\ \sigma$ *and* $t \neq_{\beta\eta} t'$*, then from the equation* $t = t'$ *as an additional axiom one would derive* $\mathtt{True} = \mathtt{False}$*.*

Proof. Immediate from the previous corollary. □

5 Proof of the main lemma

We now present the proof of the main lemma which was postponed in the previous section. To keep the proof readable, we write $\mathsf{enum}_{\mathsf{set}}$, $\mathsf{questions}_{\mathsf{set}}$, $\mathsf{find}_{\mathsf{set}}$ to emphasize the uses of the set-theoretic semantics instances of enum, $\mathsf{questions}$, find, while the free semantics instances will be written as $\mathsf{enum}_{\mathsf{syn}}$, $\mathsf{questions}_{\mathsf{syn}}$, $\mathsf{find}_{\mathsf{syn}}$. We use the fact that any functor F such as Tree has an effect on relations $R \subseteq A \times B$ denoted by $FR \subseteq FA \times FB$, which can be defined as:

$$\frac{z \in F\{(a, b) \in A \times B \mid aRb\}}{\mathsf{fmap}\ \mathsf{fst}\ z\ \ FR\ \ \mathsf{fmap}\ \mathsf{snd}\ z}$$

We first give the core of the proof and prove the lemmas this takes afterwards.

Proof (of the Main Lemma). By induction on σ.

- Case $\mathtt{Bool}$: Assume $t\mathsf{R}^{\mathtt{Bool}}b$. Then either $t =_{\beta\eta} \mathtt{True}$ and $b = \mathrm{true}$, in which case we have

  $$t =_{\beta\eta} \mathtt{True} = \mathsf{quote}^{\mathtt{Bool}}\ \mathrm{true} = \mathsf{quote}^{\mathtt{Bool}}\ b$$

  or $t =_{\beta\eta} \mathtt{False}$ and $b = \mathrm{false}$, in which case we have

  $$t =_{\beta\eta} \mathtt{False} = \mathsf{quote}^{\mathtt{Bool}}\ \mathrm{false} = \mathsf{quote}^{\mathtt{Bool}}\ b$$

- Case $\sigma :\rightarrow \tau$: Assume $t\mathsf{R}^{\sigma:\rightarrow\tau}f$, for all u, d, $u\mathsf{R}^{\sigma}d$ implies $\mathtt{App}\ t\ u\mathsf{R}^{\tau}f\ d$. We have

  $$\begin{aligned}
  t &=_{\beta\eta} \mathtt{Lam}^{\sigma}\ \mathtt{x}\ (\mathtt{App}\ t\ (\mathsf{Var}\ \mathtt{x}))\\
  &=_{\beta\eta} \text{(by Lemma 5.2 below)}\\
  &\qquad \mathtt{Lam}^{\sigma}\ \mathtt{x}\ (\mathtt{App}\ t\ (\mathsf{find}_{\mathsf{syn}}\ [q\ (\mathsf{Var}\ \mathtt{x}) \mid q \leftarrow \mathsf{questions}^{\sigma}_{\mathsf{syn}}]\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}))\\
  &= \mathtt{Lam}^{\sigma}\ \mathtt{x}\ (\mathsf{find}_{\mathsf{syn}}\ [q\ (\mathsf{Var}\ \mathtt{x}) \mid q \leftarrow \mathsf{questions}^{\sigma}_{\mathsf{syn}}]\\
  &\qquad\qquad (\mathsf{fmap}\ (\mathtt{App}\ t)\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}))\\
  &=_{\beta\eta} \text{(by Sublemma)}\\
  &\qquad \mathtt{Lam}^{\sigma}\ \mathtt{x}\ (\mathsf{find}_{\mathsf{syn}}\ [q\ (\mathsf{Var}\ \mathtt{x}) \mid q \leftarrow \mathsf{questions}^{\sigma}_{\mathsf{syn}}]\\
  &\qquad\qquad (\mathsf{fmap}\ (\mathsf{quote}^{\tau} \circ f)\ \mathsf{enum}^{\sigma}_{\mathsf{set}}))\\
  &= \mathsf{quote}^{\sigma:\rightarrow\tau}\ f
  \end{aligned}$$

