JFP **14** (6): 623–633, 2004.
DOI: 10.1017/S0956796804005131 Printed in the United Kingdom

FUNCTIONAL PEARL

Linear lambda calculus and PTIME-completeness

HARRY G. MAIRSON

Computer Science Department, Volen Center for Complex Numbers, Brandeis University, Waltham, MA 02254, UA
(*e-mail:* mairson@cs.brandeis.edu)

Abstract

We give transparent proofs of the PTIME-completeness of two decision problems for terms in the λ-calculus. The first is a reproof of the theorem that type inference for the simply-typed λ-calculus is PTIME-complete. Our proof is interesting because it uses no more than the standard combinators Church knew of some 70 years ago, in which the terms are *linear affine* – each bound variable occurs at most once. We then derive a modification of Church's coding of Booleans that is linear, where each bound variable occurs exactly once. A consequence of this construction is that any interpreter for linear λ-calculus requires polynomial time. The logical interpretation of this consequence is that the problem of *normalizing proofnets* for *multiplicative linear logic* (MLL) is also PTIME-complete.

1 Type inference for simply typed λ-calculus

The Circuit Value Problem (CVP) is to determine the output of a circuit, given an input to that circuit. CVP is complete for PTIME, because polynomial-time computations can be described by polynomial-sized circuits (Ladner, 1975). The Cook-Levin NP-completeness theorem, it should be noticed, merely augments these circuits with extra inputs which correspond to nondeterministic choices during a polynomial-time computation. We show how to code CVP into simply-typed λ-terms, where both type inference and term evaluation are synonymous with circuit evaluation.

The programs we write to evaluate circuits are not perverse: they are completely natural, and are built out of the standard Church coding of Boolean logic (e.g. see Hindley & Seldin (1986)). We use ML as a presentation device, without exploiting its `let`-polymorphism. That is, we use the convenience of naming to identify λ-terms of constant size, used to build circuits. Had we expanded the definitions, the term representing the circuit would grow by only a constant factor, and become harder to read. Here, then, are the standard, classical combinators, coded in ML:

```
- fun True x y= x;
val True = fn : 'a -> 'b -> 'a
- fun False x y= y;
val False = fn : 'a -> 'b -> 'b
- fun Not p= p False True;
val Not = fn: (('a -> 'b -> 'b) -> ('c -> 'd -> 'c) -> 'e) -> 'e
- fun And p q= p q False;
val And = fn : ('a -> ('b -> 'c -> 'c) -> 'd) -> 'a -> 'd
- fun Or p q= p True q;
val Or = fn : (('a -> 'b -> 'a) -> 'c -> 'd) -> 'c -> 'd
```

Read `True` as the constant *if true*, where "if true then `x` else `y`" reduces to `x`. Similarly, read the body of `And` as "if `p` then `q` else `False`." Observe that `True` and `False` have different types. Notice the read-eval-print loop where the interpreter reads untyped code, and then returns procedures with inferred type information automatically computed. We already know from our study of the untyped λ-calculus how these terms work to code Boolean functions. However, when these functions are used, notice that they output functions as values; moreover, the principal types (i.e. the *most general types*, constrained only by the typing rules of the simply-typed lambda calculus, and nothing else) of these functions identify them uniquely as `True` or `False`:

```
- Or False True;
val True = fn : 'a -> 'b -> 'a
- And True False;
val False = fn : 'a -> 'b -> 'b
- Not True;
val False = fn : 'a -> 'b -> 'b
```

As a consequence, while the compiler does not explicitly reduce the above expressions to normal form, hence computing an "answer," its type inference mechanism implicitly carries out that reduction to normal form, expressed in the language of first-order unification.

The computations over Boolean values can be understood in the context of the graph representations of the types (see Figure 1). Notice how the nodes in the *type* for `And` can be associated uniquely with subterms from the *code* for `And` – a *linearity* very much in the spirit of linear logic.

