EPiC Series in Computer Science
Volume 35, 2015, Pages 119–130
LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations

A New Proof of P-time Completeness of Linear Lambda Calculus

Satoshi Matsuoka

National Institute of Advanced Industrial Science and Technology (AIST),
1-1-1 Umezono, Tsukuba, Ibaraki, 305-8563 Japan
matsuoka@ni.aist.go.jp

Abstract

We give a new proof of P-time completeness of the Linear Lambda Calculus, which was originally given by H. Mairson in 2003. Our proof uses an essentially different Boolean type from the type that Mairson used. Moreover the correctness of our proof can be machined-checked using an implementation of Standard ML.

1 Introduction

In [5] Mairson gave a proof of P-time completeness of the Linear Lambda Calculus. It is an excellent exercise of linear functional programming. The crucial point of the proof is that the copy function of the truth values is representable by a linear term: this is relatively easy in the Affine Lambda Calculus as shown in [5], but difficult in the Linear Lambda Calculus. So, the key issue there is to avoid the use of the weakening rule. The issue was also treated from a different angle in [6], which established typed Böhm theorem without the weakening rule.

In this paper we give a new proof of P-time completeness of the Linear Lambda Calculus. Our proof is different from that of [5] with regard to the following points:

- In [5] Mairson used the base Boolean type $\mathbb{B}_{\mathrm{HM}} = p \multimap p \multimap (p \multimap p \multimap p) \multimap p$ while we use $\mathbb{B} = (p \multimap p) \multimap (p \multimap p) \otimes (p \multimap p)$. Although both have two normal forms, they are essentially different because while $\mathbb{B}_{\mathrm{HM}}$ reduces to itself by the *linear distributive transformation* given in Section 3 of [6] (which was called third order reduction in [6]), $\mathbb{B}$ reduces to

$$\mathbb{B}_{\mathrm{red}} = p \multimap p \multimap (p \multimap p) \multimap (p \multimap p \multimap p) \multimap p,$$

which has six normal forms.
- All the two variable functions that are representable over $\mathbb{B}_{\mathrm{HM}}$ *without any polymorphism* are only *exclusive or* and its negation, but in $\mathbb{B}$ they are all the boolean functions except for exclusive or and its negation, i.e., fourteen functions.
- Our proof is also an interesting application of the linear distributive transformation.

As in [5], our proof is also machine-checkable: all the linear λ-terms in this paper are also well-formed expressions of Standard ML [9]. So the reader may confirm the correctness of our proof using an implementation of Standard ML. We used the interactive system of Standard ML of New Jersey. Although this correctness verification of our result is light-weight, but it seems enough for the confirmation of our result. In particular such an effort has been rarely tried in the literature in the Linear Lambda Calculus.

Since boolean functions and circuits have a lot of applications in (theoretical or practical) computer science, our result has a lot of potential applications: for example, analysis of boolean circuits, model checking, etc. In particular, when represented as MLL proof nets (see [4]), two normal terms of $\mathbb{B}$ are planar proof nets (see Figure 1). Crossing numbers of proof nets over $\mathbb{B}$ that represent boolean functions may have an interesting connection to the theory of boolean function complexity. For example, the number of unary function over $\mathbb{B}$ is four. Among them, while three functions can be represented as planar proof nets, one that corresponds to the *not* gate is not (see Figures 2, 3, 4, and 5).

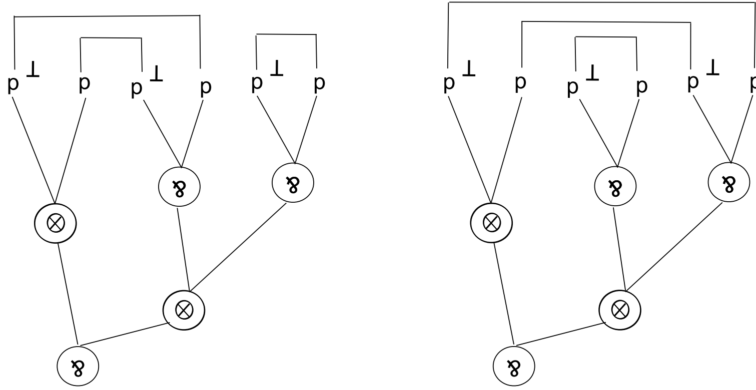

Figure 1: Two Normal Proofs of $\mathbb{B}$ (0 and 1)

One of advantages of our approach over other tools is that our boolean encoding is executable as a functional program. Of course you can use $\mathbb{B}_{\mathrm{HM}}$ instead of $\mathbb{B}$: one of advantages of $\mathbb{B}$ over $\mathbb{B}_{\mathrm{HM}}$ is that we can encode more boolean functions without ML-polymorphism in $\mathbb{B}$ than that of $\mathbb{B}_{\mathrm{HM}}$, although both classes of representable functions are disjoint.

