AMEN

Wouter Swierstra

w.s.swierstra@uu.nl
Department of Information and Computing Sciences
Universiteit Utrecht

Abstract. One of Doaitse's regular contributions to teaching at the Universiteit Utrecht, has been his lectures for the Compiler Construction course about Church encodings and combinatory logic. I think one of the reasons he enjoys these topics so much is their combination of expressive power and simplicity. It is quite surprising just how much you can achieve in a 'language' as simple as the lambda calculus. This paper covers much of the same material, but adds a new twist.

Introduction

Church encodings represent data as functions. A classic example, written here using Haskell, is that of the natural numbers:

$$\textbf{type}\ \mathit{Nat} = \forall a.(a \to a) \to (a \to a)$$

Every natural number n is represented by a higher-order function that applies its argument function n times. For example, we can define the Church encoding of the first three natural numbers as follows:

$$\begin{aligned}
&\mathit{naught} :: \mathit{Nat}\\
&\mathit{naught} = \lambda f\ x \to x\\
&\mathit{one} :: \mathit{Nat}\\
&\mathit{one} \quad\ = \lambda f\ x \to f\ x\\
&\mathit{two} :: \mathit{Nat}\\
&\mathit{two} \quad\ = \lambda f\ x \to f\ (f\ x)
\end{aligned}$$

Furthermore you can define combinators to add, multiply, or exponentiate the Church encodings of two natural numbers. Here are possible definitions for these operations:

$$\begin{aligned}
&\mathit{add} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}\\
&\mathit{add}\ m\ n = \lambda f\ x \to n\ f\ (m\ f\ x)\\
&\mathit{mul} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}\\
&\mathit{mul}\ m\ n = \lambda f \to n\ (m\ f)\\
&\mathit{exp} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}\\
&\mathit{exp}\ m\ n\ = n\ m
\end{aligned}$$

When encountered for the first time, these results are quite surprising: lambda abstractions and application are enough to do basic arithmetic.

Church encodings simplify the notion of data, but higher-order functions are still complicated beasts: capture-avoiding substitution, α-equivalence, β-reduction, and all the usual issues associated with variable binding can be a real headache.

One way to tame this complexity is by implementing the lambda calculus using a combinator calculus. Such a calculus consists of a (small) set of 'elementary functions ... which embody certain common patterns of application' (Burge, 1975, Section 1.9). The most famous choice is the three combinators S, K, and I given by the following lambda terms:

$$\begin{aligned}\mathsf{S} &= \lambda x\ y\ z \to x\ z\ (y\ z)\\ \mathsf{K} &= \lambda x\ y \to x\\ \mathsf{I} &= \lambda x \to x\end{aligned}$$

This choice of combinators is particularly popular because of the straightforward translation scheme from lambda terms to their combinator counterparts. As is common in the literature (Barendregt, 1981; Sørensen and Urzyczyn, 2006), we write $\lambda^*\ x \to t$ for the translation of the term $\lambda x \to t$ to its corresponding combinator term. The following three rules define one possible translation scheme:

$$\begin{aligned}(\lambda^*\ x \to x) &= \mathsf{I}\\ (\lambda^*\ x \to A) &= \mathsf{K}\ A \text{ provided } x \notin A\\ (\lambda^*\ x \to t_1\ t_2) &= \mathsf{S}\ (\lambda^*\ x \to t_1)\ (\lambda^*\ x \to t_2)\end{aligned}$$

A choice of combinators for which such a translation exists is said to be *combinatory complete* (Barendregt, 1981).

There are many alternative combinatory complete bases: Curry and Feys (1958) originally proposed the following combinators:

$$\begin{aligned}\mathsf{I} &= \lambda x \to x\\ \mathsf{C} &= \lambda f\ x\ y \to f\ y\ x\\ \mathsf{W} &= \lambda f\ x \to f\ x\ x\\ \mathsf{B} &= \lambda f\ g\ x \to f\ (g\ x)\\ \mathsf{K} &= \lambda x\ y \to x\end{aligned}$$

Fokker (1992) has even derived a single combinator that is still combinatory complete:

$$\mathsf{X} = \lambda f \to f\ \mathsf{S}\ (\lambda x\ y\ z \to x)$$

As Doaitse is fond of observing, this shows how a term in the lambda calculus can be reduced to a series of opening and closing brackets (placed in some series of X combinators of a certain length), which in turn can be encoded as a series of bits.

$$
\begin{aligned}
\mathsf{A} &= \lambda m\ n\ f\ x \to n\ f\ (m\ f\ x)\\
\mathsf{M} &= \lambda m\ n\ f \to n\ (m\ f)\\
\mathsf{E} &= \lambda m\ n \to n\ m\\
\mathsf{N} &= \lambda x\ y \to y
\end{aligned}
$$

Fig. 1. The AMEN combinators

So far we have seen how to represent numbers by functions and functions by combinators. Now you might wonder if it is also possible to represent combinators by numbers. Or more precisely, can we use the Church encoding of natural numbers to define a combinatory complete basis?