We now observe a certain weirdness: namely, that *nonlinearity* destroys the isomorphism between terms and types. Suppose we define a function `Same` that is a kind of identity function:

```
- fun Same p= p True False;
val Same = fn : (('a -> 'b -> 'a) -> ('c -> 'd -> 'd) -> 'e) -> 'e
- Same True;
val it = fn : 'a -> 'b -> 'a
- Same False;
val it = fn : 'a -> 'b -> 'b
```

https://doi.org/10.1017/S0956796804005131 Published online by Cambridge University Press

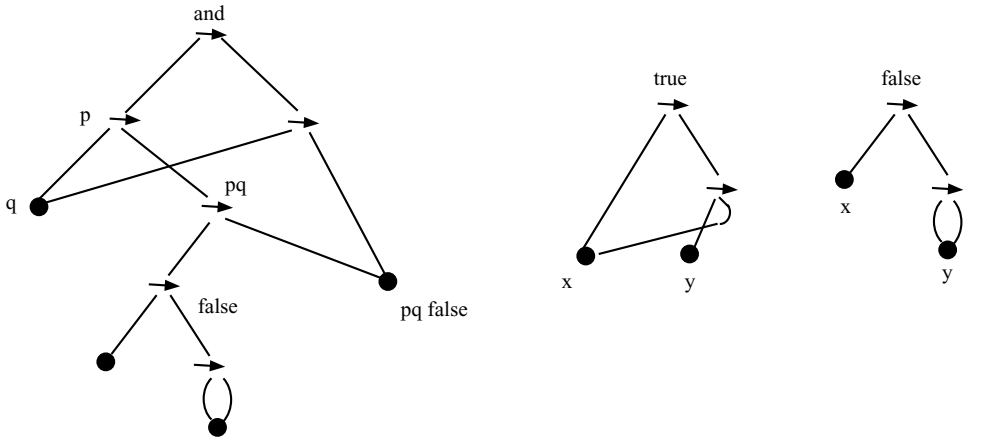

Fig. 1. Graph representations of the terms and principal types of `And`, `True`, `False`.

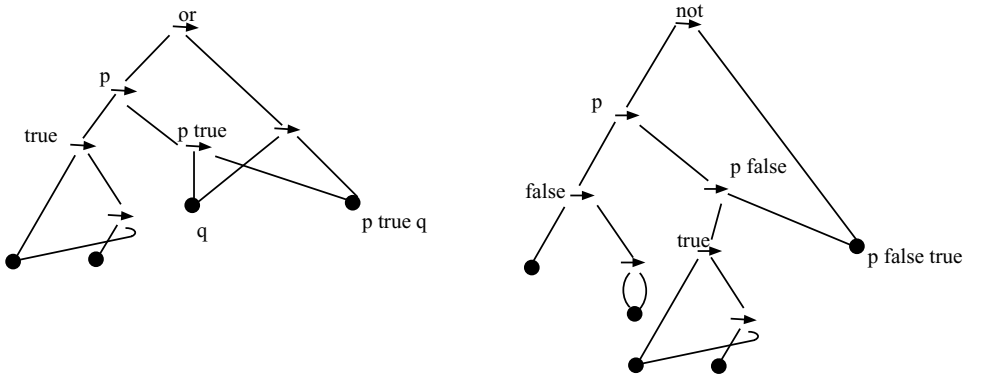

Fig. 2. Graph representations of the terms and principal types of `Or`, `Not`.