2 Typing Rules

We give our term assignment system for the Linear Lambda Calculus. Our system is based on the Natural Deduction, e.g., given in [10, 2], which is equivalent to the system based on Sequent Calculus or proof nets in [4] (see [10]). Its notation is unusual in the Linear Logic community, but its purpose is to make our proof machine-checkable.

Types

$$\texttt{A} ::= \texttt{'a} \mid \texttt{A1*A2} \mid \texttt{A1->A2}$$

The symbol `'a` stands for a type variable. On the other hand `A1*A2` stands for the tensor product $\texttt{A1} \otimes \texttt{A2}$ and `A1->A2` for the linear implication $\texttt{A1} \multimap \texttt{A2}$ in the usual notation.

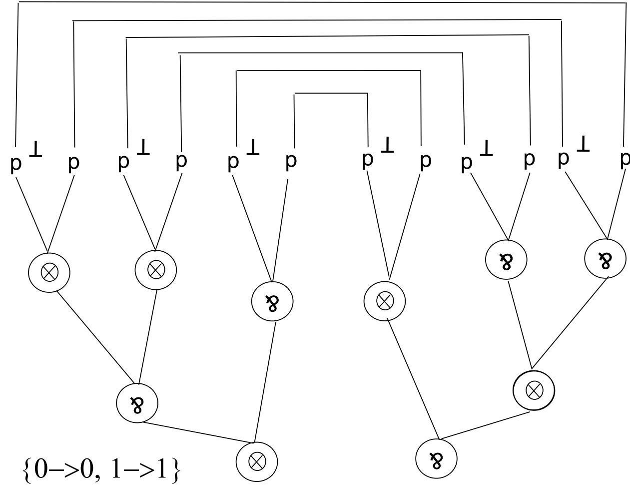

Figure 2: Normal Proof of $\mathbb{B} \multimap \mathbb{B}$ (identity)

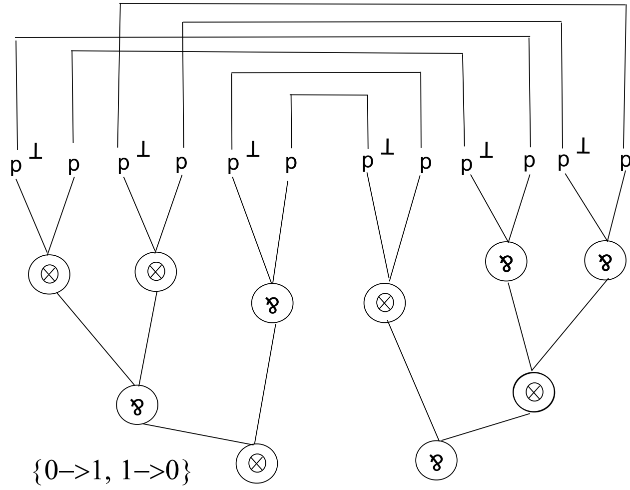

Figure 3: Normal Proof of $\mathbb{B} \multimap \mathbb{B}$ (negation)

Terms We use `x,y,z` for term variables, $\overrightarrow{\mathtt{x}}, \overrightarrow{\mathtt{y}}, \overrightarrow{\mathtt{z}}$ for finite lists of term variables, and `t,s` for general terms.

Term Assignment System

$$\frac{}{\mathtt{x:A} \vdash \mathtt{x:A}}$$

$$\frac{\mathtt{x:A}, \overrightarrow{\mathtt{y}}:\Gamma \vdash \mathtt{t:B}}{\overrightarrow{\mathtt{y}}:\Gamma \vdash \mathtt{fn\ x=>t:A->B}} \qquad \frac{\overrightarrow{\mathtt{x}}:\Gamma \vdash \mathtt{t:A->B} \quad \overrightarrow{\mathtt{y}}:\Delta \vdash \mathtt{s:A}}{\overrightarrow{\mathtt{x}}:\Gamma, \overrightarrow{\mathtt{y}}:\Delta \vdash \mathtt{ts:B}}$$