Proposition 1. *The combinators A, M, E, and N, defined in Figure 1, correspond to the lambda terms for respectively addition, multiplication, exponentiation and naught for the Church encoding of Peano arithmetic. These combinators form a combinatory complete basis.*

I learned this proposition from Peter Hancock during my PhD in Nottingham. Peter tells me that the choice of combinator names is due to Jim Laird. This is no new result. Peter tells me that this proposition dates back to Stenlund (1972). Independently Böhm (1979) also describes the combinatory completeness of these combinators. Given the occasion, and Doaitse's love for combinatory calculi, it seems appropriate to reproduce this result here.

To prove this proposition, we give the following implementations of I, C, W, B, K, and S using the AMEN combinators:

$$
\begin{aligned}
\mathsf{I} &= \mathsf{N}\ \mathsf{M}\\
\mathsf{C} &= \mathsf{M}\ \mathsf{M}\ (\mathsf{M}\ \mathsf{E})\\
\mathsf{W} &= \mathsf{C}\ (\mathsf{E}\ \mathsf{E}\ (\mathsf{A}\ \mathsf{E}))\\
\mathsf{B} &= \mathsf{M}\ \mathsf{E}\ (\mathsf{M}\ \mathsf{M})\\
\mathsf{K} &= \mathsf{M}\ \mathsf{E}\ (\mathsf{M}\ \mathsf{N})\\
\mathsf{S} &= \mathsf{M}\ (\mathsf{A}\ \mathsf{E})\ \mathsf{C}
\end{aligned}
$$

The proof that these definitions behave as required can be found in the appendix. As you might expect, these calculations are rather dull. I do not want to claim that these are the shortest or prettiest definitions of the desired combinators. They just happen to be the first definitions that a program happened to find automatically. It's much more fun to have a closer look at this program, than the proofs it generates.

Evaluation

To find the above definitions, I wrote a short Haskell program. At its heart is an evaluation function, which tries to evaluate a term using the AMEN combinators to a normal form, if it exists.

Terms and values

To start, let us define data types to represent combinatory terms:

```
data Term a =
  Const a
  | (Term a) :@: (Term a)
```

Here we represent combinatory terms as binary trees of applications, with primitive combinators in the leaves. By defining this type to be polymorphic, we keep the option open to use the same structure for different choices of primitive combinators. One such choice of primitive combinators is, of course, the AMEN combinators we described above:

```
data AMEN = A | M | E | N
```

We introduce a separate data type to represent *values*, those terms that cannot be reduced further:

```
data Val a where
  Val :: a → [Val a] → Val a
```

One invariant that our data type does not express is that a value should not be applied to enough arguments to reduce. For example, $Val\ \mathsf{A}\ [\,]$ is really a value; $Val\ \mathsf{A}\ [m, n, f, x]$ on the other hand, is not a valid value for any choice of values m, n, f, and x as the combinator A has enough arguments to trigger reduction.

Decomposition

In the following pages, we will define a small step evaluation function in the style of Danvy (2008). We start by defining a data type for *evaluation contexts* (Felleisen and Hieb, 1992):

```
data Context a where
  Empty :: Context a
  Left  :: Context a → Term a → Context a
  Right :: Val a → Context a → Context a
```

You may want to think of such contexts as a zipper (Huet, 1997) in a *Term*, with the special property that we may only navigate to the right if we have fully evaluated the term on the left.

Just as regular zippers, such evaluation contexts support a *plug* operation, that plugs a term back into an evaluation context:

```
plug :: Term a → Context a → Term a
plug t Empty          = t
plug t (Left ctx u)   = (plug t ctx) :@: u
plug t (Right v ctx) = (fromVal v) :@: (plug t ctx)
```

Note that we use an auxiliary function *fromVal* converting a *Val* to a *Term* in the obvious fashion.