Now define a nonlinear function `Weird` that uses its input twice:

```
- fun K x y= x;
val K = fn : 'a -> 'b -> 'a
- fun Weird p= K (Same p) (Not p);
val Weird = fn : (('a -> 'a -> 'a) -> ('b -> 'b -> 'b) -> 'c) -> 'c
- Weird True;
val it = fn : 'a -> 'a -> 'a
- Weird False;
val it = fn : 'a -> 'a -> 'a
```

Even though `(Weird` *p*`)` reduces to *p*, its type is not a function of *p*; thus the principal type no longer identifies the normal form. The problem with `Weird` is that its input occurs twice, and that each occurrence must have the same type. Thus the types of `Same` and `Not` are unified: each gets type `(('a -> 'a -> 'a) -> ('b -> 'b -> 'b) -> 'c) -> 'c`. And then `Same p` returns a value (`True` or `False`), but

https://doi.org/10.1017/S0956796804005131 Published online by Cambridge University Press

always of type `'a -> 'a -> 'a`. Notice also that defining `fun AlsoWeird p= Or p p` will produce a type error, for similar reasons.

We fix this problem with a functional *fanout gate*, which we call `Copy`, in the spirit of the duplication in the exponentials of linear logic:

```
- fun Pair x y z= z x y;
val Pair = fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c
- fun Copy p= p (Pair True True) (Pair False False);
val Copy = fn : ((((’a -> ’b -> ’a) -> (’c -> ’d -> ’c) -> ’e) -> ’e)
-> (((’f -> ’g -> ’g) -> (’h -> ’i -> ’i) -> ’j) -> ’j) -> ’k) -> ’k
- Copy True;
val it = fn : ((’a -> ’b -> ’a) -> (’c -> ’d -> ’c) -> ’e) -> ’e
- Copy False;
val it = fn : ((’a -> ’b -> ’b) -> (’c -> ’d -> ’d) -> ’e) -> ’e
```

`Copy` restores the linearity of terms, so that each variable occurs at most once. Now we can again compute unweirdly:

```
- fun Unweird p= (Copy p) (fn p1=> fn p2=> K (Same p1) (Not p2));
val Unweird = fn : ((((’a -> ’b -> ’a) -> (’c -> ’d -> ’c) -> ’e)
-> ’e) -> (((’f -> ’g -> ’g) -> (’h -> ’i -> ’i) -> ’j) -> ’j) -> (((’k
-> ’l -> ’k) -> (’m -> ’n -> ’n) -> ’o) -> ((’p -> ’q -> ’q) -> (’r
-> ’s -> ’r) -> ’t) -> ’o) -> ’u) -> ’u
- Unweird True;
val it = fn : ’a -> ’b -> ’a
- Unweird False;
val it = fn : ’a -> ’b -> ’b
```

In this coding, `Copy p` produces a pair of values, each β-equivalent to `p`, but not sharing any type variables. The elements of the pair are bound to `p1` and `p2`, respectively, and the computation thus proceeds linearly.

A Boolean circuit can now be coded as a λ-term by labelling its (wire) edges and traversing them bottom-up, inserting logic gates and fanout gates appropriately. We consider the circuit example shown in Figure 3; the circuit is realized by the ML code

```
- fun circuit e1 e2 e3 e4 e5 e6=
  let val e7= And e2 e3 in
    let val e8= And e4 e5 in
      let val (e9,e10)= Copy (And e7 e8) in
        let val e11= Or e1 e9 in
          let val e12= Or e10 e6 in
            Or e11 e12
          end
        end
      end
    end
  end;
```

Removing the syntactic sugar of `let`, this essentially straight-line code “compiles” to the less comprehensible simply-typed term

https://doi.org/10.1017/S0956796804005131 Published online by Cambridge University Press