{0–>0, 1–>0}

Figure 4: Normal Proof of $\mathbb{B} \multimap \mathbb{B}$ (constant function)

{0–>1, 1–>1}

Figure 5: Normal Proof of $\mathbb{B} \multimap \mathbb{B}$ (constant function)

$$\frac{\overrightarrow{\mathtt{x}}:\Gamma \vdash \mathtt{s:A} \quad \overrightarrow{\mathtt{y}}:\Delta \vdash \mathtt{s:B}}{\overrightarrow{\mathtt{x}}:\Gamma, \overrightarrow{\mathtt{y}}:\Delta \vdash \mathtt{(s,t):A*B}} \qquad \frac{\overrightarrow{\mathtt{x}}:\Gamma \vdash \mathtt{s:A*B} \quad \mathtt{x:A}, \mathtt{y:B}, \overrightarrow{\mathtt{y}}:\Delta \vdash \mathtt{t:C}}{\overrightarrow{\mathtt{x}}:\Gamma, \overrightarrow{\mathtt{y}}:\Delta \vdash \mathtt{let\ val\ (x,y)=s\ in\ t\ end:C}}$$

Moreover, function declaration
`fun f x1 x2 ··· xn = t`
is interpreted as the following term:
`f = fn x1 => fn x2 => ··· => fn xn => t`
Below we consider only closed terms (i.e., combinators)
$\vdash$ `t:A`.
The following proposition is proved easily by structural induction:

Proposition 2.1. *If* `x1:A1,...,xn:An|-t:B` *then*
`fun f x1 x2 ··· xn = t`

is a well formed function declaration of Standard ML.

Term Reduction Rules Two of our reduction rules are
(β_1): `(fn x=>t)s` $\Rightarrow_{\beta_1}$ `t[s/x]`
(β_2): `let val (x,y)=(u,v) in w end` $\Rightarrow_{\beta_2}$ `w[u/x,v/y]`
Then note that if a function `f` is defined by
`fun f x1 x2` $\cdots$ `xn = t`
and
`x1:A1,...,xn:An|-t:B`, `|-t1:A1`, ..., `|-tn:An`
then, we have
`f t1` $\cdots$ `tn` $\Rightarrow^*_{\beta_1}$ `t[t1/x1,...,tn/xn]` .
We denote the reflexive transitive closure of a relation R by R^*. In the following $\rightarrow_\beta$ denotes the congruent (one-step reduction) relation generated by the two reduction rules above and the following contexts:

$$\begin{array}{rcl} C[] & = & [] \mid C[]\texttt{t} \mid \texttt{t}C[] \mid (\texttt{t},C[]) \mid (C[],\texttt{t}) \mid \texttt{fn x=>}\, C[] \\ & & \mid \texttt{let val}\ (\texttt{x},\texttt{y}) = C[]\ \texttt{in t end} \mid \texttt{let val}\ (\texttt{x},\texttt{y}) = \texttt{t in}\ C[]\ \texttt{end} \end{array}$$

In order to establish a full and faithful embedding from linear λ-terms to IMLL proof nets, we introduce further reduction rules. Basically we follow [1]. The following are η-rules:
(η_1): `fn x=>(t x)` $\Rightarrow_{\eta_1}$ `t`
(η_2): `let val (x,y) = t in (x,y)` $\Rightarrow_{\eta_2}$ `t`
In the following $\rightarrow_{\beta\eta}$ denotes the congruent (one-step reduction) relation generated by the four reduction rules above and any context $C[]$. But these reduction rules are not enough: different normal terms may correspond to the same normal IMLL proof net. In order to make further identification we introduce the following commutative conversion rules. Before doing this, we introduce another contexts $C_0[]$, whose set is a subset of $C[]$:

$$C_0[] = [] \mid C_0[]\texttt{t} \mid \texttt{t}C_0[] \mid (\texttt{t},C_0[]) \mid (C_0[],\texttt{t})$$

Then we define the commutative conversion relation $\leftrightarrow_{\mathrm{c}}$:

$c1$. `let val (x,y)=(let val (x',y')=t in s end) in u end`
  $\leftrightarrow_{\mathrm{c}}$ `let val (x',y')=t in (let val (x,y) =s in u end) end`
$c2$. `fn z=> let val (x,y)=t in u end` $\leftrightarrow_{\mathrm{c}}$ `let val (x,y)=t in fn z=> u end`
  where $\texttt{z} \notin FV(\texttt{t})$
$c3$. C_0`[let val (x,y)=t in u end]` $\leftrightarrow_{\mathrm{c}}$ `let val (x,y)=t in` C_0`[u] end`
  where $FV(C_0[]) \cap \{\texttt{x},\texttt{y}\} = \emptyset$
$c4$. `let val (x,y)=t in (let val (x',y')=u in v end) end`
  $\leftrightarrow_{\mathrm{c}}$ `let val (x',y')=u in (let val (x,y)=t in v end) end`
  where $FV(\texttt{u}) \cap \{\texttt{x},\texttt{y}\} = \emptyset$ and $FV(\texttt{t}) \cap \{\texttt{x'},\texttt{y'}\} = \emptyset$

Let $=_{\mathrm{c}}$ be the congruent equivalence relation generated by $\leftrightarrow_{\mathrm{c}}$ and any context $C[]$. Then we define $\rightarrow_{\beta\eta\mathrm{c}}$ be the least relation satisfying the following rule:

$$\frac{\texttt{t} =_{\mathrm{c}} \texttt{t'} \quad \texttt{t'} \rightarrow_{\beta\eta} \texttt{u'} \quad \texttt{u'} =_{\mathrm{c}} \texttt{u}}{\texttt{t} \rightarrow_{\beta\eta\mathrm{c}} \texttt{u}}$$

Then the following holds.

Proposition 2.2 (Church Rosser[1]). *if* `t` $\rightarrow_{\beta\eta\mathrm{c}}$ `t'` *and* `t` $\rightarrow_{\beta\eta\mathrm{c}}$ `u'` *then for some* `w` $=_{\mathrm{c}}$ `w'`, `t'` $\rightarrow_{\beta\eta\mathrm{c}}$ `w` *and* `u'` $\rightarrow_{\beta\eta\mathrm{c}}$ `w'`.

Furthermore we can easily prove that $\rightarrow_{\beta\eta c}$ is strong normalizable as shown in [1]. We can conclude that we have the uniqueness property for normal forms under $\rightarrow_{\beta\eta c}$ up to $=_c$.

3 Review of Mairson's Proof

In this section we review the proof in [5] briefly. Below by normal forms we mean normal forms with regard to $\rightarrow_{\beta\eta c}$. The basic construct is the following term:

```
- fun Pair x y z = z x y;
val Pair = fn :  'a -> 'b -> ('a -> 'b -> 'c) -> 'c
```

Using this, we define `True` and `False`:

```
- fun True x y = Pair x y;
val True = fn :  'a -> 'b -> ('a -> 'b -> 'c) -> 'c
- fun False x y = Pair y x;
val True = fn :  'a -> 'b -> ('b -> 'a -> 'c) -> 'c
```

Note that these are the normal forms of $\mathbb{B}_{\mathrm{HM}}$. In order to define the term `Copy`, we need two auxiliary terms:

```
- fun I x = x;
val I = fn :  'a -> 'a
- fun id B = B I I I ;
val id = fn :  (('a -> 'a) -> ('b -> 'b) -> ('c -> 'c) -> 'd) -> 'd
```

The formal argument `B` is supposed to receive `True` or `False`. It is easy to see that

$$\texttt{id True} \rightarrow_\beta^* \texttt{I}, \quad \texttt{id False} \rightarrow_\beta^* \texttt{I},$$

Then the term `Copy` is defined as follows:

```
- fun Copy P = P (Pair True True) (Pair False False)
(fn U => fn V =>
U (fn u1 => fn u2 =>
V (fn v1 => fn v2 =>
((id v1) u1, (id v2) u2)))))
```

We omit its type since it is too long. The formal argument `P` is supposed to receive `True` or `False`. While the encoding in [5] uses continuation passing style, the above term does not since we have the $\otimes(= *)$-connective and can do a direct encoding using this connective. Then

```
Copy True;
val it=(fn,fn):('a -> 'b -> ('a -> 'b -> 'c) -> 'c)*('d -> 'e -> ('d -> 'e -> 'f) -> 'f)
Copy False;
val it=(fn,fn):('a -> 'b -> ('b -> 'a -> 'c) -> 'c)*('d -> 'e -> ('e -> 'd -> 'f) -> 'f)
```

(in fact, since SML/NJ does not allow any function values, it gives warnings, but the results are basically the same). These are `(True,True)` and `(False,False)` respectively since

$$\texttt{Copy True} \rightarrow_\beta^* ((\texttt{id False})\ \texttt{True},\ (\texttt{id False})\ \texttt{True})$$
$$\texttt{Copy False} \rightarrow_\beta^* ((\texttt{id True})\ \texttt{False},\ (\texttt{id True})\ \texttt{False})$$

The basic observations here are that

- the type of the formal argument `P` of `Copy` is unifiable with that of both `True` and `False`;
- the types of `Copy True` and `Copy False` are desirable ones, i.e., both have $\mathbb{B}_{\mathrm{HM}} \otimes \mathbb{B}_{\mathrm{HM}}$ as a instance.