Although we have defined a data type to represent evaluation contexts, we have not yet written a function that *produces* an evaluation context. To do so, we make the observation that every term is either a value or a redex in some evaluation context. In line with Danvy, we introduce the data type *Decomposition* to express this choice.

```
type Redex a = (Val a, Val a)

data Decomposition a where
  IsVal    :: Val a → Decomposition a
  IsRedex :: Redex a → Context a → Decomposition a
```

The *Decomposition* data type can be thought of as describing a *view* (Wadler, 1987; McBride and McKinna, 2004) on terms. A redex consists of a value, together with a new argument for this value that may trigger further reduction.

We can construct a decomposition of any term using a pair of mutually recursive functions, *load* and *unload*:

```
decompose :: Term a → Decomposition a
decompose t = load t Empty

load :: Term a → Context a → Decomposition a
load (Const x) ctx = unload (Val x [ ]) ctx
load (f :@: x)   ctx = load f (Left ctx x)

unload :: Val a → Context a → Decomposition a
unload v Empty          = IsVal v
unload v (Left xs t)   = load t (Right v xs)
unload v (Right f xs) = IsRedex (f, v) xs
```

Starting with the empty context, the *load* function navigates to the leftmost innermost combinator or variable. Once found, it calls the *unload* function, defined by induction over the evaluation context. If the evaluation context is empty, the term we are decomposing is indeed a value. If the evaluation context is not empty, there are two possible cases. If this value has been applied to some argument, in which case there is a *Left* constructor on top of the evaluation context, we proceed by storing the value on the 'stack' and decomposing the term stored in the evaluation context in search for a redex. If the value we found was the argument to another value, witnessed by a *Right* constructor on top of the evaluation context, we know that we have found a redex.

Contraction

We will use the *decompose* function to define a small step evaluator for a combinator language. Such an evaluator will start by decomposing its argument term. If this yields a value, evaluation is finished; if this yields a redex, we contract the redex and continue reduction. Before completing our evaluator, we still need to define how to contract a redex. Contraction, however, is specific to the combinatory basis we choose. As we try to defer this choice for as long as possible, we introduce a separate *Contractible* class:

```
class Contractible a where
  step :: a → [Val b] → Maybe (Term b)
```

The *step* function has a slightly more general type than you might expect. Given a combinator of type a, applied to a list of arguments of type *Val b*, it may choose to rearrange these arguments and produce a new term of type *Term b*. If no reduction is possible, it should return *Nothing*. The additional flexibility of separating the type of the combinator (a) from the type of the values it manipulates (*Val b*) will turn out to be useful.

We specify how the *AMEN* combinators reduce by defining a suitable instance of the *Contractible* class:

```
instance Contractible AMEN where
  step A [v1, v2, v3, v4] = return (n :@: f :@: (m :@: f :@: x))
    where
      [m, n, f, x] = map fromVal [v1, v2, v3, v4]
  step M [v1, v2, v3]     = return (n :@: (m :@: f))
    where
      [m, n, f] = map fromVal [v1, v2, v3]
  step E [v1, v2]         = return (n :@: m)
    where
      [m, n] = map fromVal [v1, v2]
  step N [v1, v2]         = return x
    where
      [f, x] = map fromVal [v1, v2]
  step _ _                = Nothing
```

If you squint a bit, you should be able to recognize the original definition of the AMEN combinators from Figure 1. We can use this *step* function to (try to) contract a redex:

```
contract :: (Contractible b) ⇒ Val b → Val b → Maybe (Term b)
contract (Val x args) arg = step x (args ++ [arg])
```

To contract a redex, we add the new argument to the current value. Calling the *step* function then produces a new term, if the redex can now reduce.

```
eval :: Contractible a ⇒ Term a → Val a
eval t = go (decompose t)
  where
  go :: Contractible a ⇒ Decomposition a → Val a
  go (IsVal v)              = v
  go (IsRedex (f, v) ctx) =
    case (f `contract` v) of
      Just t'   → go (load t' ctx)
      Nothing → go (unload (addArg f v) ctx)
addArg :: Val a → Val a → Val a
addArg (Val x args) arg = Val x (args ++ [arg])
```

Fig. 2. The *eval* function

Evaluation

We can now define *eval* as a tail-recursive function in Figure 2. The *eval* function decomposes its argument term. If this yields a value, evaluation is complete. If we find a redex, we can try to contract it. If the contraction is successful and reduction takes place, continue evaluation with the new term and the current evaluation context. If the contraction did not (yet) reduce, we add the new argument to our stuck value and continue evaluation, in the hope that this will eventually yield further arguments for this stuck value.