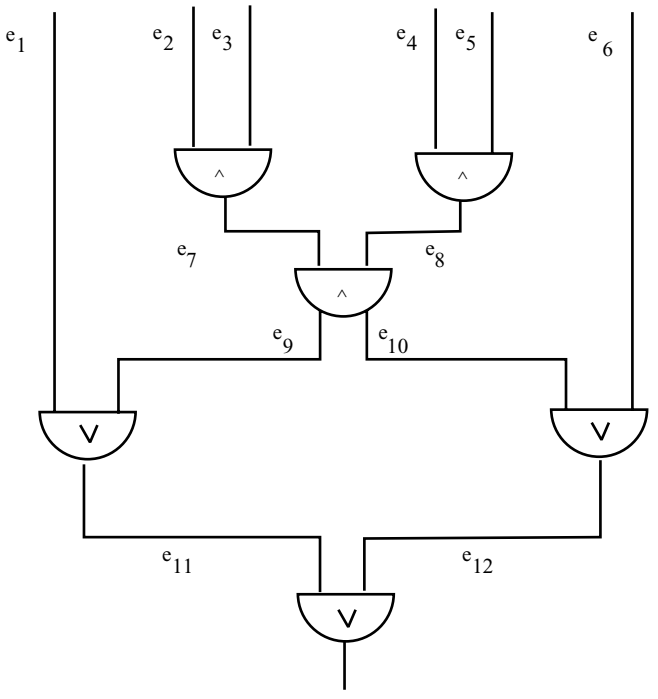

Fig. 3. Labelling of a Boolean circuit.

```
- fun circuit e1 e2 e3 e4 e5 e6=
(fn e7=>
  (fn e8=>
    (Copy (And e7 e8))
      (fn e9=> fn e10=>
        (fn e11=>
          (fn e12=> Or e11 e12)
          (Or e10 e6))
        (Or e1 e9)))
  (And e4 e5))
(And e2 e3);
```

The use of procedure naming is incidental: we could just plug in the code for the logic gates instead of using their names. We can now plug inputs into the circuit, and the type inference mechanism is forced to "evaluate" the circuit:

```
- circuit False True True True True False;
val it = fn : 'a -> 'b -> 'a
```

The output is "True"—`True` is the only closed, normal form with the given type!

We can make the code for the circuit look even more like "straight-line code" by introducing the *continuation-passing* version of unary and binary Boolean functions:

```
- fun cp1 fnc p k= k (fnc p);
val cp1 = fn : ('a -> 'b) -> 'a -> ('b -> 'c) -> 'c
- fun cp2 fnc p q k= k (fnc p q);
val cp2 = fn : ('a -> 'b -> 'c) -> 'a -> 'b -> ('c -> 'd) -> 'd
```

https://doi.org/10.1017/S0956796804005131 Published online by Cambridge University Press

```
- val Notgate= cp1 Not;
val it = fn : (('a -> 'b -> 'b) -> ('c -> 'd -> 'c) -> 'e) ->
   ('e -> 'f) -> 'f
- val Andgate= cp2 And;
val it = fn : ('a -> ('b -> 'c -> 'c) -> 'd) -> 'a -> ('d -> 'e) -> 'e
- val Orgate= cp2 Or;
val it = fn : (('a -> 'b -> 'a) -> 'c -> 'd) -> 'c -> ('d -> 'e) -> 'e
```

We now write instead:

```
- fun circuit e1 e2 e3 e4 e5 e6=
  (Andgate e2 e3 (fn e7=>
  (Andgate e4 e5 (fn e8=>
  (Andgate e7 e8 (fn f=>
  Copy f (fn e9=> fn e10=>
  (Orgate e1 e9 (fn e11=>
  (Orgate e10 e6 (fn e12=>
  Or e11 e12))))))));
val circuit = fn : (('a -> 'b -> 'a) -> 'c -> ('d -> 'e -> 'd) -> 'f
-> 'g) -> ('h -> ('i -> 'j -> 'j) -> 'k -> ('l -> 'm -> 'm) -> (((''n
-> 'o -> 'n) -> ('p -> 'q -> 'p) -> 'r) -> 'r) -> (((''s -> 't -> 't)
-> ('u -> 'v -> 'v) -> 'w) -> 'w) -> ('c -> (('x -> 'y -> 'x) -> 'z ->
'f) -> 'g) -> 'ba) -> 'h -> ('bb -> ('bc -> 'bd -> 'bd) -> 'k) -> 'bb
-> 'z -> 'ba
- circuit False True True True True False;
val it = fn : 'a -> 'b -> 'a
```

Why does the above contruction imply a PTIME-completeness result for type inference? Containment in PTIME is well known for this problem, because type inference is synonymous with finding the solution to a set of first-order constraints over type variables, possibly constants, and the binary contructor $\rightarrow$.