Since the *and* gate can be defined similarly and more easily and the *not* gate without any ML-polymorphism, it is concluded that all the boolean circuits can be defined over $\mathbb{B}_{\mathrm{HM}}$.

4 A Partial Solution

In this section we present our failed attempt. Let the following two terms be `True'` and `False'`:

```
fun True' x y f = Pair x (f y);
val True' = fn :  'a -> 'b -> ('b -> 'c) -> ('a -> 'c -> 'd) -> 'd
fun False' x y f = Pair (f x) y;
val False' = fn :  'a -> 'b -> ('a -> 'c) -> ('c -> 'b -> 'd) -> 'd
```

Both are normal forms of $\mathbb{B}_{\mathrm{red}} = p \multimap p \multimap (p \multimap p) \multimap (p \multimap p \multimap p) \multimap p$, which has six normal forms. Below we define a copy function for them as in the previous section. In order to do that, we need several auxiliary terms:

```
fun not' f x y g h = f y x g (fn u => fn v => (h v u));
val not' = fn
:  ('a -> 'b -> 'c -> ('d -> 'e -> 'f) -> 'g)
-> 'b -> 'a -> 'c -> ('e -> 'd -> 'f) -> 'g
```

The term `not'` represents the *not* gate for the new boolean values.

```
fun swap f g = f (fn u => fn v => g v u);
val swap = fn :  (('a -> 'b -> 'c) -> 'd) -> ('b -> 'a -> 'c) -> 'd
```

We note that

$$\begin{aligned} \mathtt{swap}\,(\mathtt{Pair\ False'\ True'})\,\mathtt{g} &=_\beta (\mathtt{Pair\ False'\ True'})(\mathtt{fn\,u} => \mathtt{fn\,v} => \mathtt{g\ u\ v}) \\ &=_\beta \mathtt{g\ True'\ False'} =_\beta \mathtt{Pair\ True'\ False'\ g} \end{aligned}$$

The term `newid` is similar to `id`, but receives four arguments:

```
fun newid B' = B' I I I I;
val newid = fn
:  (('a -> 'a) -> ('b -> 'b) -> ('c -> 'c) -> ('d -> 'd) -> 'e) -> 'e
```

The term `constNot` is also similar to `id`, but always returns `not'`:

```
fun constNot B' = B' I not' I I;
val constNot = fn
:  (('a -> 'a)
-> (('b -> 'c -> 'd -> ('e -> 'f -> 'g) -> 'h)
-> 'c -> 'b -> 'd -> ('f -> 'e -> 'g) -> 'h)
-> ('i -> 'i) -> ('j -> 'j) -> 'k)
-> 'k
```

The formal argument `B'` in `newid` and `constNot` is supposed to receive `True'` and `False'`. We can easily see

$$\begin{array}{ll} \mathtt{newid\ True'} \to_\beta^* \mathtt{I}, & \mathtt{newid\ False'} \to_\beta^* \mathtt{I}, \\ \mathtt{constNot\ True'} \to_\beta^* \mathtt{not}, & \mathtt{constNot\ False'} \to_\beta^* \mathtt{not} \end{array}$$

Under the preparation above, we can define `Copy'` as follows:

```
fun Copy' P' = P' (Pair False' True') (Pair False' True') swap
(fn U => fn V =>
U (fn u1 => fn u2 =>
V (fn v1 => fn v2 =>
((constNot v1) u1, (newid v2) u2))));
```

Again we omit the type. The formal parameter `P` is supposed to receive `True'` or `False'`. Then

$$\begin{array}{rcl} \texttt{Copy' True'} & \to_\beta^* & ((\texttt{constNot True})\ \texttt{False},\ (\texttt{newid False})\ \texttt{True}) \\ & \to_\beta^* & (\texttt{True},\ \texttt{True}) \\ \texttt{Copy' False'} & \to_\beta^* & ((\texttt{constNot False})\ \texttt{True},\ (\texttt{newid True})\ \texttt{False}) \\ & \to_\beta^* & (\texttt{False},\ \texttt{False}) \end{array}$$