The Sublemma is:

$$\mathsf{fmap}\ (\mathtt{App}\ t)\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}\ (\mathsf{Tree}\ =_{\beta\eta})\ \mathsf{fmap}\ (\mathsf{quote}^{\tau} \circ f)\ \mathsf{enum}^{\sigma}_{\mathsf{set}}$$

For proof, we notice that, by Lemma 5.1 (1) below for σ,

$$\mathsf{enum}^{\sigma}_{\mathsf{syn}}\ (\mathsf{Tree}\ \mathsf{R}^{\sigma})\ \mathsf{enum}^{\sigma}_{\mathsf{set}}$$

Hence, by assumption and the fact that fmap commutes with the effect on relations

$$\mathsf{fmap}\ (\mathtt{App}\ t)\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}\ (\mathsf{Tree}\ \mathsf{R}^{\tau})\ \mathsf{fmap}\ f\ \mathsf{enum}^{\sigma}_{\mathsf{set}}$$

Hence, by IH of the Lemma for τ,

$$\mathsf{fmap}\ (\mathtt{App}\ t)\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}\ (\mathsf{Tree}\ =_{\beta\eta})\ \mathsf{fmap}\ (\mathsf{quote}^{\tau} \circ f)\ \mathsf{enum}^{\sigma}_{\mathsf{set}}$$

□

The proof above used two lemmas. One is essentially free, but the other is technical.

Lemma 5.1 ("Free" Lemma).

1. $\mathsf{enum}^{\sigma}_{\mathsf{syn}}\ (\mathsf{Tree}\ \mathsf{R}^{\sigma})\ \mathsf{enum}^{\sigma}_{\mathsf{set}}$.
2. $\mathsf{questions}^{\sigma}_{\mathsf{syn}}\ [\mathsf{R}^{\sigma}\ \to \mathsf{R}^{\mathtt{Bool}}]\ \mathsf{questions}^{\sigma}_{\mathsf{set}}$.

Proof. The proof is simultaneous for (1) and (2) by induction on σ.

- Case `Bool`: Trivial.
- Case $\sigma :\to \tau$: Proof of (1) uses IH (2) for σ and IH (1) for τ; proof of (2) uses IH (1) for σ and IH (2) for τ.

□

Lemma 5.2 (Technical Lemma). *For* $t \in \mathsf{Tm}_{\Gamma}\ \sigma$*:*

$$t =_{\beta\eta} \mathsf{find}_{\mathsf{syn}}\ [q\ t \mid q \leftarrow \mathsf{questions}^{\sigma}_{\mathsf{syn}}]\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}$$

Proof. By induction on σ.

- Case `Bool`:

$$\begin{aligned} t &= \mathtt{If'}\ t\ \mathtt{True}\ \mathtt{False} \\ &= \mathtt{If'}\ t\ (\mathsf{find}_{\mathsf{syn}}\ []\ (\mathsf{Val}\ \mathtt{True}))\ (\mathsf{find}_{\mathsf{syn}}\ []\ (\mathsf{Val}\ \mathtt{False})) \\ &= \mathsf{find}_{\mathsf{syn}}\ [t]\ (\mathsf{Choice}\ (\mathsf{Val}\ \mathtt{True})\ (\mathsf{Val}\ \mathtt{False})) \\ &= \mathsf{find}_{\mathsf{syn}}\ [q\ t \mid q \leftarrow \mathsf{questions}^{\mathtt{Bool}}_{\mathsf{syn}}]\ \mathsf{enum}^{\mathtt{Bool}}_{\mathsf{syn}} \end{aligned}$$