The PTIME-hardness is more interesting: a property ϕ is PTIME-hard if, given any fixed PTIME Turing machine M and an input x, the computation of M on x can be compiled, using $O(\log |x|)$ space, into a problem instance I, where I has property ϕ iff M accepts x. The circuit value problem is such a problem: given circuit C and input $\vec{v}$, is the output of C "true"? Such a circuit can be computed by a compiler using only logarithmic space, and a further logarithmic space algorithm (which we have essentially given here) compiles C and $\vec{v}$ into a λ-term E, where E has the principal type of `True` iff C outputs "true" on $\vec{v}$. Composing these two compilers gives the PTIME-hardness of type inference.

2 Linear λ-calculus and multiplicative linear logic

The above coding uses K-redexes as well as linearity in an essential way: the codings of `True` and `False` each discard one of their arguments. Now we construct Boolean terms and associated gates where each bound argument is used *exactly* once, so that we have constructions in *linear λ-calculus*. We then see that the process of normalizing terms is complete for PTIME.

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

https://doi.org/10.1017/S0956796804005131 Published online by Cambridge University Press

Observe that the Booleans are now coded by abstract *pairs*, where the first component is our old, "real" value, and the second is some kind of garbage that preserves linearity. Negation is easy:

```
- fun Not P x y= P y x;
val Not = fn : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
- Not True;
val False = fn : 'a -> 'b -> ('b -> 'a -> 'c) -> 'c
- Not False;
val True = fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c
```

Now consider what happens if we code `Or P Q` as `P True Q`—it doesn't work. Observe that `Or True False` = `True True False` = `Pair True False`. In general, `Or` P Q will equal `Pair` $(P \vee Q)$ $(P \to Q)$. The second component of the pair is garbage: we need to dispose of it while preserving linearity. The function `id` below turns Boolean garbage into the identity function:

```
- 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
- id True;
val it = fn : 'a -> 'a
- id False;
val it = fn : 'a -> 'a
```

We use `id` to collect garbage in Boolean gates:

```
- fun Or P Q= P True Q (fn u=> fn v=> (id v) u);
val Or = fn  :
(('a -> 'b -> ('a -> 'b -> 'c) -> 'c) -> 'd -> ('e -> (('f -> 'f) ->
('g -> 'g) -> ('h -> 'h) -> 'e -> 'i) -> 'i) -> 'j) -> 'd -> 'j
- Or True False;
val it = fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c
- Or False False;
val it = fn : 'a -> 'b -> ('b -> 'a -> 'c) -> 'c
- fun And P Q= P Q False (fn u=> fn v=> (id v) u);
val And = fn  :
('a -> ('b -> 'c -> ('c -> 'b -> 'd) -> 'd) -> ('e -> (('f -> 'f) ->
('g -> 'g) -> ('h -> 'h) -> 'e -> 'i) -> 'i) -> 'j) -> 'a -> 'j
- And True True;
val it = fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c
- And False True;
val it = fn : 'a -> 'b -> ('b -> 'a -> 'c) -> 'c
```

Again, we have a gate that copies Boolean values, with a slightly more complicated garbage collector:

https://doi.org/10.1017/S0956796804005131 Published online by Cambridge University Press

```
- fun Copy P= P (Pair True True) (Pair False False)
               (fn U=> fn V=>
                  U (fn u1=> fn u2=>
                       V (fn v1=> fn v2=>
                            (Pair ((id v1) u1) ((id v2) u2)))));
val Copy = fn  :
((((’a -> ’b -> (’a -> ’b -> ’c) -> ’c) -> (’d -> ’e -> (’d -> ’e ->
’f) -> ’f) -> ’g) -> ’g) -> (((’h -> ’i -> (’i -> ’h -> ’j) -> ’j) ->
(’k -> ’l -> (’l -> ’k -> ’m) -> ’m) -> ’n) -> ’n) -> (((’o -> ’p ->
’q) -> ’r) -> ((((’s -> ’s) -> (’t -> ’t) -> (’u -> ’u) -> ’o -> ’v)
-> ((’w -> ’w) -> (’x -> ’x) -> (’y -> ’y) -> ’p -> ’z) -> (’v -> ’z
-> ’ba) -> ’ba) -> ’q) -> ’r) -> ’bb) -> ’bb
```