It is easy to adapt this evaluator to produce an evaluation trace, printing every intermediate reduction step during execution. By doing so, and adding some extra lhs2TEX pragmas, we are able to produce the proofs in the appendix. The question remains, however, how we found the terms for I, C, W, B, K and S.

Searching for combinators

To find an AMEN term automatically that implements these combinators, we will specify their behaviour and use SmallCheck (Runciman et al., 2008) to search for a term that exhibits this intended behaviour.

To specify the desired behaviour of a combinator, we start by extending our *Term* language with variables. To do so, we require the following two definitions:

```
data Var = Var String
instance Contractible Var where
  step (Var x) args = Nothing
```

The *Contractible* instance for variables merely states that variables never reduce.

To grow our language, we take the coproduct of *Var* and *AMEN* as the possible values stored in the leaves of *Term* data type. Using the technology from

```
infixr 6 :+:
data (a :+: b) = Inl a | Inr b
class (:<:) sub sup
  where inj :: sub → sup
instance (:<:) a a
  where inj = id
instance (:<:) a (a :+: b)
  where inj = Inl
instance ((:<:) a c) ⇒ (:<:) a (b :+: c)
  where inj = Inr . inj
instance (Contractible a, Contractible b) ⇒ Contractible (a :+: b) where
  step (Inl x) args = step x args
  step (Inr y) args = step y args
```

Fig. 3. Automated injections

Swierstra (2008), summarized here in Figure 3 we can write a smart constructor to create variables.

```
var :: (Var :<: a) ⇒ String → Term a
var x = Const (inj (Var x))
```

Now finally, we can state the property that a term behaves precisely as the combinator K should:

```
isK :: Term (AMEN :+: Var) → Bool
isK t = eval (t :@: var "x" :@: var "y") ≡ Val (inj (Var "x")) []
```

Now we can call SmallCheck and ask it to exhaustively search up to a certain depth, trying to find a term t that satisfies this property. We could formulate the test that this property holds as:

```
exists isK
```

Unfortunately, SmallCheck does not report the witness that satisfies such a property. Instead we negate the statement and search for a counterexample as follows:

```
*Main> smallCheck 5 (forAll (not . isK))
Failed test no. 118.
there exists M E (M N) such that
  condition is false
```

To run this test, we need to write a few lines of boilerplate to help SmallCheck generate terms. This is entirely standard using the combinators provided by the

SmallCheck library. The only interesting case is that for variables. As we only want SmallCheck to generate closed terms using combinators, the generator for variables always fails.

In a similar fashion, we can find the definitions of I, C, and B. All these searches return a result almost immediately. The other combinators are harder to find. While we could increase the depth of our search, this does not help much: the search space quickly becomes too large.

Instead, we exploit the fact that we know how to implement C using our original AMEN combinators. It is therefore safe to add it as a primitive combinator to our combinator language:

```
data C = C
instance Contractible C where
  step C [v1, v2, v3] = return (f :@: y :@: x)
    where
      [f, x, y] = map fromVal [v1, v2, v3]
  step C _ = Nothing
```

To permit the usage of C in our search, all we need to do is change the *type signature* of our predicate over terms. For instance, the predicate characterizing the W combinator becomes:

```
isW :: Term (AMEN :+: Var :+: C) → Bool
isW t = eval (t :@: (var "f") :@: (var "x"))
  ≡ Val (inj (Var "f")) [Val (inj (Var "x")) [], Val (inj (Var "x")) []]
```

With this modification, SmallCheck finds the terms for all the six combinators on page 3 in less than thirty seconds.

Further fun

No paper for Doaitse would be complete without some mention of parsing. When writing this paper, I was never really happy with some of the SmallCheck properties. They quickly devolve into a large values and terms, full of parentheses, string literals, and lists—the *isW* property above is a good example. Surely we can do better! Doaitse is certainly not one to shy away from experimental GHC extensions, which we will use for this last diversion.