– Case $\sigma :\to \tau$:

$$
\begin{aligned}
t &=_{\beta\eta} \texttt{Lam}^{\sigma}\ z\ (\texttt{App}\ t\ (\mathsf{Var}\ z)) \qquad (z \text{ fresh wrt } \Gamma)\\
&=_{\beta\eta} \text{(by IH for } \sigma)\\
&\quad \texttt{Lam}^{\sigma}\ z\ (\mathsf{find}_{\mathsf{syn}}\ [q(\mathsf{Var}\ z) \mid q \leftarrow \mathsf{questions}^{\sigma}_{\mathsf{syn}}]\ \mathsf{enum}^{\sigma}_{\mathsf{syn}})\\
&=_{\beta\eta} \text{(by Sublemma below)}\\
&\quad \texttt{Lam}^{\sigma}\ z\\
&\qquad (\mathsf{find}_{\mathsf{syn}}\ [q\ (\texttt{App}\ t\ u) \mid u \leftarrow \mathsf{flatten}\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}, q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ (\mathsf{Var}\ z))\ (\mathsf{mkenum}_{\mathsf{syn}}\ \mathsf{questions}^{\sigma}_{\mathsf{syn}}\ \mathsf{enum}^{\tau}_{\mathsf{syn}})))\\
&=_{\beta\eta} \text{by Lemma 5.3 and functor laws}\\
&\quad \mathsf{find}_{\mathsf{syn}}\ [q\ (\texttt{App}\ t\ u) \mid u \leftarrow \mathsf{flatten}\ \mathsf{enum}^{\sigma}_{\mathsf{syn}}, q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\mathsf{lam}\ \sigma)\ (\mathsf{mkenum}_{\mathsf{syn}}\ \mathsf{questions}^{\sigma}_{\mathsf{syn}}\ \mathsf{enum}^{\tau}_{\mathsf{syn}})))\\
&= \mathsf{find}_{\mathsf{syn}}\ [q\ t \mid q \leftarrow \mathsf{questions}^{\sigma:\to\tau}_{\mathsf{syn}}]\ \mathsf{enum}^{\sigma:\to\tau}_{\mathsf{syn}}
\end{aligned}
$$

The sublemma is: Given $qs \diamond us$ then

$$
\begin{aligned}
&\texttt{App}\ t\ (\mathsf{find}_{\mathsf{syn}}\ [q\ u \mid q \leftarrow qs]\ us)\\
&=_{\beta\eta} \mathsf{find}_{\mathsf{syn}}\ [q\ (\texttt{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ us, q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad\quad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\tau}_{\mathsf{syn}}))
\end{aligned}
$$

The proof is by induction on $qs \diamond us$.

- Case $[] \diamond \mathsf{Val}\ u^\star$:
  Assume $u =_{\beta\eta} \mathsf{find}_{\mathsf{syn}}\ []\ (\mathsf{Val}\ u^\star)$, i.e., $u =_{\beta\eta} u^\star$. We get

$$
\begin{aligned}
&\texttt{App}\ t\ (\mathsf{find}_{\mathsf{syn}}\ [q\ u \mid q \leftarrow qs]\ us)\\
&=_{\beta\eta} \text{(by IH of the Lemma for } \tau)\\
&\quad \mathsf{find}_{\mathsf{syn}}\ [q\ (\texttt{App}\ t\ u^\star) \mid q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\ \mathsf{enum}^{\tau}_{\mathsf{syn}}\\
&= \mathsf{find}_{\mathsf{syn}}\ [q\ (\texttt{App}\ t\ u^\star) \mid q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{fmap}\ (\lambda v \lambda u'\ v)\ \mathsf{enum}^{\tau}_{\mathsf{syn}}))\\
&= \mathsf{find}_{\mathsf{syn}}\ [q\ (\texttt{App}\ t\ u') \mid u' \leftarrow [u^\star], q \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ []\ \mathsf{enum}^{\tau}_{\mathsf{syn}}))
\end{aligned}
$$