Now we take the continuation-passing versions of the logic gates:

```
- val Notgate= cp1 Not;
val Notgate = fn : (’a -> ’b -> ’c) -> ((’b -> ’a -> ’c) -> ’d) -> ’d
- val Orgate= cp2 Or;
val Orgate = fn  :
((’a -> ’b -> (’a -> ’b -> ’c) -> ’c) -> ’d -> (’e -> ((’f -> ’f) ->
(’g -> ’g) -> (’h -> ’h) -> ’e -> ’i) -> ’i) -> ’j) -> ’d ->
   (’j -> ’k) -> ’k
- val Andgate= cp2 And;
val Andgate = fn  :
(’a -> (’b -> ’c -> (’c -> ’b -> ’d) -> ’d) -> (’e -> ((’f -> ’f) ->
(’g -> ’g) -> (’h -> ’h) -> ’e -> ’i) -> ’i) -> ’j) -> ’a ->
   (’j -> ’k) -> ’k
```

Once again, we have a circuit simulator:

```
- fun circuit e1 e2 e3 e4 e5 e6=
  Andgate e2 e3 (fn e7=>
  Andgate e4 e5 (fn e8=>
  Andgate e7 e8 (fn f=>
  Copy f (fn e9=> fn e10=>
  Orgate e1 e9 (fn e11=>
  Orgate e10 e6 (fn e12=>
  Or e11 e12))))));
val circuit = fn : ((’a -> ’b -> (’a -> ’b -> ’c) -> ’c) -> ’d -> (’e
-> ((’f -> ’f) -> (’g -> ’g) -> (’h -> ’h) -> ’e -> ’i) -> ’i) -> (’j
-> ’k -> (’j -> ’k -> ’l) -> ’l) -> ’m -> (’n -> ((’o -> ’o) -> (’p ->
’p) -> (’q -> ’q) -> ’n -> ’r) -> ’r) -> ’s) -> (’t -> (’u -> ’v ->
(’v -> ’u -> ’w) -> ’w) -> (’x -> ((’y -> ’y) -> (’z -> ’z) -> (’ba ->
’ba) -> ’x -> ’bb) -> ’bb) -> ’bc -> (’bd -> ’be -> (’be -> ’bd ->
’bf) -> ’bf) -> (’bg -> ((’bh -> ’bh) -> (’bi -> ’bi) -> (’bj -> ’bj)
-> ’bg -> ’bk) -> ’bk) -> (((’bl -> ’bm -> (’bl -> ’bm -> ’bn) -> ’bn)
-> (’bo -> ’bp -> (’bo -> ’bp -> ’bq) -> ’bq) -> ’br) -> ’br) ->
(((’bs -> ’bt -> (’bt -> ’bs -> ’bu) -> ’bu) -> (’bv -> ’bw -> (’bw ->
’bv -> ’bx) -> ’bx) -> ’by) -> ’by) -> (((’bz -> ’ca -> ’cb) -> ’cc)
-> ((((’cd -> ’cd) -> (’ce -> ’ce) -> (’cf -> ’cf) -> ’bz -> ’cg) ->
((’ch -> ’ch) -> (’ci -> ’ci) -> (’cj -> ’cj) -> ’ca -> ’ck) -> (’cg
-> ’ck -> ’cl) -> ’cl) -> ’cb) -> ’cc) -> (’d -> ((’cm -> ’cn -> (’cm
```