Unfortunately we could not find a term that represents the *and* gate over `True'` and `False'`. Without ML polymorphism, only projections and constant functions are representable in $\mathbb{B}_{\mathrm{red}}$, which does not includes the *and* gate. Moreover unlike [5], with ML polymorphism, a term for the *and* gate can not be obtained as a simplified version of `Copy'` over $\mathbb{B}_{\mathrm{red}}$ in a naive way. Note that in our encoding we can not swap the order of two arguments: we can only choose which argument is applied to the `swap` function. That's why the *and* gate is not representable over $\mathbb{B}_{\mathrm{red}}$ by imitating the method in [5]. This indicates that our encoding is quite different from that of [5].

So, we must find a similar, but different substitute. Fortunately we have found a solution described in the next section.

5 Our Solution

Our solution uses the type $\mathbb{B} = (p \multimap p) \multimap (p \multimap p) \otimes (p \multimap p)$, which has two normal forms:

```
fun True'' x = (fn z => z, fn y => x y);
val True'' = fn : ('a -> 'b) -> ('c -> 'c) * ('a -> 'b)
fun False'' x = (fn y => x y, fn z => z);
val False'' = fn : ('a -> 'b) -> ('a -> 'b) * ('c -> 'c)
```

The linear distributive transformation (which was called third-order reduction in [6]) turn $\mathbb{B}$ into $\mathbb{B}_{\mathrm{red}}$. The next term is its internalized version:

```
fun LDTr h x y f z
= let val (k, l) = h f in z (k x) (l y) end;
val LDTr = fn
: ('a -> ('b -> 'c) * ('d -> 'e))
-> 'b -> 'd -> 'a -> ('c -> 'e -> 'f) -> 'f
```

Then

$$\texttt{LDTr True''} =_\beta \texttt{True'}, \quad \texttt{LDTr False''} =_\beta \texttt{False'}$$

Our term for the *not* gate for `True''` and `False''` is

```
fun not'' h f = let val (k, l) = h f in (l, k) end;
val not'' = fn : ('a -> 'b * 'c) -> 'a -> 'c * 'b
```

Then

$$\texttt{not'' True''} =_\beta \texttt{False''}, \quad \texttt{not'' False''} =_\beta \texttt{True''}$$

Moreover we can write down a term for the *and* gate as follows:

```
fun and'' f g h = let val (u, v) = g (fn k => h k) in
(let val (x, y) = f (fn w => v w) in
(fn s => x (u s), fn t => y t) end) end;
val and'' = fn
: (('a -> 'b) -> ('c -> 'd) * ('e -> 'f))
-> (('g -> 'h) -> ('i -> 'c) * ('a -> 'b))
```

```
-> ('g -> 'h) -> ('i -> 'd) * ('e -> 'f)
```

Note that the definition of `and''` does not use any ML-polymorphism. Then

$$\texttt{and'' True'' True''} =_\beta \texttt{True''}, \quad \texttt{and'' False'' False''} =_\beta \texttt{False''},$$
$$\texttt{and'' True'' False''} =_\beta \texttt{False''}, \quad \texttt{and'' False'' True''} =_\beta \texttt{False''}$$

Next, we define a copy function for `True''` and `False''`. In order to do that, we need a modified version of `constNot`:

```
fun constNot'' B'' = B'' I not'' I I;
```

Then we can easily see

$$\texttt{newid True''} \rightarrow^*_\beta \texttt{I}, \quad \texttt{newid False''} \rightarrow^*_\beta \texttt{I},$$
$$\texttt{constNot'' True''} \rightarrow^*_\beta \texttt{not}, \quad \texttt{constNot'' False''} \rightarrow^*_\beta \texttt{not}$$

Under the preparation above, we can define a copy function for $\mathbb{B}$, which is a modified version of `Copy'` as follows:

```
fun Copy'' P'' =
LDTr P'' (Pair False'' True'') (Pair False'' True'') swap
(fn U => fn V =>
U (fn u1 => fn u2 =>
V (fn v1 => fn v2 =>
((constNot'' (LDTr v1)) u1, (newid (LDTr v2)) u2))));
```

Then

$$\texttt{Copy'' True''} \rightarrow^*_\beta (\texttt{True''}, \texttt{True''}), \quad \texttt{Copy'' False''} \rightarrow^*_\beta (\texttt{False''}, \texttt{False''})$$

Note that the linear distributive transformation is applied to the formal parameter `P''` in the term. It transforms a term of the domain with high-order or tensor connectives to a term with order less than fourth-order and without tensor connectives injectively. In general, while the elimination of weakening rule (i.e., a kind of garbage collection) is easier with low order types, it tends to be able to represent more boolean functions with higher order types or types with tensor connectives. For example the number of two variable functions that are representable over $\mathbb{B}$ is 14, but only 6 functions among them are representable over $\mathbb{B}_{\mathrm{red}}$. The role of the linear distributive transformation is used to fill the gap. That's why `Copy''` includes the subterm `LDtr`.