Let's start by defining a parser for values using Doaitse's `uu-parsinglib`. As values can contain arbitrary combinators, we define a type class to represent types that have some associated parser:

```
type Parser a = P (Str Char String LineColPos) a
class Parsable a where
  parser :: Parser a
```

Using this type class, we can parse any value containing *Parsable* combinators.

```
pVal :: Parsable a ⇒ Parser (Val a)
pVal = foldl1 addArg <$> pList1Sep pSpaces pValAtom
  where
  pValAtom = (λx → Val x []) <$> parser
           <|> pParens pVal
```

This parser recognizes a series of atomic values, separated by whitespace. An atomic value is either a primitive combinator, variable, or a compound value surrounded by parentheses. For example, to parse various combinators or variables we define straightforward parsers.

```
instance Parsable AMEN where
  parser = A <$ pSym 'A'
         <|> M <$ pSym 'M'
         <|> E <$ pSym 'E'
         <|> N <$ pSym 'N'
instance Parsable C where
  parser = C <$ pSym 'C'
instance Parsable Var where
  parser = (λx xs → Var (x : xs)) <$> pLower <*> pMany pLetter
```

We can also define a parser for sum of parsible types:

```
instance (Parsable a, Parsable b) ⇒ Parsable (a :+: b) where
  parser = Inl <$> parser
         <|> Inr <$> parser
```

Of course, just defining these parsers is not very useful just yet, as we have not yet defined how to run them.

Using GHC's recent extension for quasiquoting (Mainland, 2007), however, we can add special syntax for values. To do so, we need to define a record of type *QuasiQuoter*. As we will only use this quasiquoter to quote expressions, as opposed to types, patterns, or declarations, we only fill in one field, *quoteExp*. The *quoteExp* field requires a function of type *String* → *Q Exp* which parses any string to a Template Haskell abstract syntax tree. By running our value parser, and quoting the result using the Template Haskell *lift* function, we can assemble the desired quasiquoter:

```
val :: QuasiQuoter
val = let p = pVal :: Parser (Val (AMEN :+: Var :+: C))
      in QuasiQuoter { quoteExp = lift . run p }
```

The *run* function discards any leading or trailing whitespace and runs its argument parser on the input string. The Template Haskell *lift* function turns a Haskell value into the corresponding abstract syntax tree. The complete code, once again, requires a few trivial instances of the Template Haskell *Lift* class. These instances are easy to write by hand.

Using this QuasiQuoter, makes it a bit easier to write complicated SmallCheck properties using non-trivial values.[1] For example, to specify the behaviour of the S combinator, we can now write:

```
isS :: Term (AMEN :+: Var :+: C) → Bool
isS t = eval (t :@: var "x" :@: var "y" :@: var "z")
        ≡ [val | x z (y z) |]
```

Of course, we can define a quasiquoter for our *Term* type as well. To be useful, however, we would need to extend our language with *anti-quotation*, that is, the ability to refer to the Haskell variable t in such a quoted expression. But perhaps that is a story for another time.

Closure

Thank you Doaitse, for all your work. Your uninhibited enthusiasm for functional programming has certainly proved to be contagious. I fondly remember your hospitality when I stayed in Houten during my first week at university as a young student. Whenever I would stop by your office, you would always recommend an interesting paper to read. At the end of my third year in Utrecht, you suggested I visit Oege de Moor to write my BSc thesis. This turned out to be my first encounter with dependent types. At the time, I could never have imagined teaching a seminar in Utrecht on the topic ten years later.

You will be missed. AMEN.

References

Barendregt, H.: The Lambda Calculus: Its Syntax and Semantics, Studies in Logic and the Foundations of Mathematics, vol. 103. Elsevier (1981)

Böhm, C.: Un modèle arithmétique des termes de la logique combinatoire. In: Robinet, B. (ed.) Proc. Sixième Ecole de Printemps d'Informatique Theorique – Lambda Calcul et Semantique Formelle des Langages de Programmation. pp. 97–108 (1979)

Burge, W.: Recursive Programming Techniques. Addison-Wesley Publishing Company (1975)

Curry, H., Feys, R.: Combinatory Logic, vol. 1. North-Holland Publishing Company (1958)

Danvy, O.: From reduction-based to reduction-free normalization. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) Proceedings of the 6th International School on Advanced Functional Programming. pp. 66–164. No. 5382 in LNCS, Springer-Verlag (2008)

Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theoretical computer science 103(2), 235–271 (1992)

[1] Using such quasi-quotation this code needs to be done in a separate file due to GHC's Template Haskell stage restriction.