https://doi.org/10.1017/S0956796804005131 Published online by Cambridge University Press

```
-> 'cn -> 'co) -> 'co) -> 'cp -> ('cq -> (('cr -> 'cr) -> ('cs -> 'cs)
-> ('ct -> 'ct) -> 'cq -> 'cu) -> 'cu) -> 'm) -> 's) -> 'cv) -> 't ->
('cw -> ('cx -> 'cy -> ('cy -> 'cx -> 'cz) -> 'cz) -> ('da -> (('db ->
'db) -> ('dc -> 'dc) -> ('dd -> 'dd) -> 'da -> 'de) -> 'de) -> 'bc) ->
'cw -> 'cp -> 'cv
```

It is said that the proof of the pudding is in the tasting; here, the proof of the coding is in the testing:

```
- circuit False True True True True False;
val it = fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c
```

We further assert the following *pearl theorem:* the $\beta\eta$-normal form of any simply-typed, *linear* λ-term E, with or without free variables and K-redexes, may be inferred from only its principal type σ. Moreover, every linear term is simply typable (Hindley, 1989). Notice that it is the *multiple* occurrence of variables, (for example, in iterators such as Church numerals) which weakens this pearl theorem, and all that is left is *parametricity*, where the type indicates a property instead of identifying the exact normal form; see, for instance, Mairson (1991) and Wadler (1989). Multiple occurrences of a variable are also the way to make this problem one of *typability*; for example, in the non-affine coding presented in this section, we may define

```
- fun Test p u v= (p u v) (fn a=> fn b=> b u);
val Test = fn : ('a -> 'b -> ('c -> ('a -> 'd) -> 'd) -> 'e) -> 'a ->
   'b -> 'e
```

Then `Test True` has a type, but `Test False` does not.

We conclude by mentioning some more technical connections of the above constructions to the normalization of proofs in linear logic. Every typed linear λ-term represents a proof in *multiplicative linear logic* (MLL) (Girard, 1987), where the conclusions of the proof represent the types of the free variables and output of a term, and the structure of the proof represents the term. Here is the logic:

$$\frac{}{A, A^{\perp}}\ \text{Axiom} \qquad \frac{\Gamma, A \quad \Delta, A^{\perp}}{\Gamma, \Delta}\ \text{Cut} \qquad \frac{\Gamma, A \quad \Delta, B}{\Gamma, \Delta, A \otimes B}\ \otimes \qquad \frac{\Gamma, A, B}{\Gamma, A \parr B}\ \parr$$

These rules can type graphs and graph reduction for linear λ-calculus, an expressive subset of MLL proofnets. A *wire* (Axiom rule) has (expression) type A in one direction, and (continuation) type $A^{\perp}$ in the other. The Cut connects an expression to a continuation. The $\otimes$ pairs a continuation and input for a function, and thus represents an application (@) node. The $\parr$ unpairs them, representing a λ-node; the associated function can be thought of dually as either transforming the continuation, or the expression.

https://doi.org/10.1017/S0956796804005131 Published online by Cambridge University Press

The reduction rule for this logic is like β-reduction in the λ-calculus. We negate variables appropriately to make this analogy transparent.

$$\dfrac{\dfrac{\Gamma, B^{\perp} \quad \Delta, A}{\Gamma, \Delta, B^{\perp} \otimes A}\otimes \quad \dfrac{\Sigma, A^{\perp}, B}{\Sigma, A^{\perp} \parr B}\parr}{\Gamma, \Sigma, \Delta}\,\mathrm{Cut} \quad \Rightarrow \quad \dfrac{\dfrac{\Gamma, B^{\perp} \quad \Sigma, A^{\perp}, B}{\Gamma, \Sigma, A^{\perp}}\,\mathrm{Cut} \qquad \Delta, A}{\Gamma, \Sigma, \Delta}\,\mathrm{Cut}$$

This says: to reduce the connecting of a function (of type $A^{\perp} \parr B$) to an application context (of type $B^{\perp} \otimes A$), connect the output of the function (B) to the continuation ($B^{\perp}$), and the input (A) to the parameter request of the function ($A^{\perp}$).