- Case $q : qs \diamond \mathsf{Choice}\ l\ r$:

$$
\begin{aligned}
&\texttt{App}\ t\ (\mathsf{find}_{\mathsf{syn}}\ [q'\ u \mid q' \leftarrow q : qs]\ (\mathsf{Choice}\ l\ r))\\
&= \texttt{App}\ t\ (\mathrm{if}'\ (q\ u)\ (\mathsf{find}_{\mathsf{syn}}\ [q'\ u \mid q' \leftarrow qs]\ l)\\
&\qquad\qquad\qquad\qquad (\mathsf{find}_{\mathsf{syn}}\ [q'\ u \mid q' \leftarrow qs]\ r)\\
&=_{\beta\eta} \mathrm{if}'\ (q\ u)\ (\texttt{App}\ t\ (\mathsf{find}_{\mathsf{syn}}\ [q'\ u \mid q' \leftarrow qs]\ l))\\
&\qquad\qquad\quad (\texttt{App}\ t\ (\mathsf{find}_{\mathsf{syn}}\ [q'\ u \mid q' \leftarrow qs]\ r))\\
&=_{\beta\eta} \text{(by IH of the Sublemma for } qs \diamond l,\ qs \diamond r)\\
&\quad \mathrm{if}'\ (q\ u)\\
&\qquad (\mathsf{find}_{\mathsf{syn}}\ [q'\ (\texttt{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ l, q' \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad\quad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\tau}_{\mathsf{syn}})))
\end{aligned}
$$

$$
\begin{aligned}
&\qquad (\mathsf{find}_{\mathsf{syn}}\ [q'\ (\mathtt{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ r, q' \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad\quad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\tau}_{\mathsf{syn}})))\\
=\ & \mathsf{find}_{\mathsf{syn}}\ []\ (\mathsf{Val}\ (\mathtt{If}'\ (q\ u)\\
&\qquad (\mathsf{find}_{\mathsf{syn}}\ [q'\ (\mathtt{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ l, q' \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad\quad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\tau}_{\mathsf{syn}})))\\
&\qquad (\mathsf{find}_{\mathsf{syn}}\ [q'\ (\mathtt{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ r, q' \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad\quad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\tau}_{\mathsf{syn}})))))\\
=_{\beta\eta}\ & (\text{by twice Lemma 5.4})\\
& \mathsf{find}_{\mathsf{syn}}\ ([q'\ (\mathtt{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ l, q' \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad\quad +\!\!+\ ([q'\ (\mathtt{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ r, q' \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad\quad +\!\!+\ []))\\
&\qquad ((\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\tau}_{\mathsf{syn}})) \gg\!= \lambda v_0\\
&\qquad\quad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\tau}_{\mathsf{syn}})) \gg\!= \lambda v_1\\
&\qquad\quad \mathsf{Val}\ (\mathtt{If}'\ (q\ u)\ v_0\ v_1))\\
=\ & \mathsf{find}_{\mathsf{syn}}\ ([q'\ (\mathtt{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ l, q' \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad\quad +\!\!+\ [q'\ (\mathtt{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ r, q' \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}])\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ u)\\
&\qquad\quad ((\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\tau}_{\mathsf{syn}}) \gg\!= \lambda g_0\\
&\qquad\qquad (\mathsf{mkenum}_{\mathsf{syn}}\ qs\ \mathsf{enum}^{\tau}_{\mathsf{syn}} \gg\!= \lambda g_1\\
&\qquad\qquad \mathsf{Val}\ (\lambda u'\ \mathtt{If}'\ (q\ u')\ (g_0\ u')\ (g_1\ u')))))\\
=\ & \mathsf{find}_{\mathsf{syn}}\ [q'\ (\mathtt{App}\ t\ u') \mid t \leftarrow (\mathsf{flatten}\ l) +\!\!+ (\mathsf{flatten}\ r),\\
&\qquad\qquad\qquad\qquad\qquad\quad q' \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ (q : qs)\ \mathsf{enum}^{\tau}_{\mathsf{syn}}))\\
=\ & \mathsf{find}_{\mathsf{syn}}\ [q'\ (\mathtt{App}\ t\ u') \mid u' \leftarrow \mathsf{flatten}\ (\mathsf{Choice}\ l\ r),\\
&\qquad\qquad\qquad\qquad\qquad\quad q' \leftarrow \mathsf{questions}^{\tau}_{\mathsf{syn}}]\\
&\qquad (\mathsf{fmap}\ (\lambda g\ g\ u)\ (\mathsf{mkenum}_{\mathsf{syn}}\ (q : qs)\ \mathsf{enum}^{\tau}_{\mathsf{syn}}))
\end{aligned}
$$