From what precedes we can conclude that we can represent all the boolean circuits over $\mathbb{B}$.

As described before, while the number of two variable functions that are representable over $\mathbb{B}$ are fourteen, it is six over $\mathbb{B}_{\mathrm{red}}$. The difference is closely related to the fact that linear distributive law $A \otimes (B \parr C) \multimap (A \otimes B) \parr C$ is provable in Linear Logic ([3]), but the inverse $(A \otimes B) \parr C \multimap A \otimes (B \parr C)$ is not. In order to pursue the topic further we need to introduce the knowledge of proof nets and proof structures (e.g., see [7]).

6 Concluding Remarks

We note that we found the term `and''` manually using the proof net syntax (and then translating the proof net into `and''`), but `Copy'` and `Copy''` interactively with Standard ML of New Jersey.

In this paper we showed that $\mathbb{B}_{\mathrm{HM}}$ is not the only choice in order to establish P-time completeness of Linear Lambda Calculus. Let us call such a type like $\mathbb{B}_{\mathrm{HM}}$ and $\mathbb{B}$ a *P-time complete* type.

From our result a natural question comes up: which linear type other than $\mathbb{B}_{\mathrm{HM}}$ and $\mathbb{B}$ and its two normal forms establishes P-time completeness of Linear Lambda Calculus? For example it is unlikely that $\mathbb{B}' = p \multimap (p \multimap p) \multimap (p \multimap p) \multimap p$ and its two normal forms establish that: at least completely new ideas are needed to represent the *and* gate and copy function in $\mathbb{B}'$. Since $\mathbb{B}'$ can represent only constant functions and projections without ML polymorphism as shown in [6], ML polymorphism must be used for the goal. But it is an easy exercise to show that $\mathbb{B}_{\mathrm{TWIST}} = (p \otimes p) \multimap (p \otimes p)$ and its two normal forms are P-time complete by using [5] described in Section 3 and the linear distributive transformation. Our answer is given in Appendix A.

After the completion of the first draft, we were informed that a proof of P-time completeness of $\mathbb{B}_{\mathrm{TWIST}} \otimes \mathbb{B}_{\mathrm{TWIST}}$ was given in [11]. That encoding refers to the internal structure of $\mathbb{B}_{\mathrm{TWIST}} \otimes \mathbb{B}_{\mathrm{TWIST}}$ using the symmetry of de Morgan duality of the “and” and “or” gates. We note that in order to establish P-time completeness of $\mathbb{B}_{\mathrm{TWIST}} \otimes \mathbb{B}_{\mathrm{TWIST}}$, another method is possible.

Proposition 6.1. *Let*

$$\langle \texttt{Ty1}, \texttt{T1}, \texttt{F1}, \texttt{Copy1}, \texttt{not1}, \texttt{and1} \rangle \quad \textit{and} \quad \langle \texttt{Ty2}, \texttt{T2}, \texttt{F2}, \texttt{Copy2}, \texttt{not2}, \texttt{and2} \rangle$$

be both P-time complete. Then $\langle \texttt{Ty1*Ty2}, \texttt{(T1,F2)}, \texttt{(F1,T2)} \rangle$ *is also P-time complete.*

Proof. We can define `not`, `Copy`, `and` for `Ty1*Ty2` as follows:

```
fun not (b1, b2) = (not1 b1, not2 b2);
fun Copy (b1, b2) =
let val (b11, b12) = Copy1 b1 in
let val (b21, b22) = Copy2 b2 in
((b11, b21), (b12, b22)) end end;
fun or2 b21 b22 = not2 (and2 (not2 b21) (not2 b22);
fun and b b’ =
let val (b11, b21) = b in
let val (b12, b22) = b’ in
(and1 b11 b12, or2 b21 b22) end end;
```

□

Note that the construction is an instance of the direct product construction of Boolean algebras. Hence $\langle \texttt{Ty1*Ty2}, \texttt{(T1,T2)}, \texttt{(F1,F2)} \rangle$ is also P-time complete, etc. Then we can conclude that for example, $\mathbb{B} \otimes \mathbb{B}_{\mathrm{HM}}$ as well as $\mathbb{B}_{\mathrm{TWIST}} \otimes \mathbb{B}_{\mathrm{TWIST}}$ is P-time complete.