Every time an MLL proof is reduced, the proof gets smaller. Reduction is then easily seen to be polynomial time. Given that our circuit simulation is a linear λ-term, it is representable by an MLL proof. Thus proof(net) normalization for multiplicative linear logic is complete for polynomial time. We observe finally that with the introduction of second-order quantification in the style of System F (Girard, 1972), we can define the linear type $\mathsf{Bool} = \forall\alpha.\alpha \multimap \alpha \multimap \alpha \otimes \alpha$, and give binary Boolean functions the uniform type $\mathsf{Bool} \multimap \mathsf{Bool} \multimap \mathsf{Bool}$. The programming techniques we have described above then let us code projection functions of type $\mathsf{Bool} \otimes \mathsf{Bool} \multimap \mathsf{Bool}$ and duplication functions $\mathsf{Bool} \multimap \mathsf{Bool} \otimes \mathsf{Bool}$, *without* weakening or contraction. In this limited way, we recover the features of exponentials from linear logic. This analysis is generalized in Mairson & Terui (2003), where the types other than Bool for which exponentials can be simulated are characterized; in addition, we show that *light* multiplicative linear logic, a proper subset of light affine logic (where copying is restricted), can represent all the PTIME functions. At this point, however, the discussion is no longer a pearl.

Historical note

The analysis of type inference through linear λ-calculus is something I understood while working on this subject roughly a decade ago. While the linear features were very apparent, it is only recently that I saw the real connection with linear logic. The results of section 1 were outlined in more technical terms in a paper joint with Fritz Henglein (Henglein & Mairson, 1994), where they were used to derive lower bounds on type inference for Systems F and F_{ω}. I wish to acknowledge both his collaboration in this presentation, as well as the previous appearance of these ideas in this very journal. I take the liberty of repeating them here – it is not for nothing that this endeavor is called *re*search – not only because the ideas form a pearl that deserves to be known more widely, but also because the clarity of the original version in the journal was lost due to numerous production errors.

Acknowledgements

I am deeply grateful to Kazushige Terui, whose invited lecture at the 2002 Linear Logic workshop in Copenhagen (Terui, 2002), as well as continued discussions afterwards, have inspired me to think more about many of these issues.

References

Girard, J.-Y. (1972) *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de doctorat d'état, Université Paris VII.

Girard, J.-Y. (1987) Linear logic. *Theor. Comput. Sci.* **50**.

Henglein, F. and Mairson, H. G. (1994) The complexity of type inference for higher-order typed lambda calculi. *J. Functional Program.* **4**(4), 435–477.

Hindley, J. R. (1989) BCK-combinators and linear lambda-terms have types. *Theor. Comput. Sci.* **64(1)**, 97–105.

Hindley, J. R. and Seldin, J. P. (1986) *Introduction to Combinatory Logic and Lambda Calculus.* Cambridge University Press.

Ladner, R. (1975) The circuit value problem is log space complete for P. *SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, **7**.

Mairson, H. G. (1991) Outline of a proof theory of parametricity. *Proceedings International Conference on Functional Programming Languages and Computer Architecture: LNCS 523*, pp. 313–327. Springer-Verlag.

Mairson, H. G. and Terui, K. (2003) On the computational complexity of cut elimination in linear logic. *Proceedings Italian Conference on Theoretical Computer Science: LNCS 2841*, pp. 23–36. Springer-Verlag.

Terui, K. (2002) On the complexity of cut-elimination in linear logic (invited lecture). *Federated Logic Conference (FLOC) 2002, Workshop on Linear Logic*, Copenhagen, Denmark.

Wadler, P. (1989) Theorems for free! *Proceedings International Conference on Functional Programming Languages and Computer Architecture*, pp. 347–359. ACM Press.