□

We have used two lemmas, which are easy to prove:

Lemma 5.3. *Given $as \diamond us$ it is true that*

$$\mathsf{find}_{\mathsf{syn}}\ as\ (\mathsf{fmap}\ f\ us) =_{\beta\eta} f\ (\mathsf{find}_{\mathsf{syn}}\ as\ us)$$

Proof. Simple induction on $as \diamond us$.

Lemma 5.4. *Given $as \diamond ts$ and, for all $u \in \mathsf{Tm}_{\Gamma}\ \sigma$, $bs \diamond h\ u$, then*

$$\mathsf{find}_{\mathsf{syn}}\ (as +\!\!+ bs)\ (ts \gg\!= h) =_{\beta\eta} \mathsf{find}_{\mathsf{syn}}\ bs\ (h\ (\mathsf{find}_{\mathsf{syn}}\ as\ ts))$$

Proof. By induction on $as \diamond ts$.

– Case $[] \diamond \mathsf{Val}\ t$:

$$\begin{aligned}
&\mathsf{find}_{\mathsf{syn}}\ ([] +\!\!+ bs)\ ((\mathsf{Val}\ t) \ggg\!= h)\\
&= \mathsf{find}_{\mathsf{syn}}\ bs\ ((\mathsf{Val}\ t) \ggg\!= h)\\
&= \mathsf{find}_{\mathsf{syn}}\ bs\ (h\ t)\\
&= \mathsf{find}_{\mathsf{syn}}\ bs\ (h\ (\mathsf{find}_{\mathsf{syn}}\ []\ (\mathsf{Val}\ t))
\end{aligned}$$

– Case $a : as \diamond \mathsf{Choice}\ l\ r$:

$$\begin{aligned}
&\mathsf{find}_{\mathsf{syn}}\ ((a : as) +\!\!+ bs)\ ((\mathsf{Choice}\ l\ r) \ggg\!= h)\\
&= \ \mathsf{find}_{\mathsf{syn}}\ (a : (as +\!\!+ bs))\ ((\mathsf{Choice}\ l\ r) \ggg\!= h)\\
&= \ \mathtt{If'}\ a\ (\mathsf{find}_{\mathsf{syn}}\ (as +\!\!+ bs)\ (l \ggg\!= h))\ (\mathsf{find}_{\mathsf{syn}}\ (as +\!\!+ bs)\ (r \ggg\!= h))\\
&= \ (\text{by IH for } as \diamond l,\ as \diamond r)\\
&\quad\ \mathtt{If'}\ a\ (\mathsf{find}_{\mathsf{syn}}\ bs\ (h\ (\mathsf{find}_{\mathsf{syn}}\ as\ l)))\ (\mathsf{find}_{\mathsf{syn}}\ bs\ (h\ (\mathsf{find}_{\mathsf{syn}}\ as\ r)))\\
&=_{\beta\eta} \mathsf{find}_{\mathsf{syn}}\ bs\ (h\ (\mathtt{If'}\ a\ (\mathsf{find}_{\mathsf{syn}}\ as\ l)\ (\mathsf{find}_{\mathsf{syn}}\ as\ r)))\\
&= \ \mathsf{find}_{\mathsf{syn}}\ bs\ (h\ (\mathsf{find}_{\mathsf{syn}}\ (a : as)\ (\mathsf{Choice}\ l\ r)))
\end{aligned}$$