Fokker, J.: The systematic construction of a one-combinator basis for lambda-terms. Formal Aspects of Computing 4, 776–780 (1992)
Huet, G.: The zipper. Journal of Functional Programming 7(5), 549–554 (1997)
Mainland, G.: Why it's nice to be quoted: Quasiquoting for Haskell. In: Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop. pp. 73–82. ACM, New York, NY, USA (2007)
McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming 14(1) (2004)
Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In: Proceedings of the first ACM SIGPLAN symposium on Haskell. pp. 37–48. Haskell '08 (2008)
Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism, Studies in Logic and the Foundations of Mathematics, vol. 149. Elsevier Science Inc. (2006)
Stenlund, S.: Combinators, λ-Terms and Proof Theory, Synthese Library, vol. 42. Reidel, Dordrecht (1972)
Swierstra, W.: Data types à la carte. Journal of Functional Programming 18(4), 423–436 (2008)
Wadler, P.: Views: A way for pattern matching to cohabit with data abstraction. In: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 307–313 (1987)

A Proofs

A.1 Correctness of the definition of I

$$
\begin{array}{ll}
& \mathsf{I}\ x \\
= & \quad \{\text{by definition of } \mathsf{I}\} \\
& \mathsf{N}\ \mathsf{M}\ x \\
= & \quad \{\text{by definition of } \mathsf{N}\} \\
& x
\end{array}
$$

A.2 Correctness of the definition of C

$$
\begin{array}{ll}
& \mathsf{C}\ f\ x\ y \\
= & \quad \{\text{by definition of } \mathsf{C}\} \\
& \mathsf{M}\ \mathsf{M}\ (\mathsf{M}\ \mathsf{E})\ f\ x\ y \\
= & \quad \{\text{by definition of } \mathsf{M}\} \\
& \mathsf{M}\ \mathsf{E}\ (\mathsf{M}\ f)\ y\ x \\
= & \quad \{\text{by definition of } \mathsf{M}\} \\
& \mathsf{M}\ f\ (\mathsf{E}\ x)\ y \\
= & \quad \{\text{by definition of } \mathsf{M}\} \\
& \mathsf{E}\ x\ (f\ y) \\
= & \quad \{\text{by definition of } \mathsf{E}\} \\
& f\ y\ x
\end{array}
$$

A.3 Correctness of the definition of W

$W\ f\ x$
$=$ {by definition of W}
$\mathsf{C}\ (\mathsf{E}\ \mathsf{E}\ (\mathsf{A}\ \mathsf{E}))\ f\ x$
$=$ {by definition of E}
$\mathsf{C}\ (\mathsf{A}\ \mathsf{E}\ \mathsf{E}\ x\ f)$
$=$ {by definition of C}
$\mathsf{A}\ \mathsf{E}\ \mathsf{E}\ x\ f$
$=$ {by definition of A}
$\mathsf{E}\ x\ (\mathsf{E}\ x\ f)$
$=$ {by definition of E}
$\mathsf{E}\ x\ (f\ x)$
$=$ {by definition of E}
$f\ x\ x$

A.4 Correctness of the definition of B

$B\ f\ g\ x$
$=$ {by definition of B}
$\mathsf{M}\ \mathsf{E}\ (\mathsf{M}\ \mathsf{M})\ f\ g\ x$
$=$ {by definition of M}
$\mathsf{M}\ \mathsf{M}\ (\mathsf{E}\ f)\ x\ g$
$=$ {by definition of M}
$\mathsf{E}\ f\ (\mathsf{M}\ g)\ x$
$=$ {by definition of E}
$\mathsf{M}\ g\ f\ x$
$=$ {by definition of M}
$f\ (g\ x)$

A.5 Correctness of the definition of K

$\mathsf{K}\ x\ y$
$=$ {by definition of K}
$\mathsf{M}\ \mathsf{E}\ (\mathsf{M}\ \mathsf{N})\ x\ y$
$=$ {by definition of M}
$\mathsf{M}\ \mathsf{N}\ (\mathsf{E}\ x)\ y$
$=$ {by definition of M}
$\mathsf{E}\ x\ (\mathsf{N}\ y)$
$=$ {by definition of E}
$\mathsf{N}\ y\ x$
$=$ {by definition of N}
x

A.6 Correctness of the definition of S

$S\ x\ y\ z$
$=$ {by definition of S}
$\mathsf{M}\ (\mathsf{A}\ \mathsf{E})\ \mathsf{C}\ x\ y\ z$
$=$ {by definition of M}
$\mathsf{C}\ (\mathsf{A}\ \mathsf{E}\ x)\ z\ y$
$=$ {by definition of C}
$\mathsf{A}\ \mathsf{E}\ x\ z\ y$
$=$ {by definition of A}
$x\ z\ (\mathsf{E}\ z\ y)$
$=$ {by definition of E}
$x\ z\ (y\ z)$