It is also obvious that if $\mathbb{T}$ is a P-time complete type and $\mathbb{T}'$ is a type, then $\mathbb{T}[\mathbb{T}'/p]$ is P-time complete.

As wrote before, however, we could not prove that $\mathbb{B}_{\mathrm{red}}$ and its normal forms `True’` and `False’` establish P-time completeness of Linear Lambda Calculus. But we also could not prove that they cannot establish that. At this moment we do not have any idea to do that.

Finally, our type $\mathbb{B}$ and its generalization $(p \multimap p) \multimap \overbrace{(p \multimap p) \otimes \cdots \otimes (p \multimap p)}^{n}$ have further interesting properties. For example we can establish weak typed Böhm theorem over $\mathbb{B}$. But the subject is beyond the scope of this paper, and will be discussed elsewhere.

7 Postscript

This result was obtain December 2012. But it has never been published. After that, a more general result has been obtained in 2014 by extending the author’s previous result [6]. It is

under submission (see [8]). In addition, a tool in order to deal with the Linear Lambda Calculus has been developed by the author, which is available from

`https://staff.aist.go.jp/s-matsuoka/PNCalculator/index.html`

at this moment.

References

[1] Mackie, I., Román, L., and Abramsky, S.: An Internal Language for Autonomous Categories, *Applied Categorical Structures* **1** (1993) 311–343.

[2] Benton, P.N., Bierman, G.M., de Paiva, V.C.V., Hyland, J.M.E.: Term calculus for intuitionistic linear logic, LNCS 664 75–90 (1993)

[3] Blute, R., Scott, P.: Category Theory for Linear Logicians. *LMS Lecture Note Series* **316** 3–64 Cambridge University Press (2004)

[4] Girard, J.-Y.: Linear Logic. *Theoretical Computer Science* **50** 1–102 (1987)

[5] Mairson, H.G.: Linear Lambda Calculus and PTIME-completeness. *Journal of Functional Programing* **14**(6) 623–633 (2004)

[6] Matsuoka, S.: Weak Typed Böhm Theorem on IMLL. *Annals of Pure and Applied Logic* **145**(1) 37–90 (2007)

[7] Matsuoka, S.: A Coding Theoretic Study of MLL Proof Nets. *Mathematical Structures in Computer Science* **22**(3) 409–449 (2012)

[8] Matsuoka, S.: Strong Typed Böhm Theorem and Functional Completeness on the Linear Lambda Calculus. submitted. Available at `http://arxiv.org/abs/1505.01326` (2015) Also presented in Workshop on Constructivism and Computability (`http://www.jaist.ac.jp/is/labs/ishihara-lab/jlws2015/`)

[9] Milner, R., Tofte, M., Harper, R., and MacQueen, D.: The Definition of Standard ML (Revised). MIT Press (1997)

[10] Troelstra, A.S.: Lectures on Linear Logic. CSLI (1992)

[11] van Horn, D. and Mairson, H.G.: Relating complexity and precision in control flow analysis. *ICFP' 07* 85–96 ACM press (2007)

A A P-time completeness proof for $\mathbb{B}_{\mathrm{TWIST}}$

```
fun True0 (x, y) = (x, y);
val True0 = fn : 'a * 'b -> 'a * 'b
fun False0 (x, y) = (y, x);
val False0 = fn : 'a * 'b -> 'b * 'a
fun notTWIST0 B = fn p => let val (u, v) = p in B (v, u) end;
fun LDTrHM0 f x y g = let val (s, t) = f (x, y) in g s t end;
fun CopyTWIST0 P =
LDTrHM0 P (Pair True0 True0) (Pair False0 False0)
(fn U => fn V => U
(fn u1 => fn u2 =>
V (fn v1 => fn v2 =>
( (id (LDTrHM0 v1)) u1, (id (LDTrHM0 v2)) u2 ) ) ) );
fun andTWIST0 P Q
```

```
= LDTrHM0 P Q False0
(fn u => fn v => (id (LDTrHM0 v) u));
```