□

6 Discussion and further work

Instead of decision trees we could have used a direct encoding of the graph of a function, we call this the truth-table semantics. However, this approach leads not only too much longer normal forms but also the semantic equality is less efficient. On the other hand it is possible to go further and use Binary Decision Diagrams (BDDs) [7] instead of decision trees. We plan to explore this in further work and also give a detailed analysis of the normal forms returned by our algorithm.

We have argued that $\lambda^{\to 2}$ is the simplest λ-calculus with closed types, however we are confident that the technique described here works also for closed types in $\lambda^{0+1\times\to}$ (the finitary λ-calculus). We leave this extension for a journal version of this work.

One can go even further and implement finitary Type Theory, i.e. $\lambda^{012\Sigma\Pi}$ (note that $A + B = \Sigma x \in 2.\text{if}\ x\ \text{then}\ A\ \text{else}\ B$). This could provide an interesting base for a type-theoretic hardware description and verification language.

The approach presented here works only for calculi without type variables. It remains open to see whether this approach can be merged with the the standard techniques for NBE for systems with type variables, leading to an alternative proof of completeness and maybe even finite completeness for the calculi discussed above.

References

1. T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Proc. of 16th Annual IEEE Symposium on Logic in Computer Science*, pages 303–310. IEEE CS Press, 2001.

2. T. Altenkirch, M. Hofmann, and T. Streicher. Categorical reconstruction of a reduction free normalization proof. In D. Pitt, D. E. Rydeheard, and P. Johnstone, editors, *Category Theory and Computer Science*, number 953 in LNCS, pages 182–199, Springer-Verlag, 1995.
3. T. Altenkirch, M. Hofmann, and T. Streicher. Reduction-free normalisation for a polymorphic system. In *Proc. of 11th Annual IEEE Symposium on Logic in Computer Science*, pages 98–106. IEEE CS Press, 1996.
4. T. Altenkirch, M. Hofmann, and T. Streicher. Reduction-free normalisation for system F. Unpublished, available on WWW at `http://www.cs.nott.ac.uk/~txa/publ/f97.pdf`, 1997.
5. V. Balat. *Une étude des sommes fortes : isomorphismes et formes normales*. PhD thesis, Université Denis Diderot, 2002.
6. U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proc. of 6th Annual IEEE Symposium on Logic in Computer Science*, pages 202–211. IEEE CS Press, 1991.
7. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, 1986.
8. D. J. Dougherty and R. Subrahmanyam. Equality between functionals in the presence of coproducts. *Information and Computation*, 157(1–2):52–83, 2000.
9. P. Dybjer and A. Filinski. Normalization and partial evaluation. In G. Barthe et al., editors, *Applied Semantics*, number 2395 in LNCS, pages 137–192. Springer-Verlag, 2002.
10. N. Ghani. *Adjoint Rewriting*. PhD thesis, LFCS, Univ. of Edinburgh, 1995.
11. N. Ghani. $\beta\eta$-equality for coproducts. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Typed Lambda Calculus and Applications*, number 902 in LNCS, pages 171–185. Springer-Verlag, 1995.
12. C. McBride and J. McKinna. The view from the left. To appear in the Journal of Functional Programming, Special Issue: Dependent Type Theory Meets Programming Practice, 2004.
13. G. D. Plotkin. Lambda-definability and logical relations. Memorandum SAI-RM-4, Lab. for Artif. Intell., Univ. of Edinburgh, Oct. 1973.